Лекция 5

Динамическое выделение памяти

Два основных способа хранения информации в оперативной памяти.

1) Использование глобальных и локальных переменных.

В случае глобальных переменных выделяемые под них поля памяти остаются неизменными на все время выполнения программы. Под локальные переменные программа отводит память из стекового пространства, требуется предварительное определение объема памяти, выделяемой для каждой ситуации.

2) Использование системы динамического распределения. Память распределяется из свободной области памяти по мере необходимости. Динамическое размещение удобно, когда неизвестно, сколько элементов данных будет обрабатываться.

Память, выделяемая в процессе выполнения программы, называется динамической. После выделения динамической памяти она сохраняется до ее явного освобождения, что может быть выполнено только с помощью специальной операции или библиотечной функции.

Если динамическая память не освобождена до окончания программы, то она освобождается автоматически при завершении программы. Но явное освобождение ставшей ненужной памяти является признаком хорошего стиля программирования.

В процессе выполнения программы участок динамической памяти доступен везде, где доступен указатель, адресующий этот участок.

Для работы с динамической памятью используют указатели. С их помощью осуществляется доступ к участкам динамической памяти, которые называются динамическими переменными.

Для хранения динамических переменных - специальная область памяти - "куча".

Динамические переменные создаются с помощью специальных функций и операций. Они существуют либо до конца работы программы, либо до тех пор, пока не будет освобождена выделенная под них память .

В C++ два способа работы с динамической памятью:

использование операций new и delete;

использование семейства функций malloc (calloc), free (унаследовано из C).

Таблица 4.1. Функции выделения и освобождения памяти

Функции выделения и освобождения памяти Заголовочные файлы malloc.h и stdlib.h стандартной библиотеки (файл malloc.h).	
Функция	Прототип и краткое описание
malloc	void * malloc (unsigned s); возвращает указатель на начало области (блока) динамической памяти длинной в s байт. При неудачном завершении возвращает значение NULL.
calloc	void * calloc (unsigned n, unsigned m); возвращает указатель на начало области (блока) обнуленной динамической памяти, выделенной для размещения n элементов по m байт каждый. При неудачном завершении возвращает значение NULL.

realloc	void * realloc (void * bl, unsigned ns); изменяет размер блока ранее выделенной динамической памяти до размера ns байт, bl – адрес начала изменяемого блока. Если bl равен NULL (память не выделялась), то функция выполняется как malloc.
free	void * free (void * bl); освобождает ранее выделенный участок (блок) динамической памяти, адрес первого байта которого равен значению bl.

Память по требованию. new, delete

Разработчик может хотеть иметь полный контроль, сколько памяти используется и когда именно она используется. Это особенно важно, когда вы заранее не знаете, каков размер обрабатываемых данных.

Для управления выделением и освобождением памяти C++ предоставляет два специализированных ключевых слова:

```
new

delete
```

Ключевое слово new используется для запроса на создание нового блока памяти.

Когда выделенная память больше не нужна и/или не используется, следует вернуть ее в ОС - ключевое слово delete.

Ключевое слово new

```
float *array = new float[20];

int count = new int;
```

• нужны точные спецификации относительно создаваемой сущности; это должно быть выражено в виде описания типа; если создаваемая сущность - одномерный массив (фактически вектор), размер массива также должен быть указан (как в первом примере).;

• new возвращает указатель типа, соответствующего вновь созданному объекту;

• вновь выделенная область памяти не заполнена (не инициирована), поэтому следует ожидать, что она будет содержать только мусор.

Примечание: массивы, с которыми мы имеем дело, используя ключевое слово new , не имеют ничего общего с этими объектами, воплощенными в жизнь как объекты типа vector<type_name>. Первые управляются низкоуровневым кодом, сгенерированным компилятором, вторые являются высокоуровневыми созданиями, управляемыми библиотечным кодом.

Память можно освободить:

```
delete [] array;

delete count;
```

• используем форму delete [], если хотим освободить память, выделенную для массива, в противном случае используем delete;

• можно освободить только весь выделенный блок, а не его часть;

• после выполнения функции delete все указатели, указывающие на данные внутри освобожденной области, становятся незаконными; попытка их использования может привести к аварийному завершению программы.

```
#include <iostream>
using namespace std;
int main()
{
     float *arr = new float[5];
     for (int i = 0; i < 5; i++)
     arr[i] = i * i;
     for (int i = 0; i < 5; i++)
     cout << arr[i] << endl;
     delete[] arr;
}
0
1
4
9
16
```

• объявляем переменную с именем arr , которая будет указывать на данные типа float (тип указателя - float *); этой переменной изначально не присваивается значение;

• используем ключевое слово new для выделения блока памяти, достаточного для хранения массива с плавающей точкой, состоящего из 5 элементов;

• используем только что выделенный массив (точнее, вектор), а затем освобождаем его с помощью ключевого слова delete

Пример программы сортировки.

Возможность выделить действительно необходимый объем памяти позволяет писать программы, которые могут адаптироваться к размеру обрабатываемых в данный момент данных.

Программа сортировки чисел (их может быть 5, 100, 10 000 или сотни тысяч).

```
int main()
{  cout << "How many numbers are you going to sort? ";
  int how_many_numbers;
  cin >> how_many_numbers;
  if(how_many_numbers <= 0 || how_many_numbers > 1000000) {
     cout << "Are you kidding?" << endl;
     return 1;
  }
  int *numbers = new int [how_many_numbers];
  for (int i = 0; i < how_many_numbers; i++) {
     cout << "\nEnter the number #" << i + 1 << ": ";
     cin >> numbers[i];  }
```

```
bool swapped;
  do {
     swapped = false;
     for (int i = 0; i < how_many_numbers - 1; i++)
     if (numbers[i] > numbers[i + 1]) {
     swapped = true;
     int aux = numbers[i];
     numbers[i] = numbers[i + 1];
     numbers[i + 1] = aux;
     }
  } while (swapped);

cout << endl << "The sorted array:" << endl;
  for (int i = 0; i < how_many_numbers; i++)
     cout << numbers[i] << " ";
  cout << endl;
  delete[] numbers;
}
```

Динамический массив

Динамический массив – это массив, размер которого заранее не фиксирован и может меняться во время исполнения программы.

Для изменения размера динамического массива - специальные встроенные функции или операции. Динамические массивы дают возможность более гибкой работы с данными, позволяют не прогнозировать хранимые объемы данных, а регулировать размер массива в соответствии с реально необходимыми объемами.

Работа с динамическими массивами:

-объявление;

-выделение памяти;

-обращение к элементам массива, реализация алгоритма ;

-освобождение памяти

Объявление одномерного динамического массива - объявление указателя на переменную заданного, эту переменную можно использовать как динамический массив: Тип * ИмяМассива;

```
int *a;    double *d;
```

a и d - указатели на начало выделяемого участка памяти.

Объявление двумерного динамического массива - объявление двойного указателя, то есть имя двойного указателя для выделяемого блока памяти :

```
Тип ** ИмяМассива;
```

```
int **a;    double **d;
```

2 способа для выделения памяти под одномерный динамический массив:

1) операция new - выделяет для размещения массива участок динамической памяти соответствующего размера и не позволяет инициализировать элементы массива.

```
ИмяМассива = new Тип [ВыражениеТипаКонстанты];
```

ИмяМассива – идентификатор массива, имя указателя для выделяемого блока памяти.

Тип – тип указателя на массив.

ВыражениеТипаКонстанты – задает количество элементов (размерность) массива.

Выражение константного типа вычисляется на этапе компиляции.

Пример:

```
int *mas;

mas = new int [100];     /*выделение динамической памяти размером

100*sizeof(int) байтов*/

double *m = new double [n];       /*выделение динамической  памяти размером n*sizeof(double) байтов*/

long (*lm)[4];

lm = new long [2] [4];   /*выделение динамической памяти размером 2*4*sizeof(long) байтов*/
```

При выделении динамической памяти размеры массива должны быть полностью определены.

При помощи библиотечной функции malloc (calloc)

Синтаксис: ИмяМассива = (Тип *) malloc(N*sizeof(Тип));

Или ИмяМассива = (Тип *) calloc(N, sizeof(Тип));

ИмяМассива – идентификатор массива

Тип – тип указателя на массив. N – количество элементов массива.

Пример:

```
float *a;
a=(float *)malloc(10*sizeof(float));  // или
a=(float *)calloc(10,sizeof(float));
/*выделение динамической памяти размером 10*sizeof(float) байтов*/
```

Так как функция malloc (calloc) возвращает нетипизированный указатель void *, то необходимо выполнять преобразование полученного нетипизированного указателя в указатель объявленного типа.

При формировании двумерного динамического массива сначала выделяется память для массива указателей на одномерные массивы, а затем в цикле память под одномерные массивы.

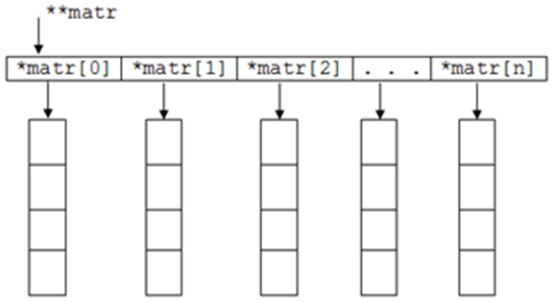

Рис. 4.3. Выделение памяти под двумерный массив

2 способа выделения памяти под двумерный динамический массив.

1) при помощи операции new

Синтаксис выделения памяти под массив указателей:

ИмяМассива = new Тип * [ВыражениеТипаКонстанты];

Синтаксис выделения памяти для массива значений:

```
ИмяМассива[ЗначениеИндекса] = new Тип [ВыражениеТипаКонстанты];
```

ИмяМассива – идентификатор массива, имя двойного указателя для выделяемого блока памяти.

```
int n, m;  //n и m – количество строк и столбцов матрицы
float **matr;   //указатель для массива указателей
matr = new float * [n];    //выделение динамической памяти под массив указателей
for (int i=0; i<n; i++)
   matr[i] = new float [m];   //выделение динамической памяти для массива значений
```

2) функции malloc (calloc) - для выделения динамической памяти.

Синтаксис выделения памяти под массив указателей:

```
ИмяМассива = (Тип **) malloc(N*sizeof(Тип *));
```

или

```
ИмяМассива = (Тип **) calloc(N, sizeof(Тип *));
```

Синтаксис выделения памяти для массива значений:

```
ИмяМассива[ЗначениеИндекса]=(Тип*)malloc(M*sizeof(Тип));
```

или

```
ИмяМассива[ЗначениеИндекса]=(Тип*)calloc(M,sizeof(Тип));
```

ИмяМассива – идентификатор массива, то есть имя двойного указателя для выделяемого блока памяти.

Тип – тип указателя на массив.

N – количество строк массива; M – количество столбцов массива.

```
int n, m;  //n и m – количество строк и столбцов матрицы

float **matr;   //указатель для массива указателей

matr = (float **) malloc(n*sizeof(float *));   //выделение динамической памяти          // под массив указателей

for (int i=0; i<n; i++)

   matr[i] = (float *) malloc(m*sizeof(float));   //выделение динамической             //памяти для массива значений
```

1) операция delete - освобождает участок памяти ранее выделенной операцией new.

Синтаксис: `delete [] ИмяМассива;`

```
delete [ ] mas;    /*освобождает память, выделенную под массив, если mas адресует его начало*/

delete [ ] m;

delete [ ] lm;
```

Квадратные скобки [ ] сообщают оператору, что требуется освободить память, занятую всеми элементами, а не только первым.

2) при помощи библиотечной функции free, которая служит для освобождения динамической памяти.

Синтаксис: `free (ИмяМассива);`

```
free (a); //освобождение динамической памяти
```

Освобождение памяти. Двумерный массив

1)операция delete - освобождает участок памяти ранее выделенной операцией new.

Синтаксис освобождения памяти, выделенной под массив указателей:

```
  delete [] ИмяМассива;
```

ИмяМассива – идентификатор массива, то есть имя двойного указателя для выделяемого блока памяти.

```
for (int i=0; i<n; i++)
  delete matr [i];
   //освобождает память, выделенную для массива значений
delete [] matr;
//освобождает память, выделенную под массив указателей
```

Квадратные скобки [] означают, что освобождается память, занятая всеми элементами массива, а не только первым.

2) функция free - для освобождения динамической памяти.

Синтаксис :

```
free (ИмяМассива[ЗначениеИндекса]);
```

ИмСинтаксис освобождения памяти, выделенной под массив указателей:

```
free (ИмяМассива);

for (int i=0; i<n; i++)

  free (matr[i]); //освобождает память, выделенную для массива значений

free (matr);

//освобождает память, выделенную под массив указателей
```

Треугольные матрицы

Преимущество таких массивов в том, что каждая строка может быть разной длины (рис. 5.1). Это полезно для алгоритмов, которым для запуска требуется не весь массив, а только его часть. Это относится конкретно к треугольным матрицам. Порядок освобождения памяти обратный порядку выделения.

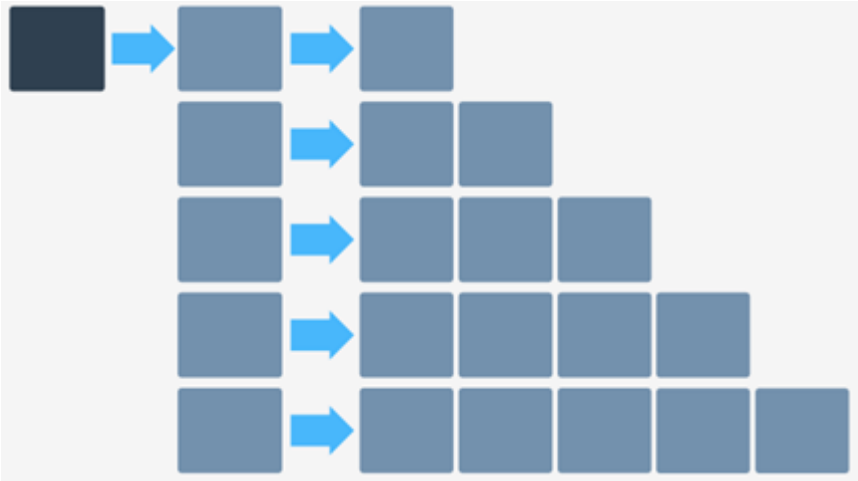

Рис. 5.1. Треугольная матрица

Пример. Обратите внимание, как получается «треугольность» (размер выделенного блока памяти зависит от номера строки) и как значение, присвоенное элементам, отражает их расположение в массиве.

Программный код:

```
#include <iostream>
using namespace std;
int main()
{
  int rows = 5;
   //выделение памяти и
  //  инициализация
  int **arr = new int* [rows];
  for (int r = 0; r < rows; r++) {
     arr[r] = new int[r + 1];
     for (int c = 0; c <= r; c++)
     arr[r][c] = (r + 1) * 10 + c + 1;
  }
// вывод
for (int r = 0; r < rows; r++) {
     for (int c = 0; c <= r; c++)
     cout << arr[r][c] << " ";
     cout << endl;
  }
   //освобождение
  for (int r = 0; r < rows; r++)
     delete[] arr[r];
  delete[] arr;
}
```

Результат:

11
21 22
31 32 33
41 42 43 44

51 52 53 54 55

Структуры

Сначала – о новом типе string (подробнее будет далее).

Переменные типа string способны хранить строки символов.

Для использования типа string в коде:

- исходный файл должен содержать директиву: `#include <string>`
- using namespace std; нужно поместить в начало кода, если вы не хотите использовать префикс std:: каждый раз при использовании string.

```
string name_dish = "pizza";
```

Задача: разработать БД, для хранения информации о студентах, посещающих курс. Необходимо хранить имя каждого студента, время, затраченное на изучение глав, и номер последней завершенной главы. Общее число всех студентов <= 100 000.

Тогда можно: `vector<string> student_name(100000);`

Массив позволяет хранить до 100 000 имен.

Действия с массивом. Пусть первым зарегистрированным студентом был Бонд:

```
student_name[0] = "Bond";
```

Время будет - в виде float. Количество - в виде десятичной дроби.

```
vector<float> student_time_spent(100000);
```

Пусть студент Бонд потратил 3 часа и 30 минут на изучение курса:

```
student_time_spent[0] = 3.5;
```

Номер последней главы является int:

```
vector<int> student_recent_chapter(100000);
```

Пусть у Бонда номер последней завершенной главы = № 7:

```
student_recent_chapter[0] = 7;
```

Проблема: данные, касающиеся одного и того же объекта (учащегося), распределены между тремя переменными, хотя по логике они должны существовать как единое целое. Обработка нескольких массивов является громоздкой и подверженной ошибкам. Если надо собрать дополнительную информацию (например, адрес электронной почты), потребуется объявить другой массив и внести другие изменения по всей программе.

Знаем, что вектор - это совокупность элементов. Элементы пронумерованы и имеют один и тот же тип.

Можно ли использовать агрегат, элементы которого могут быть разных типов?

Можно ли их идентифицировать по именам, а не по номерам?

Да, этот агрегат - структура.

Структура содержит любое количество элементов любого типа.

Каждый из этих элементов называется полем.

Каждое поле идентифицируется по его названию, а не по номеру.

Имена полей должны быть уникальными удвоены в рамках одной структуры.

Объявление структуры:

```
struct Student {
  string name;
  float time_spent;
  int recent_chapter;
};
```

Объявление структуры - с ключевого слова struct, после него - название структуры.

Существует соглашение - название структуры начинать с заглавной буквы (чтобы отличать их от обычных переменных).

Объявление полей – между { и }в конце ;

В структуре Student три поля:

- string с именем name;
- float с именем time_spent;
- int с именем recent_chapter.

Объявление не создает переменную, а только описывает структуру. Объявление переменной как структуры - один из двух способов:

```
Student stdnt;
Student stdnt2;
```

Тип переменных - `struct STUDENT` или просто `STUDENT` (объявление структуры создает новое имя типа).

Эта переменная состоит из трех полей.

Язык C++ есть:

- оператор индексирования [ ] для массивов;

-оператор выбора для структур и обозначаемый одним символом . (точка)

Приоритет оператора выбора высокий, равен приоритету оператора [ ].

Это двоичный оператор. Левый аргумент должен идентифицировать структуру, правый аргумент должен быть именем поля структуры.

Результат оператора - выбранное поле структуры, поэтому выражение, содержащее этот оператор, иногда называют селектором.

Здесь селектор `stdnt.time_spent` приводит к выбору поля с именем `time_spent`.

Типом выражения является тип выбранного поля, это выражение представляет собой l-значение (значение, которое можно поместить слева от оператора присваивания =).

Использование:

```
stdnt.time_spent = 1.5;
```

```
float t = stdnt.time_spent;
```

Современный C++ обрабатывает имена структур так же, как и имена других типов (Student в примере), также возможно использовать в качестве имени типа фразу struct Student (она - производная от C, где это был единственный способ назвать структурированные типы).

Следующее заявление также верно, но выглядит хуже:

```
struct Student stdnt;
```

В качестве поля структуры можно использовать: скаляры, векторы и другие массивы. Сама структура не может быть полем.

Структуры могут быть объединены внутри вектора.

Объявление вектора, состоящего из Student учащихся:

```
vector<Student> stdnt(100000);
```

Доступ к выбранным полям требует двух операций:

1) оператор [ ] индексирует вектор, чтобы получить доступ к нужной структуре;

2) оператор выбора выбирает нужное поле.

Выбор поля time_spent четвертого элемента stdnts:

```
  stdnts[3].time_spent
```

Эти задания для трех отдельных массивов.

```
stndts[0].name = "Bond";

stndts[0].time_spent = 3.5;

stdnts[0].recent_chapter = 7;
```

Пример - структура для хранения даты с 3 полями типа int: year, month, day.

1-ый способ объявления структуры:

```
  struct Date {

            int year;

    int month;
```

```
    int day;
  };
```

Более компактно:

```
struct Date {
    int year, month, day;
};
```

Объявление новой переменной: Date DateOfBirth;

Можно использовать её для хранения даты рождения:

```
DateOfBirth.year = 1980;
DateOfBirth.month = 7;
DateOfBirth.day = 31;
```

Можно использовать тег structure для объявления массива структур:

```
vector<Date> visits(100);
```

Доступ к одной структуре, хранящейся в массиве:

```
visits[0].year = 2020;
visits[0].month = 1;
visits[0].day =  1;
```

2 ой способ - можно определить тег структуры и объявить любое количество переменных одновременно в одном и том же операторе:

```
struct Date {
```

```
    int year, month, day;
} DateOfBirth, ExpirationDate, ETA;
Date current_date;
```

Можно опустить тег и объявить только переменные:

```
struct {
  int year, month, day;
}
the_date_of_the_end_of_the_world;
```

Оператор sizeof() (это оператор, а не функция!) возвращает количество байтов, занятых переменной или типом.

sizeof(Date) - сколько байт необходимо для хранения данных одной даты

Без структуры это сложнее, хотя можно использовать the_date_of_the_end_of_the_world; вместо этого.

Структура может быть полем внутри другой структуры.

Добавим в структуру Student поле для сохранения даты, когда конкретный студент в последний раз посещал курс.

```
struct Student {
  string name;
  float time_spent;
  int recent_chapter;
  Date last_visit;
```

```
};

  Student HarryPotter;
```

Используем две операции выбора: сначала выбираем структуру внутри структуры, затем - нужное поле внутренней структуры.

```
HarryPotter.last_visit.year = 2020;

HarryPotter.last_visit.month = 12;

HarryPotter.last_visit.day = 21;
```

Имена полей структуры могут перекрываться с именами тегов (обычно это не проблема, но может вызвать трудности при чтении и понимании программы).

```
struct Struct {

  int Struct;

} Structure;

Structure.Struct = 0; // Struct – это имя поля здесь
```

Может быть, что конкретному компилятору не понравится, когда имя тега структуры совпадает с именем переменной, поэтому этого лучше избегать:

```
struct Str {

  int field;

} Structure;

int Str;
```

```
Structure.field = 0;

Str = 1;
```

Две структуры могут содержать поля с одинаковыми именами:

```
struct S1 {
  int f1;
};
struct S2 {
  char f1;
};
S1 str1;
S2 str2;
str1.f1 = 32;
str2.f1 = str1.f1;
```

Структуры могут быть инициализированы уже во время объявления. Инициатор структуры заключен в фигурные скобки { } и содержит список значений, присвоенных последующим полям, начиная с первого.

Значения, перечисленные в инициаторе, должны соответствовать типам полей.

Если инициатор содержит меньше элементов, чем количество полей структуры, предполагается, что список автоматически расширяется нулями.

Пустой инициализатор структуры { } также обнулит все поля структуры. Это правило «нулевого расширения».

Обнуление может вызвать разные эффекты в зависимости от типа поля - целые числа и числа с плавающей точкой будут фактически обнулены, а строковым полям будет присвоена пустая строка.

Если конкретное поле является массивом или структурой, оно должно иметь свой собственный инициатор, который также подчиняется правилу нулевого расширения. Если “внутренний” инициатор завершен, можем опустить { }.

```
struct Date moon_landing = { 1969, 7, 20 };
```

Это эквивалентно последовательности назначений:

```
date.year = 1969;

date.month = 7;

date.day = 20;
```

Пример инициатора для структуры Student:

```
Student he = { "Bond", 3.5, 4, {2020, 12, 21} };
```

Это эквивалентно:

```
he.name = "Bond";

he.time_spent = 3.5;

he.recent_chapter = 4;

he.last_visit.year = 2012;

he.last_visit.month = 12;
```

```
he.last_visit.day = 21;
```

Упрощенная форма:

```
Student he = { "Bond", 3.5, 4, 2012, 12, 21};
```

Это упрощение не может быть применено в следующем случае (такой способ не рекомендуется, некоторые компиляторы могут выдавать предупреждающее сообщение):

```
Student she = { "Mata Hari", 12., 12, { 2012 }  };
```

Внутренний инициатор, ссылающийся на поле last_visit, не охватывает все поля. Это будет эквивалентно последовательности назначений:

```
she.name= "Mata Hari";

she.time_spent = 12.;

she.recent_chapter = 12;

she.last_visit.year = 2012;

she.last_visit.month = 0;

she.last_visit.day = 0;
```

Применим “пустой” инициализатор:

```
Student nobody = {};
```

Ответ:

```
nobody.name = "";

nobody.time = 0.0;

nobody.recent_chapter = 0;

nobody.last_visit.year = 0

nobody.last_visit.month = 0;

nobody.last_visit.day = 0;
```

Пример. Сложение комплексных чисел.

```
#include "stdafx.h"#

include <iostream>

using namespace std;

typedef  struct {

            double real;

            double imag;

            } complex;

int main( ) {

  complex x,y,z;

  printf("\n Введите два комплексных числа:");
```

```
        printf("\n Вещественная часть:");
        scanf("%lf", &x.real);
        printf("\n Мнимая часть:");
        scanf("%lf", &x.imag);
        printf("\n Вещественная часть:");
        scanf("%lf", &y.real);
        printf("\n Мнимая часть:");
        scanf("%lf", &y.imag);
        z.real=x.real+y.real;
        z.imag=x.imag+y.imag;
        printf("\n   Результат:   z.real=%f   z.imag=%f",
z.real,z.imag);
        system("pause");
        return 0;
    }
```