[13. Указатели.](#_Toc531685261)

[13.1. Определение переменных-указателей. Разыменование указателей.](#_Toc531685262)

[13.1.1. Объявление переменных-указателей.](#_Toc531685263)

[13.1.2. Простые операции с указателями.](#_Toc531685264)

[13.1.3. Инициализация указателей.](#_Toc531685265)

[13.1.4. Неправильное использование операции определения адреса.](#_Toc531685266)

[13.1.5. Указатели на массивы.](#_Toc531685267)

[13.1.5.1. Указатели и многомерные массивы.](#_Toc531685268)

[13.1.6. Указатели на указатели.](#_Toc531685269)

[13.1.7. Указатели на строки.](#_Toc531685270)

[13.1.8. Арифметические операции с указателями.](#_Toc531685271)

[13.1.9. Арифметические операции с указателями и массивы.](#_Toc531685272)

[13.1.10. Операции с указателями.](#_Toc531685273)

[13.1.11. Применение к указателям оператора sizeof.](#_Toc531685274)

[13.1.12. Сложности при использовании операций ++ и --.](#_Toc531685275)

[13.1.13. Сравнение указателей.](#_Toc531685276)

[13.1.14. Переносимость указателей.](#_Toc531685277)

[13.1.15. Использование функции sizeof() с указателями в среде DOS.](#_Toc531685278)

[13.2. Указатели на функции.](#_Toc531685279)

[13.3. Динамическая память.](#_Toc531685280)

[13.3.1. Использование указателей типа void.](#_Toc531685281)

[13.4. Указатели и массивы.](#_Toc531685282)

[13.4.1. Функции, массивы и указатели.](#_Toc531685283)

[13.4.2. Использование указателей при работе с массивами.](#_Toc531685284)

[13.4.3. Строки (массивы типа char).](#_Toc531685285)

[13.4.4. Массивы указателей.](#_Toc531685286)

[13.4.5. Дополнительная информация об указателях на указатели.](#_Toc531685287)

[13.4.6. Массивы указателей на строки.](#_Toc531685288)

[13.5. Ссылочный тип в C++ (reference type).](#_Toc531685289)

[13.5.1. Адрес в качестве возвращаемого значения функции.](#_Toc531685290)

[13.5.2. Передача параметров по ссылке и по значению.](#_Toc531685291)

[13.5.3. Использование встроенного отладчика.](#_Toc531685292)

[13.5.4. Использование ссылочного типа.](#_Toc531685293)

[13.6. Использование указателей и ссылок с ключевым словом const](#_Toc531685294)

[13.7. Интеллектуальные указатели (современный C++)](#_Toc531685295)

[13.7.1. Интеллектуальные указатели стандартной библиотеки C++](#_Toc531685296)

1. Указатели.

## Определение переменных-указателей. Разыменование указателей.

Иной, зачастую более удобный и эффективный способ доступа к переменной, — использовать другую переменную, содержащую адрес той, которая вам необходима. Предположим, к примеру, что имеются целая переменная imemorycell\_contents ("содержимое ячейки памяти") и переменная pimemorycell\_address ("адрес ячейки памяти"), в которой может храниться адрес первой переменной (имена произвольные, но весьма символические). Как вы уже видели, в С перед именем переменной можно использовать операцию определения адреса (&); при этом возвращается не значение переменной, а ее адрес. Следовательно, вас не удивит синтаксис выражения для присвоения адреса некоторой переменной другой, в которой этот адрес хранится:

pimemorycell\_address = &imemorycell\_contents;

10_01

Рис. 13.1. Пример переменной-указателя

10_02

Рис. 13.2. Переменная pimemorycell\_address указывает на imemorycell\_contents

Переменная, в которой хранится адрес, например pimemorycell\_address, называется переменной-указателем или просто указателем (pointer). Это отношение между переменными показано на рис. 13.1.

Переменная imemorycell\_contents была помещена в память с адресом 7751. После выполнения приведенного выше оператора адрес переменной imemorycell\_contents присваивается указателю pimemorycell\_address. Это описывается следующим образом: pimemorycell\_address указывает на imemorycell\_contents, что иллюстрируется на рис. 13.2. Стрелка направлена из ячейки, содержащей адрес, к ячейке, адрес которой хранится.

Для обращения к содержимому ячейки, адрес которой хранится в pimemorycell\_address, достаточно перед указателем поставить звездочку: \*pimemorycell\_address. Это действие называется разыменованием (dereference) указателя pimemorycell\_address. Если, к примеру, выполнить два следующих оператора, то значение переменной imemorycell\_contents будет равно 20. (см. рис. 13.3).

10_03

Рис. 13.3. Использование указателя в операторе присваивания

pimemorycell\_address = &imemorycell\_contents;

\*pimemorycell\_address = 20;

Звездочку (\*) можно рассматривать как директиву поиска адресуемой ячейки памяти согласно стрелке на рис. 13.3. Заметьте, что если указатель pimemorycell\_address содержит адрес переменной imemorycell\_contents, то следующие два оператора будут эквивалентны; они оба записывают значение 20 в imemorycell\_contents:

imemorycell\_contents = 20;

\*pimemorycell\_address = 20;

Любой объект программы, будь то переменная базового или же производного типа, занимает в памяти определенную область. Местоположение объекта в памяти определяется его адресом. Как уже говорилось, при объявлении переменной для нее резервируется место в памяти, размер которого зависит от типа данной переменной, а для доступа к содержимому объекта служит его имя (идентификатор). Для того чтобы узнать адрес конкретной переменной, служит унарная операция взятия адреса. При этом перед именем переменной ставится знак амперсанда (&). Следующий ниже пример программы выведет на печать сначала значение переменной Var1 и Var2, а затем ее адрес:

#include <iostream>

using namespace std;

#define endl '\n'

int main()

{

setlocale(LC\_ALL, "Rus");

unsigned int Var1 =40000;

unsigned int Var2 =300;

cout << "Значение1: " <<Var1 << endl;

cout << "Адрес: " << &Var1 << endl;

cout << "Значение2: " <<Var2 << endl;

cout << "Адрес: " << &Var2 << endl;

getchar(); getchar();

return 0;

}

В результате будет выведено:

Значение1: 40000

Адрес: 0x0068fe00

Значение2: 300;

Адрес: 0x0068£dfc



/\* Simple Program for Print size of different types Using Pointer in C\*/

/\* Print Pointer Address Program,C Pointer Examples \*/

#include <stdio.h>

int main() {

int a = 10;

int \*pa = &a;

char b = 'x';

char \*pb = &b;

float c = 10.01;

float \*pc = &c;

double d = 10.01;

double \*pd = &d;

long e = 10.01;

long \*pe = &e;

printf("Pointer Example Program : Print Size of Different types Using sizeof\n");

printf("\n[sizeof(a) ]: = %d", sizeof(a));

printf("\n[sizeof(\*pa) ]: = %d", sizeof(\*pa));

printf("\n[sizeof(b) ]: = %d", sizeof(b));

printf("\n[sizeof(\*pb) ]: = %d", sizeof(\*pb));

printf("\n[sizeof(c) ]: = %d", sizeof(c));

printf("\n[sizeof(\*pc) ]: = %d", sizeof(\*pc));

printf("\n[sizeof(d) ]: = %d", sizeof(d));

printf("\n[sizeof(\*pd) ]: = %d", sizeof(\*pd));

printf("\n[sizeof(e) ]: = %d", sizeof(e));

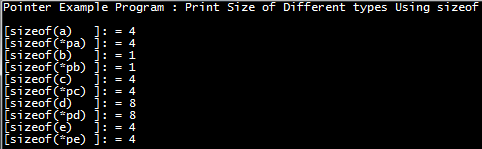
printf("\n[sizeof(\*pe) ]: = %d", sizeof(\*pe));

getchar();

getchar();

return 0;

}



#include <iostream>

struct Empty {};

struct Base { int a; };

struct Derived : Base { int b; };

struct Bit { unsigned bit: 1; };

int main()

{

Empty e;

Derived d;

Base& b = d;

Bit bit;//[[maybe\_unused]]

int a[10];

std::cout << "size of empty class: " << sizeof e << '\n'

<< "size of pointer: " << sizeof &e << '\n'

// << "size of function: " << sizeof(void()) << '\n' // error

// << "size of incomplete type: " << sizeof(int[]) << '\n' // error

// << "size of bit field: " << sizeof bit.bit << '\n' // error

<< "size of array of 10 int: " << sizeof(int[10]) << '\n'

<< "size of array of 10 int (2): " << sizeof a << '\n'

<< "length of array of 10 int: " << ((sizeof a) / (sizeof \*a)) << '\n'

<< "length of array of 10 int (2): " << ((sizeof a) / (sizeof a[0])) << '\n'

<< "size of the Derived: " << sizeof d << '\n'

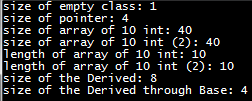
<< "size of the Derived through Base: " << sizeof b << '\n';

getchar();

getchar();

return 0;

}



Как видно из примера, адреса локальных переменных, размещаемых в стеке, следуют в обратном порядке (стек растет в направлении младших адресов). Результат может отличаться даже при повторном запуске программы, так как невозможно предугадать, по какому адресу начнут размещаться переменные. Важно другое: разница в адресах первой и второй переменной всегда будет одинакова и при четырехбайтном представлении типа int составит 4 байта (см. рис. 0501).



**Рис. 0501.** Переменные в памяти

Мощным средством разработчика программного обеспечения на C++ является возможность осуществления непосредственного доступа к памяти. Для этой цели предусматривается специальный тип переменных - указатели.

**Указатель (pointer) представляет собой переменную, значение которой является адресом ячейки памяти. Указатель может ссылаться на переменную (базового или производного типа) или функцию. Наибольшая эффективность применения указателей в разработке приложении достигается при использовании их с массивами и символьными строками.**

Объявление указателя имеет следующий синтаксис:

тип\_объекта\* идентификатор;

**Здесь тип\_объекта определяет тип данных, на которые ссылается указатель с именем идентификатор. Символ 'звездочка' (\*) сообщает компилятору, что объявленная переменная является указателем и не зависимо от того, сколько памяти требуется отвести под сам объект, для указателя резервируется два или четыре байта в зависимости от используемой модели памяти.**



**Рис. 0502.** Смысл указателя

**Поскольку указатель представляет отбой ссылку на некоторую область памяти, ему может быть присвоен только адрес некоторой переменной (или функции), а не само ее значение. В случае некорректного присвоения компилятор выдаст соответствующее сообщение об ошибке.**

Рассмотрим пример объявления и инициализации указателя

char Symbol = 'Y';

char \*pSymbol = &Symbol;

long Capital = 304L;

long\* pLong;

pLong = &Capital;

В приведенном фрагменте объявляется символьная переменная Symbol инициализируется значением 'Y', затем определяется указатель на символьный тип данных pSymbol, значение которого назначается равным адресу переменной Symbol. Вслед за этим объявляется переменная Capital типа long и указатель на этот же тип pLong, после чего производится инициализация указателя адресом переменной Capital.

**Разыменование указателей**

Как уже отмечалось, **указатели помогают осуществлять непосредственный доступ к памяти. Для того чтобы получить (прочитать) значение, записанное в некоторой области, на которую ссылается указатель, используют операцию косвенного обращения, или разыменования (\*). При этом используется имя указателя со звездочкой перед ним:**

long double Num = 10;

long double Flag;

long double \*ptr = &Num;

Flag = \*ptr;

cout << Flag;

В приведенном фрагменте объявляются две переменные двойной точности Num и Flag и указатель (Ptr) на тип long double, проинициализированный адресом переменно Num. После этого посредством косвенного доступа к переменной Flag присваивается значение, хранящееся по адресу указанному в ptr. то есть фактически значение переменной Num, что и подтверждает вывод на печать.

**На практике довольно широко применяется так называемый пустой указатель (типа void), который может указывать на объект любого типа. Для получения доступа к объекту, на который ссылается указатель void, его необходимо предварительно привести к тому же типу, что и тип самого объекта.**

Рассмотрим пример, иллюстрирующий использование пустого указателя.

#include <iostream>

using namespace std;

int main()

{

setlocale(LC\_ALL, "Rus");

char Let = 'T';

int nNum = 9;

void \*ptr;

ptr = &Let;

\*(char\*)ptr = 'L';

ptr = &nNum;

\*(int\*)ptr = 43;

cout << Let << '\n';

cout << nNum;

getchar(); getchar();

return 0;

}



Сначала создаются два разнотипных объекта Let и nNum и пустой указатель ptr, который инициализируется адресом символьной переменной. Далее ptr разыменовывается, приводится к типу char\* и посредством косвенной адресации модифицируется значение символа Let. Аналогичным образом указатель инициализируется значением адреса переменной nNum, приводится к целочисленному типу, после чего становится возможным изменение содержимого переменной nNum.

### Объявление переменных-указателей.

В языке С, также как и во многих других языках, необходимо описывать каждую переменную. Для описания переменной-указателя pimemorycell\_address, которая содержит адрес некоторой переменной типа int, нужно написать:

int \*pimemorycell\_address;

На самом деле это объявление состоит из двух частей.

Типа переменной pimemorycell\_address: int \*и идентификатора переменной pimemorycell\_address.

Звездочка, расположенная следом за типом int, означает "указывает на". То есть, этот тип данных представляет собой некоторую переменную-указатель, в которой хранится адрес переменной int:

int \*

**Этот момент следует запомнить. В языке С, в отличие от многих других языков, указатель содержит адрес конкретного типа данных.**

Взгляните на пример:

char \*pcaddress; int \*piaddress;

**Тип указателя pcaddress совершенно отличен от типа указателя piaddress. Если описан некоторый указатель одного типа, который затем используется с данными другого типа, то во время выполнения программы возникнут ошибки, а при ее компиляции предупреждения.** Нельзя назвать хорошим стилем программирования такой, когда указатель описывается одним образом, а затем используется совершенно по-другому. Для примера рассмотрим следующий фрагмент программы:

int \*pi;

float real\_value = 98.26;

pi = &real\_value;

Здесь переменная pi имеет тип int \*, это означает, что в ней может содержаться адрес переменной типа int.

В третьем операторе сделана попытка присвоить переменной pi адрес (Arealvalue) переменной типа float.

### Простые операции с указателями.

В следующем фрагменте программы выполняется обмен содержимым переменных iresult\_a и iresult\_b; для этого используются операции определения адреса и разыменования:

int iresult\_a = 15, iresult\_b = 37, itemporary;

int \*piresult;

piresult = &iresult\_a;

itemporary = \*piresult;

\*piresult = iresult\_b;

iresult\_b = itemporary;

Первая строка программы — это стандартные описание и инициализация переменных: выделяется три ячейки памяти для целых чисел, каждой ячейке присваивается имя, и две переменные инициализируются (см. рис. 13.4). Предположим для примера, что ячейка с именем iresult\_a располагается по адресу 5328, ячейка с именем iresult\_b — по адресу 7916, а ячейка с именем itemporary — по адресу 2385.

**13_04**

Рис. 13.4.Создание и инициализация ячеек памяти

Во втором операторе программы переменная piresult описывается как указатель на тип int. Этот оператор выделяет ячейку памяти и присваивает ей имя (переменная располагается по адресу 1920). Напоминаем, что если вместе с типом данных (в нашем случае int) указана звездочка, то переменная содержит адрес ячейки этого типа. Поскольку переменная piresult не инициализирована, она не указывает ни на какую конкретную переменную типа int. Если попытаться в программе использовать piresult, то компилятор не выдаст никаких предупреждений, и ссылка будет на неинициализированное содержимое переменной. В четвертом операторе указателю piresult присваивается адрес переменной iresult\_a (см. рис. 13.5).

13_05

Рис. 13.5. Присваивание указателю piresult адреса переменной iresult\_a

Следующий оператор программы для доступа к содержимому ячейки, на которую указывает piresult, то есть iresult\_a, использует выражение \*piresult.

itemporary = \*piresult;

Следовательно, в переменную itemporary записывается целое число 15 (см. рис. 13.6). Если убрать звездочку, расположенную перед именем piresult, то при выполнении оператора присваивания в ячейку с именем itemporary будет ошибочно записано содержимое указателя piresult (адрес 5328); хотя предполагается, что в переменной itemporary должно храниться целое число, а не адрес. Эту ошибку обнаружить очень трудно, так как многие компиляторы в таких случаях не выдают никаких предупреждений и/или сообщений об ошибке. (Компилятор Visual C/C++ выдает предупреждение "different levels of indirection" — различные уровни косвенной адресации.)

13_06

Рис. 13.6. Использование piresult для присваивания значения переменной itemporary

Еще больше осложняет ситуацию то, что большинство указателей имеют спецификатор near (ближний); это означает, что они занимают 2 байта памяти, то есть столько же, сколько целочисленный тип данных в РС-совместимых компьютерах. В пятом операторе программы содержимое переменной iresult\_b копируется в ячейку, адрес которой хранится в piresult (см. рис. 13.7):

\*piresult = iresult\_b;

13_07

Рис. 13.7. Еще одно присваивание с использованием piresult

Последний оператор в программе просто копирует содержимое одной целочисленной переменной itemporary в другую такую же переменную iresult\_b (см. рис. 13.8). Убедимся в том, что вы поняли разницу между объектами, которые обозначает переменная-указатель с операцией разыменования (\*piresult) и без нее (piresult). В нашем примере первое выражение является некоторым указателем на ячейку, содержащую целое

число. Второе выражение относится к ячейке, в которой хранится адрес другой ячейки, содержащей целое число.

13_08

Рис. 13.8. Обычное присваивание целого значения

Следующая небольшая программа показывает, как обращаться с адресами в переменных-указателях. В отличие от предыдущего примера, в котором выполнялся обмен данными между переменными, в этом примере происходит обмен адресами, в которых располагаются данные:

char cswitch1 = 'S', cswitch2 = "T";

char \*pcswitch1, \*pcswitch2, \*pctemporary;

pcswitch1 = &cswitch1;

pcswitch2 = &cswitch2;

pctemporary = pcswitch1;

pcswitch1 = pcswitch2;

pcswitch2 = pctemporary;

printf ( "%c%c", \*pcswitch1, \*pcswitch2);

На рис. 13.9 показано состояние ячеек и их значений после выполнения первых четырех операторов программы. При выполнении пятого оператора содержимое pcswitch1 копируется в переменную pctemporary, при этом обе переменных, pcswitch1 и pctemporary, указывают на cswitch1 (см. рис. 13.10)

13_09

Рис.13.9. Начальное соотношение между программными переменными

13_10

Рис.13.10. Переменной pctemporary присваивается адрес ячейки переменной cswitch1

При выполнении следующего оператора содержимое указателя pcswitch2 копируется в pcswitch1, и оба указателя ссылаются на cswitch2 (см. рис. 13.11):

pcswitch1 = pcswitch2;

13_11

Рис. 13.11. Присваивание указателю pcswitch1 адреса, хранящегося в pcswitch2

Обратите внимание на то, что, если бы адрес переменной cswitch1 не был сохранен во временной переменной pctemporary, то не было бы указателей для доступа к cswitch1. Предпоследний оператор копирует адрес, хранящийся в pctemporary, в указатель pcswitch2 (см. рис. 13.12). Поскольку значение \*pcswitch1 равно ‘T’, а значение \*pcswitch2 равно ‘S’, то при выполнении оператора printf получится:

13_12

Рис.13.12. Указателю pcswitch2 присваивается адрес, хранящийся в pctemporary

Обратите внимание на то, что реальные значения, хранящиеся в переменных cswitch1 и cswitch2, не изменились с момента инициализации. Однако, поскольку поменялось содержимое соответствующих указателей \*pcswitch1 и \*pcswitch2, то кажется, что эти реальные значения поменялись местами. Этот важный момент стоит запомнить. Вне зависимости от размера объекта (данных), может быть проще перемещать указатель на этот объект, чем копировать все содержимое объекта.

### Инициализация указателей.

Как и многие другие переменные С, указатели можно инициализировать при их описании. Например: следующие два оператора выделяют память для двух переменных iresult и piresult.

int iresult;

int \*piresult = &iresult;

Переменная iresult — это обычная целая переменная, a piresult — указатель на целое число. Кроме этого, указатель piresult при инициализации получает значение адреса переменной iresult. Будьте осторожны! Эта запись может в чем-то запутать: инициализируется не значение \*piresult (которое должно быть целым числом), а указатель piresult (который является адресом целой переменной). Второй из приведенных выше операторов можно записать при помощи двух следующих эквивалентных операторов:

int \*piresult;

piresult = &iresult;

Далее в примерах используется функция **StopWait()** приостановки работы программы

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

StopWait.cpp

Приостанавка работы программы

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//Stoping a program

#include "stdafx.h"

#include "E:\LECTURE\AlgorithmProgramming 02\Universal\_HederFile.h"

void StopWait(void)

{

printf ("\n\nPress any key to finish\n");

\_getch();

}

и единый заголовочный файл **Universal\_HederFile.h**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Universal\_HederFile.h

Универсальный Н файл

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Universal H file for examples

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

#include <stdarg.h>

#include <string.h>

using namespace std;

Следующий пример показывает, как объявить и инициализировать указатель на строку:

/\*10PSZ.C

Программа на С, инициализирующая указатель на строку и печатающая

строку-палиндром в обратном и в прямом порядке\*/

#include "stdafx.h"

#include "E:\LECTURE\AlgorithmProgramming 02\Universal\_HederFile.h"

void StopWait(void);

main()

{

char \*pszpalindrome=" MADAM I'M ADAM"; /\* МАДАМ Я АДАМ" \*/

int i;

for(i=strlen(pszpalindrome)-1; i >= 0; i--)

printf("%c",pszpalindrome[i]);

printf("%s\n",pszpalindrome);

StopWait(); /\* Wait a little \*/

}

Фактически, компилятор С запоминает адрес первого символа строки "MADAM I'M ADAM" в переменной pszpalindrome. При работе программы можно использовать переменную pszpalindrome так же, как и любые другие строки. Это объясняется тем, что компилятор С создает таблицу строк, в которой он хранит строковые константы программы.

Функция strlen(), имеющая прототип в файле string.h, определяет длину строки. Она получает в качестве параметра указатель на строку, заканчивающуюся null-символом, и подсчитывает все символы, за исключением этого null-символа. Переменная-индекс i получает начальное значение, на единицу меньшее числа, возвращаемого функцией strlen(), поскольку в цикле for строка psz обрабатывается как символьный массив. В палиндроме 14 букв. Если строка psz рассматривается как массив символов, то индексы его элементов имеют значения от 0 до 13. Данный пример помогает прояснить не вполне очевидную связь между указателями на символьные строки и массивами символов. Однако, если вы вспомните, что имя массива — это на самом деле адрес его первого элемента, вас не удивит то, что компилятор не выдает ошибки.

### Неправильное использование операции определения адреса.

Операцию определения адреса можно использовать не во всех выражениях С. В следующих примерах показаны ситуации, в которых эту операцию (&) применять нельзя:

/\*нельзя использовать вместе с константами\*/

pivariable = &48;

/\*нельзя использовать в выражениях с арифметическими операциями\*/

int iresult = 5;

pivariable = &(iresult + 15);

/\*нельзя использовать вместе с регистровыми переменными\*/

register register1;

pivariable = &register1;

В первом операторе ошибочно делается попытка получить адрес постоянного значения. Это не имеет смысла, так как с числом 48 не связана никакая ячейка памяти.

Во втором операторе присваивания делается попытка получить адрес выражения iresult + 15. Поскольку вычисление выражения представляет собой манипулирование со стеком, с выражением не связан никакой адрес.

В третьем примере переменная register1 описана как регистровая, а не как ячейка оперативной памяти. В этом случае невозможно получить и запомнить ее адрес. Microsoft Visual C/C++ выделяет такой переменной не регистровую, а оперативную память.

### Указатели на массивы.

Как уже упоминалось, указатели и массивы связаны между собой. Известно, что имя массива представляет собой константу, значение которой равно адресу первого элемента массива. Поэтому имя массива нельзя менять при помощи операции присваивания или любой другой операции. В следующем примере при объявлении данных имя массива ftemperatures представляет собой константу, которая равна адресу первого элемента массива из 20 чисел с плавающей точкой:

#define IMAXREADINGS 20

float ftemperatures[IMAXREADINGS];

float \*pftemp;

В следующем операторе адрес первого элемента массива присваивается указателю pftemp:

pftemp = ftemperatures;

Эквивалентный оператор выглядит так:

pftemp = &ftemperatures [0];

Однако, если pftemp содержит адрес числа с плавающей запятой, то следующие операторы будут неправильными:

ftemperatures = pftemp;

&ftemperatures[0] = pftemp;

В этих операторах делается попытка присвоить некоторое значение константе ftemperatures или ее эквиваленту ftemperatures[0], что имеет смысла не больше, чем выражение:

10 = pftemp;

Указатели и многомерные массивы.

Как создать указатели для многомерных массивов? Чтобы найти ответ на этот вопрос, рассмотрим несколько примеров.

Предположим, что у нас есть описания

/\*указатель на целый тип \*/

Тогда на что

pri = zippo;

указывает? На первый столбец первой строки:

zippo == &zippo[0][0]

А на что указывает pri + 1? На zippo[0][1], т. е. на 1-ю строку 2-го столбца? Или на zippo [1][0], элемент, находящийся во второй строке первого столбца? Чтобы ответить на поставленный вопрос, нужно знать, как располагается в памяти двумерный массив. Он размещается, подобно одномерным массивам, занимая последовательные ячейки памяти. Порядок элементов определяется тем, что самый правый индекс массива изменяется первым, т. е. элементы массива располагаются следующим образом:

zippo[0][0] zippo[0][1] zippo[1][0] zippo[1][1] zippo[2][0]

Сначала запоминается первая строка, за ней вторая, затем третья и т. д. Таким образом, в нашем примере:

|  |  |
| --- | --- |
| pri == &zippo[0][0] /\* 1-я строка, 1 столбец \*/  pri + 1 == &zippo[0][1] /\* 1-я строка, 2 столбец \*/  pri + 2 == &zippo[1][0] /\* 2-я строка, 1 столбец \*/  pri + 3 == &zippo[1][1] /\* 2-я строка, 2 столбец \*/ | 00 01  10 11  20 21  30 31  40 41 |

Получилось? Хорошо, а на что указывает pri + 5? Правильно, на

zippo[2][1].

Мы описали двумерный массив как массив массивов. Если zippo является именем нашего двумерного массива, то каковы имена четырех строк, каждая из которых является массивом из двух элементов? Имя первой строки zippo [0], имя четвертой строки zippo [3]; вы можете заполнить пропущенные имена. Однако имя массива является также указателем на этот массив в том смысле, что оно ссылается на первый его элемент. Значит,

zippo[0] == &zippo[0][0]

zippo[1] == &zippo[1][0]

zippo[2] == &zippo[2][0]

zippo[3] == &zippo[3][0]

Это свойство является более, чем новшеством. Оно позволяет использовать функцию, предназначенную для одномерного массива, для работы с двумерным массивом! Вот доказательство *(хотя мы надеемся, что теперь вы бы поверили нам и так*) использования двумерного массива в нашей программе нахождения среднего значения:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **0** | **1** | **2** | **3** |
| **0** | 00/**0** | 01/**1** | 02/**2** | 03/**3** |
| **1** | 10/**4** | 11/**5** | 12/**6** | 13/**7** |
| **2** | 20/**8** | 21/**9** | 22/**10** | 23/**11** |

/\* одномерная функция, двумерный массив \*/

#include<stdio.h>

int mean(int array[],int n)

{

int index; long sum;

if (n > 0)

{

for (index = 0, sum = 0; index < n; index++)

sum += array [index];

return((sum/n) );

}

else

{

printf("There is no an array\n");

return(0);

}

}

main ()

{

static int junk [3] [4] = {

{2, 4, 6, 8}, // строка 0

{100, 200, 300, 400}, // строка 1

{10, 40, 60, 90} // строка 2

};

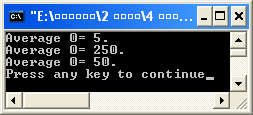
int row;

for (row = 0; row < 3; row++ )

printf(" Average %d = %d.\n", row, mean(junk[row], 4);

}

Результат работы программы:



### Указатели на указатели.

В языке С можно описать переменные-указатели, указывающие на другие указатели, которые в свою очередь ссылаются на данные, например на целое число. На рис. 13.13 показано это отношение: ppi — это указатель на другую переменную-указатель, содержимое которой указывает на число 10.

10_13

Рис. 13.13. Указатель на указатель, ссылающийся на целое число

Вы можете удивиться: зачем это нужно? Появление программных сред Windows 95 и Windows NT свидетельствует о создании многозадачных операционных систем, разработанных в целях максимально полного использования памяти. Для такой оптимизации операционная система должна иметь возможность перемещать объекты в памяти. Если прикладная программа непосредственно адресует ячейку физической памяти, где хранится объект, то при его перемещении в системе возникнет аварийная ситуация. Вместо того, чтобы непосредственно обращаться к объекту данных, приложение может обращаться к некоторому адресу памяти, который не изменяется при работе программы (назовем его для примера виртуальным адресом), и этот виртуальный адрес содержит текущий физический адрес объекта. Тогда при перемещении этого объекта операционной системе достаточно обновить текущий физический адрес, на который ссылается виртуальный адрес. Приложение же по-прежнему использует неизменный виртуальный адрес для ссылки на изменившийся текущий физический адрес.

В языке С для описания указателя на указатель достаточно добавить еще одну звездочку перед идентификатором:

int \*\*ppi;

В этом примере переменная ppi описана как указатель на указатель, который указывает на переменную типа int. Тип переменной ppi:

int \*\*

Каждая звездочка читается как "указывает на". Число указателей, необходимых для доступа к данным, или эквивалентное число звездочек, добавляемых к имени переменной для обращения к значению, на которое она ссылается, называется уровнем косвенности (level of indirection) переменной-указателя. Уровень косвенности указателя определяет число операций разыменования, которые нужно выполнить для доступа к описанным данным. На рис. 13.14 показано несколько переменных с разными уровнями косвенности.

13_14

Рис. 13.14. Использование различных уровней косвенности

Первые четыре строки кода на рис. 13.14 описывают четыре переменных: целую переменную ivalue; переменную pi, указывающую на целое число (первый уровень косвенности); переменную ppi, указывающую на указатель, ссылающийся на целое число (второй уровень косвенности); и переменную pppi, указывающую на то, что процесс косвенности может превышать два уровня. Пятая строка кода:

pi = &ivalue;

Это оператор присваивания, в котором используется операция определения адреса. Данное выражение присваивает адрес &ivalue переменной pi. Следовательно, pi содержит 1111. Обратите внимание на то, что между pi и ivalue имеется только одна стрелка. Это указывает на то, что значение ivalue или 10 можно получить, один раз используя по отношению к pi операцию разыменования. Следующий оператор и соответствующий рисунок иллюстрируют двойную косвенность:

ppi = &pi;

Поскольку переменная ppi имеет тип int\*\*, то для доступа к целому числу необходимо дважды выполнить операцию разыменования переменной. После выполнения указанного оператора, ppi содержит адрес переменной pi (но не содержимое pi); следовательно, ppi указывает на pi, которая в свою очередь указывает на ivalue. Обратите внимание на то, что между ppi и ivalue имеется уже две стрелки.

Последний оператор иллюстрирует три уровня косвенности:

pppi = &ppi;

Здесь адрес переменной ppi (но не содержимое ppi) присваивается переменной pppi. На рис. 13.14 видно, что для обращения к ivalue нужно пройти три стрелки.

Переменной pppi присвоен адрес указателя, который косвенно указывает на целое число. Однако \*\*\*pppi (ячейка, на которую указывает) может получать только целое значение, а не адрес, поскольку \*\*\*pppi является целым числом:

\*\*\*pppi = 10;

В языке С указатели можно инициализировать так же, как и любые другие переменные. Например, переменную pppi можно было бы описать и инициализировать при помощи одного следующего оператора:

int \*\*\*pppi = &ppi;

Указатели могут сами ссылаться на другие указатели. При этом в ячейках памяти, на которые ссылается указатель, содержится не значение, а адрес какого-либо объекта. Сам объект может также являться указателем и т.д. На рис. 5.3 представлен вариант размещения в памяти указателя на указатель, который в свою очередь ссылается на однобайтный тип данных (например, char или bооl).



**Рис. 0503.** Размещение указателя на указатель в памяти

Здесь по Адресу+1 хранится значение некоторой переменной, на которую указывает обычный указатель, расположенный по Адресу+3 (хранит значение Адрес+1), на который в свою очередь ссылается указатель, расположенный по Адресу+5 (содержит в качестве значения Адрес+3).

Синтаксис указателя на указатель выглядит следующим образом:

// Объявление указателя на указатель

int \*\*pPtrInt;

// ppSymbol — указатель на указатель,

// который сам является указателем

char \* \* \*ppSymbol;

При объявлении указатель на указатель может инициализироваться адресом объекта:

char\* pSymb = &myChar;

char\*\* p\_ptr = &pSymb;

char\*\*\* ppptr = &p\_ptr;

Число символов 'звездочка' (\*) при объявлении говорит о "порядке" указателя. Чтобы получить доступ к значению, такой указатель должен быть разыменован соответствующее количество раз (по числу символов ‘\*’).

В приведенном ниже примере создается указатель второго порядка pPDX, содержащий адрес указателя pDX, который в свою очередь ссылается на переменную двойной точности.

#include <iostream>

using namespace std;

int main()

{

setlocale(LC\_ALL, "Rus");

double dX = 903.456;

double \*pDX = &dX;

double \*\*pPDX = &pDX;

cout << \*\*pPDX;

getchar(); getchar();

return 0;

}



В результате будет выведено значение переменной dX.

### Указатели на строки.

Некоторая строковая константа, например "File not ready", на самом деле хранится в виде массива символов, последним элементом которого является null-символ (см. рис. 13.15). Поскольку указатель типа char может хранить адрес символа, такой указатель можно описать и инициализировать. Например:

char \*psz = "File not ready"

13_15

Рис. 13.15. Строка с null-символом в памяти компьютера

В этом операторе описан указатель psz типа char, и ему присвоено начальное значение — адрес первого символа строки (см. рис. 13.16). Кроме этого, выделена память для самой строки. Этот оператор можно было бы записать следующим образом:

char \*psz;

psz = "File not ready";

13_16

Пис.13.16. Инициализация указателя на строку

Опять-таки, нужно помнить о том, что адрес присвоен переменной psz, a не \*psz, ссылающейся на символ "F". Приведенный выше пример помогает прояснить этот момент, поскольку в нем используются отдельные операторы для описания и для инициализации указателя.

Следующий пример иллюстрирует принципиальное различие при работе с указателями на строки и указателями на массивы символов:

char \*psz = "File not ready";

char pszarray[] = "Drive not ready";

char pszarray = "Drive not ready"; //ERROR cannot convert

Главное отличие между двумя этими операторами заключается в том, что значение psz можно изменить (поскольку это — указатель-переменная), значение pszarray изменить нельзя (поскольку это — указатель-константа). Для достижения аналогичного результата следующий оператор присваивания использовать нельзя:

/\* НЕПРАВИЛЬНО! \*/

char pszarray[16];

pszarray = "Drive not ready";//ERROR cannot convert

Хотя эта запись и похожа на приведенный ранее правильный код, в операторе присвоения сделана попытка копирования адреса первой ячейки памяти, в которой хранится строка "Drive not ready", в pszarray. Поскольку pszarray является указателем-константой, а не указателем-переменной, возникает ошибка.

Следующий оператор ввода будет неправильным, так как указатель psz не инициализирован:

/\* НЕПРАВИЛЬНО! \*/

char \*psz; // psz не определен

cin >> psz;

Для решения проблемы достаточно зарезервировать память и инициализировать указатель psz:

/\* ПРАВИЛЬНО! \*/

char sztring[10];

char \*psz = sztring;

cin.get(psz,10);

Поскольку значение sztring является адресом первого элемента массива, то второй оператор в программе не только выделяет память для указателя, но и инициализирует его, присваивая ему адрес первого элемента массива sztring. После этого можно выполнять оператор cin.get, так как ему передается действительный адрес символьного массива.

### Арифметические операции с указателями.

Если вы знакомы с программированием на языке ассемблера, то вам уже привычно использовать действительные физические адреса для доступа к информации, хранящейся в таблицах. Те же, кто работал только с индексами массивов, могут поверить на слово, что фактически они пользовались тем же ассемблерным эквивалентом. Единственное отличие состоит в том, что в последнем случае вместо вас с адресами работал компилятор.

Вспомним, что одним из достоинств С является его близость к аппаратному обеспечению. В С можно манипулировать переменными-указателями. Во многих рассмотренных примерах было показано, как адрес одного указателя или просто адрес можно присвоить другому указателю того же типа. В языке С с адресом указателя можно выполнять только две арифметические операции: сложение и вычитание. Давайте рассмотрим два указателя различных типов и выполним с ними несколько простых арифметических операций:

// 10PTARTH.CPP

// Программа на C++, иллюстрирующая арифметические операции

// с указателями

#include "stdafx.h"

#include "E:\LECTURE\AlgorithmProgramming 02\Universal\_HederFile.h"

void StopWait(void);

main()

{

int an\_integer, \*pi;

float a\_real, \*pf;

pi=&an\_integer;

pf=&a\_real;

cout << "pi= " << pi << " pf= " << pf << endl;

pi++;

pf++;

cout << "pi= " << pi << " pf= " << pf << endl;

StopWait(); /\* Wait a little \*/

return (0);

}

Предположим, что целое число занимает 2 байта, а число с плавающей точкой — 4 байта. Допустим, что переменная an\_integer хранится в ячейке с адресом 2000, а переменная a\_real расположена в памяти по адресу 4000. После выполнения последних двух строк программы pi будет содержать адрес 2002, a pf— 4004. Но, спросите вы, разве операция инкремента увеличивает число не на 1? Да, это так для символьных переменных, но не всегда верно для указателей. В данном примере, поскольку переменная pi описана как указатель на целое число (которое в нашей системе занимает 2 байта), то при выполнении операции инкремента происходит проверка типа переменной, а затем выбирается соответствующее значение для самой операции. Для целых чисел это значение равно 2, а для чисел с плавающей точкой — 4 (в нашей системе). Этот принцип соблюдается для любого типа данных, на который ссылается указатель. Если бы указатель ссылался на структуру размером в 20 байт, то операции инкремента или декремента меняли бы текущий адрес, хранящийся в указателе, на 20.

Адрес в указателе можно модифицировать не только при помощи операций ++ и —, но и используя целочисленное сложение и вычитание. Например: при помощи следующего оператора можно сместиться на четыре числа типа float относительно того числа, на которое в настоящий момент ссылается указатель:

pf = pf + 4;

Взгляните на следующую программу: сможете ли вы предсказать результат? Происходит ли в программе переход указателя pf, имеющего тип float, к следующему числу?

// 10SIZEPT.CPP

// Программа на C++, использующая операцию sizeof и арифметические

// операции с указателями

#include "stdafx.h"

#include "E:\LECTURE\AlgorithmProgramming 02\Universal\_HederFile.h"

void StopWait(void);

main()

{

float fvalues[]={(float) 15.38,(float) 12.34,(float) 91.88,

(float) 11.11,(float) 22.22}, \*pf;

size\_t fwidth;

pf=&fvalues[0];

cout << "pf= " << pf << endl;

fwidth=sizeof(float);

pf=pf+fwidth;

cout << "pf= " << pf << " fwidth= " << fwidth << endl;

StopWait(); /\* Wait a little \*/

return (0);

}

Попробуйте выполнить программу в пошаговом режиме с использованием встроенного отладчика. При помощи окна Trace (трассировка) проследите изменение переменных pi и fwidth.

Предположим, что отладчик присвоил переменной pf значение адреса fwidth, и pf содержит значение FFCA. Переменной fwidth присваивается значение sizeof (float), которое равно 4. Что произойдет при выполнении последнего оператора программы? Значение переменной pf становится равным FFDA, а не FFCE. Почему? Не забывайте о том, что при выполнении арифметических операций с указателями учитывается размер того объекта, на который указатель ссылается (4 умножить на (4-х байтное число float)=16). На самом деле указатель pf в программе смещается на четыре числа float, то есть к значению 22.22.

Мы намеренно исказили смысл программы, назвав переменную fwidth ("ширина"). Для того, чтобы программа имела логический смысл, ее нужно переписать следующим образом:

// 10PTSIZE.CPP

// Программа на C++, использующая операцию sizeof и арифметические

// операции с указателями

#include "stdafx.h"

#include "E:\LECTURE\AlgorithmProgramming 02\Universal\_HederFile.h"

void StopWait(void);

main()

{

float fvalues[]={(float) 15.38,(float) 12.34,(float) 91.88,

(float) 11.11,(float) 22.22}, \*pf;

int inumber\_of\_elements\_to\_skip; //число\_элементов\_которые\_пропустить

pf=&fvalues[0];

inumber\_of\_elements\_to\_skip=1;

cout << "pf= " << pf << " \*pf= " << \*pf

<< " inumber\_of\_elements\_to\_skip= "

<< inumber\_of\_elements\_to\_skip << endl;

pf=pf+inumber\_of\_elements\_to\_skip;

cout << "pf= " << pf << " \*pf= " << \*pf << endl;

StopWait(); /\* Wait a little \*/

return (0);

}

### Арифметические операции с указателями и массивы.

В следующих двух программах с помощью индекса выполняется обработка массива из десяти символов. Обе программы считывают десять символов и затем печатают их в обратном порядке. В первой программе используется более традиционный для языков высокого уровня подход: обращение к элементам массива при помощи индексов. Вторая программа работает аналогично за исключением того, что обращение к элементам массива выполняется по адресу с использованием арифметических операций с указателями. Вот первая программа:

/\*10ARYSUB.C

Программа на С, использующая обычные индексы массива\*/

#include "stdafx.h"

#include "E:\LECTURE\AlgorithmProgramming 02\Universal\_HederFile.h"

#define ISIZE 10

void StopWait(void);

main()

{

char string10[ISIZE];

int i;

fputs("Enter 11 symbols\n",stdout);

for(i=0; i < ISIZE; i++)

string10[i]=getchar();

for(i=ISIZE-1;i >+ 0; i--)

putchar(string10[i]);

fputs("\n",stdout);

StopWait(); /\* Wait a little \*/

return (0);

}

/\*10ARYPTR.C

Программа на С, в которой для доступа к элементам массива используются арифметические

операции с указателями.\*/

#include "stdafx.h"

#include "E:\LECTURE\AlgorithmProgramming 02\Universal\_HederFile.h"

#define ISIZE 10

void StopWait(void);

main()

{

char string10[ISIZE], \*pc;

int icount;

pc=&string10[0];

fputs("Enter 11 symbols\n",stdout);

for(icount=0; icount < ISIZE; icount++)

{

\*pc=getchar();

pc++;

}

fputs("\n",stdout);

pc=&string10[ISIZE-1];

for(icount=0; icount < ISIZE; icount++)

{

putchar(\*pc);

pc--;

}

fputs("\n",stdout);

StopWait(); /\* Wait a little \*/

return (0);

}

Поскольку первая программа достаточно очевидна, обсудим вторую программу, в которой используются арифметические операции с указателями. Переменная рс имеет тип char \*, то есть она является указателем на символ. Так как каждая ячейка массива string10 является символом, переменная рс может указывать на любой из них. В следующем операторе запоминается адрес первой ячейки массива string10 в переменной рс

pc=string10[0];

В цикле for выполняется чтение ISIZE символов, которые запоминаются в массиве string10. В следующем операторе используется операция разыменования (\*), благодаря которой в левой части операции присваивания (вот еще один пример ivalue) используется не значение переменной рс (которая содержит просто адрес), а ячейка, на которую рс указывает:

\*pc=getchar();

Смысл этого оператора — запомнить символ в некоторой ячейке массива string10, а не в переменной рс.

Для печати массива в обратном направлении сначала в программе указателю рс присваивается адрес последнего элемента массива:

pc=string10 + (ISIZE - 1);

После прибавления значения (ISIZE — 1) к начальному адресу массива string10 переменная рс будет указывать на десятый элемент. Напоминаем, что это смещение. Первый элемент массива имеет нулевое смещение. В цикле for значение рс уменьшается на 1, при этом указатель перемещается к начальным элементам массива. Для того чтобы полностью понять процесс изменения указателя рс, можно выполнить трассировку этого примера с помощью встроенного отладчика.

### Операции с указателями.

Что же мы теперь умеем делать с указателями? Язык Си предлагает пять основных операций, которые можно применять к указателям, а нижеследующая программа демонстрирует эти возможности. Чтобы показать результаты каждой операции, мы будем печатать значение указателя (являющегося адресом, на который ссылается указатель), значение, находящееся по этому адресу, и адрес самого указателя.

/\* операции с указателями \*/

#include<stdio.h>

#define PR(X) printf(" X = %u, \*X = %d, &X = %u\n", X, \*X, &X)

int urn[]= {100, 200, 300};

main()

{int \*ptr1, \*ptr2;

ptr1 = urn;

ptr2 = &urn[2];

PR(ptr1);

ptr1++;

PR(ptr1);

PR(ptr2);

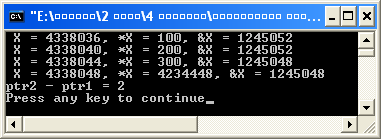
++ptr2;

PR(ptr2);

printf("ptr2 - ptr1 = %u\n", ptr2 - ptr1);

}

В результате работы программы получены следующие результаты:



Программа демонстрирует пять основных операций, которые можно выполнять над переменными типа указатель.

1. ПРИСВАИВАНИЕ. Указателю можно присвоить адрес. Обычно мы выполняем это действие, используя имя массива или операцию получения адреса (&). Программа присваивает переменной ptr1 адрес начала массива urn; этот адрес принадлежит ячейке памяти с номером 18. (В нашей системе статические переменные запоминаются в ячейках оперативной памяти.) Переменная ptr2 получает адрес третьего и последнего элемента массива, т. е. urn [2].
2. ОПРЕДЕЛЕНИЕ ЗНАЧЕНИЯ. Операция \* выдает значение, хранящееся в указанной ячейке. Поэтому результатом операции \*ptr1 в самом начале работы программы является число 100, находящееся в ячейке с номером 18.
3. ПОЛУЧЕНИЕ АДРЕСА УКАЗАТЕЛЯ. Подобно любым переменным переменная типа указатель имеет адрес и значение. Операция & сообщает нам, где находится сам указатель. В нашем примере указатель ptr1 находится в ячейке с номером 55990. Эта ячейка содержит число 18, являющееся адресом начала массива urn.
4. УВЕЛИЧЕНИЕ УКАЗАТЕЛЯ. Мы можем выполнять это действие с помощью обычной операции сложения либо с помощью операции увеличения. Увеличивая указатель, мы перемещаем его на следующий элемент массива. Поэтому операция ptr1++ увеличивает числовое значение переменной ptr1 на 2 (два байта на каждый элемент массива целых чисел), после чего указатель ptr1 ссылается уже на urn[1] (смотри рисунок). Теперь ptr1 имеет значение 20 (адрес следующего элемента массива), а операция \*ptr1 выдает число 200, являющееся значением элемента urn[1]. Заметим, что адрес самой ячейки ptr1 остается неизменным, т. е. 55990. После выполнения операции сама переменная не переместилась, потому что она только изменила значение!

13_02

Аналогичным образом можно и уменьшить указатель.

Однако при этом следует соблюдать осторожность. Машина не следит, ссылается ли еще указатель на массив или уже нет. Операция ++ptr2 перемещает указатель ptr2 на следующие два байта, и теперь он ссылается на некоторую информацию, расположенную в памяти за массивом.

Кроме того, оператор увеличения можно использовать для переменной типа указатель, но не для констант этого типа подобно тому, как вы не можете применять оператор увеличения для обычных констант. Для переменных и констант типа указатель можно использовать простое сложение:

**Правильно Неправильно**

ptr1 ++ ; urn++ ;

х++ ; 3++;

ptr2 = ptr1 + 2; ptr2 = urn++;

ptr2 = urn + 1; x = у + 3++;

1. РАЗНОСТЬ. Можно находить разность двух указателей. Обычно это делается для указателей, ссылающихся на элементы одного и того же массива, чтобы определить, на каком расстоянии друг от друга находятся элементы. Помните, что результат имеет тот же тип, что и переменная, содержащая размер массива.

Перечисленные выше операции открывают большие возможности. Программисты на языке Си создают массивы указателей, указатели на функции, массивы указателей на указатели, массивы указателей на функции и т. д. Мы будем придерживаться основных применений, которые уже упоминались. Первое из них — передача информации в функцию и из нее. Мы использовали указатели, когда хотели, чтобы функция изменила переменные, находящиеся в вызывающей программе. Второе — использование указателей в функциях, работающих с многомерными массивами.

К указателям (кроме указателей на переменные типа void) могут применяться арифметические операции. Для изменения пустого указателя он должен быть предварительно приведен к какому-либо типу (не void). Рассмотрим подробнее суть арифметических операций для указателей.

Компилятор, зная тип указателя, вычисляет размер переменной этого же типа, после чего модифицирует адрес, содержащийся в указателе в соответствии с заданной арифметической операцией, но с учетом вычисленного для данного типа размера. Это означает, что если объявлен указатель типа double, занимающего в памяти 8 байт, операция, например, инкремента указателя увеличит значение адреса не на один, а на восемь байт:

#include <iostream>

using namespace std;

int main()

{

setlocale(LC\_ALL, "Rus");

double Var;

double\* ptr = &Var;

cout << ptr << '\n';

ptr++;

cout << ptr;

getchar(); getchar();

return 0;

}

В результате будет выведено, например:

0x0068fdfc

0x0068fe04



Указатели можно вычитать друг из друга, тем самым, определяя количество элементов (того же типа, на который указывают указатели), расположенных между ними. Этот прием бывает полезным при операциях с массивами и символьными строками, которые будут рассмотрены позже.

Существует возможность использования при объявлении указателей ключевого слова const. При этом следует принимать во внимание, что применение данного спецификатора трактуется компилятором следующим образом:

// Указатель на константу типа char

// разыменованное значение неизменно

const char\* myKey;

// Константный указатель типа int,

// ВСЕГДА указывает на один и тот же адрес

int\* const Cell;

Попытка модификации содержимого указателя на константу, как и применение любой арифметической операции к константному указателю, приведет к сообщению об ошибке.

В таблице приводится перечень допустимых арифметических операций над указателями.

**Таблица. Арифметические операции над указателями**

|  |  |
| --- | --- |
| **Наименование операции** | **Пример** |
| Сравнение на равенство | Pi == Pi |
| Сравнение на неравенство | Pi != Pi |
| Сравнение на меньше | Pi < Pi |
| Сравнение на меньше или равно | Pi <= Pi |
| Сравнение на больше | Pi > Pi |
| Сравнение на больше или равно | Pi >= Pi |
| Вычисление числа элементов между указателями | Рi - Pi |
| Вычисление указателя, отстоящего от заданного на определенное число элементов | Pi Pi + n - n |

### Применение к указателям оператора sizeof.

Как и к любой переменной или типу данных, к указателям можно применять операцию определения размера sizeof. Выше отмечалось, что размер указателя может принимать одно из двух значений: два или четыре байта, что позволяет указателю адресовать 2(2\*8) = 65 Кбайт или 2(4\*'8) = 4 Гбайта памяти соответственно. На размер указателя (2 или 4 байта) влияет выбранная модель памяти и ряд других причин, которые будут рассмотрены в главах, посвященных моделям памяти и модификаторам.

К указателям можно применять не только оператор sizeof, но и одноименную функцию, что и демонстрирует следующий пример.

#include <iostream>

using namespace std;

int main ()

{

char endl = '\n';

unsigned long ulCone = 546213;

bool IsTrue = false;

unsigned long\* pUL = &ulCone;

bool\* pBool = &IsTrue;

cout << sizeof pUL << endl;

cout << sizeof(pUL) << endl;

cout << endl;

cout << sizeof pBool << endl;

cout << sizeof(pBool) << endl;

setlocale(LC\_ALL, "Rus");

getchar(); getchar();

return 0;

}



В начале программы объявляются целочисленная и логическая переменные, а также соответствующие указатели, после чего выводится информация о размере указателей.

### Сложности при использовании операций ++ и --.

В качестве напоминания заметим, что два следующих оператора обращаются к разным ячейкам памяти:

\*pc++=getchar();

\*++pc=getchar() ;

В первом операторе символ, полученный функцией getchar(), запоминается в текущей ячейке, на которую указывает рс, а затем рс инкрементируется. Во втором операторе сначала инкрементируется адрес, хранящийся в рс, а затем символ, полученный от функции, запоминается в той ячейке, на которую указывает обновленный адрес. Далее вы увидите, как эти два различных типа присваивания значений указателям используются для обращения к элементам списка argv.

### Сравнение указателей.

Вы уже познакомились с примерами, в которых указатели инкрементируются и декрементируются при помощи операций ++ и --, а также с результатами сложения указателей и целых чисел. С указателями можно выполнять и другие операции. В их числе следующие:

* Вычитание целого числа из указателя
* Вычитание одного указателя из другого (обычно они указывают на один и тот же объект)
* Сравнение указателей при помощи операций отношения, например, = или >=

Поскольку операция вычитания (указатель — число) похожа на операцию сложения (указатель + число) (которая уже рассматривалась), то очевидно, что в этом случае результирующее значение указателя будет ссылаться на ячейку памяти, отстоящую на целое число элементов относительно первоначального значения.

При вычитании двух указателей получается некоторое постоянное значение, представляющее собой число элементов, располагающихся между двумя указателями. Подразумевается, что оба указателя одного типа и указывают на один и тот же массив. Вычитание указателей разных типов или указателей, ссылающихся на различные массивы, дает непредсказуемые результаты.

**ПРИМЕЧАНИЕ.**

Для всех арифметических операций с указателями отсутствует какая-либо проверка, определяющая, что вычисленное значение указателя выходит за указанные границы массива.

Указатели одинаковых типов (то есть указатели, которые ссылаются на один и тот же тип данных) можно также сравнивать между собой. Полученный результат — ИСТИНА (!0) или ЛОЖЬ (0) — можно или анализировать, или присвоить некоторой целой переменной — так же, как и результат любого другого логического выражения. Путем сравнения можно определить: равны указатели или нет, какой из них больше, а какой меньше. Один указатель будет меньше другого, если первый указатель ссылается на элемент массива с меньшим индексом. (Напоминаем, что указатели и индексы практически идентичны.) При выполнении данной операции предполагается, что указатели ссылаются на один и тот же массив.

И наконец, указатель можно сравнивать с null-значением — с нулем. В этом случае возможен только результат "равно" или "не равно", так как проверка указателей на отрицательные значения не имеет смысла. Null-значение указателя означает то, что он не имеет значения или не на что не ссылается. Null или ноль — это единственное числовое значение, которое можно непосредственно присвоить указателю, не выполняя операции приведения типа.

Необходимо заметить, что над операндами-указателями выполняется преобразование указателей. Это означает, что любой указатель можно сравнить с постоянным выражением, дающим значение ноль, и что любой указатель можно сравнить с указателем на тип void \*. (В последнем случае указатель предварительно преобразуется в тип void \*.)

### Переносимость указателей.

В приведенных примерах адреса представлялись целыми числами. Вы можете предположить, что в С указатели имеют тип int. Это не так. Указатель содержит адрес переменной некоторого типа, однако сам указатель не относится ни какому простому типу данных вроде int, float и им подобному. В конкретной системе С указатель может копироваться в переменную типа int, а переменная int может копироваться в указатель; однако, язык С не гарантирует, что указатели могут храниться в переменных типа int. Для обеспечения переносимости программного кода таких операций следует избегать.

Кроме того, разрешены не все арифметические операции с указателями. Например, запрещается складывать или перемножать два указателя, или делить один указатель на другой.

### Использование функции sizeof() с указателями в среде DOS.

Действительная длина указателя зависит от двух вещей: от размера модели памяти, выбранной для приложения, и от того, как используются непереносимые, зависящие от реализации ключевые слова near, \_far и huge.

В микропроцессорах от 80486 до 8088 используется сегментная (страничная) адресация, при которой адрес разбивается на две части: на сегмент и смещение. Во многих почтовых отделениях имеется по несколько стоек с почтовыми ящиками, каждый из которых имеет собственный номер. Адресация по схеме "сегмент-смещение" следует тому же принципу. Для обращения к вашему почтовому ящику сначала необходимо определить, в какой из стоек он находится (сегмент), а затем узнать номер ящика (смещение).

Когда вы знаете, что код и данные вашего приложения умещаются в одном 64К блоке памяти, то можно выбирать малую (small) модель памяти. Если следовать аналогии с почтовым отделением, то это означает, что код программы и данные целиком располагаются в одной стойке (сегменте), при этом код и данные имеют уникальные номера почтовых ящиков (смещения) в стойке.

Если же приложение не вписывается в такие рамки — из-за его размера и из-за количества хранимых и обрабатываемых данных, — то нужно выбирать большую (large) модель памяти. Следуя указанной аналогии, это означает, что код программы должен храниться в одной стойке, а данные — совершенно в другой.

Когда приложение использует общий сегмент памяти для кода и данных, то для вычисления адреса объекта достаточно определить его смещение в этом сегменте. Это очень простая операция. Если в приложении для кода и данных используются разные сегменты памяти, то вычисление адреса некоторого объекта немного сложнее. Во-первых, нужно определить сегмент для кода и сегмент для данных, а затем — смещение в соответствующем сегменте. Для этого, конечно же, требуется процессорное время.

В языке C++ можно переопределять размер указателя, заданный по умолчанию для конкретной переменной, при помощи ключевых слов near, \_far и huge. Нужно, однако, заметить, что при их использовании в приложении код программы становится менее переносимым, так как на различных компьютерах эти ключевые слова дадут разный результат. Ключевое слово near задает указатели, имеющие только смещение, в тех случаях, когда по умолчанию указатели имеют вид сегмент-смещение. Ключевое слово \_far создает указатель типа сегмент-смещение тогда, когда по умолчанию указатели имеют только смещение. Ключевое слово huge также создает указатель вида сегмент-смещение вместо нормализованного указателя. Обычно ключевое слово near используется для увеличения скорости выполнения программы, а использование ключевого слова \_far обеспечивает правильную работу указателя для любой выбранной модели памяти.

Для многих приложений можно просто игнорировать эту проблему, позволяя компилятору самому выбрать модель памяти по умолчанию. Но при таком подходе могут возникнуть проблемы, если вы, к примеру, попытаетесь обратиться по абсолютному адресу (возможно, к аппаратному регистру или к особой области памяти), лежащему за пределами сегмента вашей программы.

С другой стороны, можно спросить: а почему бы просто не использовать самую большую модель памяти, доступную для приложения? Это можно сделать, но при этом теряется эффективность. Если все данные находятся в одном сегменте, то указатель имеет размер смещения. Если же данные и код программы располагаются в разных областях памяти, указатель имеет размер сегмента и смещения, и при каждом изменении указателя нужно вычислять и сегмент, и смещение. В следующей программе функция sizeof() используется для печати наименьшего и наибольшего из допустимых размеров указателя.

В этой программе на C++ печатается размер указателей по умолчанию, и их размер с моделями памяти \_\_far и near. Также используется директива препроцессора (#), определяющая PRINT\_SIZEOF с аргументом A\_POINTER, поэтому печатается не только размер указателя, но и его название.

//10STRIZE.CPP

//sizeof с указателями

#include <stdio.h>

#define PRINT\_SIZEOF(A\_POINTER) \

printf("sizeof\t("#A\_POINTER")\t= %d\n", \

sizeof(A\_POINTER))

main()

{

char \*reg\_pc;

long double \*reg\_pldbl;

char \_\_far \*far\_pc;

long double \_\_far \*far\_pldbl;

char \_\_near \*near\_pc;

long double \_\_near \*near\_pldbl;

PRINT\_SIZEOF(reg\_pc);

PRINT\_SIZEOF(reg\_pldbl);

PRINT\_SIZEOF(far\_pc);

PRINT\_SIZEOF(far\_pldbl);

PRINT\_SIZEOF(near\_pc);

PRINT\_SIZEOF(near\_pldbl);

return (0);

}

Результат выполнения программы:

|  |  |  |
| --- | --- | --- |
| sizeof | (reg\_pc) | = 2 |
| sizeof | (reg\_pldbl) | = 2 |
| sizeof | (far\_pc) | = 4 |
| sizeof | (far\_pldbl) | = 4 |
| sizeof | (near\_pc) | = 2 |
| sizeof | (near\_pldbl) | = 2 |

#include <iostream>

using namespace std;

#if !defined(ARRAY\_SIZE)

#define ARRAY\_SIZE(x) (sizeof((x)) / sizeof((x)[0]))

#endif

int main()

{

int days[] = {1,2,3,4,5};

int \*ptr = days;

printf("%u\n", ARRAY\_SIZE(days));

printf("%u\n", sizeof(ptr));

getchar();

getchar();

return 0;

}



#include <stdio.h>

#define NUM\_DAYS 5

typedef int days\_t[ NUM\_DAYS ];

#define SIZEOF\_DAYS ( sizeof( days\_t ) )

int main() {

days\_t days;

days\_t \*ptr = &days;

printf( "SIZEOF\_DAYS: %u\n", SIZEOF\_DAYS );

printf( "sizeof(days): %u\n", sizeof(days) );

printf( "sizeof(\*ptr): %u\n", sizeof(\*ptr) );

printf( "sizeof(ptr): %u\n", sizeof(ptr) );

getchar();

getchar();

return 0;

}



## Указатели на функции.

В C++ указатели могут ссылаться на функции. Имя функции само по себе представляет константный указатель на эту функцию, то есть содержит адрес входа в нее. Однако можно задать свой собственный указатель на данную функцию:

тип (\*имя указателя) (список\_типа\_аргументов)

Например,

bool (\*MyFuncPtr)(char, long);

объявляет указатель MyFuncPtr. ссылающийся на функцию, возвращающую логическое значение и принимающую в качестве параметров одну символьную и одну целую длинную переменную.

Вызов функции через указатель осуществляется так, будто имя указателя является просто именем вызываемой функции. То есть после имени указателя следует список аргументов, ожидаемых функцией. Так, для приведенного выше указателя на функцию MyFuncPtr вызов может выглядеть, например, следующим образом:

bool bVar;

char Symbol = 'x';

long lNum = 0L;

bVar = MyFuncPtr(Symbol, lNum);

Рассмотрим пример использования указателя на функцию. Пусть задано какое-либо целое число, характеризующее месяц в году. Требуется определить, к какому- сезону года относится данный месяц.

#include <iostream>

using namespace std;

bool IsSpring(int);

bool IsSummer(int);

bool IsAutumn(int);

bool IsWinter(int);

int main()

{

setlocale(LC\_ALL, "Rus");

int Month = 4;

bool (\*pFunc)(int) = IsWinter;

if(pFunc(Month))

cout << "Зима";

pFunc = IsAutumn;

if(pFunc(Month))

cout << "Осень";

pFunc = IsSummer;

if(pFunc(Month))

cout << "Лето";

pFunc = IsSpring; if(pFunc(Month))

cout << "Весна";

getchar(); getchar();

return 0;

}

bool IsSpring(int x)

{return (x>2 && x<6);}

bool IsSummer(int x)

{return (x>5 && x<9);}

bool IsAutumn(int x)

{return (x>8 && x<12);}

bool IsWinter(int x)

{return (x>11 && x<3);}



В приведенной программе создается указатель (pFunc) на функцию (принимающую один целочисленный аргумент и возвращающую логическое значение), инициализированный адресом функции IsWinter. Далее через указатель поочередно вызываются функции, определяющие, попадает ли принятый аргумент в заданный диапазон. Возвращаемые функциями логические значения определяют, выводить ли на экран сообщение.

Указатели на функции используются часто в качестве аргументов других функций. Таким образом создаются универсальные функции, значительно упрощающие текст сложной программы (например, численное решение уравнений дифференцирование интегрирование). Некоторые библиотечные функции в качестве параметра также принимают указатели на функции.

Во всех рассмотренных до сих пор примерах показывалось, как можно обращаться к данным различных типов при помощи указателей. Оказывается, что используя указатель на функцию можно обратиться также и к фрагментам кода программы. Указатели на функции служат тем же целям, что и указатели на данные: они позволяют обращаться к функции косвенно — так же, как указатели на данные обеспечивают косвенное обращение к данным.

Указатель на функцию можно использовать в нескольких важных случаях. Рассмотрим, например, функцию qsort(). Одним из ее параметров является указатель на функцию. Адресуемая функция осуществляет необходимое сравнение, которое должно выполняться для элементов сортируемого массива. Применение в qsort() указателя на функцию вызвано тем, что процесс сравнения двух элементов может быть весьма сложным, и при помощи одного управляющего флага его представить нельзя. Функцию нельзя передать по значению — то есть, передать ее код. Однако, в С можно передать указатель на код или указатель на функцию.

Концепцию указателей на функцию обычно иллюстрируют при помощи поставляемой вместе с компилятором функции qsort(). К сожалению, нередко встречаются объявления указателей и на другие библиотечные функции. Следующие ниже программы на С и C++ показывают, как описать указатель на функцию и как передать пользовательскую функцию в функцию qsort(), объявленную в файле stdlib.h.

Вот программа на С:

/\*10FNCPTR.C

Программа на С, иллюстрирующая объявление пользовательской функции

и указателя на функцию, используемого с qsort().\*/

#include "stdafx.h"

#include "E:\LECTURE\AlgorithmProgramming 02\Universal\_HederFile.h"

#define IMAXVALUES 10

int icompare\_funct(const void \*iresult\_a, const void \*iresult\_b);

int (\*ifunct\_ptr)(const void \*,const void \*);

void StopWait(void);

main()

{

int i;

int iarray[IMAXVALUES]={0,5,3,2,8,7,9,1,4,6};

ifunct\_ptr=icompare\_funct;

qsort(iarray,IMAXVALUES, sizeof(int), ifunct\_ptr);

for(i=0; i < IMAXVALUES; i++)

printf("%d ",iarray[i]);

printf("\n");

StopWait(); /\* Wait a little \*/

return (0);

}

int icompare\_funct(const void \*iresult\_a, const void \*iresult\_b)

{

return((\*(int \*)iresult\_a)-(\*(int \*) iresult\_b));

}

Функция icompare\_funct() (которая будет называться вызываемой функцией) имеет прототип, соответствующий требованиям четвертого параметра функции qsort() (которая будет называться вызывающей функцией).

Несколько отклоняясь в сторону, можно пояснить: четвертый параметр функции qsort() должен быть указателем на функцию. Вызываемая функция должна получать два параметра типа const void \* и возвращать значение типа int. Это объясняется тем, что qsort() использует вызываемую функцию для реализации алгоритма сравнения при сортировке. Теперь, когда вам стал понятен прототип вызываемой функции icompare\_funct(), уделите немного внимания изучению ее программного кода.

Если вызываемая функция возвращает значение < 0, то первый ее параметр меньше, чем значение второго параметра. Ноль свидетельствует о равенстве параметров, а значение > 0 указывает на то, что второй параметр больше первого. Все эти отношения реализованы в функции icompare\_funct() при помощи единственного оператора:

return((\*(int \*)iresult\_a) - (\*(int \*) iresult\_b));

Поскольку оба указателя передаются как тип void \*, они преобразуются в соответствующий тип указателей int \*, а затем по отношению к ним применяется операция разыменования (\*). Результат вычитания двух величин, на которые ссылаются указатели, удовлетворяет критерию сравнения функции qsort().

После анализа прототипа функции icompare\_funct() нужно обратить внимание на объявление указателя на функцию, которое следует за этим прототипом:

int icompare\_funct(const void \*iresult\_a, const void \*iresult\_b);

int (\*ifunct\_ptr)(const void \*, const void \*);

Тип функции определяется ее возвращаемым значением и списком аргументов. Указатель на icompare\_funct() должен иметь такой же список параметров и тип возвращаемого значения. Можно предположить, что следующий оператор отвечает этим требованиям:

int \*ifunct\_ptr(const void \*, const void \*);

Это почти правильно. Проблема в том, что компилятор интерпретирует этот оператор как описание функции ifunct\_ptr(), получающей два параметра и возвращающей указатель типа int \*. Операция разыменования связана не с функцией ifunct\_ptr(), а с описателем типа. Для того чтобы связать операцию разыменования с ifunct\_ptr(), необходимы круглые скобки.

Исправленный таким образом оператор объявляет ifunct\_ptr() как указатель на функцию, получающую два параметра и возвращающую тип int, -то есть как указатель того типа, который нужен в качестве четвертого параметра функции qsort().

Единственное, что остается сделать в теле функции main(), это инициализировать указатель ifunct\_ptr и присвоить ему адрес функции icompare\_funct(). Параметрами функции qsort() являются следующие величины: адрес нулевого элемента сортируемой таблицы (iarray), количество элементов в таблице (IMAXVALUES), размер каждого элемента (sizeof(int)) и указатель на функцию сравнения (ifunct\_ptr()).

Эквивалентная программа на C++:

// 10QSORT.CPP

// Программа на C++, иллюстрирующая объявление пользовательской

// функции и указателя на функцию, которая используется вместе с qsort()

#include "stdafx.h"

#include "E:\LECTURE\AlgorithmProgramming 02\Universal\_HederFile.h"

#define IMAXVALUES 10

int icompare\_funct(const void \*iresult\_a, const void \*iresult\_b);

int (\*ifunct\_ptr)(const void \*,const void \*);

void StopWait(void);

main()

{

int i;

int iarray[IMAXVALUES]={0,5,3,2,8,7,9,1,4,6};

ifunct\_ptr=icompare\_funct;

qsort(iarray,IMAXVALUES, sizeof(int), ifunct\_ptr);

for(i=0; i < IMAXVALUES; i++)

cout << " " << iarray[i];

cout << endl;

StopWait(); /\* Wait a little \*/

return (0);

}

int icompare\_funct(const void \*iresult\_a, const void \*iresult\_b)

{

return((\*(int \*)iresult\_a)-(\*(int \*) iresult\_b));

}

Понимание синтаксиса указателей на функцию может оказаться непростым. Рассмотрим несколько примеров:

int \*(\* (\*ifunct\_ptr) (int)) [5];

float (\*(\*ffunct\_ptr)(int,int))(float);

typedef double (\*(\*(\*dfunct\_ptr)())[5])();

dfunct\_ptr A\_dfunct\_ptr; (\*(\* function\_ary\_ptrs()) [5]) () ;

Первый оператор описывает ifunct\_ptr() как указатель на функцию, которая получает целочисленный параметр и возвращает указатель на массив из пяти указателей типа int.

Второй оператор описывает ffunct\_ptr() как указатель на функцию, получающую два целых параметра и возвращающую указатель на функцию, у которой аргумент и возвращаемое значение типа float.

Используя описание typedef, можно избежать ненужного повторения сложных объявлений. Описание typedef читается следующим образом: dfunct\_ptr() определяется как указатель на функцию без параметров, которая возвращает указатель на массив из пяти указателей, ссылающихся на функции, не имеющие параметров и возвращающие значение типа double.

Последний оператор — это объявление функции, а не переменной. Оператор описывает function\_ary\_ptrs() как функцию без параметров, которая возвращает указатель на массив из пяти указателей, ссылающихся на функции, не имеющие аргументов и возвращающие значение типа int (тип int задается в С и C++ по умолчанию).

Здесь может порадовать только то, что такие сложные объявления и описания встречаются достаточно редко. Но если вы разберетесь с подобным синтаксисом, то сможете уверенно анализировать любые конструкции.

## Динамическая память.

При компиляции программы на С память компьютера разбивается на четыре области, содержащие код программы, все глобальные данные, стек и динамически распределяемую область памяти (иногда ее называют heap-куча). Динамическая память (heap) — это просто свободная область памяти, с которой работают при помощи функций динамического выделения памяти malloc() и free().

При вызове функции malloc() выделяется непрерывный блок памяти для указанного объекта и возвращается указатель на начало этого блока. Функция free() возвращает выделенный блок обратно в динамическую область памяти для повторного использования.

Аргумент, передаваемый функции malloc(), представляет собой объем необходимой памяти в байтах. Если имеется свободная память, функция возвращает указатель типа void \*, который можно преобразовать в любой необходимый тип указателя. Концепция указателей типа void была предложена стандартом ANSI С; она распространяется на указатели неизвестного типа или родовые (generic) указатели. Сам по себе указатель типа void не может использоваться для ссылки на какой бы то ни было объект (поскольку он не указывает ни на какой тип данных), однако он может содержать указатель любого другого типа. Следовательно, любой указатель можно преобразовать в указатель типа void и наоборот, без потери информации. В следующем фрагменте кода выделяется память для 300 чисел типа float:

float \*pf;

int inum\_floats = 300;

pf = (float \*) malloc(inum\_floats \* sizeof(float));

Функции malloc() был сделан запрос на выделение памяти некоторого объема; этот объем определяется как 300, умноженное на действительный размер числа типа float, и выражается в байтах. Для того чтобы получить указатель типа float, использована операция приведения типа (float \*). Каждый блок выделяемой памяти совершенно отделен и независим от любых других блоков памяти. О расположении блоков нельзя сделать абсолютно никаких предположений. Блоки обычно "помечаются" некоторыми признаками, позволяющими операционной системе управлять их положением и размером. Когда блок становится ненужным, его можно вернуть операционной системе, используя, следующий оператор:

free((void \*) pf);

Так же как в С, в языке C++ имеющаяся память распределяется двумя способами. Когда объявляются переменные, они создаются в стеке, и при этом указатель стека сдвигается вниз. Когда область действия переменных заканчивается (например, когда локальная переменная больше не нужна), память для таких переменных автоматически освобождается путем смещения указателя стека вверх. Размер стековой памяти всегда должен быть известен при компиляции.

В программе возможно существование переменных, размер которых при компиляции неизвестен. Для них необходимо самостоятельно выделять память из свободной области. Свободная память занимает нижнюю часть памяти программы и растет снизу вверх, а стек занимает верхнюю часть и растет сверху вниз.

Программы на С и C++ могут в любой момент выделять и освобождать динамическую память. Важно помнить, что на переменные, память для которых выделена в динамической области, не распространяются правила области действия, которые справедливы для других переменных. Эти переменные действуют всегда; поэтому, если вы выделили динамическую область памяти, то не должны забывать о ее освобождении. Если повторно выделить динамическую память, предварительно не освободив ее, то выполнение программы, скорее всего закончится крахом.

Для распределения динамической памяти в большинстве компиляторов С используются описанные выше библиотечные функции malloc() и free(), однако, в C++ эти средства считаются настолько важными, что они включены в ядро языка. В C++ для выделения и освобождения динамической памяти используются операторы new и delete. Для new аргументом является выражение, определяющее число выделяемых байт, а возвращаемое значение — это указатель на начало блока памяти. Аргументом для delete является начальный адрес освобождаемого блока памяти. Следующие две программы иллюстрируют сходство и различия приложений С и C++, использующих выделение динамической памяти. Вот пример на С:

/\*10MALLOC.C, версия 1

Программа на С, использующая функции malloc() и free()\*/

#include <stdio.h>

#include <stdlib.h>

#define ISIZE 512

void main()

{

int \*pimemory\_buffer;

pimemory\_buffer=malloc(ISIZE \* sizeof(int));

if(pimemory\_buffer == NULL)

printf("Insufficient memory\n");

else

printf("Memory allocated\n");

free(pimemory\_buffer);

}

/\*10MALLOC.C, версия 2

Программа на С, использующая функции malloc() и free()\*/

#include <stdio.h>

#include <stdlib.h>

#define ISIZE 512

void main()

{

int pimemory\_buffer;

pimemory\_buffer=(int)malloc(ISIZE \* sizeof(int));

if(pimemory\_buffer == NULL)

printf("Insufficient memory\n");

else

printf("Memory allocated\n");

free((void\*)pimemory\_buffer);

}

Первое, что представляет интерес, это второй оператор #include, включающий в программу заголовочный файл stdlib.h, в котором содержатся описания функций malloc() и free(). После описания указателя pimemory\_buffer, имеющего тип int \*, вызывается функция malloc(), которая возвращает адрес блока памяти размером ISIZE \* sizeof(int) байт. Для надежности всегда нужно проверять результат операции выделения памяти, для этого служит следующий далее оператор if-else. Если памяти для выделения блока недостаточно, то функция malloc() всегда возвращает null. В конце этой простой программы выделенная область возвращается обратно в динамическую память при помощи функции free(), которой передается начальный адрес выделенного блока.

Программа на C++ не слишком отличается:

// 10NEWDEL.CPP

// Простая программа на C++, использующая операции new и delete

#include "stdafx.h"

#include "E:\LECTURE\AlgorithmProgramming 02\Universal\_HederFile.h"

#define NULL 0

#define ISIZE 512

void StopWait(void);

main()

{

int \*primemory\_buffer;

primemory\_buffer=new int[ISIZE];

if(primemory\_buffer == NULL)

cout << "Insufficient memory" << endl; //Недостаточно памяти

else

cout << "Memory allocated" << endl; //Память выделена

delete(primemory\_buffer);

StopWait(); /\* Wait a little \*/

return (0);

}

Единственным существенным отличием между этими двумя программами является синтаксис, используемый в функции malloc() и операции new. В то время, как для правильной работы функции malloс() необходима операция sizeof, операция new автоматически выполняет функцию sizeof() с объявленным типом данных. Обе программы выделяют 512 2-х байтных блоков непрерывной памяти (для систем, в которых целое число занимает 2 байта).

### Использование указателей типа void.

Теперь, когда вы уже глубоко понимаете природу переменных-указателей, становится понятной необходимость указателей типа void. Суть указателя в том, что это переменная, в которой хранится адрес другой переменной. Если бы вы всегда знали размер указателя, то на этапе компиляции не нужно было бы определять его тип. Тогда можно было бы передать в функцию адрес данных любого типа. Затем функция могла бы преобразовать этот адрес в указатель нужного типа (на основании другой информации) и выполнить операции, определяющие результат. Этот процесс позволил бы создавать функции, работающие со множеством различных типов данных.

Именно поэтому в C++ появился тип указателей void. Если описатель void используется с указателем, то его смысл отличается от того, который подразумевается при описании списка параметров функции и ее возвращаемого значения (там он означает "пусто"). Указатель на void ссылается на любой тип данных. Следующая программа на C++ иллюстрирует использование указателей void:

// 10VOIDPT.CPP

// Программа на C++, использующая указатели void

#include "stdafx.h"

#include "E:\LECTURE\AlgorithmProgramming 02\Universal\_HederFile.h"

#define ISTRING\_MAX 50

void voutput(void \*pobject, char cflag);

void StopWait(void);

void main()

{

int \*pi;

char \*psz;

float \*pf;

char cresponse, cnewline;

cout << "Please enter the dynamic data type\n";

cout << "you woud like to create.\n\n";

cout << "Use (s)tring, (i)nt or (f)loat";

cout << endl;

cin >> cresponse;

cin.get(cnewline);

switch(cresponse)

{

case 's':

psz=new char[ISTRING\_MAX];

cout << "\nPliase enter a string: ";

cin.get(psz,ISTRING\_MAX);

voutput(psz, cresponse);

break;

case 'i':

pi=new int;

cout << "\nPliase enter an integer: ";

cin >> \*pi;

voutput(pi, cresponse);

break;

case 'f':

pf=new float;

cout << "\nPliase enter a float: ";

cin >> \*pf;

voutput(pf, cresponse);

break;

default:

cout << "\n\nObject type not implemented!";

}

StopWait(); /\* Wait a little \*/

}

void voutput(void \*pobject, char cflag)

{

switch(cflag)

{

cout << "\n\nThe string read in: " << (char \*) pobject;

delete pobject;

break;

case 'i':

cout << "\n\nThe integer read in: " << \*((int \*) pobject);

delete pobject;

break;

case 'f':

cout << "\n\nThe float read in: " << \*((float \*) pobject);

delete pobject;

break;

}

cout << endl;

}

Первый оператор программы, который представляет интерес, это прототип функции voutput(). Обратите внимание на то, что первый формальный параметр, pobject, имеет тип void \*, то есть является родовым указателем. Просматривая объявления данных, вы обнаружите указатели трех различных типов: int \*, char \* и float \*. Очевидно, что им присваиваются реальные адреса ячеек памяти соответствующих типов.

Выполнение программы начинается с ввода типа данных, который пользователь хотел бы динамически создать. Вас может удивить то, что для обработки ответа пользователя используются два отдельных оператора ввода. Первый оператор cin считывает ответ в виде одного символа, но оставляет символ перевода строки \n. Эта ситуация исправляется при помощи второго оператора ввода cin.get(cnewline).

Оператор switch анализирует ответ пользователя и вызывает соответствующий запрос и инициализацию указателя. Указатели инициализируются при помощи одного из трех способов:

psz=new char; pi=new int; pf=new float;

Следующий оператор используется для ввода строки символов; в данном примере длина строки ограничена ISTRING\_MAX (50) символами.

cin.get(psz,ISTRING\_MAX);

Поскольку первый параметр оператора ввода cin.get() является указателем на строку, при вызове функции voutput() нет необходимости выполнять разыменование переменной:

voutput(psz,cresponse);

Ситуация упрощается, если пользователь вводит целое или вещественное число. Два этих случая аналогичны, и отличаются только сообщением на экране и типом ссылочной переменной.

Обратите внимание на то, что три вызова функции voutput() имеют различные типы указателей:

voutput(psz,cresponse);

voutput(pi,cresponse);

voutput(pf,cresponse);

Функция voutput() может получать эти параметры только потому, что тип соответствующего формального параметра — void \*. Напоминаем, что для использования этих указателей необходимо сначала преобразовать их в соответствующий тип. Если в cout используется указатель на строку, то необходимо преобразовать указатель в тип char \*.

Создание и печать динамических переменных целого и вещественного типов аналогичны. Отличия только в последних двух операторах case, в которых выдается сообщение и выполняется операция приведения типа.

Хотя по завершении программы все динамические переменные бесследно исчезают, в каждой опции оператора case выполняется явное удаление указателя. Когда и где программа создает и удаляет динамические объекты — зависит от приложения.

## Указатели и массивы.

В следующих разделах приводится множество примеров, в которых затрагиваются концепция массивов и их связь с указателями.

Указатели позволяют нам работать с символическими адресами. Поскольку в реализуемых аппаратно командах вычислительной машины интенсивно используются адреса, указатели предоставляют возможность применять адреса примерно так, как это делается в самой машине, и тем самым повышать эффективность программ. В частности, указатели позволяют эффективно организовать работу с массивами. Действительно, как мы могли убедиться, наше обозначение массива представляет собой просто скрытую форму использования указателей.

Например, имя массива определяет также его первый элемент, т.е. если flizny[] — массив, то flizny == &flizny[0] и обe части равенства определяют адрес первого элемента массива. (Вспомним, что операция & выдает адрес.) Оба обозначения являются константами типа указатель, поскольку они не изменяются на протяжении всей программы. Однако их можно присваивать (как значения) переменной типа указатель и изменять значение переменной, как показано в нижеследующем примере. Посмотрите, что происходит со значением указателя, если к нему прибавить число.

/\* прибавление к указателю \*/

#include<stdio.h>

main ()

{

int dates[4], \*pti, index;

float bills [4], \*ptf;

pti = dates; /\* присваивает адрес указателю массива \*/

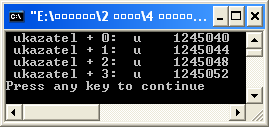
ptf = bills;

for (index = 0; index < 4; index++ )

printf(“ ukazatel + %d: %10 u %10u\n", index, pti + index, ptf + index);

}

Результат работы программы



Первая напечатанная строка содержит начальные адреса двух массивов, а следующая строка — результат прибавления единицы к адресу и т. д. Почему так получается?

56014 + 1 = 56016?

56026 + 1 = 56030?

В нашей системе единицей адресации является байт, но тип int использует два байта, а тип float — четыре. Что произойдет, если вы скажете: «прибавить единицу к указателю?» Компилятор языка Си добавит единицу памяти. Для массивов это означает, что мы перейдем к адресу следующего элемента, а не следующего байта. Вот почему мы должны специально оговаривать тип объекта, на который ссылается указатель; одного адреса здесь недостаточно, так как машина должна знать, сколько байтов потребуется для запоминания объекта. (Это справедливо также для указателей на скалярные переменные; иными словами, при помощи операции \*pt нельзя получить значение.)

13_01

Благодаря тому, что компилятор языка Си умеет это делать, мы имеем следующие равенства:

dates + 2 == &dates[2] /\* один и тот же адрес \*/

\*(dates + 2) == dates[2] /\* одно и то же значение \*/

Эти соотношения суммируют тесную связь между массивами и указателями. Они показывают, что можно использовать указатель для определения отдельного элемента массива, а также для получения его значения. По существу мы имеем два различных обозначения для одного и того же. Действительно, компилятор превращает обозначение массива в указатели, поэтому метод указателей более предпочтителен.

Между прочим, постарайтесь различать выражения \* (dates + 2), и \*dates + 2. Операция (\*) имеет более высокий приоритет, чем +, поэтому последнее выражение означает (\* dates) + 2:

\*(dates + 2) /\* значение 3-го элемента массива dates \*/

\*dates + 2 /\* 2 добавляется к значению 1-го элемента массива \*/

Связь между массивами и указателями часто позволяет нам применять оба подхода при создании программ. Одним из примеров этого является функция с массивом в качестве аргумента.

### Функции, массивы и указатели.

Массивы можно использовать в программе двояко. Во-первых, их можно описать в теле функции. Во-вторых, они могут быть аргументами функции. Все, что было сказано в этой главе о массивах, относится к первому их применению; теперь рассмотрим массивы в качестве аргументов.

Об этом уже говорилось в гл. 10. Сейчас, когда мы познакомились с указателями, можно заняться более глубоким изучением массивов-аргументов. Давайте проанализируем скелет программы, обращая внимание на описания:

/\* массив-аргумент \*/

main ()

{

int ages [50]; /\* массив из 50 элементов \*/

convert(ages);

…

}

convert (years)

int years []; /\* каков размер массива? \*/

{

…

}

Очевидно, что массив ages состоит из 50 элементов. А что можно сказать о массиве years? Оказывается, в программе нет такого массива. Описатель

int years[];

создает не массив, а указатель на него.

Вот вызов нашей функции:

convert(ages);

ages — аргумент функции convert. Вы помните, что имя ages является указателем на первый элемент массива, состоящего из 50 элементов. Таким образом, оператор вызова функции передает ей указатель, т. е. адрес функции convert (). Это значит, что аргумент функции является указателем, и мы можем написать функцию convert () следующим образом:

convert (уears)

int \*years;

{

}

Действительно, операторы

int years [];

int \*years;

синонимы. Оба они объявляют переменную years указателем массива целых чисел. Однако главное их отличие состоит в том, что первый из них напоминает нам, что указатель years ссылается на массив.

Как теперь связать его с массивом ages? Вспомним, что при использовании указателя в качестве аргумента, функция взаимодействует с соответствующей переменной в вызывающей программе, т.е. операторы, использующие указатель years в функции convert (), фактически работают с массивом ages, находящимся в теле функции main ().

Посмотрим, как работает этот механизм. Во-первых, вызов функции инициализирует указатель years, ссылаясь на ages[0]. Теперь предположим, что где-то внутри функции convert () есть выражение years [3]. Как вы видели в предыдущем разделе, оно аналогично (years + 3). Однако если years указывает на ages[0], то years + 3 ссылается на ages[3]. Это приводит к тому, что \*(years + 3) означает ages[3]. Если внимательно проследить данную цепочку, то мы увидим, что years [3] аналогично (years + 3), которое в свою очередь совпадает с ages[3]. Что и требовалось доказать, т. е. операции над указателем years приводят к тем же результатам, что и операции над массивом ages. Короче говоря, когда имя массива применяется в качестве аргумента, функции передается указатель. Затем функция использует этот указатель для выполнения изменений в исходном массиве, принадлежащем программе, вызвавшей функцию.

### Использование указателей при работе с массивами.

Попробуем написать функцию, использующую массивы, а затем перепишем ее, применяя указатели.

Рассмотрим простую функцию, которая находит (или пытается найти) среднее значение массива целых чисел. На входе функции мы имеем имя массива и количество элементов. На выходе получаем среднее значение, которое передается при помощи оператора return. Оператор вызова функции может выглядеть следующим образом:

/\* X54.C \*/

#include <stdio.h>

void main(void)

{

int n1=9,i, \*pa1,\*pa,n;

int array[] = {9,5,3,1,0,-9,-5,-3,-1};

pa1=array;

int mean(int \*pa,int n);

i = mean(pa1,n1);

printf("Average is %d\n",i);

}

int mean(int \*pa,int n)

{

int index;

long sum;

if(n > 0)

{

for(index = 0, sum = 0; index < n; index++)

sum += \*(pa+index);

return((int) (sum/n));

}

else

{

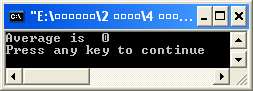
printf("Net massiva\n");

return(0);

}

}

Результат работы программы



Эту программу легко переделать, применяя указатели. Объявим ра указателем на тип int. Затем заменим элемент массива array[index] на соответствующее значение: \*(ра + index).

/\* Использование указателей для нахождения среднего значения

массива n целых чисел \*/

int mean(pa, n)

int \*pa, n;

{

int index;

long sum; /\*Если целых слишком много, их можно суммировать в формате long int \*/

if (n > 0)

{

for (index = 0, sum = 0; index < n: index++)

sum += \*(pa + index);

return((int) (sum/n) ); /\* Возвращает целое \*/

}

else

{

printf(" Нет массива. \n");

return(0);

}

}

Это оказалось несложным, но возникает вопрос: должны ли мы изменить при этом вызов функции, в частности numbs, который был именем массива в операторе mean (numbs, size)? Ничего не нужно менять, поскольку имя массива является указателем. Как мы уже говорили в предыдущем разделе, операторы описания

int pa[];

и

int \*pa;

идентичны по действию: оба объявляют ра указателем. В программе можно применять любой из них, хотя до сих пор мы использовали второй в виде \*(ра + index).

Понятно ли вам, как работать с указателями? Указатель устанавливается на первый элемент массива, и значение, находящееся там, добавляется в sum. Затем указатель передвигается на следующий элемент (к указателю прибавляется единица), и значение, находящееся в нем, также прибавляется к sum и т. д. Это похоже на механизм работы с массивом, где индекс действует как стрелка часов, показывающая по очереди на каждый элемент массива.

Теперь у нас есть два подхода: какой же из них выбрать? Во-первых, хотя массивы и указатели тесно связаны, у них есть отличия. Указатели являются более общим и широко применяемым средством, однако многие пользователи (по крайней мере, начинающие) считают, что массивы более привычны и понятны. Во-вторых, при использовании указателей у нас нет простого эквивалента для задания размера массива. Самую типичную ситуацию, в которой можно применять указатель, мы уже показали: это функция работающая с массивом, который находится где-то в другой части программы. Мы предлагаем использовать любой из подходов по вашему желанию. Однако несомненное преимущество использования указателей в приведенном выше примере должно научить вас легко применять их, когда в этом возникает необходимость.

### Строки (массивы типа char).

Многие строковые операции в С обычно выполняются с использованием указателей и арифметических операций с указателями для доступа к элементам символьного массива. Это обусловлено тем, что символьные массивы или строки, как правило, обрабатываются строго последовательно. Напоминаем, что все строки в С заканчиваются символом null (\0). Следующая программа на C++ представляет собой модификацию описанного ранее примера печати палиндрома; она иллюстрирует использование указателей с символьными массивами:

// 10CHRARY.CPP

// Программа на C++, печатающая массив символов в обратном порядке

// и использующая указатель на символ и оператор декремента

#include "stdafx.h"

#include "E:\LECTURE\AlgorithmProgramming 02\Universal\_HederFile.h"

void StopWait(void);

void main()

{

char pszpalindrome[]="POOR DAN IN A DROOP";

char \*pc;

pc=&pszpalindrome[0]+(strlen(pszpalindrome)-1);

do

{

cout << \*pc;

pc--;

} while (pc >= pszpalindrome);

cout << endl;

StopWait(); /\* Wait a little \*/

}

После того, как в программе объявлен и инициализирован палиндром pszpalindrome, создается указатель рс типа char \*. Напоминаем, что имя массива само по себе является адресной переменной. Выполнение программы начинается с того, что указателю рс присваивается адрес последнего символа массива. Затем выполняется вызов функции strlen(), которая определяет размер символьного массива.

**ПРИМЕЧАНИЕ.**

Функция strlen() считает только символы. Она не включает в результат null - символ \0

На то, что из результата, полученного функцией, вычитается единица, есть своя причина. Программа учитывает тот факт, что адрес первого элемента массива имеет нулевое смещение. Следовательно, адрес смещения в указателе нужно увеличить на число, которое на единицу меньше реального количества символов.

После вычисления значения указателя, для последнего символа массива выполняется цикл do-while, в котором указатель используется для обращения к ячейке памяти, хранящей выводимый символ, и этот символ печатается. Затем вычисляется адрес следующего символа и его значение сравнивается с начальным адресом pszpalindrome. Если результат сравнения >=, то цикл повторяется.

### Массивы указателей.

В языках С и C++ можно создавать не только простые массивы и указатели. Их можно объединить в очень полезную конструкцию — массив указателей. Массив указателей — это такой массив, элементы которого являются указателями на другие объекты. Эти объекты, в свою очередь, могут быть указателями. Это означает, что можно иметь массив указателей, которые ссылаются на другие указатели.

Концепция массива указателей на указатели широко используется для аргументов командной строки argc и argv. Следующая программа находит наибольшее и наименьшее значение, введенное из командной строки. Аргументы командной строки могут быть только числами, перед которыми вводится команда, определяющая поиск наименьшего (-s,-S) или наибольшего значения (-1,-L).

// 10ARGCGV.CPP

// Программа на C++, использующая массив указателей для обработки

// аргументов командной строки argc and argv

#include "stdafx.h"

#include "E:\LECTURE\AlgorithmProgramming 02\Universal\_HederFile.h"

#define IFIND\_SMALLEST 0

#define IFIND\_LARGEST 1

void StopWait(void);

int main(int argc, char \*argv[])

{

char \*psz;

int ihow\_many;

int iwhich\_extreme=0;

int irange\_boundary=32767;

if(argc < 2) // сколько аргументов?

{

// Вам нужно ввести -S,-s,-L,-l и хотя бы одно число

cout << "\nYou need to enter an -S, -s, -L, -l"

<< " and at least one integer value" << endl;

exit(1);

}

// проверка аргументов слева направо

// далее следует ключ «-»

while(--argc > 0 && (\*++argv)[0] == '-')

{

for(psz=argv[0]+1; \*psz != '\0'; psz++) // второй элемент аргумента S или L

{

switch(\*psz) // вычисляется наименьшее или наибольшее

{

case 's':

case 'S':

iwhich\_extreme=IFIND\_SMALLEST;

irange\_boundary=32767;

break;

case 'l':

case 'L':

iwhich\_extreme=IFIND\_LARGEST;

irange\_boundary=0;

break;

default:

// Неизвестный аргумент – это не S и не L

cout << "\n\nUnknow argument! " << \*psz << endl;

exit(1);

}

}

}

if(argc == 0) // не уменьшилось argc до 0

{

// Введите хотя бы одно число

cout << "Pliase enter at least one number\n";

exit(1);

}

ihow\_many=argc; // сколько чисел будет сравниваться

while(argc--) // сколько чисел задано, столько и сравнивать

{

int present\_value;

present\_value=atoi(\*(argv++)); // преобразование строки в число

if(iwhich\_extreme == IFIND\_LARGEST && present\_value > irange\_boundary)

irange\_boundary=present\_value;

if(iwhich\_extreme == IFIND\_SMALLEST && present\_value < irange\_boundary)

irange\_boundary=present\_value;

}

// Самое большее / меньшее из ... введенных значений

cout << "The ";

cout << ((iwhich\_extreme) ? "largest" : "smallest");

cout << " of the " << ihow\_many << " value(s) input is " << irange\_boundary

<< endl;

StopWait(); /\* Wait a little \*/

return(0);

}

До того как разбирать текст программы, давайте познакомимся с возможными комбинациями команд, которые можно использовать при вызове этой программы. Ниже следует список возможных комбинаций команд:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **0** | **1** | **2** | **3** |
| **0** | 10argcgv |  |  |  |
| **1** | 10argcgv | 98 |  |  |
| **2** | 10argcgv | 98 | 21 |  |
| **3** | 10argcgv | -s | 98 |  |
| **4** | 10argcgv | -S | 98 | 21 |
| **5** | 10argcgv | -I | 14 |  |
| **6** | 10argcgv | -L | 14 | 67 |

Взглянув на функцию main(), вы заметите формальные параметры argc и argv, с которыми познакомились ранее. Напомним, что argc — это целое число, равное количеству отдельных элементов или аргументов, появляющихся в командной строке. Переменная argv ссылается на массив указателей на символьные строки.

**ПРИМЕЧАНИЕ.**

Переменная argv не является константой. Это переменная, значение которой можно изменять, и это является ключом к пониманию того, как она используется. Первый элемент массива, argv[0], является указателем на строку символов, которая содержит имя программы.

Перейдем к самой программе. Первый оператор if анализирует значение argc. Если оно меньше двух, то это означает, что пользователь ввел только имя программы без каких бы то ни было ключей. Поскольку это означает, что пользователь не знаком с опциями ключей, то программа перечисляет на экране правильные опции и затем завершается (выполняется функция ехit()).

В цикле while слева направо проверяются аргументы, при этом сначала декрёментируется argc. Если значение argc остается положительным, то анализируется правая часть логического выражения.

Сначала в правой части этого логического выражения инкрементируется указатель массива argv (++argv); при этом пропускается первый элемент массива, то есть имя программы, и указатель ссылается на второй его элемент. После операции инкремента, указатель используется для ссылки (\*++argv) на нулевое смещение ((\*++argv)[0]) первого элемента адресуемой строки. Если полученный символ равен "-", то программа определяет, что далее, возможно, следует ключ, например, -s или -L.

Начальное значение цикла for определяется текущим значением указателя argv, который инкрементируется для ссылки на второй элемент массива. Поскольку второй элемент массива argv является указателем на символьную строку, этот указатель может иметь индекс (argv[0]). Полное выражение, argv[0]+1, указывает на второй символ второй строки, на которую ссылается текущий адрес, хранящийся в argv. Этот второй элемент следует непосредственно за символом командного ключа "-". После того, как программа вычислит адрес этого символа, она запоминает адрес в переменной psz. Цикл for повторяется до тех пор, пока символ, на который указывает \*psz, не станет равным null-символу \0.

Далее в программе анализируется ключ, указывающий на то, какое значение хочет получить пользователь: наименьшее или наибольшее. В зависимости от значения ключа, переменной iwhich\_extreme присваивается значение соответствующей константы. Кроме этого, в каждом операторе case инициализируется переменная irange\_boundary, которой задается соответствующее значение для последующих сравнений. Если же пользователь ввел недопустимый ключ, например - d, то по опции default об этом выводится сообщение.

Во втором операторе if выполняется проверка: не уменьшилось ли значение argc до нуля. Сообщение печатается в случае, если анализ ключей в командной строке выполнен, но числа для обработки отсутствуют. Тогда программа завершается с кодом выхода, равным десятичной единице.

Успешное выполнение этой проверки (if) означает, что теперь в командной строке появились значения, которые нужно анализировать. Поскольку программа будет декрементировать argc, его текущее значение присваивается переменной ihowmany ("сколько чисел").

Цикл while выполняется до тех пор, пока имеются хотя бы два значения, которые нужно сравнивать. Этот цикл нужно начинать только в том случае, когда для сравнения имеется два или более чисел, поскольку оператор cout, следующий за циклом while, может обрабатывать командную строку с единственным числом.

Функция atoi() преобразует каждый оставшийся аргумент в целое число и запоминает результат в переменной present\_value ("текущее значение"). Напоминаем, что сначала нужно инкрементировать argv++, чтобы он указывал на первое сравниваемое число. Кроме этого, чтобы цикл while не выполнялся, если в командной строке имеется только одно число, счетчик там уже декрементирован.

Следующие два оператора if обновляют переменную irange\_boundary с учетом того, какое значение ищется по требованию пользователя: наименьшее или наибольшее. И, наконец, печатается результат выполнения программы; при этом используется интересная комбинация строковых литералов и условной операции.

### Дополнительная информация об указателях на указатели.

В следующей программе демонстрируются переменные-указатели, ссылающиеся на другие указатели. Этот пример рассматривается здесь потому, что в нем используется динамическое распределение памяти.

/\*10DBLPTR.C

Программа на С, использующая указатели с двумя уровнями косвенности\*/

#include "stdafx.h"

#include "E:\LECTURE\AlgorithmProgramming 02\Universal\_HederFile.h"

#define IMAXELEMENTS 3

void voutput(int \*\*ppiresult\_a, int \*\*ppiresult\_b, int \*\*ppiresult\_c);

void vassign(int \*pivirtual\_array[], int \*pinewblock);

void StopWait(void);

main()

{

int \*\*ppiresult\_a, \*\*ppiresult\_b, \*\*ppiresult\_c;

int \*pivirtual\_array[IMAXELEMENTS];

int \*pinewblock, \*pioldblock;

ppiresult\_a=&pivirtual\_array[0];

ppiresult\_b=&pivirtual\_array[1];

ppiresult\_c=&pivirtual\_array[2];

pinewblock=(int \*)malloc(IMAXELEMENTS\*sizeof(int));

pioldblock=pinewblock;

vassign(pivirtual\_array, pinewblock);

\*\*ppiresult\_a=7;

\*\*ppiresult\_b=10;

\*\*ppiresult\_c=15;

voutput(ppiresult\_a, ppiresult\_b, ppiresult\_c);

pinewblock=(int \*)malloc(IMAXELEMENTS\*sizeof(int));

\*pinewblock=\*\*ppiresult\_a;

\*(pinewblock+1)=\*\*ppiresult\_b;

\*(pinewblock+2)=\*\*ppiresult\_c;

free(pioldblock);

vassign(pivirtual\_array, pinewblock);

voutput(ppiresult\_a, ppiresult\_b, ppiresult\_c);

StopWait(); /\* Wait a little \*/

return(0);

}

void vassign(int \*pivirtual\_array[], int \*pinewblock)

{

pivirtual\_array[0]=pinewblock;

pivirtual\_array[1]=pinewblock+1;

pivirtual\_array[2]=pinewblock+2;

}

void voutput(int \*\*ppiresult\_a, int \*\*ppiresult\_b, int \*\*ppiresult\_c)

{

printf("%d\n",\*\*ppiresult\_a);

printf("%d\n",\*\*ppiresult\_b);

printf("%d\n",\*\*ppiresult\_c);

}

Эта программа иллюстрирует концепцию переменной-указателя (ppiresult\_a, ppiresult\_b, ppiresult\_c), ссылающихся на постоянный адрес (&pivirtual\_array[0], &pivirtual\_array[1], &pivirtual\_array[2]), содержащий ссылочный адрес на ячейку, содержимое которой может динамически изменяться.

Взгляните на объявления данных в main(). Переменные ppiresult\_a, ppiresult\_b, ppiresult\_c описаны как указатели на указатели, ссылающиеся на целые числа. Обсудим это подробно на основе разных синтаксических конструкций:

ppiresult\_a

\*ppiresult\_a

\*\*ppiresult\_a

Первая запись относится к адресу, хранящемуся в указателе ppiresult\_a. Вторая запись относится к адресу, на который указывает адрес, хранящийся в ppiresult\_a. Последняя запись относится к целому числу, на которое указывает ссылочный адрес, на который ссылается указатель ppiresult\_a. He продолжайте чтение, пока не разберетесь с этими тремя уровнями косвенности.

Все три переменные ppiresult\_a, ppiresult\_b и ppiresul\_c описаны как указатели на указатели, ссылающиеся на целые числа, то есть они имеют тип int \*\*. Переменная pivirtual\_array описана как массив указателей на целые числа (тип int \*), имеющий длину IMAXELEMENTS. Последние две переменные, pinewblock и pioldblock, аналогичны переменной pivirtual\_array, за исключением того, что они являются отдельными переменными, указывающими на целое число (тип int \*). На рис. 13.17 показано положение этих семи переменных в памяти и как, в частности, выглядят переменные ppiresult\_a, ppiresult\_b и ppiresult\_c после того, как им присвоены адреса соответствующих элементов массива pivirtualarray.

10_17

Рис. 13.17. Вид переменных после того, как ppiresult\_a, ppiresult\_b и ppiresult\_c получили начальные значения адресов

Именно в массиве pivirtualarray будут храниться адреса динамически изменяемых ячеек памяти. На практике нечто похожее происходит в истинно многозадачных системах. Прикладная программа считает, что она работает с реальным физическим адресом переменной, хранящейся в памяти, в то время, как на самом деле она обращается к фиксированному адресу массива указателей, которые в свою очередь указывают на текущий физический адрес объекта данных в памяти. Когда многозадачной среде нужно освободить память путем перемещения объектов данных прикладной программы, она просто изменяет местоположение этих объектов и обновляет массив указателей. Однако, переменные в программе продолжают ссылаться на тот же физический адрес, хотя это не физический адрес данных, а адрес массива указателей.

Чтобы понять, как работает эта конструкция, обратите особое внимание на тот факт, что физические адреса, хранящиеся в указателях ppiresult\_a, ppiresult\_b и ppiresult\_c, после инициализации указателей нигде не меняются.

На рис. 13.18 показано, что происходит с переменными после того, как создан динамический массив pinewblock и переменной pioldblock присвоен адрес нового массива. Обратите особое внимание на то, как физические адреса отдельных элементов массива pinewblock присвоены соответствующим элементам pivirtual\_array.

10_18

Рис.13.18. Динамическое выделение блока памяти

Все присваивания указателей выполняются при помощи функции vassign(), которой передается массив pivirtual\_array (вызов по значению) и адрес только что выделенного блока динамической памяти, хранящийся в переменной pinewblock. Функция присваивает адреса динамически выделенной памяти каждому элементу массива pivirtual\_array. Поскольку массив передавался по значению, изменения отражены в main().

Если теперь при помощи отладчика напечатать значение ppiresult\_a, то вы увидите АСС8 (адрес первого элемента pivirtual\_array), а значение \*ppiresult\_a будет равно 1630 (то есть содержимое адреса, на который происходит ссылка). Дамп памяти для двух других указателей, ppiresult\_b и ppiresult\_c, выглядит аналогично.

На рис. 13.19 показано, как ячейкам физической памяти присваиваются три целочисленных значения. Обратите внимание на синтаксис соответствующих операторов:

\*\*ppiresult\_a=7;

\*\*ppiresult\_b=10;

\*\*ppiresult\_c=15;

10_19

Рис.13.19. Заполнение данными блока памяти

Теперь, после вызова функции voutput(), программа печатает значения 7, 10 и 15. Заметьте, что согласно описанию функция должна получать три переменных типа int \*\*, и что в списке действительных параметров перед именами переменных не нужно ставить символ двойной операции косвенного обращения \*\*, поскольку этот тип переменных указан при объявлении.

Как показано на рис. 13.20, ситуация стала очень интересной. При помощи функции malloc() выделен новый блок динамической памяти, при этом его физический адрес записан в указатель pinewblock. Указатель pioldblock по-прежнему ссылается на ранее выделенный блок динамической памяти. Если следовать приблизительной аналогии с многозадачной средой, то этот рисунок мог бы проиллюстрировать намерение операционной системы физически переместить ячейки памяти, хранящие объекты данных.

10_20

Рис. 13.20. Динамическое выделение и заполнение данными второго блока памяти

Также на рис. 10.20 показано, что сами данные скопированы в новые ячейки памяти. Программа это выполняет в трех строках кода:

\*pinewblock=\*\*ppiresult\_a;

(pmewblock+1)=\*\*ppiresult\_b;

(pmewblock+2)=\*\*ppiresult\_c;

Поскольку указатель pinewblock хранит адрес первого элемента динамически выделенного блока, по отношению к адресу применяется операция разыменования (\*), что указывает на саму ячейку памяти, в которую записывается 7. При помощи простых арифметических операций и инкрементирования указателя выполняется обращение к двум другим ячейкам памяти.

10_21

Рис.13.21. Результат обновления массива pivirtual\_array другими физическими адресами

Для того чтобы адрес в указателе инкрементировался раньше, чем применяется операция разыменования (\*), необходимы круглые скобки.

На рис. 13.21 показан результат выполнения функций free() и vassign(), последняя связывает новые физические адреса блока динамически выделенной памяти с указателями, хранящимися в pivirtual\_array.

Самое главное, на что следует обратить внимание на этом последнем рисунке, это то, что реальные физические адреса указателей ppiresult\_a, ppiresult\_b и ppiresult\_c не изменились. Следовательно, при распечатке значений, на которые ссылаются \*\*ppiresult\_a и другие указатели, вы опять увидите значения 7, 10 и 15, хотя их физическое положение в памяти изменилось.

### Массивы указателей на строки.

Одним из самых простых способов определения массива строк является описание массива указателей на строки. Это намного проще, чем описывать двухмерный массив символов. В следующей программе для обработки трех сообщений об ошибках функции используется массив указателей на строки:

/\*10AOFPTR.C

Программа на С, демонстрирующая описание и использование массивов

указателей\*/

#include "stdafx.h"

#include "E:\LECTURE\AlgorithmProgramming 02\Universal\_HederFile.h"

#define INUMBER\_OF\_ERRORS 3

char \*pszarray[INUMBER\_OF\_ERRORS];

{

"\nFile not available.\n", /\* Файл отсутствует \*/

"\nNot an alpha character.\n", /\* He буква \*/

"\nValue not between 1 and 10.\n" /\*He значение от 1 до 10 \*/

};

FILE \*fopen\_a\_file(char \*psz);

char cget\_a\_char(void);

int iget\_an\_integer(void);

FILE \*pfa\_file;

void StopWait(void);

void main()

{

char cvalue;

int ivalue;

fopen\_a\_file("input.dat");

cvalue=cget\_a\_char();

ivalue=iget\_an\_integer();

StopWait(); /\* Wait a little \*/

}

FILE \*fopen\_a\_file(char \*psz)

{

const ifopen\_a\_file\_error=0;

pfa\_file=fopen(psz,"r");

if(!pfa\_file)

printf("%s",pszarray[ifopen\_a\_file\_error]);

return(pfa\_file);

}

char cget\_a\_char(void)

{

char cvalue;

const icget\_a\_char\_error=1;

printf("\nEnter a character: "); /\* Введите символ \*/

scanf("%c", &cvalue);

if(!isalpha(cvalue))

printf("%s",pszarray[icget\_a\_char\_error]);

return(cvalue);

}

int iget\_an\_integer(void)

{

int ivalue;

const iiget\_an\_integer=2;

printf("\nEnter an integer between 1 and 10: "); /\*Введите целое \*/

scanf("%d", &ivalue);

if((ivalue < 1) || (ivalue > 10))

printf("%s", pszarray[iiget\_an\_integer]);

return(ivalue);

}

Массив pszarray инициализируется за пределами объявлений всех функций; это обеспечивает ему глобальное время жизни. В больших программах подобные массивы могут храниться в отдельном исходном файле, предназначенном для управления всеми сообщениями об ошибках. Обратите внимание на то, что в каждой из функций fopen\_a\_file(), cget\_a\_char() и iget\_an\_integer() определяется собственный постоянный индекс массива. Такое сочетание массива сообщений об ошибках и уникального индекса функции обеспечивает модульное решение обработки исключительных ситуаций. Если в проекте нужно создать новую функцию, то к новому фрагменту кода привязывается свободное значение индекса и в pszarray добавляется одно сообщение об ошибке. Такой эффективный подход позволяет при создании каждого модуля программы быстро обновлять все приложение согласно конкретным требованиям к вводу/выводу и не беспокоиться о разработке специального механизма распознавания ошибок.

## Ссылочный тип в C++ (reference type).

**Ссылка особый тип данных, являющийся скрытой формой указателя, который при использовании автоматически разыменовывается. Иными словами, он может использоваться просто как другое имя или псевдоним объекта. При объявлении ссылки перед ее именем ставится знак амперсанда, а сама она должна быть тут же проинициализирована именем того объекта, на который ссылается:**

тип &имя\_ссылки = переменной;

**Тип объекта, на который указывается ссылка, может быть любым. Объявление неинициализированной ссылки вызовет сообщение компилятора (кроме ситуации, когда ссылка объявляется как extern).** Рассмотрим пример объявления ссылок:

chаr Letter\_A = 'А';

chаr &ref = Letter\_A;

Здесь объявляется и инициализируется символьная переменная Letter\_A и ссылка на нее ref.

Любое изменение значения ссылки повлечет за собой изменение того объекта, на который данная ссылка указывает:

int i=0;

int &ref =i;.

ref += 10; // то же, что i+=10;

После выполнения приведенного фрагмента значение обеих переменных i и ref будет равно 10.

Использование ссылок не связано с дополнительными затратами.

**Следует отметить, что ссылки нельзя переназначать, инициализировав ссылку однажды адресом некоторой переменной, любое действие со ссылкой сказывается на самом объекте. Попытка переназначить имеющуюся ссылку какой-либо другой переменной приведет к присвоению оригиналу объекта значения второй переменной**:

char letA = 'A';

char &refA = letA;

char B\_Letter = 'B';

refA = B\_Letter;

//то есть letA = B\_Letter

**Кроме того, следует учесть, что ссылаться можно только на сам объект. Нельзя объявить ссылку на тип объекта.** Ниже приводится корректный вариант объявления.

bool Flag = true;

bool &ref << Flag; // а не &ref >> bool

**Еще одно ограничение, налагаемое на ссылки, заключается в том, что они не могут указывать на нулевой объект (принимающий значение NULL).** Таким образом, если есть вероятность того, что объект в результате работы приложения станет нулевым, от ссылки следует отказаться в пользу применения указателя.

В языке C++ имеется способ вызова по ссылке, который использовать еще проще, чем указатели. Для начала рассмотрим использование ссылочных переменных в C++. Так же, как и в С, в языке C++ можно объявить обычные переменные или переменные-указатели. В первом случае для данных действительно выделяется память; во втором случае память резервируется для адреса объекта, который будет создан в другое время. В C++ имеется третий тип объявлений — ссылки. Так же, как указатель, ссылочная переменная указывает на положение другой переменной, однако, так же как и в случае обычной переменной, для ссылки не требуется специальной операции разыменования. Синтаксис ссылочной переменной понятен:

int iresult\_a=5;

int& riresult\_a=iresult\_a; // правильно

int& riresult\_b; // неправильно: нет начального значения

В этом примере определяется ссылочная переменная riresult\_a и присваивается существующей переменной iresult\_a. После этого адресуемое значение имеет два имени — iresult\_a и riresul\_a. Поскольку обе переменные указывают на одну и ту же ячейку памяти, они представляют собой, по сути, одну переменную. Любое присваивание, сделанное по отношению к riresult\_a, отражается на iresult\_a; справедливо и обратное утверждение. Следовательно, при помощи ссылки можно создать нечто подобное псевдониму (alias) переменной.

Ссылки имеют ограничение, позволяющее отличать их от указателей, которые ведут себя очень похоже. **Значение ссылочного типа должно быть задано при объявлении и не может меняться в процессе выполнения программы. После инициализации типа при объявлении, ссылка всегда указывает на одну и ту же ячейку памяти. Следовательно, при любом присваивании значения ссылочной переменной изменяются только данные в памяти, а не адрес самой переменной. Другими словами, ссылку можно считать указателем на ячейку памяти с постоянным адресом.**

Если, к примеру, использовать ранее сделанные объявления, то при помощи следующего оператора можно удвоить содержимое iresult\_a, умножив 5 на 2:

riresult\_a \*= 2;

Далее переменной icopy\_value (предполагая, что она имеет тип int) присваивается значение, связанное с riresult\_v.

icopy\_value = riresult\_a;

Следующий оператор также допустим при использовании ссылочных типов:

int \*piresult\_a = &riresult\_a;

Этот оператор присваивает адрес riresult\_a переменной piresult\_a, имеющей тип int \*.

В первую очередь ссылки используются в качестве аргументов функций и их возвращаемых значений, в особенности, если речь идет о классах, описываемых пользователем.

### Адрес в качестве возвращаемого значения функции.

Когда функция возвращает адрес либо в виде переменной-указателя, либо в виде ссылки, пользователь получает некоторый адрес в памяти. Он может считать значение, находящееся по этому адресу, и, если тип указателя не объявлен как const, всегда может что-нибудь туда записать. Если функция возвращает адрес, то пользователь получает возможность читать и — в случае указателей, имеющих тип, отличный от const — обновлять локальные данные. Это важно для проектирования приложений. Сможете ли вы описать работу следующей программы:

// 10REFVAR.CPP

// Программа на C++, показывающая что НЕ НУЖНО делать с адресными

// переменными

#include "stdafx.h"

#include "E:\LECTURE\AlgorithmProgramming 02\Universal\_HederFile.h"

int \*ifirst\_function(void);

int \*isecond\_function(void);

void StopWait(void);

void main ()

{

int \*pi=ifirst\_function();

cout << "Correct value? " << \*pi << "\n"; // Правильное значение?

isecond\_function();

cout << "Correct value? " << \*pi << "\n"; // Правильное значение?

StopWait(); /\* Wait a little \*/

}

int \*ifirst\_function(void)

{

int ilocal\_to\_first=11;

return &ilocal\_to\_first;

}

int \*isecond\_function(void)

{

int ilocal\_to\_second=44;

return &ilocal\_to\_second;

}



### Передача параметров по ссылке и по значению.

Параметры в функцию могут передаваться одним из следующих способов:

■ по значению;

■ по ссылке.

При передаче аргументов по значению компилятор создает временную копию объекта, который должен быть передан, и размешает ее в области стековой памяти, предназначенной для хpaнения локальных объектов. Вызываемая функция оперирует именно с этой копией, не оказывая влияния на оригинал объекта. Прототипы функций, принимающих аргументы по значению, предусматривают в качестве параметров указание типа объекта, а не его адреса. Например, функция

int GetMaх (int, int);

принимает два целочисленных аргумента по значению. Если же необходимо, чтобы функция модифицировала оригинал объекта, используется передача параметров по ссылке. При этом в функцию передается не сам объект, а только его адрес. Таким образом, все модификации в теле функции, переданных ей по ссылке аргументов воздействуют на объект. Принимая во внимание тот факт, что функция может возвращать лишь единственное значение, использование передачи адреса объекта оказывается весьма эффективным способом работы с большим числом данных. Кроме того, так как передается адрес, а не сам объест, существенно экономится стековая память.

В C++ передача по ссылке может осуществляться двумя способами:

■ используя непосредственно ссылки;

■ с помощью указателей.

Синтаксис передачи с использованием ссылок подразумевает применение в качестве аргумента ссылки на тип объекта. Например, функция

double Glue(long& var1, int& var2);

получает две ссылки на переменные типа long и int. При передаче в функцию параметра-ссылки компилятор автоматически передает в функцию адрес переменной, указанной в качестве аргумента. Ставить знак амперсанда перед аргументом в вызове функции не нужно. Например, для предыдущей функции вызов с передачей параметров по ссылке выглядит следующим образом:

Glue(varl, var2);

Пример прототипа функции при передаче параметров через указатель приведен ниже:

void SetNumber(int\*, long\*);

Кроме того, функции могут возвращать не только значение некоторой переменной, но и указатель или ссылку на него. Например, функции, прототип которых:

\*int Count(int);

&int Increase();

возвращают указатель и ссылку соответственно на целочисленную переменную типа int. Следует иметь в виду, что возвращение ссылки или указателя из функции может привести к проблемам, если переменная, на которую делается ссылка, вышла из области видимости. Например,

int& func()

{

int x;

return x;

}

В этом случае попытка возвратить ссылку на локальную переменную х приведет к ошибке, которая, к сожалению, выяснится только в ходе выполнения программы.

Эффективность передачи адреса объекта вместо самой переменной ощутима и в скорости работы, особенно, если используются большие объекты, в частности массивы (будут рассмотрены позже).

Если требуется в функцию передать довольно большой объект, однако его модификация не предусматривается, на практике используется передача константного указателя. Данный тип вызова предполагает использование ключевого слова const, например, функция

const int\* FName(int\* const Number)

принимает и возвращает указатель на константный объект типа int.. Любая попытка модифицировать такой объект в пределах тела вызываемой функции вызовет сообщение компилятора об ошибке. Рассмотрим пример, иллюстрирующий использование константных указателей.

#include <iostream>

using namespace std;

int\* const call(int\* const);

int main()

{

setlocale(LC\_ALL, "Rus");

int X = 13;

int\* pX = &X;

call(pX);

getchar(); getchar();

return 0;

}

int\* const call(int\* const x)

{

cout << \*x;

//\*x++; // нельзя модифицировать объект!

return x;

}



Вместо приведенного выше синтаксиса константного указателя в качестве альтернативы при передаче параметров можно использовать константные ссылки, например:

const int& FName (const int& Number)

имеющие тот же смысл, что и константные указатели.

#include <iostream>

using namespace std;

const int& call(const int& x)

{

cout << x;

//x++; // нельзя модифицировать объект!

return x;

}

int main()

{

setlocale(LC\_ALL, "Rus");

int X = 13;

int& rX = X;

call(rX);

getchar(); getchar();

return 0;

}



### Использование встроенного отладчика.

Чтобы увидеть работу этой программы на C++ в реальных условиях, можно воспользоваться встроенным отладчиком. Проследим при помощи окна Trace изменения переменной pi.

Что происходит? При вызове функции ifirst\_function() для переменной ilocal\_to\_first в стеке выделяется локальная область памяти, в которую записывается число 11. После этого функция ifirst\_function() возвращает адрес этой локальной переменной (очень плохая новость!). Во второй строке функции main() вызывается функция isecond\_function(), которая в свою очередь выделяет локальную область памяти для переменной ilocal\_to\_second и записывает туда число 44. Почему же оператор printf печатает значение 44, хотя ему при вызове функции ifirst\_function() был передан адрес переменной ilocal\_to\_first?

Фактически происходит следующее. Когда при вызове функции ifirst\_function() адрес временной локальной переменной ilocal\_to\_first был присвоен переменной pi, то этот адрес был сохранен, несмотря на завершение области действия ilocal\_to\_first. При вызове функции isecond\_function(), ей также понадобилась локальная память. Поскольку переменной ilocal\_to\_first не стало, переменная ilocal\_to\_second получила ту же локальную .область, что и ее предшественница. Так как pi указывает на ту же занятую ячейку памяти, то становится понятным, почему при печати адресуемой ячейки появляется число 44. Следует быть чрезвычайно внимательным и стараться не возвращать адреса локальных переменных.

### Использование ссылочного типа.

Подводя черту, можно выделить четыре основных причины использования ссылок C++:

* Ссылки упрощают восприятие программы, поскольку можно игнорировать детали процесса передачи параметра.
* Ссылки передают ответственность за передачу аргумента тому программисту, который пишет функцию, а не тому, который ею пользуется.
* Ссылки являются необходимой составной частью перегрузки операций.
* Ссылки используются при передаче функциям классов, при этом конструкторы и деструкторы не вызываются.

Пример для указателя и ссылки

#include <iostream>

using namespace std;

int main()

{

int i=0;

int &ref=i;

cout << "&i=" << &i << " &ref=" << &ref << endl;

cout << "i=" << i << " ref=" << ref << endl;

ref += 10; // то же, что i+=10;

cout << "i=" << i << " ref=" << ref << endl;

cout << endl;

int j=0;

int\* pj =&j;

cout << "&j=" << &j << " &pj=" << &pj << endl;

cout << "j=" << j << " \*pj=" << \*pj << endl;

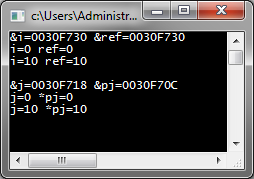
\*pj=\*pj+10; // то же, что j+=10;

cout << "j=" << j << " \*pj=" << \*pj << endl;

cin.get();

return 0;

}



## Использование указателей и ссылок с ключевым словом const

Некоторые конструкции языка C++ являются источником путаницы. Одной из таких конструкций является использование ключевого слова const с указателями и ссылками. Следующие примеры помогут вам прояснить ситуацию.

// Объявление данных

int number;

const int count=0;

// Указатель является константой

// Указатель указывает на константу

// (указываемое Значение есть const)

const int\* n2=&count;

// И указатель, и указываемое значение

// являются константами

const int\* const n3=&count;

// Указатели на строки

// строка является константной

const char\* strl="text";

// Указатель на строку является константой

char\* const str2="text";

// Указатель и сама строка - константы

const char\* const str3="text";

/\* Массивы указателей на символы \*/

// Символы являются константами

const char\* textl[]={"lne1","lnе2","lnеЗ"};

// Указатели являются константами

char\* const text2[]={"lne1","lnе2","lnеЗ"};

// Указатели и символы являются константами

const char\* const text3={"st1","st2","st3"};

Указатели сложно начинать осваивать, но необходимость их применения очень сильно требуют понимания указателей. Одна из запутанных тем с указателями — это тема указателей на константу и константных указателей. Такие похожие названия и совершенно разный смысл. Новички часто путают или смешивают одно с другим, но давайте разберемся что к чему.

Посмотрим на короткий пример с использованием Константного указателя и Указателя на константу

#include <iostream>

using namespace std;

int main()

{

int a = 100;//два обычных объекта типа int

int b = 222;

int \*const P2 = &a; //Константный указатель

\*P2 = 987; //Менять значение разрешено

//P2=&b; //Но изменять адрес не разрешается

const int \*P1 = &a; //Указатель на константу

//\*P1=110; //Менять значение нельзя

P1 = &b; //Но менять адрес разрешено

const int \*const P3 = &a;//Константный указатель на константу

//\*P3=155; //Изменять нельзя ни значение

//P3=&b; //Ни адрес к которому такой указатель привязан

cin.get();

return 0;

}

Чтобы не путаться где что можно и где тут чего нельзя достаточно вспомнить определение указателя. Так как указателем называется адрес памяти, то константный указатель = неизменяемый адрес памяти.  
Другое дело если мы не хотим изменять то значение, на которое указывает указатель, но предполагаем, что указатель будет указывать на разные объекты, каждый из которых изменению не подлежит. Таким образом у нас получается изменяемый указатель на неизменяемый объект или просто указатель на константу.

В описанных случаях есть что-то такое, что разрешено и чего не разрешено изменять, но можно запретить любые изменения, запретив менять и адрес и значение по этому адресу. Для такой жесткой привязки используется константный указатель на константу. Это как бы объединение ограничений двух первых вариантов друг с другом.

Каждый из трех описанных вариантов бывает нужен. Сам пример ничего на экран не выводит, так как его задача показать смысл существования таких понятий и помочь разобраться в столь запутанном материале.

## Интеллектуальные указатели (современный C++)

Интеллектуальные указатели определены в пространстве имен std в файле заголовка [<memory>](https://msdn.microsoft.com/ru-ru/library/k11k2x83.aspx). Они чрезвычайно важны для идиомы программирования [**RAII**](https://msdn.microsoft.com/ru-ru/library/hh438480.aspx) **или *Resource Acquisition Is Initialialization — получение ресурса является инициализацией***. Главная задача этой идиомы — обеспечить, чтобы одновременно с получением ресурса производилась инициализация объекта, чтобы все ресурсы для объекта создавались и подготавливались в одной строке кода. На практике основным принципом RAII является предоставление владения любым ресурсом в куче (например, динамически выделенной памятью или дескрипторами системных объектов) объекту, выделенному стеком, деструктор которого содержит код для удаления или освобождения ресурса, а также весь связанный код очистки.

В большинстве случаев при инициализации необработанного указателя или дескриптора ресурса для указания на фактический ресурс следует сразу же передать указатель в интеллектуальный указатель. В современном C++ необработанные указатели используются только в небольших блоках кода с ограниченной областью, циклах или вспомогательных функциях, когда важна производительность и вероятность проблем с владением низкая.

### Интеллектуальные указатели стандартной библиотеки C++

**POCO** (или C++ Portable Components) — это коллекция [библиотек](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) [классов](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) с [открытым исходным кодом](https://ru.wikipedia.org/wiki/%D0%9E%D1%82%D0%BA%D1%80%D1%8B%D1%82%D1%8B%D0%B9_%D0%B8%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4), которая упрощает и ускоряет разработку сетевых [мультиплатформенных](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F) приложений на C++. Библиотеки хорошо интегрируются со [Стандартной библиотекой шаблонов](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%BD%D0%B0%D1%8F_%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D0%BE%D0%B2) (STL) и органично заполняют множество функциональных пробелов в ней. Модульная структура и эффективная реализация делает POCO идеальным кандидатом для использования в разработке для [встраиваемых устройств](https://ru.wikipedia.org/wiki/%D0%92%D1%81%D1%82%D1%80%D0%B0%D0%B8%D0%B2%D0%B0%D0%B5%D0%BC%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) ([прошивки](https://ru.wikipedia.org/wiki/%D0%9C%D0%B8%D0%BA%D1%80%D0%BE%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0) и прочее), область, в которой C++ становится все более и более популярным, так как подходит как для низкоуровневой ([устройства ввода-вывода](https://ru.wikipedia.org/wiki/%D0%A3%D1%81%D1%82%D1%80%D0%BE%D0%B9%D1%81%D1%82%D0%B2%D0%BE_%D0%B2%D0%B2%D0%BE%D0%B4%D0%B0-%D0%B2%D1%8B%D0%B2%D0%BE%D0%B4%D0%B0), [обработчики прерываний](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D1%87%D0%B8%D0%BA_%D0%BF%D1%80%D0%B5%D1%80%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) и прочее), так и для [высокоуровневой](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%81%D0%BE%D0%BA%D0%BE%D1%83%D1%80%D0%BE%D0%B2%D0%BD%D0%B5%D0%B2%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) объектно-ориентированной разработки. Естественно, POCO подходит и для разработки [корпоративных приложений](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D1%80%D0%BF%D0%BE%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0).

Библиотеки POCO освобождают разработчика от необходимости повторной реализации типовых задач, которая могла бы повлечь опечатки, неточную реализацию спецификаций и неверную обработку ошибок.

Используйте эти интеллектуальные указатели как основной вариант для инкапсуляции указателей на простые старые объекты C++ (POCO).

**unique\_ptr**

Обеспечивает, чтобы у базового указателя был только один владелец. Используйте как вариант по умолчанию для POCO, кроме случая, когда вы точно знаете, что требуется shared\_ptr. Может быть передан новому владельцу, но не может быть скопирован или сделан общим. Заменяет auto\_ptr, использовать который не рекомендуется. Сравните с boost::scoped\_ptr. unique\_ptr мал и эффективен; размер — один указатель, и он поддерживает ссылки rvalue для быстрой вставки и извлечения из коллекций STL. Файл заголовка: <memory>. Дополнительные сведения см. в разделах [Практическое руководство. Создание и использование экземпляров unique\_ptr](https://msdn.microsoft.com/ru-ru/library/hh279676.aspx) и [Класс unique\_ptr](https://msdn.microsoft.com/ru-ru/library/ee410601.aspx).

**shared\_ptr**

Интеллектуальный указатель с подсчитанными ссылками. Используйте, когда необходимо присвоить один необработанный указатель нескольким владельцам, например, когда копия указателя возвращается из контейнера, но требуется сохранить оригинал. Необработанный указатель не будет удален до тех пор, пока все владельцы shared\_ptr не выйдут из области или не откажутся от владения. Размер — 2 указателя; один — для объекта и второй — для блока общего элемента управления, который содержит счетчик ссылок. Файл заголовка: <memory>. Дополнительные сведения см. в разделах [Практическое руководство. Создание и использование экземпляров shared\_ptr](https://msdn.microsoft.com/ru-ru/library/hh279669.aspx) и [Класс shared\_ptr](https://msdn.microsoft.com/ru-ru/library/bb982026.aspx).

**weak\_ptr**

Интеллектуальный указатель для особых случаев использования с shared\_ptr. weak\_ptr предоставляет доступ к объекту, который принадлежит одному или нескольким экземплярам shared\_ptr, но не участвует в подсчете ссылок. Используйте, когда требуется отслеживать объект, но не требуется, чтобы он оставался в активном состоянии. Требуется в некоторых случаях для разрыва циклических ссылок между экземплярами shared\_ptr. Файл заголовка: <memory>. Дополнительные сведения см. в разделах [Практическое руководство. Создание и использование экземпляров weak\_ptr](https://msdn.microsoft.com/ru-ru/library/hh279672.aspx) и [Класс weak\_ptr](https://msdn.microsoft.com/ru-ru/library/bb982126.aspx).

**Что вы должны были узнать.**

* Как объявить одномерный массив: long id\_ no [200];
* Как объявить двумерный массив: short chess[8] [8];
* Какие массивы можно инициализировать: внешние и статические.
* Как инициализировать массив: static int hats[3] = {10,20,15};
* Другой способ инициализации: static int caps[] = {3,56,2};
* Как получить адрес переменной: использовать операцию &
* Как получить значение, ссылаясь на указатель: использовать операцию \*
* Смысл имени массива: hats == &hats[0]
* Соответствие массива и указателя: если ptr = hats; то ptr + 2 == &hat[2]; и \*(ptr + 2) == hat[2];
* Пять операций, которые можно применять для переменных типа указатель: см. текст.
* Метод указателей для функций, работающих с массивами.

**Вопросы и ответы.**

**Вопросы.**

1. Что напечатается в результате работы этой программы?

#define PC(X, Y) printf("%c %c\n", X, Y)

char ref[] = { D, О, L, Т};

main ()

{

char \*ptr;

int index;

for (index = 0; ptr = ref; index < 4; index++, ptr++ )

PC(ref[index], \*ptr);

}

1. Почему в вопросе 1 массив ref описан до оператора main ()?
2. Определите значение \*ptr и \*(ptr + 2) в каждом случае:

int \*ptr;

static int boop[4] = {12, 21, 121, 212};

ptr = boop;

* 1. float \*ptr;

static float awk[2][2] = { { 1.0, 2.0}, {3.0, 4.0}};

ptr = awk[0];

* 1. int \*ptr;

static int jirb[4] = {10023, 7};

ptr = jirb;

* 1. int = \*ptr;

static int torf[2][2) = {12, 14, 16};

ptr = torf[0];

* 1. int \*ptr;

static int fort[2][2] = {{ 12}, {14, 16}};

ptr = fort[0];

1. Предположим, у нас есть описание static int grid[30][100];
   1. Выразите адрес grid [22] [56] иначе.
   2. Выразите адрес grid[22] [0] двумя способами.
   3. Выразите адрес grid[0][0] тремя способами.

**Ответы.**

D D

О О

L L

Т Т

1. По умолчанию такое положение ref относит его к классу памяти типа extern, a массивы этого класса памяти можно инициализировать.
2. 12 и 121
3. 1.0 и 3.0
4. 10023 и 0 (автоматическая инициализация нулем)
5. 12 и 16
6. 12 и 14 (именно 12 появляется в первой строке из-за скобок).
7. &grid[22][56]
8. &grid[22][0] и grid[22]
9. &grid[][] и grld[0] и grid

**Упражнение.**

Модифицируйте нашу метеорологическую программу таким образом, чтобы она выполняла вычисления, используя указатели вместо индексов. (Вы по-прежнему должны объявить и инициализировать массив.)