Министерство образования Республики Беларусь

ПОЛОЦКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Кафедра технологий программирования

**Методические указания**

**к лабораторной работе № 6  
 по курсу «Основы алгоритмизации   
и программирования»**

«Указатели. Арифметика указателей»

Преподаватель: Войтехович  
Агния Витольдовна

Составитель: Войтехович А.В.

Полоцк, 2016ЦЕЛЬ РАБОТЫ

Научиться работать с указателями. Изучить операции, которые можно произвести над указателями и научиться правильно их применять.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

# 1 Что такое указатели

**Указатель** — это переменная, значением которой является адрес некоторого объекта (обычно другой переменной) в памяти компьютера. Например, если одна переменная содержит адрес другой переменной, то говорят, что первая переменная указывает (ссылается) на вторую. Это иллюстрируется с помощью рисунка 1.

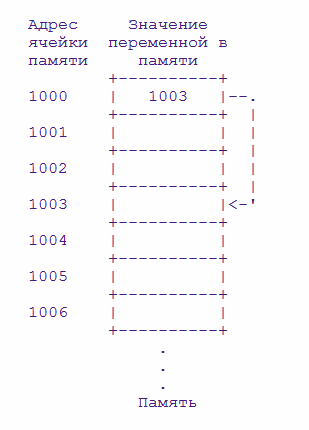


Рисунок 1 – Одна переменная ссылается на другую

# 2 Указательные переменные

Как известно, переменную, являющуюся указателем, нужно соответствующим образом объявить. Объявление указателя состоит из имени базового типа, символа \* и имени переменной. Общая форма объявления указателя следующая:

тип \*имя;

Здесь тип — это базовый тип указателя, им может быть любой правильный тип. Имя определяет имя переменной-указателя.

Базовый тип указателя определяет тип объекта, на который указатель будет ссылаться. Фактически указатель любого типа может ссылаться на любое место в памяти. Однако выполняемые с указателем операции существенно зависят от его типа. Например, если объявлен указатель типа int \*, компилятор предполагает, что любой адрес, на который он ссылается, содержит переменную типа int, хоть это может быть и не так. Следовательно, объявляя указатель, необходимо убедиться, что его тип совместим с типом объекта, на который он будет ссылаться.

# 3 Операция получения адреса (&) и раскрытия ссылки (\*)

Для работы с указателями, в основном, используются два оператора.

Первый из них — оператор **&**, это унарный оператор, возвращающий адрес операнда в памяти (унарной операцией называется операция, имеющая только один операнд). Он называется также оператором получения (взятия) адреса. Например, оператор

m = &count;

записывает в переменную m адрес переменной count. Этот адрес представляет собой адрес ячейки памяти компьютера, в которой размещена переменная. Адрес и значение переменной — совершенно разные понятия. Выражение "&переменная" означает "адрес переменной". Следовательно, инструкция m = &scount; означает: "Переменной m присвоить адрес, по которому расположена переменная count;".

Допустим, переменная count расположена в памяти в ячейке с адресом 2000, а ее значение равно 100. Тогда в предыдущем примере переменной m будет присвоено значение 2000.

Второй рассматриваемый оператор **\*** является двойственным (дополняющим) по отношению к **&**. Оператор **\*** называется также оператором косвенности, оператором раскрытия ссылки и оператором разыменования адреса. Оператор \* является унарным оператором, он возвращает значение объекта, расположенного по указанному адресу. Операндом для **\*** служит адрес объекта (переменной). Например, если переменная m содержит адрес переменной count, то оператор

q = \*m;

записывает значение переменной count в переменную q. В нашем примере переменная q получит значение 100, потому что по адресу 2000 записано число 100, причем этот адрес записан в переменной m. Выражение "\* адрес" означает "по адресу". Наш фрагмент программы можно прочесть как "q получает значение, расположенное по адресу m".

К сожалению, символ операции раскрытия ссылки совпадает с символом операции умножения, а символ операции получения адреса — с символом операции побитового И. Необходимо помнить, что эти операторы не имеют никакого отношения друг к другу. Операторы **\*** и **&** имеют более высокий приоритет, чем любая арифметическая операция, кроме унарного минуса, имеющего такой же приоритет.

Если переменная является указателем, то в объявлении перед ее именем нужно поставить символ \*, он сообщит компилятору о том, что это указатель на переменную данного типа. Например, объявление указателя на переменную типа char записывается так:

char \*ch;

Необходимо понимать, что ch — это не переменная типа char, а указатель на переменную данного типа, это совершенно разные вещи. Тип данных, на который указывает указатель (в данном случае это char), называется базовым типом указателя (иногда называется также основным или исходным типом.). Сам указатель является переменной, содержащей адрес объекта базового типа. Компилятор учтет размер указателя в архитектуре компьютера и выделит для него необходимое количество байтов, чтобы в указатель поместился адрес. Базовый тип указателя определяет тип объекта, хранящегося по этому адресу.

В одном операторе объявления можно одновременно объявить и указатель, и переменную, не являющуюся указателем. Например, оператор

int x, \*y, count;

объявляет х и count как переменные целого типа, а у — как указатель на переменную целого типа.

В следующей программе операторы **\*** и **&** используются для записи значения 10 в переменную target. Программа выведет значение **10** на экран.

#include <iostream>

using namespace std;

int main()

{

int target, source;

int \*m;

source = 10;

m = &source;

target = \*m;

cout << target << endl;

return 0;

}

# 4 Указательные выражения

В общем случае выражения с указателями подчиняются тем же правилам, что и обычные выражения. В этом разделе рассматривается применение указательных выражений в операциях присваивания, преобразования типов, а также в операциях "указательной" арифметики.

## 4.1 Присваивание указателей

Указатель можно использовать в правой части оператора присваивания для присваивания его значения другому указателю. Если оба указателя имеют один и тот же тип, то выполняется простое присваивание, без преобразования типа. В следующем примере

**#include <iostream>**

**using namespace** std;

**int** main(**void**)

{

**int** x = 99;

**int** \*p1, \*p2;

p1 = &x;

p2 = p1;

/\* печать значения x дважды \*/

**cout** << "\tZna4enie po ADRESU p1 & p2:\t";

**cout** << \*p1 <<"\t\t"<< \*p2 << endl << endl;

/\* печать адреса x дважды \*/

**cout** << "\tZna4enie UKAZATELEJ p1 & p2:\t";

**cout** << p1 <<"\t"<< p2 << endl << endl;

**return** 0;

}

после присваивания

p1 = &x;

p2 = p1;

оба указателя (**p1** и **р2**) ссылаются на **х**. То есть, оба указателя ссылаются на один и тот же объект. Программа выводит на экран следующее:

Zna4enie po ADRESU p1 & p2 : 99 99

tZna4enie UKAZATELEJ p1 & p2: 0063FDF0 0063FDF0

Допускается присваивание указателя одного типа указателю другого типа. Однако для этого необходимо выполнить явное преобразование типа указателя (операция приведения типов), которая рассматривается в следующем разделе.

## 4.2 Преобразование типа указателя

Указатель можно преобразовать к другому типу. Эти преобразования бывают двух видов: с использованием указателя типа void \* и без его использования.

В языке С допускается присваивание указателя типа void \* указателю любого другого типа (и наоборот) без явного преобразования типа указателя. Тип указателя void \* используется, если тип объекта неизвестен. Например, использование типа void \* в качестве параметра функции позволяет передавать в функцию указатель на объект любого типа, при этом сообщение об ошибке не генерируется. Также он полезен для ссылки на произвольный участок памяти, независимо от размещенных там объектов. Например, функция размещения malloc() (будет рассматриватся позже) возвращает значение типа void \*, что позволяет использовать ее для размещения в памяти объектов любого типа.

В отличие от void \*, преобразования всех остальных типов указателей должны быть всегда явными (т.е. должна быть указана операция приведения типов). Однако следует учитывать, что преобразование одного типа указателя к другому может вызвать непредсказуемое поведение программы. Например, в следующей программе делается попытка присвоить значение **х** переменной **у** посредством указателя **р**. При компиляции программы сообщение об ошибке не генерируется, однако результат работы программы неверен.

#include <iostream>

using namespace std;

int main(void)

{

double x = 100.1f, y;

int \*p;

/\* В следующем операторе указателю на целое p

(присваивается значение, ссылающееся на double. \*/

p = (int \*) &x;

/\* Следующий оператор работает не так,

как ожидается. \*/

y = \*p;

/\* Следующий оператор не выведет число 100.1. \*/

cout << "\n\t X = " << y << endl << endl;

return 0;

}

Обратите внимание на то, что операция приведения типов применяется в операторе присваивания адреса переменной **х** (он имеет тип **double \***) указателю **p**, тип которого **int \***. Преобразование типа выполнено корректно, однако программа работает не так, как ожидается (по крайней мере, в большинстве оболочек). Для разъяснения проблемы предположим, что переменная **int** занимает в памяти 4 байта, а **double** — 8 байтов. Указатель **p** объявлен как указатель на целую переменную (т.е. типа **int**), поэтому оператор присваивания

y = \*р;

передаст переменной y только 4 байта информации, а не 8 байтов, необходимых для **double**. Несмотря на то, что **p** ссылается на объект **double**, оператор присваивания выполнит действие с объектом типа **int**, потому что p объявлен как указатель на **int**. Поэтому такое использование указателя **p** неправильное.

Приведенный пример подтверждает то, что операции с указателями выполняются в зависимости от базового типа указателей. Синтаксически допускается ссылка на объект с типом, отличным от типа указателя, однако при этом указатель будет "думать", что он ссылается на объект своего типа. Таким образом, операции с указателями управляются типом указателя, а не типом объекта, на который он ссылается.

Разрешен еще один тип преобразований: преобразование целого в указатель и наоборот. В этом случае необходимо применить операцию приведения типов (явное преобразование типа). Однако пользоваться этим средством нужно очень осторожно, потому что при этом легко получить непредсказуемое поведение программы. Явное преобразование типа не обязательно, если преобразуется нуль, то есть нулевой указатель.

## 4.3 Адресная арифметика

В языке С допустимы только две арифметические операции над указателями: суммирование и вычитание. Предположим, текущее значение указателя **p1** типа **int \*** равно 2000. Предположим также, что переменная типа **int** занимает в памяти 2 байта. Тогда после операции увеличения

p1++;

указатель **p1** принимает значение 2002, а не 2001. То есть, при увеличении на 1 указатель p1 будет ссылаться на следующее целое число. Это же справедливо и для операции уменьшения. Например, если **p1** равно 2000, то после выполнения оператора

p1--;

значение **p1** будет равно 1998.

Операции адресной арифметики подчиняются следующим правилам. После выполнения операции увеличения над указателем, данный указатель будет ссылаться на следующий объект своего базового типа. После выполнения операции уменьшения — на предыдущий объект. Применительно к указателям на char, операций адресной арифметики выполняются как обычные арифметические операции, потому что длина объекта char всегда равна 1. Для всех указателей адрес увеличивается или уменьшается на величину, равную размеру объекта того типа, на который они указывают. Поэтому указатель всегда ссылается на объект с типом, тождественным базовому типу указателя. Эта концепция иллюстрируется с помощью рисунка 2.

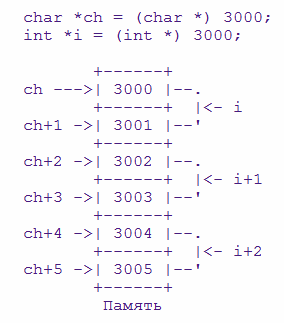


Рисунок 2 – Пример размещения в памяти переменных  
 char (слева) и int (справа)

Операции адресной арифметики не ограничены увеличением (инкрементом) и уменьшением (декрементом). Например, к указателям можно добавлять целые числа или вычитать из них целые числа. Выполнение оператора

p1 = p1 + 12;

"передвигает" указатель **p1** на 12 объектов в сторону увеличения адресов.

Кроме суммирования и вычитания указателя и целого, разрешена еще только одна операция адресной арифметики: можно вычитать два указателя. Благодаря этому можно определить количество объектов, расположенных между адресами, на которые указывают данные два указателя; правда, при этом считается, что тип объектов совпадает с базовым типом указателей. Все остальные арифметические операции запрещены. А именно: **нельзя делить и умножать указатели, суммировать два указателя, выполнять над указателями побитовые операции, суммировать указатель со значениями, имеющими тип float или double и т.д.**

## 4.4 Сравнение указателей

Стандартом С допускается сравнение двух указателей. Например, если объявлены два указателя **р** и **q**, то следующий оператор является правильным:

if(p < q)

cout << "p ссылается на меньший адрес, чем q\n";

Как правило, сравнение указателей может оказаться полезным, только тогда, когда два указателя ссылаются на общий объект, например, на массив. В качестве примера рассмотрим программу с двумя стековыми функциями, предназна-ченными для записи и считывания целых чисел. Стек — это список, использующий систему доступа "первым вошел — последним вышел". Иногда стек сравнивают со стопкой тарелок на столе: первая, поставленная на стол, будет взята последней. Стеки часто используются в компиляторах, интерпретаторах, программах обработки крупноформатных таблиц и в других системных программах. Для создания стека необходимы две функции: push() и pop(). Функция push() заносит числа в стек, a pop() — извлекает их. В данном примере эти функции используются в main(). При вводе числа с клавиатуры, программа помещает его в стек. Если ввести 0, то число извлекается из стека. Программа завершает работу при вводе -1.

#include <iostream>

#include <windows.h>

#include <clocale>

using namespace std;

#define SIZE 50

void push(int i);

int pop(void);

int \*tos, \*p1, stack[SIZE];

int main(void)

{

setlocale(LC\_ALL, "Russian");

SetConsoleOutputCP(1251);

SetConsoleCP(1251);

int value;

tos = stack; /\* tos ссылается на основание стека \*/

p1 = stack; /\* инициализация p1 \*/

do {

cout << "Введите значение: ";

cin >> value;

if(value != 0) push(value);

else

{ cout << "значение на вершине равно ";

cout << pop()<< endl;

}

} while(value != -1);

return 0;

}

void push(int i)

{

p1++;

if(p1 == (tos+SIZE)) {

cout << "Переполнение стека.\n";

exit(1);

}

\*p1 = i;

}

int pop(void)

{

if(p1 == tos) {

cout << "Стек пуст.\n";

exit(1);

}

p1--;

return \*(p1+1);

}

Стек хранится в массиве **stack**. Сначала указатели **p1** и **tos** устанавливаются на первый элемент массива **stack**. В дальнейшем **p1** ссылается на верхний элемент стека, a **tos** продолжает хранить адрес основания стека. После инициализации стека используются функции push() и pop(). Они выполняют запись в стек и считывание из него, проверяя каждый раз соблюдение границы стека. В функции push() проверяется, что указатель **p1** не превышает верхней границы стека **tos+SIZE**. Это предотвращает переполнение стека. В функции pop() проверяется, что указатель **p1** не выходит за нижнюю границу стека.

В операторе return функции pop() скобки необходимы потому, что без них оператор

return \*p1+1;

вернул бы значение, расположенное по адресу p1, увеличенное на 1, а не значение по адресу p1+1.

# 5 Указатели и массивы

Понятия указателей и массивов тесно связаны. Рассмотрим следующий фрагмент программы:

char str[80], \*p1;

p1 = str;

Здесь **p1** указывает на первый элемент массива **str**. Обратиться к пятому элементу массива **str** можно с помощью любого из двух выражений:

str[4]

\* (p1+4)

Массив начинается с нуля. Поэтому для пятого элемента массива **str** нужно использовать индекс 4. Можно также увеличить **p1** на 4, тогда он будет указывать на пятый элемент. (Напомним, что имя массива без индекса возвращает адрес первого элемента массива.)

В языке С существуют два метода обращения к элементу массива: адресная арифметика и индексация массива. Стандартная запись массивов с индексами наглядна и удобна в использовании, однако с помощью адресной арифметики иногда удается сократить время доступа к элементам массива. Поэтому адресная арифметика часто используется в программах, где существенную роль играет быстродействие.

В следующем фрагменте программы приведены две версии функции **putstr()**, выводящей строку на экран. В первой версии используется индексация массива, а во второй — адресная арифметика:

/\* Индексация указателя s как массива. \*/

void putstr(char \*s)

{

register int t;

for(t=0; s[t]; ++t) putchar(s[t]);

}

/\* Использование адресной арифметики. \*/

void putstr(char \*s)

{

while(\*s) putchar(\*s++);

}

Большинство профессиональных программистов сочтут вторую версию более наглядной и удобной. Для большинства компиляторов она также более быстродействующая. Поэтому в процедурах такого типа приемы адресной арифметики используются довольно часто.

## 5.1 Массивы указателей

Как и объекты любых других типов, указатели могут быть собраны в массив. В следующем операторе объявлен массив из 10 указателей на объекты типа int:

int \*x[10];

Для присвоения, например, адреса переменной var третьему элементу массива указателей, необходимо написать:

x[2] = &var;

В результате этой операции, следующее выражение принимает то же значение, что и var:

\*x[2]

Для передачи массива указателей в функцию используется тот же метод, что и для любого другого массива: имя массива без индекса записывается как формальный параметр функции. Например, следующая функция может принять массив x в качестве аргумента:

void display\_array(int \*q[])

{

int t;

for(t=0; t<10; t++)

cout << ”\t” << \*q[t];

}

Необходимо помнить, что q — это не указатель на целые, а указатель на массив указателей на целые. Поэтому параметр q нужно объявить как массив указателей на целые. Нельзя объявить q просто как указатель на целые, потому что он представляет собой указатель на указатель.

Массивы указателей часто используются при работе со строками. Например, можно написать функцию, выводящую нужную строку с сообщением об ошибке по индексу **num**:

void syntax\_error(int num)

{

static char \*err[] = {

"Нельзя открыть файл\n",

"Ошибка при чтении\n",

"Ошибка при записи\n",

"Некачественный носитель\n"

};

Cout << err[num];

}

Массив **err** содержит указатели на строки с сообщениями об ошибках. Здесь строковые константы в выражении инициализации создают указатели на строки. Аргументом функции **printf()** служит один из указателей массива **err**, который в соответствии с индексом **num** указывает на нужную строку с сообщением об ошибке. Например, если в функцию **syntax\_error()** передается **num** со значением 2, то выводится сообщение “***Ошибка при записи”.***

Отметим, что аргумент командной строки **argv** также является массивом указателей на строковые константы.

# 6 Многоуровневая адресация

Иногда указатель может ссылаться на указатель, который ссылается на число. Это называется многоуровневой адресацией. Иногда применение таких указателей существенно усложняет программу, делает ее плохо читаемой и подверженной ошибкам. Рисунок 3 иллюстрирует концепцию многоуровневой адресации. На рисунке видно, что значением "нормального" указателя является адрес объекта, содержащего нужное значение. В случае двухуровневой адресации первый указатель содержит адрес второго указателя, который содержит адрес объекта с нужным значением.

Многоуровневая адресация может иметь сколько угодно уровней, однако уровни глубже второго, т.е. указатели более глубокие, чем "указатели на указатели" применяются крайне редко. Дело в том, что при использовании таких указателей часто встречаются концептуальные ошибки из-за того, что смысл таких указателей представить трудно.

**ПРИМЕЧАНИЕ.** Не следует путать многоуровневую адресацию с многоуровневыми структурами данных, использующими указатели, такими, например, как связные списки. Это фундаментально различные концепции.

Переменная, являющаяся указателем на указатель, должна быть соответствующим образом объявлена. Это делается с помощью двух звездочек перед именем переменной. Например, в следующем операторе **newbalance** объявлена как указатель на указатель на переменную типа **float**:

float \*\*newbalance;

Следует хорошо понимать, что **newbalance** — это не указатель на число типа **float**, а указатель на указатель на число типа **float**.

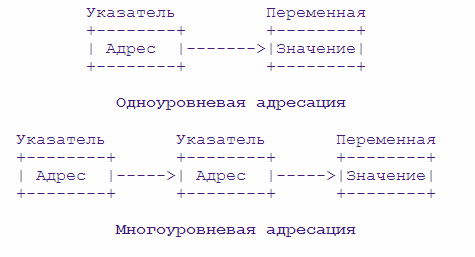


Рисунок 3. Одноуровневая и многоуровневая адресация

При двухуровневой адресации для доступа к значению объекта нужно поставить перед идентификатором две звездочки:

int main(void)

{

int x, \*p, \*\*q;

x = 10;

p = &x;

q = &p;

cout << \*\*q << endl; /\* печать значения x \*/

return 0;

}

Здесь **p** объявлена как указатель на целое, a **q** — как указатель на указатель на целое. Функция **printf()** выводит на экран число ***10***.

# 7 Инициализация указателей

После объявления нестатического локального указателя до первого присвоения он содержит неопределенное значение. (Глобальные и статические локальные указатели при объявлении неявно инициализируются нулем.) Если попытаться использовать указатель перед присвоением ему нужного значения, то скорее всего он мгновенно разрушит программу или всю операционную систему. Это очень досадная ошибка.

При работе с указателями большинство программистов придерживаются следующего важного соглашения: ***указатель, не ссылающийся в текущий момент времени должным образом на конкретный объект, должен содержать нулевое значение.*** Нуль используется потому, что С гарантирует отсутствие чего-либо по нулевому адресу. Следовательно, если указатель равен нулю, то это значит, во-первых, что он ни на что не ссылается, а во-вторых — что его сейчас нельзя использовать.

Указателю можно задать нулевое значение, присвоив ему 0. Например, следующий оператор инициализирует **р** нулем:

char \*p = 0;

Дополнительно к этому во многих заголовочных файлах языка С, например, в **<stdio.h>** определен макрос ***NULL***, являющийся нулевой указательной константой. Поэтому в программах на С часто можно увидеть следующее присваивание:

p = NULL;

Однако равенство указателя нулю не делает его абсолютно "безопасным". Использование нуля в качестве признака неподготовленности указателя — это только соглашение программистов, но не правило языка С. В следующем примере компиляция пройдет без ошибки, а результат, тем не менее, будет неправильным:

int \*p = 0;

\*p = 10; /\* ошибка! \*/

В этом случае присваивание посредством **p** будет присваиванием по нулевому адресу, что обычно вызывает разрушение программы.

Во многих процедурах для повышения эффективности программы можно использовать то, что нулевой указатель заведомо считается неподготовленным для использования. Например, можно использовать нулевой указатель как признак конца массива указателей (по аналогии с нулевым терминатором строки). Процедура, использующая массив указателей, таким образом узнает о конце массива. Такой подход иллюстрируется в таком примере. Просматривая список имен, функция **search()** определяет, есть ли в этом списке заданное имя.

#include <iostream>

#include <windows.h>

#include <clocale>

#include <string>

using namespace std;

int search(char \*p[], char \*name);

char \*names[] = { "Сергей", "Юрий", "Ольга", "Игорь", NULL}; /\* Нулевая константа заканчивает список \*/

int main(void)

{

setlocale(LC\_ALL, "Russian");

SetConsoleOutputCP(1251);

SetConsoleCP(1251);

if(search(names, "Ольга") != -1)

cout << "Ольга есть в списке.\n";

if(search(names, "Павел") == -1)

cout << "Павел в списке не найден.\n";

return 0;

}

/\* Просмотр имен. \*/

int search(char \*p[], char \*name)

{

register int t;

for(t=0; p[t]; ++t)

if(!strcmp(p[t], name)) return t;

return -1; /\* имя не найдено \*/

}

В функцию **search()** передаются два параметра. Первый из них, **p** — массив указателей на строки, представляющие собой имена из списка. Второй параметр **name** является указателем на строку с заданным именем. Функция **search()** просматривает массив указателей, пока не найдет строку, совпадающую со строкой, на которую указывает **name**. Итерации цикла **for** повторяются до тех пор, пока не произойдет совпадение имен, или не встретится нулевой указатель. Конец массива отмечен нулевым указателем, поэтому при достижении конца массива управляющее условие цикла примет значение ***ЛОЖЬ***. Иными словами, **p[t]** имеет значение ***ЛОЖЬ***, когда **p[t]** является нулевым указателем. В рассмотренном примере именно это и происходит, когда идет поиск имени "Павел", которого в списке нет.

В программах на С указатель типа **char \*** часто инициализируют строковой константой (как в предыдущем примере). Рассмотрим следующий пример:

char \*p = "тестовая строка";

Переменная **р** является указателем, а не массивом. Поэтому возникает логичный вопрос: где хранится строковая константа "тестовая строка"? Так как **p** не является массивом, она не может храниться в **p**, тем не менее, она где-то записана. Чтобы ответить на этот вопрос, нужно знать, что происходит, когда компилятор встречает строковую константу. Компилятор создает так называемую таблицу строк, в ней он сохраняет строковые константы, которые встречаются ему по ходу чтения текста программы. Следовательно, когда встречается объявление с инициализацией, компилятор сохраняет строку "тестовая строка" в таблице строк, а в указатель **p** записывает ее адрес. Дальше в программе указатель **p** может быть использован как любая другая строка. Это иллюстрируется следующим примером:

#include <iostream>

#include <windows.h>

#include <clocale>

#include <string>

using namespace std;

char \*p = "тестовая строка";

int main(void)

{

setlocale(LC\_ALL, "Russian");

SetConsoleOutputCP(1251);

SetConsoleCP(1251);

register int t;

/\* печать строки слева направо и справа налево \*/

cout << p << endl;

for(t=strlen(p)-1; t>-1; t--)

cout << p[t];

cout << endl << endl;

return 0;

}

ЗАДАНИЯ ДЛЯ ЛАБОРАТОРНЫХ РАБОТ

**Задание 1.** Создайте переменную типа **a)** и считайте с клавиатуры ее значение. Создайте указатель, который будет ссылаться на эту переменную. Выведите на экран адрес указателя и значение, которое хранится по этому адресу.

**Задание 2.** Пусть дано значение **x** типа **б)** и указатель **p** на переменную **х**. Получите значение переменной **n** в следующих примерах:

1) n = \*(p++);

2) n= (\*p)++;

3) n = ++(\*p);

4) n = \*(++p).

Сравните результаты и сделайте выводы.

**Задание 3.** Создайте один указатель типа **a)** и другой - типа **б)**. Создайте дополнительную переменную типа **а)** и считайте с клавиатуры ее значение. Присвойте ее адрес указателю соответствующего типа.

1. Выведите на экран значения, которые хранятся в обоих указателях, а также их адреса в формате, используемом компилятором.
2. \* Присвойте указателю типа **б)** значение второго указателя с применением оператора приведения типов и повторите действие 1).

**\*\*Задание 4.** Написать программу, создающую массив из 10 случайных целых чисел из отрезка [-50;50]. Вывести на экран весь массив и на отдельной строке — значение минимального элемента массива. Для обхода массива использовать указатели (запрещено обращаться к элементам массива по индексам).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **1.4** | а) double  б)char | **2.4** | а) double  б) int | **3.4** | а) double  б) int |

**Внимание!!! Варианты заданий представлены ниже ↓**

# ВАРИАНТЫ ЗАДАНИЙ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Вариант** | **Задание** | **Вариант** | **Задание** | **Вариант** | **Задание** |
| **1.1** | а) int  б)char | **2.1** | а) int  б) double | **3.1** | а) int  б) float |
| **1.2** | а) float  б)char | **2.2** | а) float  б) int | **3.2** | а) float  б) int |
| **1.3** | а) char  б) int | **2.3** | а) char  б) float | **3.3** | а) char  б) float |
| **1.4** | а) double  б)char | **2.4** | а) double  б) int | **3.4** | а) double  б) int |
| **1.5** | а) int  б) double | **2.5** | а) int  б)char | **3.5** | а) int  б) double |
| **1.6** | а) float  б) int | **2.6** | а) float  б)char | **3.6** | а) float  б) int |
| **1.7** | а) char  б) double | **2.7** | а) char  б) int | **3.7** | а) char  б) double |
| **1.8** | а) double  б) int | **2.8** | а) double  б)char | **3.8** | а) double  б)char |
| **1.9** | а) int  б) float | **2.9** | а) int  б) float | **3.9** | а) int  б)char |
| **1.10** | а) float  б) int | **2.10** | а) float  б) int | **3.10** | а) float  б)char |
| **1.11** | а) char  б) float | **2.11** | а) char  б) double | **3.11** | а) char  б) int |