Лекция 7

Указатели

Указатели и адреса

Каждая переменная в программе – это объект, имеющий имя и значение. По имени можно обратиться к переменной и получить её значение.

Указатели - это переменные, значениями которых являются *адреса* других переменных (описывает расположение переменной в машинной памяти).

Синтаксис: при объявлении указателя используется символ звездочка (*):

ТИП* ИМЯ_УКАЗАТЕЛЯ;

Здесь ТИП это тип переменной, адрес которой может храниться в указателе.

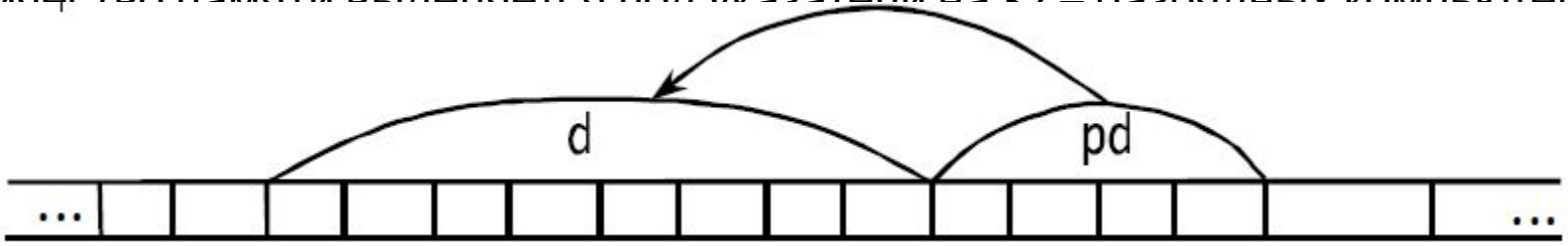
Пример

Унарный оператор & выдает адрес своего операнда.

```
double* pd;          // pd – указатель на переменную типа double
double d = 3.14159;  // d - переменная типа double
pd = &d;             // В указатель pd скопирован адрес переменной d
```

Указатель pd будет содержать адрес переменной d (говорят, что pd ***указывает*** или ***ссылается*** на d).

Переменная d типа double занимает 8 байтов памяти, а указатель pd – четыре. Такое количество памяти выделяется под указатели на 32 – разрядных компьютерах.



Доступ к объектам через указатели

Для доступа к объекту по его адресу используется унарный оператор раскрытия ссылки “*“ (оператор разадресации).

Пример

```
ini x=1,y=2;
int *p;
p=&x;          // теперь p указывает на x
y=*p;          // y теперь равен единице
*p=0;          // x теперь равен нулю
```

Приоритет операций.

Операции взятия адреса имеют более высокий приоритет, чем все арифметические операции, поэтому следующие выражения эквивалентны и обозначают увеличение переменно i на единицу (pi=&i).

$$i = i + 1 \approx *pi = *pi + 1 \approx ++(*pi) \approx (*pi)++$$

Замечание.

1) При использовании адресной операции “*” в арифметических выражениях следует остерегаться случайного сочетания знаков операции деления “/” и разыменования “*”, так как комбинация “/*”воспринимается компилятором как начало комментария.

Выражение i/*pi следует заменить так: i/(*pi).

2) К указателям применимы операции сравнения . Таким образом указатели можно использовать в отношениях, но сравнивать указатели допустимо только с

Программа «Работа с указателями»

```
#include <iostream>
#include <clocale>
#include <cstdlib>
using namespace std;
int main()
{    setlocale(LC_ALL, "Russian");
double* pd;                                    // Указатель на double
cout << "Адрес указателя pd: &pd = " << &pd << endl;
double d = 3.14159;                            // Переменная типа double
cout << "Адрес переменной d: &d = " << &d << endl;
pd = &d;                                       // В указатель pd скопирован адрес переменной d
cout << "Значение указателя pd: pd = " << pd << endl;
cout << "Значение переменной d: d = " << d << endl;
cout << "Значение переменной, на которую указывает pd: *pd = "<< *pd << endl;
*pd = 2.71828;                                 // Изменение переменной d через указатель pd  на нее
cout << "Значение переменной d: d = " << d << endl;
char* pc;                                      // Указатель на символ
int* pi;                                       // Указатель на целое
cout << "Размеры указателей:\n";
cout << "sizeof(char*) = " << sizeof(pc) << "\n"<< "sizeof(int*) = " << sizeof(pi) << "\n"<< "sizeof(double*) = "
<< sizeof(pd) << "\n";
system("pause");    return 0;    }
```

Указатели как аргументы функций

В C++ аргументы функции передаются по значению, то есть функция получает лишь локальные копии аргументов. Передав функции адреса переменных, можно изменить их значение в вызывающей программе. При этом аргументы должны быть декларированы как указатели.

Если указатель на переменную является аргументом функции, то внутри функции становится известен адрес этой внешней по отношению к функции переменной, что позволяет работать с этой переменной, в том числе изменять ее значение.

Программа «Обмен значениями»

```
#include <iostream>
#include <clocale>
#include <cstdlib>
using namespace std;
void swap1 (int a, int b)
{
	int tmp=a;
	a=b;
	b=tmp;
}
void swap2 (int* a, int* b)
{
	int tmp=*a;
	*a=*b;
	*b=tmp;
}
int main()
{
	setlocale(LC_ALL, "Russian");
	int a,b;
	cout<<"Введите 2 числа"<<endl;
	cin>>a>>b;
	swap1(a,b);
	cout<<"swap1: a="<<a<<", b="<<b<<endl;
	swap2(&a,&b);
	cout<<"swap2: a="<<a<<", b="<<b<<endl;
	system("pause");
	return 0;
}
```

Программа «Расчет треугольника»

Пусть требуется вычислить периметр и площадь треугольника по трем его сторонам ***a, b, c***. Напишем для этого функцию **triangle()**.

Так как треугольник существует не для любых значений длин сторон, функция должна как-то информировать об этом. Пусть она будет возвращать **true**, если для заданных длин сторон треугольник существует, и **false**, если не существует. Две остальные величины – периметр и площадь – будем возвращать из функции через аргументы, имеющее тип указателя.

Вычисления можно проводить по формулам:

Перимет $P = a + b + c$ Полупериметр $p = \frac{P}{2}$

Площад $S = \sqrt{p \cdot (p-a) \cdot (p-b) \cdot (p-c)}$

Неравенство треугольника: $a < b + c, b < a + c, c < a + b$

Программа «Расчет треугольника»

```
#include <iostream>
#include <cmath>
#include <clocale>
#include <cstdlib>
using namespace std;
// triangle: вычисление периметра и площади треугольника. Возвращает true, если треугольник существует и false
// если не существует.  a, b, c - стороны треугольника, p_perim - указатель на переменную для периметра
// p_area - указатель на переменную для площади
bool triangle(double a, double b, double c, double* p_perim, double* p_area)
{    if(a > b + c || b > a + c || c > a + b)      // Проверка существования треугольника
          return false;                           // Треугольник не существует, выход из функции
     double p = (a + b + c) / 2.0;                // Полупериметр
    *p_perim = p * 2.0;                           // Периметр
    *p_area = sqrt(p * (p - a) * (p - b) * (p - c)); // Площадь
     return true;   }
int main()
{   setlocale(LC_ALL, "Russian");
double r, s, t, P, A; // Стороны треугольника, периметр и площадь
cout << "Введите три стороны треугольника: ";
cin >> r >> s >> t;
if( triangle(r, s, t, &P, &A) == false )
cout << "Такого треугольника не существует\n";
else
cout << "Периметр: " << P << ", площадь: " << A << "\n";
system("pause");   return 0;   }
```

Взаимодействие формальных параметров и фактических аргументов функции triangle()

При вызове функции triangle() формальные параметры a, b, c получают *значения* фактических аргументов r, s, t. Размеры прямоугольников p_perim, p_area в два раза меньше, чем размеры прямоугольников a, b, c, так как a, b, c имеют тип double размером 8 байт, а указатели p_perim, p_area имеют размер 4 байта. Формальные параметры p_perim, p_area получают значения *адресов* внешних переменных P и A.

Указатели и массивы

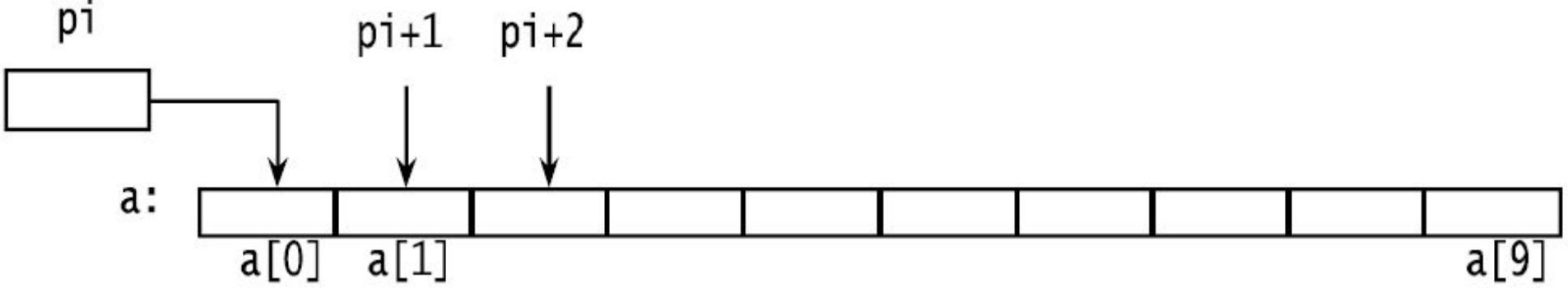
Задан целочисленный массив из 10 элементов , то есть блок из 10 расположенных последовательно переменных целого типа с именами a[0], a[1], …, a[9].

```
int a[10];
int *pi;        //  определен указатель на целое число
pi = &a[0];   // указатель pi будет содержать адрес первого элемента массива a
```

По определению, **pi + 1** указывает на следующий элемент массива, **pi + i** указывает на i-й элемент после pi, **pi - i** указывает на i-й элемент перед pi. Таким образом, имея указатель на начало массива, можно получить доступ к любому его элементу, например, ***pi** есть первый элемент массива, ***(pi + 1)** – второй и т.д. Присваивание

```
*(pi + 1) = 0;        // a[1] = 0
```

обнуляет второй элемент массива, номер которого равен 1



Указатели и массивы

По определению, ***имя массива*** имеет значение ***адреса первого элемента*** массива, соответственно имя массива имеет тип ***указателя на элемент массива***.

Пример: **a** имеет тип **int*** (указатель на целое). Доступ к **i** - му элементу массива можно получить, используя индексацию **a[i]** или выражение

***(a + i)**.

Указатель – это переменная, которой можно присваивать различные значения.

Пример:

```
pi = a + 1;    // Теперь pi указывает на второй элемент массива a.
```

Имя массива является ***константой***, так как содержит адрес конкретного участка памяти, и записи типа a = pi; a++ недопустимы.

Вывод: значение имени массива изменить нельзя, во всем остальном имя массива подобно указателю.

$$a \equiv \&a[0]$$

Пусть:

```
pi=a;
```

$$pi + i \approx a + i \approx \&a[i] \approx \&pi[i]$$

$$*(pi + i) \approx *(a + i) \approx a[i] \approx pi[i]$$

Эквивалентные выражения:

Различия между указателем и массивом

Указатель – это переменная, а имя массива это константа, равная адресу элемента с индексом 0.

```
int *pi, a[10];
pi=a;        // допустимо
pi++;        // допустимо
a=pi;        // не допустимо
a++;         // не допустимо
```

Это связано с различием в выделении памяти. Память под массив выделяется при его определении, расположение массива в памяти фиксировано. При определении указателя память отводится только для хранения адреса.

Адресная арифметика

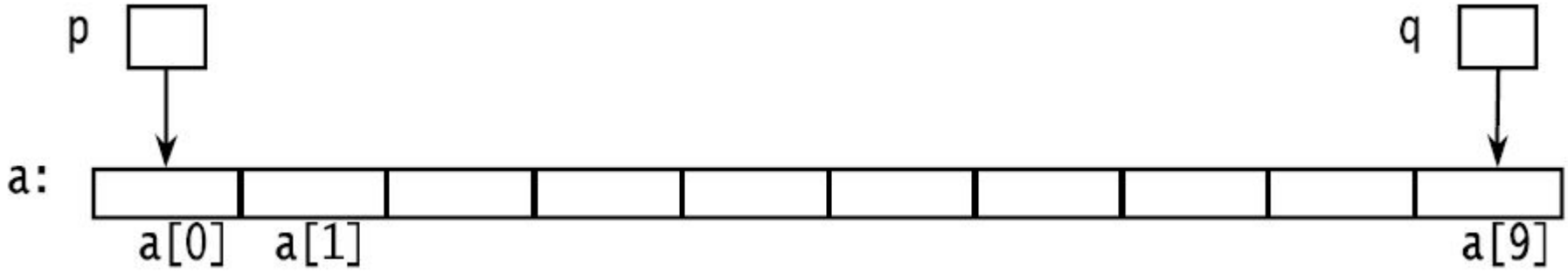
1) Указателям можно присваивать значение другого указателя такого же типа.

2) К указателям можно прибавлять и вычитать целые числа (сдвиг указателя).

Если **p** – указатель на некоторый элемент массива, то выражение **p++** или **++p** изменяет p так, чтобы он указывал на следующий элемент массива, а выражение **p--** или **--p** переводит указатель на предыдущий элемент массива. Выражение **p += i** изменяет **p** так, чтобы он указывал на **i** - й элемент, после того, на который он указывал ранее.

3) Из одного указателя можно вычесть значение другого указателя, если они ссылаются на элементы одного и того массива. Если **p** и **q** указывают на элементы одного и того же массива и **p < q**, то

с



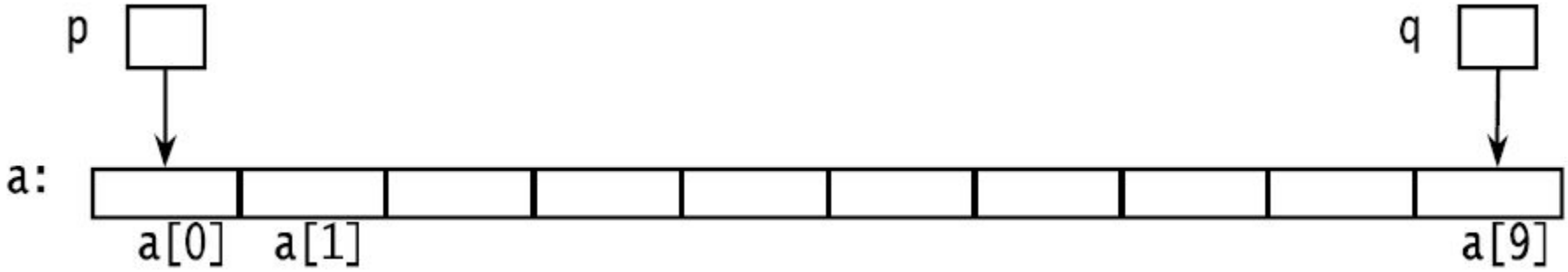
Адресная арифметика

4) Указателям можно присваивать значение, равно нулю, либо сравнивать на равенство с нулём (NULL).

Если значение указателя равно нулю, это трактуется так, что указатель никуда не указывает.

5) К указателям, ссылающимся на элементы одного и того же массива, можно применить операции сравнения (==, !=, <, <=, >, >=).

Выражение p < q истинно, если p указывает на более ранний элемент массива, чем q.



Программа «Массивы и указатели»

```
#include <iostream>
#include <locale>
using namespace std;
int main()
{  setlocale(LC_ALL, "Russian");
int a[10], *p, *q;                          // Массив и два указателя
for (int i = 0; i < 10; i++)
        a[i] = i;
p = &a[0];                                  // Указатель на первый элемент массива, можно p = a;
cout << "Значение имени массива a = " << a << endl<< "Значение указателя p = " << p << endl;
cout << "Элементы массива:\n"<< "Использование индексации:\n";
for (int i = 0; i < 10; i++)
        cout << a[i] << " ";
cout << "\nИспользование имени массива:\n";
for (int i = 0; i < 10; i++)
        cout << *(a + i) << " ";
cout << "\nИспользование указателя:\n";
for (int i = 0; i < 10; i++)
        cout << *(p + i) << " ";
q = &a[9];                                  // Указатель на последний элемент массива
cout << "\np = " << p << ", q = " << q << ", q - p = " << (q - p)<< "\nint(q) - int(p) = " << (int(q) - int(p));
++p; --q;                                   // Изменение указателей
cout << "\n++p = " << p << ", --q = " << q << ", q - p = " << q - p;
cin.get();    return 0;   }
```

Массивы как аргументы функции

Почему, когда массив является аргументом функции, он не копируется внутрь функции?

Массив в языке C++ является понятием низкого уровня. Во многих случаях массив теряет информацию о своём размере. При использовании массива в качестве аргументов функции передаётся только его базовый адрес:

$$f(char\ s[]) \approx f(char * s)$$

Пусть объявлена функция с аргументом – массивом:

```
void f(char s[]);
```

Так как имя массива – это указатель на первый элемент массива, то данное объявление эквивалентно такому:

```
void f(char* s);
```

То есть, внутри функции создается ***копия указателя*** на первый элемент массива и принцип передачи аргументов ***по значению*** остается в силе.

Элементы массива не копируются внутрь функции. Используя адрес первого элемента массива, можно внутри функции получить доступ к любому элементу массива и изменить его. Таким образом, *если*

Программа «Сортировка массива»

Сортировкой называется упорядочение массива по возрастанию или убыванию.

Реализуемые функции: функция заполнения массива; функции вывода массива на экран; функция сортировки массива.

Вспомогательные действия:

1) Размер массива задается константой **SIZEARR** достаточно большого размера, чтобы выделенной под массив памяти хватило в большинстве случаев использования программы.

2) При вводе недопустимого размера массива вызывается библиотечная функция **void exit(int k),** которая завершает работу программы и передает в вызывающую программу значение своего аргумента k. Эта функция объявлена в **stdlib.h**.

3) Массив x заполняется случайными числами. Случайные целые числа из диапазона от **0** до **RAND_MAX** генерируется функцией стандартной библиотеки **int rand()**, объявленной в **cstdlib**. Величина **RAND_MAX** (обычно 32767) определена в файле **cstdlib**.

Функция **void srand(unsigned int seed);** (объявлена в **cstdlib**) настраивает генератор случайных чисел, используя для формирования первого псевдослучайного числа параметр **seed**. Чтобы получать при каждом запуске программы разную последовательности случайных чисел, функцию **srand()** следует вызывать с разными аргументами. В программе функции **srand()**

Функция заполнения массива из n элементов

```
// get_array: заполняет массив y случайными числами
void get_array(int* y, int n)
{
srand((unsigned) time(NULL)); // Инициализация генератора случайных чисел
for(int i = 0; i < n; i++)
    y[i] = rand();          // rand() генерирует целое случайное число
}
```

Функция вывода массива на экран

```
// prn_array: вывод массива
void prn_array(int* y, int n)
{
for(int i = 0; i < n; i++)
    cout << y[i] << " ";
}
```

Функция сортировки массива методом пузырька

Алгоритм. Организуется проход по массиву, в котором сравниваются соседние элементы. Если предшествующий элемент оказывается больше следующего, они и меняются местами. В результате первого прохода наибольший элемент оказывается на своем, последнем месте (*«всплывает»*). Затем проход по массиву повторяется до предпоследнего элемента, затем до третьего с конца массива и т.д. В последнем проходе по массиву сравниваются только первый и второй элементы.

```
// bubble_sort: сортировка массива y методом пузырька
void bubble_sort(int * y, int n)
{     for(int i = n - 1; i > 0; i--)              // i задает верхнюю границу
            for(int j = 0; j < i; j++)   // Цикл сравнений соседних
элементов
                  if(y[j] > y[j + 1])             // Если нет порядка,
                     { int tmp = y[j];          // перестановка местами
                       y[j] = y[j + 1];          // соседних
                       y[j + 1] = tmp;  }       // элементов
```

Программа «Сортировка массива»

```
#include <iostream>
#include <cstdlib>                  // Для exit(), rand()
#include <time.h>                   // Для time()
using namespace std;
// Объявления функций
void get_array(int[], int n);                       // Заполнение массива из n элементов
void bubble_sort(int[], int n);                     // Сортировка массива методом пузырька
void prn_array(int[], int n);                       // Вывод массива
int main()
{   setlocale(LC_ALL, "Russian");
const int SIZEARR = 500;                            // Максимальный размер массива
int x[SIZEARR], n;                                  // Массив и размер массива
cout << "Введите размер массива < " << SIZEARR << ": ";
cin >> n;
if(n > SIZEARR || n <= 0){                          // Проверка размера
        cout << "Размер массива " << n<< " недопустим " << endl;
        system("pause");   exit(1);    }            // Завершение программы
get_array(x, n);                                    // Заполнение массива
cout << "Исходный массив: \n";
prn_array(x, n);                                    // Вывод исходного массива
bubble_sort(x, n);                                  // Сортировка
cout << "\nОтсортированный массив: \n";
prn_array(x, n);                                    // Вывод упорядоченного массива
cout << endl;   system("pause");    return 0;  }
```

Символьные указатели

Для работы со строками символов часто используются указатели на char.

```
char* pc;
```

Строковая константа, написанная в виде "Я строка", есть массив символов с нулевым символом '\0' на конце. Адрес начала массива, в котором расположена строковая константа, можно присвоить указателю:

```
pc = "Я строка";
```

Здесь копируется только ***адрес*** начала строки, сами символы строки не копируются.

Указатель на строку можно использовать там, где требуются строки.

```
cout << pc << " длиной " << strlen(pc) << " символов";
```

Будет напечатано: «Я строка длиной 8 символов».

Замечания:

1) При подсчете символов строки учитываются и пробелы, а завершающий символ '\0' не учитывается.

2) В отличие от указателей других типов, при выводе символьных указателей выводится ***не адрес***, хранящийся в указателе, ***а строка***

Программа «Длина строки»

```
#include <iostream>
#include <cstdlib>
#include <locale>
using namespace std;
int strlen1 (char*s)
    {
      int n;
      for (n=0; *s!='\0'; s++)
        n++;
      return n;
    }
int strlen2 (char*s)
    {
      char *t=s;
      while (*s!='\0')
        s++;
      return s-t;
    }
int main()
{
        setlocale(LC_ALL, "Russian");
        char s[30];
        cout<<"Введите строку"<<endl;
        cin.getline(s,30);
        cout<<"Длина строки: "<<strlen1(s)<<endl;
        cout<<"Введите строку"<<endl;
        cin.getline(s,30);
        cout<<"Длина строки: "<<strlen2(s)<<endl;
        system("pause");
        return 0;
}
```

Выделение и освобождение динамической памяти под переменную

Оператор **new** выделяет память под объект во время выполнения программы.

Пример:

```
double* pd;       // pd - указатель на double, его значение не определено
```

Инструкция:

```
pd = new double;
```

выделяет память под переменную типа double, адрес которой присваивается **pd**. После оператора **new** указывается ***тип*** создаваемого объекта. Теперь динамически созданную переменную можно использовать.

```
*pd = sqrt(3.0);          // Размещение в памяти значения
cout << *pd;              // Печать значения
```

Оператор **delete** освобождает память, выделенную ранее оператором **new**.

Пример:

```
delete pd;
```

Выделение и освобождение динамической памяти под массивы

Динамические массивы можно создавать оператором **new[]**.

Пример:

```
char* s = new char[80]; // Создается динамический массив из 80 символов
```

Для удаления динамических массивов служит оператор **delete[].**

Пример:

```
delete[] s;     // Освобождение памяти, на которую указывает s
```

Замечания:

1) При освобождении памяти, выделенной оператором **new**, операторы **delete** и **delete[]** должны иметь возможность определять размер удаляемого объекта. Это обеспечивается тем, что под динамический объект памяти выделяется больше, чем под статический, обычно на одно слово, в котором хранится размер объекта.

2) С помощью оператора **new[]** можно создавать массивы

Программа «Выделение и освобождение памяти»

```
#include <iostream>
#include <clocale>
#include <cstdlib>
#include <ctime>
#include <cmath>
using namespace std;
// get_array: заполнение массива y случайными числами
void get_array(int* y, int n) // n – размер массива
{ for (int i = 0; i < n; i++)
   y[i] = rand(); } // rand() генерирует целое случайное число
// prn_array: вывод массива
void prn_array(int* y, int n)
{   for (int i = 0; i < n; i++)
      cout << y[i] << " ";   }
int main()
{
setlocale(LC_ALL, "Russian");
double* pd = 0; // Указатель на double
pd = new double; // Выделение памяти
cout << "Адрес(pd) = " << pd << ", значение(*pd) = " << *pd;
*pd = sqrt(3.0); // Занесение в память значения
cout << "\nАдрес(pd) = " << pd << ", значение(*pd) = " << *pd;
delete pd; pd = 0; // Освобождение памяти
int *pi_1 = 0, *pi_2 = 0, *pi_3 = 0; // Указатели на целое
int size1, size2, size3; // Размеры динамических массивов
cout << "\nВведите размер массива 1: ";
cin >> size1;
pi_1 = new int[size1]; // Выделение памяти под массив 1
srand(time(0)); // Инициализация генератора случайных чисел
get_array(pi_1, size1); // Заполнение массива 1
cout << "Массив 1\n";
prn_array(pi_1, size1); // Вывод массива 1
cout << "\nВведите размер массива 2: ";
cin >> size2;
pi_2 = new int[size2]; // Выделение памяти под массив 2
get_array(pi_2, size2); // Заполнение массива 2
cout << "Массив 2\n";
prn_array(pi_2, size2); // Вывод массива 2
pi_3 = new int[size1 + size2]; // Память для составного массива
int k; // Индекс для массива 3
for (k = 0; k < size1; ++k) // Копируем массив 1 в составной
pi_3[k] = pi_1[k];
for (int i = 0; i < size2; ++i, ++k) //Копируем массив 2 в общий
pi_3[k] = pi_2[i];
cout << "\nМассив 1 + Массив 2:\n";
prn_array(pi_3, size1 + size2); // Вывод составного массива
cout << endl;
delete[] pi_1; pi_1 = 0; // Удаление
delete[] pi_2; pi_2 = 0; // динамических
delete[] pi_3; pi_3 = 0; // массивов
system("pause");
return 0;
}
```

Программа «Копия строки»

```
#include <iostream>
#include <clocale>
#include <cstdlib>
#include <string>
using namespace std;
char * copy (char *s)
 {
  char* s1=new char [strlen(s)+1]; // размерность массива trlen(s)+1, так как функция strlen
                        //не учитывает символ '\0' в длине
  strcpy (s1,s);
  return s1;                  // Возвращаеи адрес выделенного участка
 }
int main()
{  setlocale(LC_ALL, "Russian");
char s1[20],*s2;  // Массив под строку и указатель под дубликат
cout<<"Введите строку:"<<endl;
cin.getline(s1,20);
s2=copy(s1);
cout<<"Копия строки длины  "<<strlen(s2)<<endl<<s2<<endl;
system("pause");  return 0;  }
```

Утечка памяти

При работе с динамической памятью следует быть внимательным, чтобы не допустить ситуации, когда память выделяется динамически, но не освобождается, когда необходимость в ней отпадает, что приводит к уменьшению доступной свободной памяти. Это явление называют «утечка памяти».

Пример:

В программе в бесконечном цикле вызывается функция f(), которая при каждом вызове создает массив из 10 миллионов целых чисел, то есть запрашивает 4 107 байт памяти.

Замечание:

Следует знать, что захват одной программой большого объема памяти может замедлить выполнение других программ.

Утечка памяти

```
#include <iostream>
using namespace std;
void f()
{
// Создание массива из 10 миллионов целых
int* pi = new int[1024 * 1024 * 1024];
}
int main()
{
for(int i = 1; ; ++i){ // Бесконечный цикл
f();
cout << i << ' ';
}
}
```

Массивы указателей

Указатели, как и любые другие переменные, можно группировать в массивы.

Удобно, например, использовать массив символьных указателей при работе с несколькими строками, которые могут иметь различную длину.

Пример:

Программа предлагает ввести номер месяца и выводит его название. Для доступа к строкам с названиями месяцев используется массив символьных указателей.

Программа «Названия месяцев»

```
#include <iostream>
#include <clocale>
#include <cstdlib>
using namespace std;
int main()
{    setlocale(LC_ALL, "Russian");
char* pmn[] = { // Массив указателей
"Неверный номер месяца", "Январь", "Февраль", "Март",
"Апрель", "Май", "Июнь", "Июль", "Август",
"Сентябрь", "Октябрь", "Ноябрь", "Декабрь " };
int month; // Номер месяца
cout << "Введите номер месяца: ";
cin >> month;
if (0 < month && month <= 12)
cout << "Это " << pmn[month] << endl;
else
cout << pmn[0] << endl;
system("pause");
return 0;  }
```

Программа «Ввод и сортировка строк»

```
#include <iostream>
#include <clocale>
#include <cstdlib>
using namespace std;
int main()
{
	setlocale(LC_ALL, "Russian");
	char* pmn[100];    // Массив указателей на строки
	char tmp [100];    // текущая  строка
	int n=0;     // Количество строк
	while(!cin.eof())  // Пока не конец потока
	{
		cin.getline(tmp,100); // Считываем текущую строку
		pmn[n]=new char [strlen(tmp)+1]; // Выделяем память под новую строку
		strcpy(pmn[n],tmp); // Копирум строку в массив
		n++;  // Увеличиваем количество строк
	}
	for(int i=0;i<n-1; i++) //Сортировка
		for (int j=i+1;j<n;j++)
			if (strcmp(pmn[i],pmn[j])>0)
			{
				char *c=pmn[i];
				pmn[i]=pmn[j];
				pmn[j]=c;
			}
	for (int i=0;i<n;i++) // Вывод отсортированных строк
		cout<<pmn[i]<<endl;
	system("pause");
	return 0;
}
```

Программа «Суммирование матриц»

```
#include <iostream>
#include <clocale>
#include <cstdlib>
using namespace std;
// Создание и заполнение матрицы (Используя оператор [])
int** scanMatrix (int n, int m)
{	int **a=new int *[n];
	for (int i=0;i<n;i++)
		a[i]=new int[m];
	cout<<"Ведите элементы матрицы "<<n<<" на "<<m<<endl;
	for (int i=0;i<n;i++)
		for (int j=0; j<m; j++)
			cin>>a[i][j];
	return a;}
// Вывод матрицы (Используя указатели)
void printMatrix(int **a, int n, int m)
{	for (int i=0;i<n;i++)
	{
		for (int j=0; j<m; j++)
			cout<<*(*(a+i)+j)<<'\t';
		cout<<endl;
	}
}
// Удаление матрицы (Используя указатели)
void deleteMatrix(int **a, int n)
{	for (int i=0;i<n;i++)
		delete [] *(a+i);
	delete [] a;
}
// Сумма матриц
int** sumMatrix(int **a, int **b, int n, int m)
{
	int** c=new int *[n];
	for (int i=0;i<n;i++)
		*(c+i)=new int[m];
	for (int i=0;i<n;i++)
		for (int j=0; j<m; j++)
			c[i][j]=*(*(a+i)+j)+*(b[i]+j);
	return c;
}
int main()
{
	setlocale(LC_ALL, "Russian");
	int **a,**b,**c;
	int n,m;
	cout<< "Введите размерности матриц"<<endl;
	cin>>n>>m;
	a=scanMatrix(n,m);
	b=scanMatrix(n,m);
	c=sumMatrix(a, b, n, m);
	cout<<"Матрица A"<<endl;
	printMatrix(a, n, m);
	cout<<"Матрица B"<<endl;
	printMatrix(b, n, m);
	cout<<"Сумма матриц"<<endl;
	printMatrix(c, n, m);
	deleteMatrix(a, n);
	deleteMatrix(b, n);
	deleteMatrix(c, n);
	system("pause");
	return 0;
}
```