Министерство экономического развития и торговли Российской Федерации
Министерство образования и науки Российской Федерации

ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ – ВЫСШАЯ ШКОЛА ЭКОНОМИКИ
Нижегородский Филиал

Практикум:
Объектно-ориентированное программирование
в примерах на C++

Рекомендовано Ученым советом Нижегородского филиала Государственного университета-Высшей школы экономики в качестве учебного пособия для студентов направления "Бизнес-информатика"

Нижний Новгород 2005

УДК 681.3.06
ББК 32.973.26
Д 30

В.М.Дёмкин. Практикум: Объектно-ориентированное программирование в примерах на C++: Учебное пособие. – Н.Новгород: НФ ГУ-ВШЭ, 2005. – 179 с.

ISBN 5-901956-11-7

Практикум предназначен для самостоятельного изучения одного из разделов курса “Информатика” – объектно-ориентированного программирования на языке C++. Обсуждаются вопросы проектирования пользователем абстрактных типов данных с помощью структур, объединений и классов. Учебные примеры в виде программных реализаций с комментариями иллюстрируют принципы объектно-ориентированной парадигмы программирования. Программный код апробирован на современных компиляторах платформ Windows и Linux.

Библиогр.: 10 назв.

Рецензенты: д-р физ.-мат. наук, проф. Е.М.Громов
д-р физ.-мат. наук, проф. С.Н.Митяков

ББК 32.973.26

ISBN 5-901956-11-7

Государственный Университет-Высшая Школа Экономики Нижегородский филиал
Н.Новгород, ул. Б.Печерская, 25

Подп. к печ. 25.03.2005. Формат 60x84/16. Бумага офсетная.
Печать офсетная. Усл. п. л.: 10,4. Тираж 200 экз. Заказ № 168.

Отпечатано ООО «Растр НН» г. Н.Новгород, ул. Белинского, 61
ИД № 05407 от 20.07.2001

Предисловие

Предлагаемый вниманию читателя практикум по объектно-ориентированному программированию на языке C++ открывает собой серию методических материалов, предназначенных для формирования у студентов младших курсов направления “Бизнес-информатика” целостного взгляда на информатику. Автор на протяжении многих лет не без успеха последовательно отстаивает свою позицию в том, что реальным помощником в реализации этой цели могут служить отдельные учебные пособия и практикумы по основным разделам курса “Информатика”, связанные единой формой изложения. Прежде всего, сама идея издания подобной литературы связана со стремлением автора рассматривать информатику под углом зрения проектирования алгоритмов и абстрактных типов данных, придерживаясь рамок требуемой для решения задачи парадигмы программирования (например, императивной, объектно-ориентированной или обобщенной) при помощи таких популярных языков программирования для персональных компьютеров, как Fortran, Basic, Turbo Pascal, C, C++ и Java.

Излагаемый материал практикума соответствует разделам учебного пособия “Основы объектно-ориентированного программирования в примерах на C++” и является одной из трех составных частей лабораторного практикума учебной дисциплины “Методы программирования”. Программный код прошел апробацию на современных компиляторах платформ Windows и Linux.

Изучение учебного материала практикума предполагает знакомство читателя с императивной парадигмой программирования для языка C++.

Язык программирования C++ является одним из представителей семейства гибридных языков, он поддерживает четыре парадигмы программирования: императивную, модульную, объектно-ориентированную и обобщенную.

C++ был создан Бьярном Страуструпом (Bjarne Stroustrup) в начале 1980-х годов. Перед Страуструпом стояли тогда две основные задачи: во-первых, сделать C++ совместимым со стандартным языком C, во-вторых, расширить C конструкциями объектно-ориентированного программирования, основанными на концепции классов из языка Simula 67. Язык C был создан Дэннисом Ритчи (Dennis M.Ritchie) как язык системного программирования в начале 1970-х годов. История C связана с разработкой операционной системы Unix для мини-компьютера DEC PDP-11.

C++ развивался не только благодаря идеям, взятых из других языков программирования, но и в соответствии с запросами и опытом большого числа пользователей различного уровня, использующих C++ в самых разнообразных прикладных областях. В августе 1998 года был ратифицирован стандарт языка C++ (ISO/IEC 14882). Стандартный C++ и его стандартная библиотека до сих пор остаются одним из основных инструментов для разработки приложений, предназначенных для широкого диапазона использования.

Структуры и объединения – абстрактные типы данных
Структуры

➢ Пример 1

Рассмотрим реализацию стека (динамическая структура данных, организованная по принципу ***LIFO – Last In First Out*** – “последним пришел, первым ушел”).

Поначалу стек будет представлен в виде статической структуры данных на основе одномерного символьного массива фиксированной длины, для доступа к которому будет использоваться индекс ***top*** (вершина стека). Коллекция операций состоит из стандартных операций работы со стеком: ***push*** (вталкивание данных в стек) и ***pop*** (выталкивание данных из стека), а также ряда вспомогательных операций: ***reset*** (привести стек в исходное состояние), ***top*** (доступ к вершине стека), ***empty*** (проверить состояние стека “пустой”) и ***full*** (проверить состояние стека “заполнен”). Далее будет сказано, как от статической структуры данных перейти к динамической.

Например, представим стек для 10 элементов:

```
// C++ Абстрактный тип данных - стек

#include <iostream>

using namespace std;

const int stackSize = 10;
const int stackEmpty = -1;
const int stackFull = stackSize - 1;
struct Stack
{
  char buffer[stackSize];     // стек
  int top;                    // вершина стека
};

// Привести стек в исходное состояние
void reset(Stack* stack)
{
  stack->top = stackEmpty;
}

// Вталкивание данных в стек
void push(char symbol, Stack* stack)
{
  stack->buffer[++stack->top] = symbol;
}

// Выталкивание данных из стека
char pop(Stack* stack)
{
```

```
  return stack->buffer[stack->top--];
}

// Доступ к вершине стека
char top(Stack* stack)
{
  return stack->buffer[stack->top];
}

// Проверить состояние стека - "пустой"
bool empty(const Stack* stack)
{
  return (stack->top == stackEmpty);
}

// Проверить состояние стека - "заполнен"
bool full(const Stack* stack)
{
  return (stack->top == stackFull);
}

int main()
{
  Stack stack;
  char line[] = "Hello, Hello!";
  int i = 0;
  cout << line << endl;
  // Привести стек в исходное состояние
  reset(&stack);
  // Вталкивание символов С-строки в стек
  while (line[i])
    if (!full(&stack))
      push(line[i++], &stack);
    else ++i;
  // Доступ к вершине стека
  if (!empty(&stack)) cout << top(&stack) << endl;
  // Выталкивание символов из стека
  while (!empty(&stack)) cout << pop(&stack);
  cout << endl;
  return 0;
}
```

Результат работы программы:

```
Hello, Hello!
l
leH ,olleH
```

➢ Пример 2

Рассмотрим реализацию комплексного числа, вещественная и мнимая части которого являются объектами встроенного типа ***double***. Коллекция операций состоит из четырех операций комплексной арифметики (сложение, вычитание, умножение и деление), а также операций инициализации и визуализации комплексных чисел.

Например, вычислим значение выражения $\frac{(-1+5i)^2(3-4i)}{1+3i}+\frac{10+7i}{5i}$:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  double re;     // вещественная часть комплексного числа
  double im;     // мнимая часть комплексного числа
};

// Инициализация комплексного числа
void define(Complex& c, double r = 0.0, double i = 0.0)
{
  c.re = r;
  c.im = i;
}

// Сложение комплексных чисел
Complex add(Complex a, Complex b)
{
  Complex temporary;
  temporary.re = a.re + b.re;
  temporary.im = a.im + b.im;
  return temporary;
}

// Вычитание комплексных чисел
Complex subtract(Complex a, Complex b)
{
  Complex temporary;
  temporary.re = a.re - b.re;
  temporary.im = a.im - b.im;
  return temporary;
}

// Умножение комплексных чисел
Complex multiply(Complex a, Complex b)
{
```

```
  Complex temporary;
  temporary.re = a.re * b.re - a.im * b.im;
  temporary.im = a.re * b.im + b.re * a.im;
  return temporary;
}

// Деление комплексных чисел
Complex divide(Complex a, Complex b)
{
  Complex temporary;
  double divider = b.re * b.re + b.im * b.im;
  temporary.re = (a.re * b.re + a.im * b.im) / divider;
  temporary.im = (b.re * a.im - a.re * b.im) / divider;
  return temporary;
}

// Визуализация комплексного числа
void print(Complex c)
{
  cout << '(' << c.re << ", " << c.im << ')' << endl;
}

int main()
{
  Complex x1, y1, z1, x2, z2;
  // Инициализация операндов выражения - комплексных чисел
  define(x1, -1, 5);              // x1 = -1 + 5i
  define(y1, 3, -4);              // y1 = 3 - 4i
  define(z1, 1, 3);               // z1 = 1 + 3i
  define(x2, 10, 7);              // x2 = 10 + 7i
  define(z2, 0, 5);               // z2 = 5i
  // Визуализация вычисленного значения выражения
  print(add(divide(multiply(multiply(x1, x1), y1), z1),
        divide(x2, z2)));
  return 0;
}
```

Результат работы программы:

```
(10, 38.2)
```

➢ Пример 3

Рассмотрим реализацию двумерного массива объектов встроенного типа ***double*** в свободной памяти (динамический массив).

Динамический двумерный массив определяется значениями своих размерностей: ***columnSize*** (размер столбца) – число строк – и ***rowSize*** (размер строки) – число столбцов массива. Указатель ***base*** – указатель на указатель на ***double***, указатель ***base*** содержит начальный адрес массива указателей (***0, 1, …, columnSize–1***), а каждый указатель этого массива – начальный адрес строки, состоящей из элементов ***double*** (***0, 1, …, rowSize–1***). Коллекция операций состоит из операций выделения и освобождения свободной памяти, а также поиска максимального элемента массива.

Например, представим двумерный массив из 5×5 элементов:

```
// C++ Абстрактный тип данных – динамический двумерный массив

#include <iostream>
#include <iomanip>

using namespace std;

struct Array2D
{
  double** base;     // указатель на массив
  int columnSize;    // число строк массива
  int rowSize;       // число столбцов массива
};

// Выделение свободной памяти для массива
void allocate(int columnSize, int rowSize, Array2D& a)
{
  a.base = new double*[columnSize];
  for (int i = 0; i < columnSize; ++i)
    a.base[i] = new double[rowSize];
  a.columnSize = columnSize;
  a.rowSize = rowSize;
}

// Освобождение выделенной для массива памяти
void deallocate(Array2D& a)
{
  for (int i = 0; i < a.columnSize; ++i)
    delete[] a.base[i];
  delete[] a.base;
}

// Поиск максимального элемента массива
double find_maximum(Array2D& a)
{
```

```
  double maximum = a.base[0][0];
  for (int i = 0; i < a.columnSize; ++i)
    for (int j = 0; j < a.rowSize; ++j)
      if (a.base[i][j] > maximum) maximum = a.base[i][j];
  return maximum;
}

int main()
{
  Array2D a;
  // Выделение свободной памяти для массива
  allocate(5, 5, a);
  // Инициализация и визуализация элементов массива
  for (int i = 0; i < a.columnSize; ++i)
  {
    for (int j = 0; j < a.rowSize; ++j)
    {
      a.base[i][j] = (i + 1) * (j + 1);
      cout << setw(4) << (i + 1) * (j + 1);
    }
    cout << endl;
  }
  // Визуализация максимального элемента массива
  cout << "maximum = " << find_maximum(a) << endl;
  // Освобождение выделенной для массива памяти
  deallocate(a);
  return 0;
}
```

Результат работы программы:

```
   1   2   3   4   5
   2   4   6   8  10
   3   6   9  12  15
   4   8  12  16  20
   5  10  15  20  25
maximum = 25
```

➢ Пример 4

Рассмотрим реализацию двухсвязного списка в виде статической структуры данных. Коллекция операций состоит из операций инициализации и визуализации элементов списка, а также операции присоединения элемента к списку.

Например, представим двухсвязный список из 3 элементов:

```
// C++ Абстрактный тип данных - двухсвязный список

#include <iostream>

using namespace std;

struct List
{
  List* predecessor;  // предыдущий элемент списка
  int data;           // данные элемента списка
  List* successor;    // следующий элемент списка
};

// Инициализация элемента списка
void define(List* pointer, int item = 0)
{
  pointer->predecessor = 0;
  pointer->data = item;
  pointer->successor = 0;
}

// Присоединение элемента к списку
void append(List* pointerPredecessor, List* pointerSuccessor)
{
  pointerPredecessor->successor = pointerSuccessor;
  pointerSuccessor->predecessor = pointerPredecessor;
}

// Визуализация элементов списка в прямом направлении
void printForward(List* pointer)
{
  while (pointer)
  {
    cout << pointer << '\t' << pointer->data << endl;
    pointer = pointer->successor;
  }
}

// Визуализация элементов списка в обратном направлении
void printBackward(List* pointer)
{
```

```
  while (pointer)
  {
    cout << pointer << '\t' << pointer->data << endl;
    pointer = pointer->predecessor;
  }
}

int main()
{
  // Первый элемент списка
  List x1;
  // Инициализация элемента списка
  define(&x1, 1);
  // Второй элемент списка
  List x2;
  // Инициализация элемента списка
  define(&x2, 2);
  // Связывание первых двух элементов списка
  append(&x1, &x2);
  // Визуализация элементов списка в прямом направлении
  printForward(&x1);
  // Визуализация элементов списка в обратном направлении
  printBackward(&x2);
  // Третий элемент списка
  List x3;
  // Инициализация элемента списка
  define(&x3, 3);
  // Присоединение элемента к списку
  append(&x2, &x3);
  // Визуализация элементов списка в прямом направлении
  printForward(&x1);
  // Визуализация элементов списка в обратном направлении
  printBackward(&x3);
  return 0;
}
```

Результат работы программы:

```
0x8fad0ff6      1
0x8fad0fec      2
0x8fad0fec      2
0x8fad0ff6      1
0x8fad0ff6      1
0x8fad0fec      2
0x8fad0fe2      3
0x8fad0fe2      3
0x8fad0fec      2
0x8fad0ff6      1
```

Объединения

➢ Пример 5

Объединение позволяет в разное время использовать одну и ту же область памяти для хранения объектов различных типов и размеров.

Для типа объединения ***ShortChar***, элементами которого являются переменная целого типа ***short s*** и беззнаковый символьный массив ***unsigned char c*[*2*]**, представим различные варианты доступа к элементам объединений ***union1***, ***union2*** и ***union3***:

```
// C++ Абстрактный тип данных - объединение short и char

#include <iostream>

using namespace std;

union ShortChar
{
  short s;
  unsigned char c[2];
} union1 = { 16961 }, union2 = { 17475 };

ShortChar union3;
short byte0, byte1;

int main()
{
  cout << union1.s << endl;
  cout << (short)union1.c[0] << ' ' << union1.c[0] << endl;
  cout << (short)union1.c[1] << ' ' << union1.c[1] << endl;
  cout << union2.s << endl;
  cout << (short)union2.c[0] << ' ' << union2.c[0] << endl;
  cout << (short)union2.c[1] << ' ' << union2.c[1] << endl;
  union1.c[0] = 'C';
  union1.c[1] = 'D';
  cout << union1.s << endl;
  cout << (short)union1.c[0] << ' ' << union1.c[0] << endl;
  cout << (short)union1.c[1] << ' ' << union1.c[1] << endl;
  union3.s = 19788;
  byte0 = (short)union3.c[0];
  byte1 = (short)union3.c[1];
  cout << union3.s << endl;
  cout << byte0 << ' ' << union3.c[0] << ' '
       << oct << byte0 << ' ' << hex << byte0 << endl;
  cout << dec << byte1 << ' ' << union3.c[1] << ' '
       << oct << byte1 << ' ' << hex << byte1 << endl;
  return 0;
}
```

Результат работы программы:

```
16961
65 A
66 B
17475
67 C
68 D
17475
67 C
68 D
19788
76 L 114 4c
77 M 115 4d
```

Не следует забывать о том, что все эти представленные структуры и объединения, вообще говоря, являются своего рода упрощенными формами класса. Это означает, что для таких классов в случае необходимости компилятор может сгенерировать два конструктора – конструктор по умолчанию (конструктор с пустым списком параметров) и конструктор копирования по умолчанию (конструктор с одним параметром), статус доступа которых ***public*** (открытый, общедоступный). Роль каждого из этих конструкторов чрезвычайно важна. Далее, пример за примером, будут представлены все возможные виды конструкторов, а пока обсуждение этой темы здесь несколько преждевременно.

➢ Пример 6

Зачастую объединения являются элементами структур или массивов. Например, комбинированное использование структуры и объединения позволяет объявлять так называемые переменные с изменяемой структурой (переменные структуры). Пусть информация о геометрических фигурах будет представлена на основе именно такого использования структуры и объединения. Общая информация о фигурах может включать, например, такие элементы, как площадь и периметр, поэтому их следует представить в виде элементов структуры. Поскольку информация о геометрических размерах фигур может оказаться различной в зависимости от их формы, то ее можно представить в виде элементов объединения. При этом пользователь вправе выбрать либо анонимное объединение, либо не анонимное, что приведет либо к упрощению механизма доступа к элементу объединения, либо к его усложнению.

Приведем определение структурного типа ***Figure*** для создания переменной структуры, где в качестве фигур выбраны круг, прямоугольник и треугольник:

```
// C++ Абстрактный тип данных - переменная структура

#include <iostream>
#include <cmath>

using namespace std;

enum figureStatus { circle, rectangle, triangle };

struct Figure
{
  // Общие элементы - параметры фигуры
  double area;         // площадь фигуры
  double perimeter;    // периметр фигуры
  // Метка активного элемента объединения
  figureStatus type;
  // Конкретные параметры фигуры
  union
  {
    double radius;               // радиус круга
    double rectangleSides[2];    // стороны прямоугольника
    double triangleSides[3];     // стороны треугольника
  } anyFigure;
};

// Площадь и периметр фигуры
void get_figureParameters(Figure* figure)
{
  switch (figure->type)
  {
    case circle :
    {
```

```
      double a = figure->anyFigure.radius;
      figure->area = 3.141592654 * a * a;
      figure->perimeter = 2 * 3.141592654 * a;
      break;
    }
    case rectangle :
    {
      double a = figure->anyFigure.rectangleSides[0];
      double b = figure->anyFigure.rectangleSides[1];
      figure->area = a * b;
      figure->perimeter = 2 * (a + b);
      break;
    }
    case triangle :
    {
      double a = figure->anyFigure.triangleSides[0];
      double b = figure->anyFigure.triangleSides[1];
      double c = figure->anyFigure.triangleSides[2];
      double p = (a + b + c) / 2;
      figure->area = sqrt(p * (p - a) * (p - b) * (p - c));
      figure->perimeter = 2 * p;
      break;
    }
  }
}

int main()
{
  Figure figure;
  figure.type = circle;
  figure.anyFigure.radius = 1;
  get_figureParameters(&figure);
  cout << "Площадь круга и длина окружности = " << figure.area
       << ", " << figure.perimeter << endl;
  figure.type = rectangle;
  figure.anyFigure.rectangleSides[0] = 1;
  figure.anyFigure.rectangleSides[1] = 2;
  get_figureParameters(&figure);
  cout << "Площадь и периметр прямоугольника = " << figure.area
       << ", " << figure.perimeter << endl;
  figure.type = triangle;
  figure.anyFigure.triangleSides[0] = 3;
  figure.anyFigure.triangleSides[1] = 4;
  figure.anyFigure.triangleSides[2] = 5;
  get_figureParameters(&figure);
  cout << "Площадь и периметр треугольника = " << figure.area
       << ", " << figure.perimeter << endl;
  return 0;
}
```

Результат работы программы:

```
Площадь круга и длина окружности = 3.141593, 6.283185
Площадь и периметр прямоугольника = 2, 6
Площадь и периметр треугольника = 6, 12
```

Здесь значение переменной ***type*** используется для указания, какой из элементов объединения ***anyFigure*** в данный момент является активным. Объект структурного типа ***Figure*** называется переменной структурой, потому что ее элементы меняются в зависимости от значения метки активного элемента.

В случае объявления анонимного объединения

```
struct Figure
{
// Общие элементы - параметры фигуры
  double area;
  double perimeter;
// Метка активного элемента объединения
  figureStatus type;
// Конкретные параметры фигуры
  union
  {
    double radius;
    double rectangleSides[2];
    double triangleSides[3];
  };
};
```

доступ к его элементам несколько упростится, например, элемент ***radius*** в функции ***main*()** будет доступен благодаря конструкции:

```
figure.radius = 1;
```

Далее будет сказано, что элементы анонимного объединения находятся в той же области видимости, что и само объединение, т.е. в данном случае в локальной области видимости структурного типа ***Figure***.

➢ **Пример 7**

Чтобы проиллюстрировать, насколько упростится механизм доступа к элементам анонимного объединения, приведем теперь другое определение структурного типа ***Figure*** для создания переменной структуры, где в качестве фигур, как и ранее, выбраны круг, прямоугольник и треугольник:

```
// C++ Абстрактный тип данных - переменная структура

#include <iostream>
#include <cmath>

using namespace std;

enum figureStatus { circle, rectangle, triangle };

struct Figure
{
  // Общие элементы - параметры фигуры
  double area;         // площадь фигуры
  double perimeter;    // периметр фигуры
  // Метка активного элемента объединения
  figureStatus type;
  // Конкретные параметры фигуры
  union
  {
    double radius;               // радиус круга
    double rectangleSides[2];    // стороны прямоугольника
    double triangleSides[3];     // стороны треугольника
  };
};

// Площадь и периметр фигуры
void get_figureParameters(Figure* figure)
{
  switch (figure->type)
  {
    case circle :
    {
      double a = figure->radius;
      figure->area = 3.141592654 * a * a;
      figure->perimeter = 2 * 3.141592654 * a;
      break;
    }
    case rectangle :
    {
      double a = figure->rectangleSides[0];
      double b = figure->rectangleSides[1];
```

```
      figure->area = a * b;
      figure->perimeter = 2 * (a + b);
      break;
    }
    case triangle :
    {
      double a = figure->triangleSides[0];
      double b = figure->triangleSides[1];
      double c = figure->triangleSides[2];
      double p = (a + b + c) / 2;
      figure->area = sqrt(p * (p - a) * (p - b) * (p - c));
      figure->perimeter = 2 * p;
      break;
    }
  }
}

int main()
{
  Figure figure;
  figure.type = circle;
  figure.radius = 1;
  get_figureParameters(&figure);
  cout << "Площадь круга и длина окружности = " << figure.area
       << ", " << figure.perimeter << endl;
  figure.type = rectangle;
  figure.rectangleSides[0] = 1;
  figure.rectangleSides[1] = 2;
  get_figureParameters(&figure);
  cout << "Площадь и периметр прямоугольника = " << figure.area
       << ", " << figure.perimeter << endl;
  figure.type = triangle;
  figure.triangleSides[0] = 3;
  figure.triangleSides[1] = 4;
  figure.triangleSides[2] = 5;
  get_figureParameters(&figure);
  cout << "Площадь и периметр треугольника = " << figure.area
       << ", " << figure.perimeter << endl;
  return 0;
}
```

Результат работы программы:

```
Площадь круга и длина окружности = 3.141593, 6.283185
Площадь и периметр прямоугольника = 2, 6
Площадь и периметр треугольника = 6, 12
```

➢ Пример 8

Рассмотрим еще раз объявление неименованного типа объединения (анонимное объединение). Основное достоинство анонимных объединений – это экономия памяти, так как доступ к элементам таких объединений реализуется как к обычным переменным программы. Различие между элементами анонимных объединений и обычными переменными лишь в том, что присваивание значения какому-либо элементу объединения приводит к утрате значений других его элементов. Элементы анонимного объединения находятся в той же области видимости, что и само объединение.

Объявим в локальной области видимости главной функции ***main*()** анонимное объединение, элементы которого будут встроенных типов ***short*** и ***long***:

```
// C++ Абстрактный тип данных - анонимное объединение

#include <iostream>

using namespace std;

int main()
{
  union
  {
    short shortInt;
    long longInt;
  };
  shortInt = 1;
  cout << "short\t" << shortInt << "\tlong\t" << longInt << endl;
  longInt = 32768;
  cout << "short\t" << shortInt << "\tlong\t" << longInt << endl;
  return 0;
}
```

Результат работы программы:

```
short   1       long    134479873
short   -32768  long    32768
```

Класс – абстрактный тип данных
Класс как расширение понятия структуры

- **Пример 9**

Получив представление о реализации комплексного числа в виде структуры, рассмотрим теперь реализацию комплексного числа в виде объекта класса, компонентные данные которого, как и оба элемента структуры, будут встроенного типа ***double***. Набор операций для этого первого класса будем строить таким образом, чтобы класс рассматривался как естественное расширение структуры. Компонентные функции пусть представляют собой всего одну операцию комплексной арифметики (сложение) и три вспомогательных операции (инициализация и визуализация комплексных чисел, а также связывание друг с другом двух комплексных чисел).

Определим класс ***Complex***, используя ключ класса ***struct*** (в этом случае все компоненты класса по умолчанию будут общедоступными):

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  // Компонентные функции - все общедоступные (public)
  // Инициализация комплексного числа
  void define(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
  }
  // Сложение комплексных чисел
  Complex add(Complex a, Complex b)
  {
    Complex temporary;
    temporary.re = a.re + b.re;
    temporary.im = a.im + b.im;
    return temporary;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
```

```
  // Связывание комплексного числа с другим комплексным числом
  void assign(Complex c)
  {
    re = c.re;
    im = c.im;
  }
};

int main()
{
  Complex x1, x2, y;
  x1.define(-1, 5);
  x2.define(10, 7);
  y.define();
  x1.print(x1);
  x2.print(x2);
  y.print(y);
  x1.print(x1.add(x1, x2));
  y.assign(y.add(x1, x2));
  y.print(y);
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(0, 0)
(9, 12)
(9, 12)
```

Еще раз обратим внимание на одно весьма важное обстоятельство, связанное с тем, что здесь компилятор в случае необходимости может сгенерировать уже известные нам конструкторы – конструктор по умолчанию (конструктор с пустым списком параметров) и конструктор копирования по умолчанию (конструктор с одним параметром). Роль конструктора по умолчанию, сгенерированного компилятором, весьма важна, но так как его здесь попросту нет, поэтому еще не пришла пора говорить об этом. А вот при вызове компонентных функций ***add*()**, ***print*()** и ***assign*()** для аргументов вызова, передаваемых по значению, всякий раз неявно вызывается конструктор копирования по умолчанию для инициализации их локальных копий. Конструктор копирования по умолчанию к тому же еще инициализирует временный объект, создаваемый для хранения значения, возвращаемого компонентной функцией ***add*()**. Далее будет сказано, что компилятор не всегда строит временные объекты с помощью конструктора копирования, и причиной тому являются детали реализации конкретного компилятора.

Конструкторы, деструкторы и доступ к компонентам класса

➢ Пример 10

Здесь и далее класс ***Complex*** станет одним из тех сквозных примеров, для которого уже нет необходимости пояснять, какие операции и в каком количестве этому классу необходимы. Выбор такого абстрактного типа данных обусловлен, прежде всего, тем, что на его основе автору удалось проиллюстрировать почти все характерные идеи по "обустраиванию" естественного поведения типа. Поначалу там, где это необходимо для иллюстрации механизма вызова конструктора и деструктора класса ***Complex***, каждый такой вызов будет сопровождаться своим идентифицирующим сообщением.

Для создания и инициализации объектов класса ***Complex*** определим в нем компонентную функцию – конструктор объектов класса, называемый конструктором по умолчанию (здесь конструктор без параметров):

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex()
  {
    re = 0.0;
    im = 0.0;
    cout << "Конструктор по умолчанию" << endl;
  }
  // Сложение комплексных чисел
  Complex add(Complex a, Complex b)
  {
    Complex temporary;
    temporary.re = a.re + b.re;
    temporary.im = a.im + b.im;
    return temporary;
  }
  // Связывание комплексного числа с другим комплексным числом
  void assign(Complex c)
  {
    re = c.re;
    im = c.im;
  }
```

```
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
};

int main()
{
  Complex x1;
  Complex x2;
  Complex y;
  x1.print(x1);
  x2.print(x2);
  y.print(y);
  x1.re = -1;
  x1.im = 5;
  x1.print(x1);
  x2.re = 10;
  x2.im = 7;
  x2.print(x2);
  y.assign(y.add(x1, x2));
  y.print(y);
  return 0;
}
```

Результат работы программы:

```
Конструктор по умолчанию
Конструктор по умолчанию
Конструктор по умолчанию
(0, 0)
(0, 0)
(0, 0)
(-1, 5)
(10, 7)
Конструктор по умолчанию
(9, 12)
```

Как видим, конструктор по умолчанию явно вызывается при создании каждого объекта класса, выполняя конструирование их значений:

```
  Complex x1;         // x1.re = 0.0, x1.im = 0.0
  Complex x2;         // x2.re = 0.0, x2.im = 0.0
  Complex y;          // y.re = 0.0, y.im = 0.0
  Complex temporary; // temporary.re = 0.0, temporary.im = 0.0
```

Однако вполне очевидно, что конструктор без параметров – это далеко не лучший конструктор для инициализации объектов.

➢ Пример 11

Следующим шагом в последовательном “обустраивании” класса ***Complex*** будет определение для него такого конструктора по умолчанию, который предоставлял бы пользователю максимально возможную степень свободы по конструированию значений создаваемых объектов.

Для создания и инициализации объектов класса ***Complex*** определим в нем теперь конструктор по умолчанию, все параметры которого имеют стандартные значения:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
    cout << "Конструктор по умолчанию" << endl;
  }
  // Сложение комплексных чисел
  Complex add(Complex a, Complex b)
  {
    Complex temporary;
    temporary.re = a.re + b.re;
    temporary.im = a.im + b.im;
    return temporary;
  }
  // Связывание комплексного числа с другим комплексным числом
  void assign(Complex c)
  {
    re = c.re;
    im = c.im;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
};
```

```
int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  Complex y;
  x1.print(x1);
  x2.print(x2);
  y.assign(y.add(x1, x2));
  y.print(y);
  x1.re += 10;
  x1.im += 7;
  x1.print(x1);
  Complex* p = &x1;
  p->re -= 10;
  (*p).im -=7;
  p->print(x1);
  (*p).print(x1);
  return 0;
}
```

Результат работы программы:

```
Конструктор по умолчанию
Конструктор по умолчанию
Конструктор по умолчанию
(-1, 5)
(10, 7)
Конструктор по умолчанию
(9, 12)
(9, 12)
(-1, 5)
(-1, 5)
```

Как видим, конструктор по умолчанию явно вызывается при создании каждого объекта класса, выполняя конструирование их значений:

```
  Complex x1(-1, 5); // x1.re = -1.0, x1.im = 5.0
  Complex x2(10, 7); // x2.re = 10.0, x2.im = 7.0
  Complex y;         // y.re = 0.0, y.im = 0.0
  Complex temporary; // temporary.re = 0.0, temporary.im = 0.0
```

➢ **Пример 12**

Определим в классе ***Complex*** компонентные функции ***get_re*()** и ***get_im*()**, позволяющие получать доступ к собственным компонентным данным объекта, для которого они были вызваны:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
  }
  // Сложение комплексных чисел
  Complex add(Complex a, Complex b)
  {
    Complex temporary;
    temporary.re = a.re + b.re;
    temporary.im = a.im + b.im;
    return temporary;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
  // Ссылка на вещественную часть комплексного числа
  double& get_re()
  {
    return re;
  }
  // Ссылка на мнимую часть комплексного числа
  double& get_im()
  {
    return im;
  }
  private:
  // Компонентные данные - все собственные (private)
  double re;
  double im;
};
```

```
int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  x1.print(x1);
  x2.print(x2);
  x1.print(x1.add(x1, x2));
  x1.get_re() += 10;
  x1.get_im() += 7;
  x1.print(x1);
  Complex* p = &x1;
  p->get_re() -= 10;
  (*p).get_im() -= 7;
  p->print(x1);
  (*p).print(x1);
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(9, 12)
(9, 12)
(-1, 5)
(-1, 5)
```

Как видим, раздел класса ***private*** ограничивает доступность компонентов класса, т.е. изолирует их внутри класса, в этом случае доступ к закрытым (собственным) компонентам данных ***re*** и ***im*** вне класса возможен только посредством его открытых (общедоступных) компонентных функций.

➢ Пример 13

Зачастую от пользователя класса требуется реализация полного контроля над процессом образования копии объекта во время выполнения инициализации, в таких случаях пользователь всегда определяет свой собственный конструктор копирования.

Перегрузим конструктор копирования по умолчанию, определив в классе ***Complex*** компонентную функцию – конструктор объектов класса, называемый конструктором копирования (конструктор с одним параметром):

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
  }
  // Конструктор объектов класса - конструктор копирования
  Complex(const Complex& c)
  {
    re = c.re;
    im = c.im;
    cout << "Конструктор копирования" << endl;
  }
  // Сложение комплексных чисел
  Complex add(Complex a, Complex b)
  {
    Complex temporary;
    temporary.re = a.re + b.re;
    temporary.im = a.im + b.im;
    return temporary;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
};
```

```
int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  x1.print(x1);
  x2.print(x2);
  Complex y = x1;
  y.print(y);
  y.print(y.add(y, x2));
  Complex z(x1);
  z.print(z);
  return 0;
}
```

Результат работы программы:

```
Конструктор копирования
(-1, 5)
Конструктор копирования
(10, 7)
Конструктор копирования
Конструктор копирования
(-1, 5)
Конструктор копирования
Конструктор копирования
Конструктор копирования
Конструктор копирования
(9, 12)
Конструктор копирования
Конструктор копирования
(-1, 5)
```

Как видим, конструктор копирования вызывается во всех трех случаях инициализации объектов класса, выполняя конструирование их значений:

```
  x1.print(x1);            // неявный вызов конструктора
  x2.print(x2);            // неявный вызов конструктора
  Complex y = x1;          // неявный вызов конструктора
  y.print(y);              // неявный вызов конструктора
  y.print(y.add(y, x2));   // неявные вызовы конструктора
  Complex z(x1);           // явный вызов конструктора
  z.print(z);              // неявный вызов конструктора
```

Следует отметить, что компилятор не всегда создает временный объект при помощи конструктора копирования для хранения значения, которое возвращается в данном примере функцией ***add*()**, а если и создает, то не всегда при этом строит его локальную копию при вызове функции ***print*()**. В таких случаях вместо четырех вызовов конструктора копирования может оказаться всего два или три.

➢ Пример 14

Чтобы уменьшить накладные расходы при выполнении программы, следует отказаться от передачи компонентным функциям ***add*()** и ***print*()** аргументов вызова по значению. Одним из приемлемых вариантов решения этой проблемы является передача аргументов вызова по ссылке.

Определим в классе ***Complex*** компонентные функции ***add*()** и ***print*()** таким образом, чтобы их аргументы передавались по ссылке:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
  }
  // Конструктор объектов класса - конструктор копирования
  Complex(const Complex& c)
  {
    re = c.re;
    im = c.im;
    cout << "Конструктор копирования" << endl;
  }
  // Сложение комплексных чисел
  Complex add(Complex& a, Complex& b)
  {
    Complex temporary;
    temporary.re = a.re + b.re;
    temporary.im = a.im + b.im;
    return temporary;
  }
  // Визуализация комплексного числа
  void print(const Complex& c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
};
```

```
int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  x1.print(x1);
  x2.print(x2);
  x1.print(x1.add(x1, x2));
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
Конструктор копирования
(9, 12)
```

Как видим, теперь конструктор копирования вызывается только в одном из трех возможных случаев инициализации объектов класса, а именно тогда, когда он инициализирует временный объект значением, которое возвращается компонентной функцией ***add*()**:

```
x1.print(x1.add(x1, x2));
```

Особый интерес здесь представляет сам вызов конструктора копирования (явный или неявный), но ответ на этот вопрос пока остается открытым. Далее будет сказано о роли спецификатора ***explicit*** при объявлении конструктора копирования и о возможных вариантах его вызова.

Однако, в тех особых случаях, когда компилятор, создавая такой временный объект, обходится без помощи конструктора копирования, то вместо одного вызова конструктора копирования может оказаться, что здесь не будет ни одного.

Напомним, что благодаря использованию параметра в виде ссылки функция ***print*()** получает доступ к той области памяти, где размещается сам объект. Константность ссылки здесь является гарантией того, что непреднамеренного изменения значения аргумента вызова не произойдет. В случае ссылки на объект, не являющегося ***lvalue***, лишь свойство ее константности обеспечивает инициализацию аргумента вызова.

➢ **Пример 15**

Для реализации полного контроля над процессом образования копии объекта во время выполнения инициализации пользователь вправе выбрать один из трех возможных способов определения своего собственного конструктора копирования. Конструктор копирования можно определить либо открытым компонентом класса, либо закрытым, либо открытым со спецификатором ***explicit***. Поначалу определим в классе ***Complex*** конструктор копирования закрытым компонентом класса (теперь будут запрещены все операции по конструированию значений, выполняемые конструктором копирования, кроме тех, которые выполняются им при создании временного объекта для хранения возвращаемого функцией значения):

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
  }
  // Сложение комплексных чисел
  Complex add(Complex& a, Complex& b)
  {
    Complex temporary;
    temporary.re = a.re + b.re;
    temporary.im = a.im + b.im;
    return temporary;
  }
  // Визуализация комплексного числа
  void print(const Complex& c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
  private:
  // Компонентные функции - все собственные (private)
  // Конструктор объектов класса - конструктор копирования
  Complex(const Complex& c)
  {
```

```
    re = c.re;
    im = c.im;
    cout << "Конструктор копирования" << endl;
  }
};

int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  x1.print(x1);
  x2.print(x2);
  x1.print(x1.add(x1, x2));
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
Конструктор копирования
(9, 12)
```

Как видим, закрытый конструктор копирования и здесь вызывается только в одном из трех возможных случаев инициализации объектов класса, а именно тогда, когда он инициализирует временный объект значением, которое возвращается компонентной функцией ***add*()**:

```
x1.print(x1.add(x1, x2));
```

Однако, в тех особых случаях, когда компилятор, создавая такой временный объект, обходится без помощи конструктора копирования, то вместо одного вызова конструктора копирования может оказаться, что здесь не будет ни одного.

Как будет сказано далее, реализация полного контроля над процессом образования копии объекта во время выполнения инициализации обязательно требует знания о форме вызова конструктора копирования. Отметим, что во всех случаях, когда конструктор копирования использовался для инициализации временного объекта значением, которое как раз и возвращалось компонентной функцией ***add*()**, нигде не оговаривалось, каким же образом вызывался конструктор копирования – явно или неявно. Следующие два примера наглядно демонстрируют, в какой мере свойство реализации компилятора для конкретной платформы определяет возможные способы конструирования значений таких временных объектов при помощи конструктора копирования.

➢ **Пример 16**

Итак, чтобы ответить на вопрос, каким же образом конструируются значения временных объектов, определим в классе ***Complex*** конструктор копирования открытым компонентом класса со спецификатором ***explicit*** (теперь будут разрешены только явные вызовы конструктора копирования при создании копий объектов и тем самым запрещены все неявные операции по конструированию значений, которые обычно выполняются конструктором копирования):

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
  }
  // Конструктор объектов класса - конструктор копирования
  explicit Complex(const Complex& c)
  {
    re = c.re;
    im = c.im;
    cout << "Конструктор копирования" << endl;
  }
  // Сложение комплексных чисел
  Complex add(Complex& a, Complex& b)
  {
    Complex temporary;
    temporary.re = a.re + b.re;
    temporary.im = a.im + b.im;
    return temporary;
  }
  // Визуализация комплексного числа
  void print(const Complex& c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
};
```

```
int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  x1.print(x1);
  x2.print(x2);
  x1.print(x1.add(x1, x2));
  Complex y(x1);
  y.print(y);
  return 0;
}
```

Результат работы программы для платформы Windows:

```
(-1, 5)
(10, 7)
Конструктор копирования
(9, 12)
Конструктор копирования
(-1, 5)
```

Как видим, здесь конструктор копирования явно вызывается только в двух из трех возможных случаев инициализации объектов класса, а именно тогда, когда он инициализирует временный объект значением, которое возвращается компонентной функцией ***add*()**, и когда он инициализирует объект ***y***:

```
  x1.print(x1.add(x1, x2));
  Complex y(x1);
```

Отметим, что лишь компиляторы для платформы Windows позволяют получить такой результат работы программы, что является доказательством явного вызова конструктора копирования для инициализации временного объекта значением, которое возвращается компонентной функцией ***add*()** при выполнении операции сложения двух комплексных чисел. Напротив, этот код уже нельзя скомпилировать для платформы Linux, так как ее компиляторы в этом случае вместо явного вызова конструктора копирования предлагают неявный вызов.

➢ **Пример 17**

Итак, порой вполне “безобидный” код может привести к неожиданным побочным эффектам времени компиляции. Одним из вариантов решения проблемы успешной компиляции кода для платформы Linux, например, может быть отказ от локальной переменной ***temporary*** компонентной функции ***add*()** и переход к ее статической переменной ***result***, что позволит этой функции вместо значения возвращать ссылку на эту переменную, как того требует для своего аргумента вызова функция ***print*()**:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
  }
  // Конструктор объектов класса - конструктор копирования
  explicit Complex(const Complex& c)
  {
    re = c.re;
    im = c.im;
    cout << "Конструктор копирования" << endl;
  }
  // Сложение комплексных чисел
  Complex& add(Complex& a, Complex& b)
  {
    static Complex result;
    result.re = a.re + b.re;
    result.im = a.im + b.im;
    return result;
  }
  // Визуализация комплексного числа
  void print(const Complex& c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
};
```

```
int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  x1.print(x1);
  x2.print(x2);
  x1.print(x1.add(x1, x2));
  Complex y(x1);
  y.print(y);
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(9, 12)
Конструктор копирования
(-1, 5)
```

Если локальная статическая переменная позволяет обычной функции “помнить о прошлом” этой переменной, то это справедливо и по отношению к компонентным функциям класса – их локальные статические переменные тоже “помнят о своем прошлом”. Поэтому их применение, как видим, может быть весьма полезным.

Далее будет сказано, как можно было бы еще и по-другому решить эту проблему, если воспользоваться указателем ***this***.

➢ Пример 18

Среди всех конструкторов класса с единственным параметром далеко не последняя роль принадлежит конструктору, параметр которого имеет тип, отличающийся от типа конструктора. Такой конструктор класса принято называть конструктором преобразования. Конструктор преобразования задает преобразование типа своего аргумента вызова в тип конструктора (по умолчанию неявное, если только это преобразование уникально).

Определим в классе ***Complex*** компонентную функцию – конструктор объектов класса, называемый конструктором преобразования, единственный аргумент вызова которого соответствует компоненту данных ***re***:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор преобразования
  Complex(double r)
  {
    re = r;
    im = 0.0;
    cout << "Конструктор преобразования" << endl;
  }
  // Сложение комплексных чисел
  Complex add(Complex& a, Complex& b)
  {
    Complex temporary(0);
    temporary.re = a.re + b.re;
    temporary.im = a.im + b.im;
    return temporary;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
};

int main()
{
```

```
  Complex x1(-1);
  Complex x2 = 10;
  x1.im = 5;
  x2.im = 7;
  x1.print(x1);
  x2.print(x2);
  x1.print(x1.add(x1, x2));
  return 0;
}
```

Результат работы программы:

```
Конструктор преобразования
Конструктор преобразования
(-1, 5)
(10, 7)
Конструктор преобразования
(9, 12)
```

Как видим, конструктор преобразования задает преобразование типа своего единственного аргумента вызова в тип конструктора:

```
  Complex x1(-1);          // явный вызов конструктора
  Complex x2 = 10;         // неявный вызов конструктора
  Complex temporary(0);    // явный вызов конструктора
```

Напомним, что конструктор преобразования, как и конструктор копирования, тоже позволяет инициализировать объект, используя оператор = в самом определении объекта. Здесь вначале создается временный объект, значение которого строится благодаря неявному вызову конструктора преобразования ***Complex*(*10*)**, которым затем инициализируется объект ***x2*** при помощи конструктора копирования по умолчанию, сгенерированного компилятором.

Поскольку здесь компилятором обязательно будет вызываться конструктор копирования по умолчанию, то необходимость в его переопределении со стороны пользователя класса, по крайней мере, в этом примере отпадает сама собой. С другой стороны, отказ от переопределения конструктора копирования по умолчанию здесь может быть оправдан также еще и стремлением избежать нежелательных побочных эффектов времени компиляции.

➢ Пример 19

Использование конструктора для преобразования типа удобно, но может иметь неприятные побочные эффекты, а в некоторых случаях неявное преобразование просто нежелательно, поэтому для его подавления следует определить конструктор со спецификатором ***explicit*** (такой конструктор всегда будет вызываться только явно). Учитывая, что и в этом случае компилятором обязательно будет вызываться конструктор копирования по умолчанию, то необходимость в его переопределении со стороны пользователя класса так же отпадает сама собой.

Определим в классе ***Complex*** теперь только один конструктор – конструктор преобразования со спецификатором ***explicit***:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор преобразования
  explicit Complex(double r)
  {
    re = r;
    im = 0.0;
    cout << "Конструктор преобразования" << endl;
  }
  // Сложение комплексных чисел
  Complex add(Complex a, Complex b)
  {
    Complex temporary(0);
    temporary.re = a.re + b.re;
    temporary.im = a.im + b.im;
    return temporary;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
};

int main()
{
```

```
  Complex x1(-1);
  Complex x2 = Complex(10);
  x1.im = 5;
  x2.im = 7;
  x1.print(x1);
  x2.print(x2);
  x1.print(x1.add(x1, x2));
  return 0;
}
```

Результат работы программы:

```
Конструктор преобразования
Конструктор преобразования
(-1, 5)
(10, 7)
Конструктор преобразования
(9, 12)
```

Как видим, конструктор преобразования задает только явное преобразование типа своего аргумента вызова в тип конструктора:

```
  Complex x1(-1);               // явный вызов конструктора
  Complex x2 = Complex(10);     // явный вызов конструктора
  Complex temporary(0);         // явный вызов конструктора
```

И напоследок о неудобствах использования конструктора преобразования в этих учебных примерах. Во-первых, речь идет о создании локального объекта ***temporary*** компонентной функции ***add*()** и его обязательной инициализации, что противоречит здравому смыслу, но, к сожалению, определено особенностями механизма вызова конструктора. Очевидно, что это неудобство связано как с реализацией операции сложения, так и с интерфейсом функции ***add*()**. И то, и другое легко изменить, но об этом позже. Во-вторых, после определения объекта класса должна осуществляться повторная инициализация его компонента данных ***im***, если только не полагаться на умолчание.

➢ Пример 20

Отметим, что конструктор по умолчанию класса ***Complex*** в случае вызова только первого аргумента может выступить также и в роли конструктора преобразования. Очевидно, что при этом не должен определяться сам конструктор преобразования, в противном случае будет иметь место ошибка времени компиляции.

Определим в классе ***Complex*** теперь только один конструктор – конструктор по умолчанию, все параметры которого имеют стандартные значения:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
    cout << "Конструктор по умолчанию" << endl;
  }
  // Сложение комплексных чисел
  Complex add(Complex a, Complex b)
  {
    Complex temporary;
    temporary.re = a.re + b.re;
    temporary.im = a.im + b.im;
    return temporary;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
};

int main()
{
  Complex x1(-1);
  Complex x2 = 10;
  Complex x3 = Complex(10);
```

```
  x1.im = 5;
  x2.im = 7;
  x3.im = 7;
  x1.print(x1);
  x2.print(x2);
  x3.print(x3);
  x1.print(x1.add(x1, x2));
  return 0;
}
```

Результат работы программы:

```
Конструктор по умолчанию
Конструктор по умолчанию
Конструктор по умолчанию
(-1, 5)
(10, 7)
(10, 7)
Конструктор по умолчанию
(9, 12)
```

Как видим, здесь конструктор по умолчанию задает преобразование типа своего первого аргумента вызова в тип конструктора:

```
  Complex x1(-1);              // явный вызов конструктора
  Complex x2 = 10;             // неявный вызов конструктора
  Complex x3 = Complex(10);    // явный вызов конструктора
  Complex temporary;           // явный вызов конструктора
```

Заметим здесь, что неявные вызовы конструктора по умолчанию можно запретить, если определить его со спецификатором ***explicit*** (такой конструктор всегда будет вызываться только явно). Однако в практике программирования такой прием следует отнести лишь к разряду учебных, например, хотя бы даже только для того, чтобы научиться различать явные и неявные вызовы как таковые. Далее, шаг за шагом, на примерах будет иллюстрироваться та подспудная роль конструктора по умолчанию, которая как раз и связана с его неявными вызовами.

➢ Пример 21

На первых порах инициализация компонентных данных осуществлялась при помощи операции присваивания в инструкциях тела конструктора класса. Однако определенным типам компонентных данных, например, константам или ссылкам, таким способом не могут быть присвоены значения. Для такого случая предусмотрен другой способ инициализации, основанный на применении списка инициализации компонентных данных объекта в определении конструктора. Очевидно, что таким образом можно инициализировать не только константы или ссылки, но и любые другие компонентные данные класса. Заметим, что инициализация компонентных данных здесь, как и прежде, осуществляется только при создании объекта класса. Заметим также, что такая форма определения конструктора класса может иногда приводить к его пустому телу.

Для инициализации константных компонентных данных определим в классе ***Complex*** конструктор по умолчанию со списком инициализации:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные данные - все общедоступные (public)
  const double re;
  const double im;
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Сложение комплексных чисел
  Complex add(Complex a, Complex b)
  {
    Complex temporary(a.re + b.re, a.im + b.im);
    return temporary;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
};

int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  Complex x3;
```

```
  x1.print(x1);
  x2.print(x2);
  x3.print(x3);
  x1.print(x1.add(x1, x2));
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(0, 0)
(9, 12)
```

Как видим, свойство константности компонентных данных класса, определяющее доступ к ним “только для чтения”, привело теперь к изменению и самой реализации операции сложения объектов с помощью компонентной функции ***add*()**. Далее будет сказано о проблеме “снятия” константности и возможных способах ее решения.

Заметим, что непонимание механизма инициализации компонентных данных объекта класса с помощью списка инициализации для его конструктора может привести к фатальным последствиям, но об этом уже в следующем примере.

➢ **Пример 22**

Представим класс ***Stack***, позволяющий реализовать стек произвольной длины в виде одномерного массива символьных объектов в свободной памяти:

```
// C++ Абстрактный тип данных - стек

#include <iostream>

using namespace std;

class Stack
{
  // Компонентные данные - все собственные (private)
  char* pointer;
  int top;
  int stackEmpty;
  int stackFull;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Stack(int stackSize = 100):
    stackEmpty(-1),
    stackFull(stackSize - 1)
  {
    pointer = new char[stackSize];
    top = stackEmpty;
    cout << "Конструктор по умолчанию" << endl;
  }
  // Вталкивание данных в стек
  void push(char symbol)
  {
    pointer[++top] = symbol;
  }
  // Выталкивание данных из стека
  char pop()
  {
    return pointer[top--];
  }
  // Проверить состояние стека - "пустой"
  bool empty()
  {
    return (top == stackEmpty);
  }
  // Проверить состояние стека - "заполнен"
  bool full()
  {
    return (top == stackFull);
  }
```

```
  // Деструктор объектов класса
  ~Stack()
  {
    delete[] pointer;
    cout << "Деструктор" << endl;
  }
};

int main()
{
  Stack stack(10);
  char line[] = "Hello, Hello!";
  int i = 0;
  cout << line << endl;
  // Вталкивание символов С-строки в стек
  while (line[i])
    if (!stack.full())
      stack.push(line[i++]);
    else ++i;
  // Выталкивание символов из стека
  while (!stack.empty()) cout << stack.pop();
  cout << endl;
  return 0;
}
```

Результат работы программы:

```
Конструктор по умолчанию
Hello, Hello!
leH ,olleH
Деструктор
```

Как видим, применение списка инициализации компонентных данных объекта в определении конструктора по умолчанию здесь вполне уместно.

А вот такое определение конструктора

```
  Stack(int stackSize = 100):
    stackEmpty(-1),
    stackFull(stackSize - 1),
    top(stackEmpty)
  {
    pointer = new char[stackSize];
  }
```

уже не может быть оправданным, так как значения компонентных данных объекта класса “проявляются” лишь в процессе его создания. Отсутствие ошибок времени компиляции – это еще не гарантия на отсутствие ошибок времени выполнения (здесь переменная ***top*** останется неинициализированной, что и приведет впоследствии к нежелательному побочному эффекту).

➢ **Пример 23**

Представим класс ***Array1D*** для иллюстрации корректного управления свободной памятью, выделяемой для динамических одномерных массивов:

```
// C++ Абстрактный тип данных – динамический одномерный массив

#include <iostream>

using namespace std;

class Array1D
{
  // Компонентные данные - все собственные (private)
  int* pointer;
  int arraySize;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Array1D(int size)
  {
    pointer = new int[size];
    arraySize = size;
  }
  // Конструктор объектов класса – конструктор копирования
  Array1D(const Array1D&);
  // Деструктор объектов класса
  ~Array1D()
  {
    delete[] pointer;
  }
  // Инициализация элементов массива
  void define();
  // Визуализация элементов массива
  void print();
};

// Конструктор объектов класса – конструктор копирования
Array1D::Array1D(const Array1D& c)
{
  pointer = new int[c.arraySize];
  arraySize = c.arraySize;
  for (int i = 0; i < c.arraySize; ++i)
    pointer[i] = c.pointer[i];
}

// Инициализация элементов массива
void Array1D::define()
{
```

```
  cout << "A(" << arraySize << ')' << endl;
  for (int i = 0; i < arraySize; ++i)
  {
    cout << "A[" << i << "]? ";
    cin >> pointer[i];
  }
}

// Визуализация элементов массива
void Array1D::print()
{
  for (int i = 0; i < arraySize; ++i)
    cout << pointer[i] << ' ';
  cout << endl;
}

int main()
{
  Array1D a(5);
  a.define();
  a.print();
  Array1D b = a;
  b.print();
  Array1D c(3);
  c.define();
  c.print();
  Array1D d(c);
  d.print();
  return 0;
}
```

Результат работы программы:

```
A(5)
A[0]? 1
A[1]? 2
A[2]? 3
A[3]? 4
A[4]? 5
1 2 3 4 5
1 2 3 4 5
A(3)
A[0]? 1
A[1]? 2
A[2]? 3
1 2 3
1 2 3
```

Компонентные данные и компонентные функции
Статические компоненты класса

➢ Пример 24

Чтобы компонентные данные класса были в единственном экземпляре и не тиражировались при определении каждого нового объекта класса, они должны быть объявлены в классе как статические (***static***). Память статическим компонентам данных класса выделяется только после их определения вне определения класса, причем в памяти они размещаются отдельно от нестатических компонентов класса.

Представим класс ***Complex***, в котором объявим два статических компонента – компонент данных ***counter*** для подсчета количества созданных в программе объектов класса ***Complex*** (конструктор будет осуществлять операцию инкрементирования, а деструктор – соответственно операцию декрементирования) и компонентную функцию ***get_counterValue*()**, которая будет возвращать значение счетчика ***counter***:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  static int counter;       // объявление статического компонента
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
    ++counter;
  }
  // Деструктор объектов класса
  ~Complex()
  {
    --counter;
  }
  // Доступ к счетчику
  static int get_counterValue()
  {
    return counter;
  }
};
```

```
int Complex::counter;       // определение статического компонента

int main()
{
  cout << "sizeof(Complex) = " << sizeof(Complex) << endl;
  cout << "Сколько объектов? " << Complex::counter << ':'
       << Complex::get_counterValue() << endl;
  Complex x1(-1, 5);
  cout << "sizeof x1 = " << sizeof x1 << endl;
  cout << "Сколько объектов? " << Complex::counter << ':'
       << Complex::get_counterValue() << ':'
       << x1.counter << ':' << x1.get_counterValue() << endl;
  Complex x2(10, 7);
  cout << "sizeof x2 = " << sizeof x2 << endl;
  cout << "Сколько объектов? " << Complex::counter << ':'
       << Complex::get_counterValue() << ':'
       << x2.counter << ':' << x2.get_counterValue() << endl;
  Complex* pointer = new Complex;
  cout << "Сколько объектов? " << Complex::counter << ':'
       << Complex::get_counterValue() << ':'
       << pointer->counter << ':' << (*pointer).counter << ':'
       << pointer->get_counterValue() << ':'
       << (*pointer).get_counterValue() << endl;
  delete pointer;
  cout << "Сколько объектов? " << Complex::counter << ':'
       << Complex::get_counterValue() << endl;
  return 0;
}
```

Результат работы программы:

```
sizeof(Complex) = 16
Сколько объектов? 0:0
sizeof x1 = 16
Сколько объектов? 1:1:1:1
sizeof x2 = 16
Сколько объектов? 2:2:2:2
Сколько объектов? 3:3:3:3:3:3
Сколько объектов? 2:2
```

➢ Пример 25

Статические компоненты данных – это просто глобальные переменные, область видимости которых ограничена классом, в котором они и были объявлены. Тем самым в C++ полностью отпадает необходимость в использовании глобальных переменных. При этом необходимо помнить, что на статические компоненты класса распространяются правила статуса доступа (объявление статического компонента можно поместить как в закрытом, так и в открытом разделах определения класса). Необходимо также еще помнить, что статические компонентные функции могут непосредственно ссылаться на статические компоненты только своего класса.

Представим класс ***Complex***, в закрытом разделе определения которого наряду с его компонентами данных ***re*** и ***im*** объявим еще и статический компонент ***counter*** для подсчета количества созданных в программе объектов класса ***Complex***:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

class Complex
{
  // Компонентные данные - все собственные (private)
  double re;
  double im;
  static int counter;       // объявление статического компонента
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
    ++counter;
  }
  // Деструктор объектов класса
  ~Complex()
  {
    --counter;
  }
  // Доступ к счетчику
  static int get_counterValue()
  {
    return counter;
  }
};

int Complex::counter;       // определение статического компонента
```

```
int main()
{
  cout << "sizeof(Complex) = " << sizeof(Complex) << endl;
  cout << "Сколько объектов? " << Complex::get_counterValue()
       << endl;
  Complex x1(-1, 5);
  cout << "sizeof x1 = " << sizeof x1 << endl;
  cout << "Сколько объектов? " << Complex::get_counterValue()
       << ':' << x1.get_counterValue() << endl;
  Complex x2(10, 7);
  cout << "sizeof x2 = " << sizeof x2 << endl;
  cout << "Сколько объектов? " << Complex::get_counterValue()
       << ':' << x2.get_counterValue() << endl;
  Complex* pointer = new Complex;
  cout << "Сколько объектов? " << Complex::get_counterValue()
       << ':' << pointer->get_counterValue() << ':'
       << (*pointer).get_counterValue() << endl;
  delete pointer;
  cout << "Сколько объектов? " << Complex::get_counterValue()
       << endl;
  return 0;
}
```

Результат работы программы:

```
sizeof(Complex) = 16
Сколько объектов? 0
sizeof x1 = 16
Сколько объектов? 1:1
sizeof x2 = 16
Сколько объектов? 2:2
Сколько объектов? 3:3:3
Сколько объектов? 2
```

Как видим, все операции непосредственного доступа к закрытому статическому компоненту класса ***counter*** стали теперь запрещены.

Указатели на компоненты класса

➢ Пример 26

Наряду с операторами выбора члена **.** и **->** в C++ есть еще два специфичных оператора выбора члена **.*** и **->***, которые предназначены для работы с указателями на общедоступные нестатические компоненты класса. Указатели на компоненты класса не могут адресовать никакого участка памяти, так как память выделяется не классу, а объектам этого класса при их создании. Указатель на компонент класса является значением, который всего лишь идентифицирует этот компонент. Мало того, даже после инициализации значение этого указателя остается неопределенным до тех пор, пока не будут выполнены операции обращения к компонентам класса с помощью операторов выбора члена **.*** или **->***.

Представим класс ***Complex*** для иллюстрации приемов работы с указателями на общедоступные нестатические компоненты класса:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
  }
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
};

int main()
{
  double(Complex::*dataMemberPointer) = &Complex::re;
  void(Complex::*memberFunctionPointer)() = &Complex::print;
  Complex x1(-1, 5);
  Complex* objectPointer = &x1;
```

```
  Complex x2;
  x1.print();
  x2.print();
  x1.*dataMemberPointer = 0;
  x2.*dataMemberPointer = -1;
  dataMemberPointer = &Complex::im;
  objectPointer->*dataMemberPointer = 0;
  objectPointer = &x2;
  objectPointer->*dataMemberPointer = 5;
  (x1.*memberFunctionPointer)();
  (objectPointer->*memberFunctionPointer)();
  return 0;
}
```

Результат работы программы для платформы Linux:

```
(-1, 5)
(0, 0)
(0, 0)
(-1, 5)
```

Как правило, при определении указателей на общедоступные нестатические компонентные функции вовсе необязательно пользоваться оператором **&** для получения адреса компонентной функции, например:

```
void(Complex::*memberFunctionPointer)() = Complex::print;
```

Однако, в тех редких случаях, когда имеет место ошибка времени компиляции, наличие этого оператора уже становится обязательным, например, как здесь:

```
void(Complex::*memberFunctionPointer)() = &Complex::print;
```

Одним из очевидных неудобств при объявлении указателей на нестатические компоненты класса в стиле C может являться излишняя громоздкость выражений. Далее будет сказано об одном из приемлемых способов решения этой проблемы, связанным с использованием возможности строить определения сложных типов при помощи ***typedef***.

➢ Пример 27

При объявлении указателей на общедоступные нестатические компоненты класса в C++ используется стиль C, принятый для объявления указателей на функцию, что нередко приводит к громоздкости выражений. Если стремиться к сохранению уже принятого стиля именования объектов программы в рассматриваемых учебных примерах, то здесь можно воспользоваться ***typedef*** для задания синонима типа.

Продолжим иллюстрацию приемов работы с указателями на общедоступные нестатические компоненты класса ***Complex***, строя определения сложных типов при помощи ***typedef***:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
};

typedef double(Complex::*dataPointer);
typedef void(Complex::*functionPointer)(Complex);

int main()
{
  dataPointer dataMemberPointer = &Complex::re;
  functionPointer memberFunctionPointer = &Complex::print;
  Complex x1(-1, 5);
  Complex* objectPointer = &x1;
  Complex x2;
  x1.print(x1);
  x2.print(x2);
```

```
  x1.*dataMemberPointer = 0;
  x2.*dataMemberPointer = -1;
  dataMemberPointer = &Complex::im;
  objectPointer->*dataMemberPointer = 0;
  objectPointer = &x2;
  objectPointer->*dataMemberPointer = 5;
  (x1.*memberFunctionPointer)(x1);
  (objectPointer->*memberFunctionPointer)(x2);
  return 0;
}
```

Результат работы программы для платформы Linux:

```
(-1, 5)
(0, 0)
(0, 0)
(-1, 5)
```

Как видим, использование ***typedef*** здесь не только облегчает определение сложных типов, но также позволяет еще сохранять стиль: одна строка – одна инструкция.

➢ Пример 28

Указатели на статические компоненты класса похожи на обычные указатели на переменную или функцию, так как статические компоненты класса не ассоциируются с конкретным объектом. Отличие лишь в том, что для их инициализации требуется полная квалификация инициализатора. Во всем остальном, что касается работы с указателями на статические компоненты класса, это ничем не отличается от работы с обычными указателями на переменную или функцию.

Продолжим иллюстрацию приемов работы с указателями на общедоступные статические компоненты класса ***Complex***, показывая все возможные способы доступа к этим компонентам:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  static int counter;       // объявление статического компонента
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
    ++counter;
  }
  // Деструктор объектов класса
  ~Complex()
  {
    --counter;
  }
  // Доступ к счетчику
  static int get_counterValue()
  {
    return counter;
  }
};

int Complex::counter;       // определение статического компонента

int main()
{
```

```
  int* dataPointer = &Complex::counter;
  int(*functionPointer)() = &Complex::get_counterValue;
  cout << "Сколько объектов? " << Complex::counter << ':'
       << Complex::get_counterValue() << ':'
       << *dataPointer << ':' << functionPointer() << ':'
       << (*functionPointer)() << endl;
  Complex x1(-1, 5);
  cout << "Сколько объектов? " << Complex::counter << ':'
       << Complex::get_counterValue() << ':'
       << x1.counter << ':' << x1.get_counterValue() << ':'
       << *dataPointer << ':' << functionPointer() << ':'
       << (*functionPointer)() << endl;
  Complex x2(10, 7);
  cout << "Сколько объектов? " << Complex::counter << ':'
       << Complex::get_counterValue() << ':'
       << x2.counter << ':' << x2.get_counterValue() << ':'
       << *dataPointer << ':' << functionPointer() << ':'
       << (*functionPointer)() << endl;
  Complex* pointer = new Complex;
  cout << "Сколько объектов? " << Complex::counter << ':'
       << Complex::get_counterValue() << ':'
       << pointer->counter << ':' << (*pointer).counter << ':'
       << pointer->get_counterValue() << ':'
       << (*pointer).get_counterValue() << ':'
       << *dataPointer << ':' << functionPointer() << ':'
       << (*functionPointer)() << endl;
  delete pointer;
  cout << "Сколько объектов? " << Complex::counter << ':'
       << Complex::get_counterValue() << ':'
       << *dataPointer << ':' << functionPointer() << ':'
       << (*functionPointer)() << endl;
  return 0;
}
```

Результат работы программы:

```
Сколько объектов? 0:0:0:0:0
Сколько объектов? 1:1:1:1:1:1:1
Сколько объектов? 2:2:2:2:2:2:2
Сколько объектов? 3:3:3:3:3:3:3:3:3
Сколько объектов? 2:2:2:2:2
```

➢ Пример 29

И в завершение приведем один из характерных приемов работы с указателем на общедоступные нестатические компоненты класса ***Complex***, когда с его помощью, например, можно организовать переключение между компонентными функциями ***get_re*()** и ***get_im*()**:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

class Complex
{
  // Компонентные данные - все собственные (private)
  double re;
  double im;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Сложение комплексных чисел
  Complex add(Complex a, Complex b)
  {
    Complex temporary;
    temporary.re = a.re + b.re;
    temporary.im = a.im + b.im;
    return temporary;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
  // Ссылка на вещественную часть комплексного числа
  double& get_re()
  {
    return re;
  }
  // Ссылка на мнимую часть комплексного числа
  double& get_im()
  {
    return im;
  }
};

typedef double&(Complex::*functionPointer)();
```

```
int main()
{
  Complex x1(-1, 5);
  Complex* objectPointer = &x1;
  Complex x2(10, 7);
  objectPointer->print(*objectPointer);
  x2.print(x2);
  objectPointer->print(objectPointer->add(*objectPointer, x2));
  functionPointer memberFunctionPointer = &Complex::get_re;
  (x1.*memberFunctionPointer)() += 10;
  memberFunctionPointer = &Complex::get_im;
  (objectPointer->*memberFunctionPointer)() += 7;
  x1.print(x1);
  return 0;
}
```

Результат работы программы для платформы Linux:

```
(-1, 5)
(10, 7)
(9, 12)
(9, 12)
```

Как видим, подобного рода переключатели для компонентных функций класса строятся достаточно просто, и пользователь вправе решить, как ему вызывать такие функции – непосредственно или при помощи указателей на компоненты класса.

С другой стороны, указатели на компонентные функции так же, как и указатели на обычные функции, часто используются тогда, когда возникает необходимость сослаться на функцию, имя которой заранее неизвестно. При этом особый интерес представляют именно виртуальные функции. Известно, что для каждого класса с виртуальными функциями компилятор строит свою таблицу указателей на его виртуальные функции (таблица виртуальных функций). Вызов виртуальной функции выполняется по ее индексу в таблице, соответствующей требуемому классу.

Определение компонентных функций

➢ Пример 30

Компонентные функции, которые определяются в пределах определения класса, являются неявно встроенными (***inline function***). Как правило, в классе должны определяться только небольшие по размеру и часто используемые компонентные функции. Так как не каждая компонентная функция может быть встроенной, то в классе такую функцию следует только объявить, а ее определение должно быть представлено после класса. Заметим, что наряду с внутренним возможно также и внешнее определение встраиваемых компонентных функций после класса, в котором были представлены их прототипы. Тогда спецификатор ***inline*** следует явно добавить либо в прототип функции, либо в ее внешнее определение, либо сделать то и другое одновременно. В случае внешнего определения компонентных функций, чтобы указать, к какому именно классу они относятся, требуется их полная квалификация.

Представим различные способы определения встраиваемых компонентных функций для класса ***Complex***:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  //   теперь уже невстраиваемая функция
  Complex(double, double);
  // Сложение комплексных чисел
  //   возможно, что еще будет встраиваемой функцией
  Complex add(Complex, Complex);
  // Визуализация комплексного числа
  //   встроенная функция
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
};

// Конструктор объектов класса – конструктор по умолчанию
Complex::Complex(double r = 0.0, double i = 0.0)
{
```

```
  re = r;
  im = i;
}

// Сложение комплексных чисел
inline Complex Complex::add(Complex a, Complex b)
{
  Complex temporary;
  temporary.re = a.re + b.re;
  temporary.im = a.im + b.im;
  return temporary;
}

int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  x1.print(x1);
  x2.print(x2);
  x1.print(x1.add(x1, x2));
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(9, 12)
```

Необходимо отметить здесь одно немаловажное обстоятельство, связанное с тем, что в случае внешнего определения компонентных функций компилятор может и проигнорировать указание по поводу их встраивания.

➢ Пример 31

В ряде случаев пользователь может объявить какие-либо компонентные функции константными. С одной стороны, это означает, что такие функции теперь не могут изменять компонентные данные класса. Далее будет сказано, что эта особенность определяется тем обстоятельством, что в константной компонентной функции класса ее указатель ***this***, который неявно передается всем нестатическим компонентным функциям, является константным указателем на константу, что как раз и предотвращает любое изменение состояния объектов этого класса. С другой стороны, константные компонентные функции можно вызывать как для константных, так и для неконстантных объектов класса, в то время как неконстантную компонентную функцию можно вызвать только для неконстантных объектов. Напомним, что объект называют константным, если при его определении используется квалификатор ***const***.

Представим класс ***Complex***, в котором определим две константные компонентные функции ***add*()** и ***print*()**:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
  }
  // Сложение комплексных чисел
  Complex add(Complex, Complex)const;
  // Связывание комплексного числа с другим комплексным числом
  void assign(Complex c)
  {
    re = c.re;
    im = c.im;
  }
  // Визуализация комплексного числа
  void print(Complex c)const
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
};
```

```
// Сложение комплексных чисел
inline Complex Complex::add(Complex a, Complex b)const
{
  Complex temporary;
  temporary.re = a.re + b.re;
  temporary.im = a.im + b.im;
  return temporary;
}

int main()
{
  const Complex x1(-1, 5);
  const Complex x2(10, 7);
  Complex y;
  x1.print(x1);
  x2.print(x2);
  y.print(y);
  x1.print(x1.add(x1, x2));
  y.assign(y.add(x1, x2));
  y.print(y);
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(0, 0)
(9, 12)
(9, 12)
```

Хотя конструкторы и деструкторы не являются константными компонентными функциями, как видим, они все же могут вызываться и для константных объектов класса. Напомним здесь правило, что объект становится константным сразу после того, как конструктор проинициализирует его, и перестает быть таковым, как только вызывается деструктор. Таким образом, объект класса с квалификатором ***const*** трактуется как константный с момента завершения работы конструктора и до вызова деструктора.

➢ Пример 32

"Снятие" ***const***, т.е. аннулирование действия квалификатора ***const*** является одним из характерных приемов при работе с какими-либо константными объектами. Одним из приемлемых решений проблемы "снятия константности" у компонентных данных класса является объявление их с квалификатором хранения ***mutable***, который указывает на то, что эти компоненты должны храниться таким способом, чтобы допускалась их модификация, даже если они являются компонентами константного объекта. Другими словами, квалификатор ***mutable*** означает: "ни при каких условиях не является константным".

Объявление имен с квалификатором ***mutable*** наиболее приемлемо, когда только часть представления класса может быть модифицирована, в то время как с логической точки зрения объект этого класса остается константным.

Проиллюстрируем теперь использование квалификатора хранения ***mutable*** для компонентных данных класса ***Complex***:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные данные - все общедоступные (public)
  mutable double re;
  mutable double im;
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
  }
  // Сложение комплексных чисел
  Complex add(Complex a, Complex b)const
  {
    Complex temporary;
    temporary.re = a.re + b.re;
    temporary.im = a.im + b.im;
    return temporary;
  }
  // Связывание комплексного числа с другим комплексным числом
  void assign(Complex c)const
  {
    re = c.re;
    im = c.im;
  }
```

```
  // Визуализация комплексного числа
  void print(Complex c)const
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
};

int main()
{
  const Complex x1(-1, 5);
  const Complex x2(10, 7);
  const Complex y;
  x1.print(x1);
  x2.print(x2);
  y.print(y);
  x1.print(x1.add(x1, x2));
  y.assign(y.add(x1, x2));
  y.print(y);
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(0, 0)
(9, 12)
(9, 12)
```

Как видим, совместное использование квалификатора хранения ***mutable*** для компонентных данных класса ***Complex*** и его константной компонентной функции ***assign*()** позволяет “снять константность” у компонентных данных ***re*** и ***im***. Далее будет сказано и о других способах решения этой проблемы, в основе которых лежит использование указателя ***this***.

Указатель this

➢ Пример 33

Когда вызывается какая-либо нестатическая компонентная функция для обработки компонентных данных конкретного объекта класса, этой функции неявно передается дополнительный (скрытый) аргумент вызова – специальный константный указатель на этот объект (***this***). В случае же вызова константной нестатической компонентной функции указатель ***this*** является константным указателем на константу. Тем самым компонентные функции знают, для каких объектов класса они были вызваны, поэтому могут явно на них ссылаться. Объект класса, который адресуется указателем ***this***, становится доступным внутри всех принадлежащих классу нестатических компонентных функций, однако, указатель ***this*** – это не совсем обычная переменная, так как невозможно получить ее адрес или присвоить ей что-нибудь.

Вернемся к решению проблемы "снятия константности" у компонентных данных класса, основанное на явном преобразовании типов (так называемое приведение типов) – **(*имя_типа*)*выражение***.

Представим класс ***Complex***, в котором его константная компонентная функция ***assign*()** осуществляет "снятие" ***const*** путем приведения типа **(*Complex**)*this***:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

class Complex
{
  // Компонентные данные - все собственные (private)
  double re;
  double im;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Связывание комплексного числа с другим комплексным числом
  void assign(Complex c)const
  {
    ((Complex*)this)->re = c.re;
    ((Complex*)this)->im = c.im;
  }
  // Визуализация комплексного числа
  void print(Complex c)const
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
};
```

```
int main()
{
  Complex x1(-1, 5);
  x1.print(x1);
  const Complex x2(10, 7);
  x2.print(x2);
  x2.assign(x1);
  x2.print(x2);
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(-1, 5)
```

Такого же “снятия” ***const*** можно добиться, например, и с помощью оператора приведения типа ***const_cast***, если в классе ***Complex*** его константную компонентную функцию ***assign*****()** определить так:

```
  void assign(Complex c)const
  {
    (const_cast<Complex*>(this))->re = c.re;
    (const_cast<Complex*>(this))->im = c.im;
  }
```

➢ Пример 34

И в завершение обратимся теперь к решению проблемы “снятия константности” у компонентных данных класса, основанное на использовании оператора приведения типа ***const_cast***. Практика, однако, показывает, что результат такого преобразования не всегда гарантирован.

Напомним, что нотация ***const_cast<имя_типа>(выражение)*** призвана заменить ***(имя_типа)выражение*** для преобразований, которым нужно получить доступ к данным с квалификатором ***const*** или ***volatile***. В ***const_cast<имя_типа>(выражение)*** тип параметра ***имя_типа*** должен совпадать с типом аргумента ***выражение*** во всем, кроме квалификаторов ***const*** или ***volatile***. Результатом этой операции будет то же значение, что у аргумента ***выражение***, только его типом станет ***имя_типа***.

Теперь представим класс ***Complex***, в котором его константная компонентная функция ***assign()*** осуществляет “снятие” ***const*** с помощью оператора приведения типа ***const_cast***:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

class Complex
{
  // Компонентные данные - все собственные (private)
  double re;
  double im;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса - конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Связывание комплексного числа с другим комплексным числом
  void assign(Complex c)const
  {
    (const_cast<Complex*>(this))->re = c.re;
    (const_cast<Complex*>(this))->im = c.im;
  }
  // Визуализация комплексного числа
  void print(Complex c)const
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
};

int main()
{
  Complex x1(-1, 5);
  x1.print(x1);
```

```
  const Complex x2(10, 7);
  x2.print(x2);
  x2.assign(x1);
  x2.print(x2);
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(-1, 5)
```

Как видим, явное использование указателя ***this*** для решения проблемы “снятия константности” у компонентных данных класса весьма полезно, а в некоторых случаях иногда просто незаменимо, например, когда внутри принадлежащей классу компонентной функции необходимо явно задать адрес объекта, для которого она была вызвана. Далее будет сказано, как с помощью указателя ***this*** решаются и другие проблемы. Так, одним из таких ярких примеров широко распространенного явного использования указателя ***this*** являются операции со связным списком из стандартной библиотеки C++. Указатель ***this*** также незаменим, например, и при выполнении некоторой последовательности операций с одним и тем же объектом класса.

➢ **Пример 35**

Обращаясь еще раз к двухсвязному списку в виде статической структуры данных, приведем определение класса ***List***, где его компонентной функции ***append*()** требуется явное использование указателя ***this*** при выполнении операции добавления элемента списка:

```
// C++ Абстрактный тип данных - двухсвязный список

#include <iostream>

using namespace std;

struct List
{
  // Компонентные данные – все общедоступные (public)
  List* predecessor;
  int data;
  List* successor;
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса - конструктор по умолчанию
  List(int item = 0)
  {
    predecessor = 0;
    data = item;
    successor = 0;
  }
  // Присоединение элемента к списку
  void append(List* pointer)
  {
    pointer->predecessor = this;
    successor = pointer;
  }
  // Визуализация элемента списка
  void print(List* pointer)
  {
    cout << pointer->data << endl;
  }
};

int main()
{
  List x1(1);           // первый элемент списка
  cout << &x1 << '\t';
  x1.print(&x1);
  List x2(2);           // второй элемент списка
  cout << &x2 << '\t';
  x2.print(&x2);
```

```
  x1.append(&x2);       // связывание первых двух элементов списка
  List x3(3);           // третий элемент списка
  cout << &x3 << '\t';
  x3.print(&x3);
  x2.append(&x3);       // присоединение элемента списка
  cout << x1.predecessor << endl;
  cout << x1.successor << endl;
  cout << x1.successor->successor << endl;
  cout << x1.successor->successor->successor << endl;
  cout << x2.predecessor << endl;
  cout << x2.predecessor->predecessor << endl;
  cout << x2.successor << endl;
  cout << x3.predecessor << endl;
  cout << x3.predecessor->predecessor << endl;
  cout << x3.predecessor->predecessor->predecessor << endl;
  cout << x3.successor << endl;
  return 0;
}
```

Результат работы программы:

```
0x8fb00ff6      1
0x8fb00fec      2
0x8fb00fe2      3
0x00000000
0x8fb00fec
0x8fb00fe2
0x00000000
0x8fb00ff6
0x00000000
0x8fb00fe2
0x8fb00fec
0x8fb00ff6
0x00000000
0x00000000
```

Как видим, чтобы присоединить элемент к двухсвязному списку, необходимо обновить объекты, на которые указывают указатели ***predecessor***, ***this*** и ***successor*** (предыдущий, текущий и следующий элементы списка). Напомним, что каждый элемент двухсвязного списка (кроме первого и последнего) должен быть связан со своим предыдущим и следующим элементом соответственно с помощью указателей ***predecessor*** и ***successor***.

➢ Пример 36

Теперь можно вернуться к той программе, код которой приводил к побочному эффекту времени компиляции для платформы Linux. Напомним здесь, что интерфейс компонентной функции ***add*()** оказался как раз тем камнем преткновения, о который и "разбился" пресловутый контроль над образованием копии временного объекта во время вызова конструктора копирования.

Представим определение класса ***Complex***, где его компонентная функция ***add*()** возвращает ссылку на тот объект, для которого она и вызывалась:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Сложение комплексных чисел
  Complex& add(Complex& a, Complex& b)
  {
    re = a.re + b.re;
    im = a.im + b.im;
    return *this;
  }
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
};

int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  x1.print();
  x2.print();
  x1.add(x1, x2);
  x1.print();
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(9, 12)
```

Если до этой поры ответ на вопрос, где и как будет храниться результат сложения объектов класса ***Complex***, а также как он затем в дальнейшем будет использоваться, всецело определялся “скудным” запасом средств по “обустраиванию” естественного поведения типа, то теперь, как видим, благодаря возможности явного использования указателя ***this*** этот запас пополнился весьма эффективным средством. Теперь этот результат можно будет хранить именно в том объекте класса, для которого и была вызвана функция ***add*()**. Удобно это или нет – другой вопрос, здесь важно, что теперь уже можно еще на один шаг приблизиться к естественному поведению операции сложения, когда ее результат присваивается первому операнду:

```
x1 = x1 + x2;
```

Далее будет сказано, как благодаря перегрузке стандартных операторов можно будет решить и остальные проблемы так называемой смешанной арифметики, где результат выполнения операций не будет зависеть от порядка расположения их операндов, которые к тому же могут быть и различных типов.

Как видим, в этом случае функции ***print*()** никакой аргумент вовсе и не нужен. От этого примера теперь легко перейти к следующему, где указатель ***this*** используется для выполнения последовательности операций с одним и тем же объектом класса ***Complex***. При этом не так важно, открытыми или закрытыми будут его компоненты данных. В дальнейшем будут приведены и другие примеры явного использования указателя ***this***.

➢ **Пример 37**

Явное использование указателя ***this*** просто незаменимо при выполнении некоторой последовательности операций с одним и тем же объектом класса ***Complex***:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса - конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Сложение комплексных чисел
  Complex& add(Complex& a, Complex& b)
  {
    re = a.re + b.re;
    im = a.im + b.im;
    return *this;
  }
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
};

int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  x1.print();
  x2.print();
  x1.add(x1, x2).print();
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(9, 12)
```

Друзья класса

- **Пример 38**

Расширить открытый интерфейс класса, т.е. предоставить доступ как к закрытым, так и к защищенным компонентам позволяют его друзья – ***дружественные функции*** (или ***функции-друзья***) и ***дружественные классы*** (или ***классы-друзья***).

Дружественные функции – это такие функции, которые, не являясь компонентами класса, предоставившего дружбу, имеют доступ к его собственным и защищенным компонентам, т.е. функции-друзья являются частью интерфейса класса в той же мере, в какой ею являются его компонентные функции. Для получения прав друга функции-друзья должны быть объявлены в теле класса, который предоставил им дружбу, со спецификатором ***friend***, причем объявление функции-друга можно поместить как в закрытом, так и в открытом разделах определения класса. При этом сами функции-друзья должны быть либо явно определены в охватывающей области видимости, либо иметь аргументы вызова класса, предоставившего им дружбу.

Представим класс ***Complex***, в закрытом разделе определения которого будут объявлены только его компонентные данные, для доступа к которым он объявляет своим другом глобальную функцию ***create*()**:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

class Complex
{
  // Компонентные данные - все собственные (private)
  double re;
  double im;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
  // Создание объектов в свободной памяти
  friend Complex* create(double, double);
};

// Создание объектов в свободной памяти
Complex* create(double r = 0.0, double i = 0.0)
{
```

```
  Complex* pointer = new Complex(r, i);
  cout << pointer << endl;
  cout << '(' << pointer->re << ", " << pointer->im << ")\n";
  pointer->print();
  return pointer;
}

int main()
{
  Complex x1(-1, 5);
  Complex* p = &x1;
  cout << p << endl;
  p->print();
  p = create();
  cout << p << endl;
  p->print();
  delete p;
  p = create(10, 7);
  cout << p << endl;
  p->print();
  delete p;
  return 0;
}
```

Результат работы программы:

```
0x8fb10fec
(-1, 5)
0x91420004
(0, 0)
(0, 0)
0x91420004
(0, 0)
0x91420004
(10, 7)
(10, 7)
0x91420004
(10, 7)
```

Отметим здесь, что поиск любой функции, в том числе и функции-друга, обычно осуществляется сначала в области видимости вызова, а затем – в пространствах имен каждого аргумента вызова (включая класс каждого аргумента и его базовые классы). В противном случае функцию-друга вызвать нельзя.

➢ **Пример 39**

В тех случаях, когда по какой-либо причине у класса нет открытого интерфейса, его функции-друзья, например, могут помочь построить этот интерфейс.

Представим класс ***Complex***, в закрытом разделе определения которого будут объявлены не только его компонентные данные, но и все его компонентные функции. Теперь, чтобы построить открытый интерфейс, объявим в классе ***Complex*** двух его друзей, например, глобальные функции ***create*()** и ***add*()**, каждая из которых помимо своей основной операции выполняет еще и операцию визуализации:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

class Complex
{
  // Закрытое представление класса (private)
  double re;
  double im;
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
  // Создание объектов в свободной памяти
  friend Complex* create(double, double);
  // Сложение комплексных чисел
  friend Complex* add(Complex*, Complex*);
};

// Создание объектов в свободной памяти
Complex* create(double r = 0.0, double i = 0.0)
{
  Complex* pointer = new Complex(r, i);
  pointer->print();
  return pointer;
}

// Сложение комплексных чисел
Complex* add(Complex* p, Complex* q)
{
  Complex* pointer = new Complex;
  pointer->re = p->re + q->re;
  pointer->im = p->im + q->im;
```

```
  pointer->print();
  return pointer;
}

int main()
{
  Complex* p = create(-1, 5);
  Complex* q = create(10, 7);
  Complex* r = add(p, q);
  delete p, q, r;
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(9, 12)
```

Как видим, чтобы построить открытый интерфейс, один из друзей класса ***Complex*** должен взять на себя роль “конструктора”, неявно вызывая настоящий конструктор объектов. Отметим, что такой подход к построению интерфейса непосредственно связан с последовательной реализацией принципа, что функции-друзья должны использоваться только для отражения тесно связанных концепций. В противном случае пользователь вправе остановить свой выбор на альтернативном (и пока единственном для себя) способе представления операций с объектами класса при помощи его компонентных функций.

Далее будет сказано и о другом альтернативном способе представления операций с объектами класса, если, например, обратиться к механизму перегрузки стандартных операторов. В частности, при помощи дружественной классу ***Complex*** операторной функции вставки можно полностью отказаться от компонентной функции ***print*()**, чтобы для рассматриваемых здесь двух функций-друзей ***create*()** и ***add*()** исключить несвойственную им операцию визуализации.

➢ **Пример 40**

Для иллюстрации возможных способов расширения открытого интерфейса класса рассмотрим теперь еще раз известную связку двух функций ***print*()** и ***add*()**, памятуя о стремлении к естественному поведению типа, т.е. когда его новые операции должны быть так же естественны, как если бы они были компонентами класса.

Представим класс ***Complex***, в закрытом разделе определения которого будут объявлены только его компонентные данные, для доступа к которым он объявляет своим другом глобальную функцию ***add*()**, возвращаемое значение которой, как и ранее, будет являться аргументом вызова открытой компонентной функции ***print*()**:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

class Complex
{
  // Компонентные данные - все собственные (private)
  double re;
  double im;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
  // Сложение комплексных чисел
  friend Complex add(Complex, Complex);
};

// Сложение комплексных чисел
Complex add(Complex a, Complex b)
{
  Complex temporary;
  temporary.re = a.re + b.re;
  temporary.im = a.im + b.im;
  return temporary;
}

int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
```

```
  x1.print(x1);
  x2.print(x2);
  x1.print(add(x1, x2));
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(9, 12)
```

Как видим, такое представление операций с объектами класса ***Complex*** в виде функции-друга ***add*()** выглядит вполне естественной альтернативой рассматриваемой ранее открытой компонентной функции ***add*()**. Тем самым наглядно демонстрируется, насколько важны детали внутренней реализации типа, которая является одной из составляющих основы первого принципа объектно-ориентированной парадигмы, называемого инкапсуляцией. Напомним здесь, что инкапсуляция включает не только детали внутренней реализации абстрактного типа данных, но и доступные извне операции и функции, которые могут оперировать объектами этого типа.

➢ Пример 41

В завершение представим теперь класс ***Array2D*** с внешним определением двух его компонентных функций (конструктора и деструктора) для реализации динамических двумерных массивов, при этом конструктору класса будет отведена роль, связанная лишь с выделением свободной памяти. В закрытом разделе определения класса, как и в предыдущем примере, будут объявлены только его компонентные данные, для доступа к которым он объявляет своими друзьями три глобальные функции – ***define*()** для инициализации элементов массива, ***print*()** для визуализации элементов массива и ***find_maximum*()** для поиска максимального элемента массива.

Напомним здесь, что указатель ***base*** – это указатель на указатель на ***double***, указатель ***base*** содержит начальный адрес массива указателей (***0, 1,*** …***, columnSize–1***), а каждый указатель этого массива – начальный адрес строки, состоящей из элементов ***double*** (***0, 1,*** …***, rowSize–1***):

```
// C++ Абстрактный тип данных - динамический двумерный массив

#include <iostream>
#include <iomanip>

using namespace std;

class Array2D
{
  // Компонентные данные - все собственные (private)
  double** base;
  int columnSize;
  int rowSize;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Array2D(int, int);
  // Деструктор объектов класса
  ~Array2D();
  // Инициализация элементов массива
  friend void define(Array2D&);
  // Визуализация элементов массива
  friend void print(Array2D&);
  // Поиск максимального элемента массива
  friend double find_maximum(Array2D&);
};

// Конструктор объектов класса
Array2D::Array2D(int arrayColumnSize, int arrayRowSize)
{
  base = new double*[arrayColumnSize];
  for (int i = 0; i < arrayColumnSize; ++i)
    base[i] = new double[arrayRowSize];
```

```
  rowSize = arrayRowSize;
  columnSize = arrayColumnSize;
}

// Деструктор объектов класса
Array2D::~Array2D()
{
  for (int i = 0; i < columnSize; ++i)
    delete[] base[i];
  delete[] base;
}

// Инициализация элементов массива
void define(Array2D& a)
{
  cout << "A(" << a.columnSize << 'x' << a.rowSize << ')' << endl;
  for (int i = 0; i < a.columnSize; ++i)
    for (int j = 0; j < a.rowSize; ++j)
    {
      cout << "A[" << i << ',' << j << "]? ";
      cin >> a.base[i][j];
    }
}

// Визуализация элементов массива
void print(Array2D& a)
{
  for (int i = 0; i < a.columnSize; ++i)
  {
    for (int j = 0; j < a.rowSize; ++j)
      cout << setw(4) << a.base[i][j];
    cout << endl;
  }
}

// Поиск максимального элемента массива
double find_maximum(Array2D& a)
{
  double maximum = a.base[0][0];
  for (int i = 0; i < a.columnSize; ++i)
    for (int j = 0; j < a.rowSize; ++j)
      if (a.base[i][j] > maximum) maximum = a.base[i][j];
  return maximum;
}

int main()
{
  Array2D a(2, 2);
  define(a);
```

```
  print(a);
  cout << "maximum = " << find_maximum(a) << endl;
  return 0;
}
```

Результат работы программы:

```
A(2x2)
A[0,0]? 1
A[0,1]? 2
A[1,0]? 3
A[1,1]? 4
   1   2
   3   4
maximum = 4
```

Как видим, такое представление операций с объектами класса ***Array2D*** в виде функций-друзей и здесь выглядит вполне естественной альтернативой. Главным в таком подходе должно быть стремление к созданию эффективного интерфейса для реализации минимально возможного доступа к представлению класса.

Заметим, что во избежание проблем, связанных с отсутствием в классе ***Array2D*** конструктора копирования, определяемого пользователем, функциям-друзьям ***define*****()**, ***print*****()** и ***find_maximum*****()** в качестве аргумента вызова вместо значения здесь передается ссылка на массив.

➢ Пример 42

Отметим, что функция-друг по отношению к классу, предоставившему ей дружбу, может быть компонентной функцией другого класса.

Рассмотрим пример для иллюстрации расширения открытого интерфейса класса ***Complex***, предоставившего дружбу компонентной функции ***add*()** класса ***Tools*** для доступа к своим закрытым компонентным данным:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

class Complex;
class Tools
{
  // Компонентные данные - все собственные (private)
  Complex& operand1;
  Complex& operand2;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Tools(Complex& a, Complex& b) : operand1(a), operand2(b) {}
  // Сложение комплексных чисел
  Complex add();
};

class Complex
{
  // Компонентные данные - все собственные (private)
  double re;
  double im;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
  // Сложение комплексных чисел
  friend Complex Tools::add();
};

Complex Tools::add()
{
```

```
  Complex temporary;
  temporary.re = operand1.re + operand2.re;
  temporary.im = operand1.im + operand2.im;
  return temporary;
}

int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  x1.print();
  x2.print();
  Tools t(x1, x2);
  Complex r = t.add();
  r.print();
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(9, 12)
```

Как видим, чтобы компонентная функция ***add*()** класса ***Tools*** могла быть объявлена другом класса ***Complex***, требуется выполнить два условия: во-первых, класс ***Complex***, который предоставляет дружбу функции ***add*()**, должен быть определен только после определения класса ***Tools***, и во-вторых, в классе ***Tools*** его компонентную функцию ***add*()** можно лишь объявить, а ее внешнее определение должно быть только после определения класса ***Complex***.

➢ Пример 43

Функция-друг также может быть дружественной по отношению и к нескольким классам. Однако создание такого интерфейса потребует немало усилий, если только при этом не стремиться к последовательной реализации принципа, что функция-друг должна использоваться только для отражения тесно связанных концепций.

Рассмотрим пример для иллюстрации расширения открытого интерфейса классов ***Complex*** и ***Tools***, предоставивших дружбу глобальной функции ***print*()** для доступа к своим закрытым компонентным данным:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

class Complex;
class Tools
{
  // Компонентные данные - все собственные (private)
  Complex& operand1;
  Complex& operand2;
  Complex& result;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Tools(Complex& a, Complex& b, Complex& c)
  : operand1(a), operand2(b), result(c) {}
  // Сложение комплексных чисел
  void add();
  // Визуализация комплексных чисел
  friend void print(Complex& a, Complex& b, Tools& c);
};

class Complex
{
  // Компонентные данные - все собственные (private)
  double re;
  double im;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Сложение комплексных чисел
  friend void Tools::add();
  // Визуализация комплексных чисел
  friend void print(Complex& a, Complex& b, Tools& c);
};
```

```
// Сложение комплексных чисел
void Tools::add()
{
  result.re = operand1.re + operand2.re;
  result.im = operand1.im + operand2.im;
}

// Визуализация комплексных чисел
void print(Complex& a, Complex& b, Tools& c)
{
  cout << '(' << a.re << ", " << a.im << ')' << endl;
  cout << '(' << b.re << ", " << b.im << ')' << endl;
  cout << '(' << c.result.re << ", " << c.result.im << ")\n";
}

int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  Complex r;
  Tools t(x1, x2, r);
  t.add();
  print(x1, x2, t);
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(9, 12)
```

➢ Пример 44

Дружественные классы – это такие классы, все компонентные функции которых являются друзьями другого класса, предоставившего дружбу, т.е. компоненты этого класса становятся доступными и всем компонентным функциям классов-друзей. Для получения прав друга классы-друзья, как и функции-друзья, должны быть объявлены в теле класса, предоставившего им дружбу, со спецификатором ***friend***. При этом сами классы-друзья должны быть либо предварительно объявлены в охватывающей области видимости, либо определены в области видимости, непосредственно охватывающей класс, предоставивший им дружбу, т.е. в одном пространстве имен. Обычно классы-друзья используются тогда, когда они не связаны отношением наследования с классом, предоставившим им дружбу.

Например, к закрытым компонентам класса ***Complex*** могут получить доступ все компонентные функции класса ***Tools***, если класс ***Complex*** объявит его своим другом:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

class Complex
{
  // Закрытый раздел определения класса (private)
  double re;
  double im;
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
  public:
  // Открытый раздел определения класса (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Класс-друг
  friend class Tools;
};

class Tools
{
  // Компонентные данные - все собственные (private)
  Complex operand1;
  Complex operand2;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Tools(Complex a, Complex b) : operand1(a), operand2(b) {}
```

```
  // Сложение комплексных чисел
  Complex add()
  {
    Complex temporary;
    temporary.re = operand1.re + operand2.re;
    temporary.im = operand1.im + operand2.im;
    return temporary;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    c.print();
  }
};

int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  Tools t(x1, x2);
  t.print(x1);
  t.print(x2);
  t.print(t.add());
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(9, 12)
```

➢ **Пример 45**

И завершим примером взаимной дружбы классов – класс ***Tools*** объявляется другом класса ***Complex***, а класс ***Complex***, в свою очередь, объявляется другом класса ***Tools***:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

class Tools;
class Complex
{
  // Закрытый раздел определения класса (private)
  double re;
  double im;
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
  public:
  // Открытый раздел определения класса (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Сложение комплексных чисел
  Complex& add(Tools&);
  // Класс-друг
  friend class Tools;
};

class Tools
{
  // Компонентные данные - все собственные (private)
  Complex operand1;
  Complex operand2;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Tools(Complex a, Complex b) : operand1(a), operand2(b) {}
  // Визуализация комплексного числа
  void print(Complex c)
  {
    c.print();
  }
  // Класс-друг
  friend class Complex;
};
```

```
// Сложение комплексных чисел
Complex& Complex::add(Tools& t)
{
  re = t.operand1.re + t.operand2.re;
  im = t.operand1.im + t.operand2.im;
  return *this;
}

int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  Complex r;
  Tools t(x1, x2);
  t.print(x1);
  t.print(x2);
  t.print(r.add(t));
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(9, 12)
```

Перегрузка стандартных операторов
Бинарные и унарные операторы

➢ Пример 46

Для распространения действия стандартного оператора на тип, определяемый пользователем, необходимо в классе определить специальную функцию, называемую ***операторной функцией***. Механизм перегрузки определяет новое поведение стандартных операторов языка, это возможно, если хотя бы один операнд для такого оператора является объектом класса. Операторные функции позволяют реализовать более привычную и удобную форму записи для манипулирования объектами классов по сравнению с той, которая прежде была доступна с использованием только базовой функциональной формы записи.

Количество параметров операторной функции зависит как от самого оператора, так и от способа определения самой функции. Так, бинарный оператор определяется для объектов класса либо как нестатическая компонентная функция с одним параметром, либо как глобальная или дружественная функция с двумя параметрами. Унарный оператор определяется для объектов класса либо как нестатическая компонентная функция без параметров, либо как глобальная или дружественная функция с одним параметром.

Например, распространим действия трех бинарных операторов (**+**, **-** и *****) на объекты класса ***Complex***, определив ***operator*+()** как дружественную функцию класса, ***operator*-()** как нестатическую компонентную функцию, ***operator**()** как глобальную функцию с параметрами типа класс:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Ссылка на вещественную часть комплексного числа
  double& get_re()
  {
    return re;
  }
  // Ссылка на мнимую часть комплексного числа
  double& get_im()
  {
    return im;
  }
```

```
  // Сложение комплексных чисел
  friend Complex operator+(Complex, Complex);
  // Вычитание комплексных чисел
  Complex operator-(Complex c)
  {
    Complex temporary;
    temporary.re = re - c.re;
    temporary.im = im - c.im;
    return temporary;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
  private:
  // Компонентные данные - все собственные (private)
  double re;
  double im;
};

// Сложение комплексных чисел
Complex operator+(Complex a, Complex b)
{
  Complex temporary;
  temporary.re = a.re + b.re;
  temporary.im = a.im + b.im;
  return temporary;
}

// Умножение комплексных чисел
Complex operator*(Complex a, Complex b)
{
  Complex temporary;
  temporary.get_re() = a.get_re() * b.get_re() -
    a.get_im() * b.get_im();
  temporary.get_im() = a.get_re() * b.get_im() +
    b.get_re() * a.get_im();
  return temporary;
}

int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  x1.print(x1);
  x2.print(x2);
  x1.print(x1 + x2);
  x1.print(operator+(x1, x2));
```

```
  x2 = x1 + x2;
  x2.print(x2);
  x1 = operator+(x1, x2);
  x1.print(x1);
  x2.print(x2 + 10);
  x1.print(-10 + x1);
  x1.print(x1 - x2);
  x1.print(x1.operator-(x2));
  x1 = x1 - x2;
  x1.print(x1);
  x2.print(x2 - 10);
  x2 = x2.operator-(x1);
  x2.print(x2);
  x1 = x1 * x2;
  x1.print(x1);
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(9, 12)
(9, 12)
(9, 12)
(8, 17)
(19, 12)
(-2, 17)
(-1, 5)
(-1, 5)
(-1, 5)
(-1, 12)
(10, 7)
(-45, 43)
```

➢ **Пример 47**

Согласно принятому соглашению при распространении действия постфиксных операторов инкремента и декремента, которые в то же время могут быть и префиксными, операторные функции должны иметь еще один дополнительный параметр типа ***int***, значение которого компилятор определяет равным нулю.

Например, распространим действия двух унарных операторов (префиксный **++** и постфиксный **--**) на объекты класса ***Complex***, определив ***operator*++()** и ***operator*--()** как нестатические компонентные функции, полагая, что операции инкремента и декремента будут определены только для вещественной части комплексного числа:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0):re(r), im(i) {}
  // Инкремент вещественной части комплексного числа
  Complex& operator++()
  {
    ++re;
    return *this;
  }
  // Декремент вещественной части комплексного числа
  Complex operator--(int)
  {
    Complex temporary = *this;
    --re;
    return temporary;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
  private:
  // Компонентные данные - все собственные (private)
  double re;
  double im;
};

int main()
{
```

```
  Complex x1(-1, 5);
  Complex x2(10, 7);
  x1.print(x1);
  x2.print(x2);
  x1.print(++x1);
  x1.print(x1);
  x2 = ++++x1;
  x2.print(x2);
  x1.print(x1);
  x1.print(x1--);
  x1.print(x1);
  x2 = x1--;
  x2.print(x2);
  x1.print(x1);
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(0, 5)
(0, 5)
(2, 5)
(2, 5)
(2, 5)
(1, 5)
(1, 5)
(0, 5)
```

Чтобы запретить здесь многократную операцию инкремента для комплексного числа, операторную функцию необходимо определить теперь так:

```
  const Complex& operator++()
  {
    ++re;
    return *this;
  }
```

Напомним здесь, что если какая-либо функция возвращает указатель или ссылку, то добавление квалификатора ***const*** к спецификации возвращаемого значения в определении этой функции означает, что она не может использовать возвращаемое значение для последующего изменения значения переменной, на которую указывает или ссылается эта функция.

Смешанная арифметика

➢ Пример 48

В терминологии языка Fortran операторы, которые работают с операндами разного типа, реализуют так называемую ***смешанную арифметику***.

Смешанную арифметику, например, для комплексных чисел легко реализовать, если, во-первых, свести к минимуму количество функций, непосредственно манипулирующих представлением объекта класса. Этого можно добиться благодаря определению в нем только тех операторов, которые будут модифицировать значение первого параметра, например, таких как **+=**. Во-вторых, определить вне класса в каком-либо пространстве имен (возможно в глобальном) операторные функции, которые просто вычисляют новое значение на основе своих параметров, например, такие как **+**, и, в-третьих, определить арифметические операторы, которые бы работали с операндами разного типа.

Например, представим определение класса ***Complex***, в котором смешанная арифметика будет реализована только для одной операции сложения с операндами типа ***Complex*** и встроенного типа ***double***:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Сложение комплексных чисел
  Complex& operator+=(Complex c)
  {
    re += c.re;
    im += c.im;
    return *this;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
  private:
  // Компонентные данные - все собственные (private)
  double re;
  double im;
};
```

```
// Сложение комплексных чисел
Complex operator+(Complex a, Complex b)
{
  Complex temporary = a;
  return temporary += b;
}

// Сложение комплексного числа с действительным числом
Complex operator+(Complex a, double b)
{
  Complex temporary = a;
  return temporary += b;
}

// Сложение действительного числа с комплексным числом
Complex operator+(double a, Complex b)
{
  Complex temporary = b;
  return temporary += a;
}

int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  x1.print(x1);
  x2.print(x2);
  x1.print(x1 + x2);
  x1 = x1 + 2;
  x1.print(x1);
  x2 = 2 + x2;
  x2.print(x2);
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(9, 12)
(1, 5)
(12, 7)
```

Вывод

➢ Пример 49

В C++ вывод иногда называют вставкой (***insertion***), а перегруженный оператор сдвига влево **<<** – соответственно оператором вставки (***inserter***). Операция вывода вставляет данные различных типов в поток вывода. Оператор вывода позволяет избежать многословности, которая получается при использовании функций вывода. Когда пользователь перегружает стандартный оператор **<<** для вывода, он создает операторную функцию вставки (глобальную или дружественную функцию с двумя параметрами). В большинстве случаев функция вставки является дружественной классу, для которого она создавалась, а его объект, вставляемый в поток, обычно передается ей в виде ссылки на константу.

Представим определение класса ***Complex***, в котором смешанная арифметика, как и ранее, будет реализована только для одной операции сложения с операндами типа ***Complex*** и встроенного типа ***double***, а операция вывода ее результата здесь будет реализована как дружественная операторная функция вставки, которой он передается в виде ссылки на константу:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Сложение комплексных чисел
  Complex& operator+=(Complex c)
  {
    re += c.re;
    im += c.im;
    return *this;
  }
  // Визуализация комплексного числа
  friend ostream& operator<<(ostream& stream, const Complex& c)
  {
    return stream << '(' << c.re << ", " << c.im << ')';
  }
  private:
  // Компонентные данные - все собственные (private)
  double re;
  double im;
};
```

```
// Сложение комплексных чисел
Complex operator+(Complex a, Complex b)
{
  Complex temporary = a;
  return temporary += b;
}

// Сложение комплексного числа с действительным числом
Complex operator+(Complex a, double b)
{
  Complex temporary = a;
  return temporary += b;
}

// Сложение действительного числа с комплексным числом
Complex operator+(double a, Complex b)
{
  Complex temporary = b;
  return temporary += a;
}

int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  cout << x1 << endl;
  cout << x2 << endl;
  cout << x1 + x2 << endl;
  x1 = x1 + 2;
  cout << x1 << endl;
  x2 = 2 + x2;
  cout << x2 << endl;
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(9, 12)
(1, 5)
(12, 7)
```

Копирующее присваивание

➢ Пример 50

Ранее уже было сказано, что операция копирования объектов класса требует особого внимания со стороны пользователя класса. Отсутствие контроля при выполнении этой операции может привести к нежелательным и порой неожиданным эффектам. Напомним, что значение одного объекта может быть передано другому лишь в двух случаях – при присваивании и при инициализации. Напомним также, что выражение присваивания для объектов класса по умолчанию означает почленное копирование данных одного объекта в другой.

Следует помнить также, что такое копирование обычно является неправильным при копировании объектов, имеющих ресурсы, управляемые конструктором и деструктором. Не менее катастрофичной такая ситуация наблюдается, например, при наличии в классе членов, являющихся указателями или ссылками. Кроме того, присваивание по умолчанию не может быть сгенерировано, если нестатический компонент класса является ссылкой, константой или типом, определяемым пользователем, не имеющим копирующего оператора присваивания.

Отметим, что семантику присваивания и копирования по умолчанию зачастую называют поверхностным копированием, так как при этом копируются только члены класса, но не те объекты, на которые эти члены указывают. Рекурсивное копирование указываемых объектов (так называемое глубокое копирование) пользователю следует определять явно.

Представим класс ***Complex***, объекты которого копируются при помощи копирующего оператора присваивания, реализующего механизм защиты от присваивания самому себе:

```
// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

class Complex
{
  // Компонентные данные - все собственные (private)
  double re;
  double im;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Визуализация комплексного числа
  friend ostream& operator<<(ostream& stream, const Complex& c)
  {
    return stream << '(' << c.re << ", " << c.im << ')';
  }
```

```
  // Копирующее присваивание
  Complex& operator=(const Complex& c)
  {
    if (this != &c)
    {
      re = c.re;
      im = c.im;
    }
    return *this;
  }
};

int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  Complex x3(9, 12);
  cout << x1 << ' ' << x2 << ' ' << x3 << endl;
  x1 = x2;
  cout << x1 << endl;
  x2 = x3 = x1;
  cout << x1 << ' ' << x2 << ' ' << x3 << endl;
  return 0;
}
```

Результат работы программы:

```
(-1, 5) (10, 7) (9, 12)
(10, 7)
(10, 7) (10, 7) (10, 7)
```

Вызов функции

➢ Пример 51

Одним из наиболее очевидных и в то же время важных приемов использования перегруженного бинарного оператора вызова функции **()** является предоставление синтаксиса стандартного вызова функций для объектов класса, которые в некотором смысле ведут себя как функции – так называемые ***объекты-функции*** или ***функторы***. Класс, в котором определен оператор вызова функции, называется ***функциональным***.

Объекты-функции обеспечивают механизм, посредством которого пользователь может приспособить стандартные алгоритмы библиотеки C++ для работы со своими данными. Алгоритмами называют функции, выполняющие некоторые стандартные действия, например, поиск, копирование, слияние. Ряд обобщенных алгоритмов стандартной библиотеки требуют функций в качестве аргументов. Одним из простых примеров служит обобщенный алгоритм ***for_each*()**, который вызывает функцию или объект-функцию, переданную ему в качестве одного из аргументов, для каждого элемента последовательности. Отметим здесь, что представление данных в виде последовательности объектов – как содержимого некоторого контейнера – является одной из тех концепций, что составляют основу проектирования шаблонов стандартной библиотеки C++. Доступ к элементам последовательности реализуется с помощью итераторов, представляющих собой абстракцию понятия указателя.

Отметим также, что для иллюстрации ряда примеров, связанных с использованием стандартных алгоритмов библиотеки, аргументами которых могут быть итераторы и объекты-функции, автор вынужден здесь приводить фрагменты кода, характерные только для обобщенной парадигмы программирования.

Прежде чем перейти к функциональным классам, сначала рассмотрим пример, который составит основу перехода от традиционной функции, используемой для каждого элемента последовательности, к объекту-функции.

Например, визуализацию элементов одномерного массива, значение которых превышает некоторый заданный порог, можно осуществить с помощью алгоритма ***for_each*()**, если ему в качестве последнего его аргумента передать указатель на функцию ***greater_then*()**, а в качестве первых двух его аргументов (которые в общем случае являются итераторами ввода) передать указатели – соответственно на первый элемент массива и за его последний элемент:

```
// C++ Иллюстрация стандартного алгоритма библиотеки for_each()

#include <iostream>
#include <algorithm>

using namespace std;

void greater_then(int data)
{
  if (data > 5) cout << data << ' ';
}
```

```
int main()
{
  int a[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  for_each(a, a + 10, greater_then);
  cout << endl;
  return 0;
}
```

Результат работы программы:

```
6 7 8 9
```

Как видим, стандартный алгоритм ***for_each*()** последовательно друг за другом берет элементы массива (в общем случае стандартного контейнера), на которые ссылается указатель (в общем случае итератор ввода), и передает их как аргумент функции ***greater_then*()**, что в целом позволяет отказаться от явной формы организации цикла. Отметим, что ситуация, когда библиотечная функция ***for_each*()** вызывает функцию пользователя ***greater_then*()**, называется ***обратным вызовом*** (от слова ***callback***).

Напомним, что к синтаксису вызова функции помимо объектов-функций приводит использование и других конструкций языка: функций, функциеобразных макросов, указателей на функции, ссылок на функции, функций-членов и указателей на них, среди которых функторами можно считать лишь указатели на функции.

➢ **Пример 52**

Отметим, что стандартная библиотека C++ предоставляет множество полезных объектов-функций – логические предикаты (функторы, возвращающие булево значение), арифметические операции и адаптеры (унарные и бинарные функторы). Объекты-функции (как стандартные, так и созданные самим пользователем) позволяют записывать код с использованием нетривиальных операций в качестве параметра. Зачастую перегрузка оператора вызова функции оказывается весьма полезной для определения типов с одной единственной операцией или типов с одной главной операцией.

Рассмотрим три функциональных класса, объекты-функции которых реализуют единственные для каждого определяемого пользователем типа данных операции, например, сравнения и сложения в целочисленной и комплексной арифметике.

Представим первый функциональный класс – ***GreaterThen***, объект-функция которого реализует операцию сравнения данных встроенного типа ***int***:

```
// C++ Функциональный класс

#include <iostream>

using namespace std;

struct GreaterThen
{
  // Компонентные функции - все общедоступные (public)
  // Сравнение целых чисел
  int operator()(int a, int b)const
  {
    return a > b;
  }
};

int main()
{
  GreaterThen x;
  cout << x(-1, 5) << endl;
  cout << x(5, -1) << endl;
  return 0;
}
```

Результат работы программы:

```
0
1
```

Как видим, от такого функционального класса даже не требуется наличия других его компонентов.

➢ **Пример 53**

От определения функционального класса ***GreaterThen*** теперь легко перейти к определению логического предиката ***GreaterThen*** – функции общего вида, которая выполняет операцию целочисленного сравнения с некоторым пороговым значением, определяемым при создании экземпляра класса ***GreaterThen*** его конструктором.

Представим второй функциональный класс, объект-функция которого является логическим предикатом:

```
// C++ Функциональный класс

#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

class GreaterThen
{
  // Компонентные данные - все собственные (private)
  const int value;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  GreaterThen(int data) : value(data) {}
  // Сравнение целых чисел
  bool operator()(int data)const
  {
    return data > value;
  }
};

int main()
{
  // Объявление стандартного контейнера – пустой вектор элементов
  vector<int> v;
  // Инициализация вектора - добавление элементов в контейнер
  for (int i = 1; i <= 5; ++i) v.push_back(i);
  // Визуализация вектора – копирование элементов в поток вывода
  copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
  cout << endl;
  // Объявление итератора произвольного доступа
  vector<int>::iterator first;
  // Поиск первого элемента, удовлетворяющего предикату
  first = find_if(v.begin(), v.end(), GreaterThen(2));
  if (first != v.end()) cout << *first << endl;
  return 0;
}
```

Результат работы программы для платформы Windows:

```
1 2 3 4 5
3
```

Как видим, определить предикат не трудно, а как только он введен для типов, определяемых пользователем, применение этих типов со стандартными алгоритмами становится уже весьма простым делом. Вместе с тем, даже этот код является яркой иллюстрацией выразительной мощи обобщенной парадигмы программирования.

Каждый алгоритм стандартной библиотеки выражается шаблоном функции или набором шаблонов функций. Таким образом, алгоритм может работать с разными последовательностями, содержащими значения разнообразных типов.

Тип функций ***begin*()** и ***end*()** принадлежит к категории итераторов ввода. Итератор, возвращаемый ***begin*()**, указывает на первый элемент последовательности, а итератор, возвращаемый ***end*()**, указывает на элемент, следующий за последним.

Стандартный контейнер ***vector*** – это шаблон класса. Стандартный вектор является одним из примеров абстрактного представления последовательности, для которой характерно использование операций со стеком. Например, функция-член ***push_back*()** добавляет элемент в конец последовательности.

Алгоритм ***copy*()** копирует одну последовательность, начиная с первого элемента, в другую, причем результат копирующего алгоритма не обязан быть контейнером. Здесь результатом является выходной поток ***cout*** в виде последовательности, для которой указывается итератор вывода ***ostream_iterator***.

Алгоритм ***find_if*()** находит первое соответствие предикату ***GreaterThen*(*2*)** в последовательности. Результатом поиска является либо итератор на найденный элемент, либо итератор, возвращаемый ***end*()**.

Отметим здесь, что лишь компиляторы для платформы Windows сразу позволяют получить такой результат работы программы, а для платформы Linux этот код уже нельзя скомпилировать, так как ее компиляторы к указанным заголовочным файлам предлагают подключить еще один файл – ***<iterator>***, в котором находится объявление стандартного шаблона класса для итератора вывода ***ostream_iterator***.

➢ **Пример 54**

И в завершение представим теперь третий функциональный класс – ***Complex***, объект-функция которого реализует операцию сложения данных типа ***Complex***:

```
// C++ Функциональный класс

#include <iostream>

using namespace std;

class Complex
{
  // Компонентные данные - все собственные (private)
  double re;
  double im;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Визуализация комплексного числа
  friend ostream& operator<<(ostream& stream, const Complex& c)
  {
    return stream << '(' << c.re << ", " << c.im << ')';
  }
  // Прибавление комплексного числа
  Complex& operator()(Complex c)
  {
    re += c.re;
    im += c.im;
    return *this;
  }
  // Прибавление комплексного числа
  Complex& operator()(double a, double b)
  {
    re += a;
    im += b;
    return *this;
  }
  // Прибавление комплексного числа
  Complex& operator()(double a)
  {
    re += a;
    return *this;
  }
};

int main()
{
```

```
  Complex x1(-1, 5);
  Complex x2(10, 7);
  cout << x1 << endl;
  cout << x2 << endl;
  cout << x1(x2) << endl;
  cout << x1(-10, -7) << endl;
  cout << x1(10) << endl;
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(9, 12)
(-1, 5)
(9, 5)
```

Как видим, перегрузка оператора вызова функции позволяет реализовать даже такой непривычный подход к выполнению операций комплексной арифметики. Здесь лишь необходимо помнить, что подобным образом можно реализовать только одну единственную операцию.

Индексация

➢ Пример 55

Как в C, так и в C++ во время выполнения программы можно выйти за границы диапазона встроенного массива без генерации сообщений об ошибках времени выполнения. Класс ***vector*** из стандартной библиотеки по умолчанию так же не обеспечивает проверку диапазона вектора. Для создания защищенного одномерного массива, например, можно определить класс, который позволяет объявить этот массив и разрешить доступ к его элементам только через перегруженный оператор индексации **[]**. С помощью операторной функции ***operator*[]()** теперь можно будет перехватывать индекс, который выходит за границы диапазона массива.

Приведем определение класса ***Array1D*** для защищенного одномерного массива, элементы которого принадлежат встроенному типу ***int***:

```
// C++ Абстрактный тип данных - защищенный одномерный массив

#include <iostream>
#include <cstdlib>
#include <iomanip>

using namespace std;

class Array1D
{
  // Компонентные данные - все собственные (private)
  int* base1D;
  int size1D;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Array1D(int);
  // Деструктор объектов класса
  ~Array1D();
  // Индексация массива
  int& operator[](int);
};

// Конструктор объектов класса
Array1D::Array1D(int size)
{
  base1D = new int[size];
  size1D = size;
}

// Деструктор объектов класса
Array1D::~Array1D()
{
```

```
  delete[] base1D;
}

// Индексация массива
int& Array1D::operator[](int index)
{
  if (index < 0 || index > size1D - 1)
  {
    cout << "Индекс за границами диапазона!" << endl;
    exit(1);
  }
  return base1D[index];
}

int main()
{
  Array1D a(5);
  // Инициализация и визуализация элементов массива
  for (int i = 0; i < 5; ++i)
  {
    a[i] = i;
    cout << setw(3) << a[i];
  }
  cout << endl;
  return 0;
}
```

Результат работы программы:

```
  0  1  2  3  4
```

Как видим, после определения объекта класса ***Array1D*** все операции индексации массива ***a*** будут ограничены диапазоном изменения индекса от ***0*** до ***4***.

Отметим здесь, что операции индексации для индекса, выходящего за границы допустимого диапазона, как правило, должны приводить к аварийному завершению программы.

➢ Пример 56

Для создания защищенного двумерного массива необходимо воспользоваться особенностями механизма индексации многомерных массивов – определить класс, который позволяет объявить этот массив, и разрешить доступ к его элементам только через перегруженный оператор **[]**. Например, это можно сделать, определив класс ***Array2D*** таким образом, чтобы его операторная функция ***operator*[]()** перехватывала бы индекс, выходящий за границы первой размерности массива, и возвращала бы ссылку на объект вложенного класса ***Array1D***. В свою очередь, операторная функция ***operator*[]()** вложенного класса ***Array1D*** перехватывала бы индекс, выходящий за границы второй размерности массива, и возвращала бы ссылку на элемент массива.

Пользователи класса ***Array2D*** не должны знать о существовании класса ***Array1D***, объекты класса ***Array1D*** представляют собой одномерные массивы и называются proxy-объектами (от слов ***proxy objects***). Соответственно, классы, объекты которых являются proxy-объектами, называются proxy-классами (от слов ***proxy classes***).

Приведем определение класса ***Array2D*** для защищенного двумерного массива, элементы которого принадлежат встроенному типу ***int***:

```
// C++ Абстрактный тип данных - защищенный двумерный массив

#include <iostream>
#include <cstdlib>
#include <iomanip>

using namespace std;

class Array2D
{
  class Array1D
  {
    // Компонентные данные - все собственные (private)
    int* base1D;
    int size1D;
    public:
    // Компонентные функции - все общедоступные (public)
    // Конструктор объектов класса – конструктор по умолчанию
    Array1D();
    // Деструктор объектов класса
    ~Array1D()
    {
      delete[] base1D;
    }
    // Выделение свободной памяти для одномерного массива
    void allocate1D(int);
    // Индексация массива
    int& operator[](int);
  };
```

```
  // Компонентные данные - все собственные (private)
  Array1D* base2D;
  int size2D;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Array2D(int, int);
  // Деструктор объектов класса
  ~Array2D()
  {
    delete[] base2D;
  }
  // Индексация массива
  Array1D& operator[](int);
};

// Конструктор объектов класса Array2D
Array2D::Array2D(int columnSize, int rowSize)
{
  base2D = new Array1D[columnSize];
  for (int i = 0; i < columnSize; ++i)
    base2D[i].allocate1D(rowSize);
  size2D = columnSize;
}

// Индексация массива
Array2D::Array1D& Array2D::operator[](int index)
{
  if (index < 0 || index > size2D - 1)
  {
    cout << "Первый индекс за границами диапазона!" << endl;
    exit(1);
  }
  return base2D[index];
}

// Конструктор объектов класса Array1D – конструктор по умолчанию
Array2D::Array1D::Array1D()
{
  base1D = 0;
  size1D = 0;
}

// Выделение свободной памяти для одномерного массива
void Array2D::Array1D::allocate1D(int rowSize)
{
  base1D = new int[rowSize];
  size1D = rowSize;
}
```

```
// Индексация массива
int& Array2D::Array1D::operator[](int index)
{
  if (index < 0 || index > size1D - 1)
  {
    cout << "Второй индекс за границами диапазона!" << endl;
    exit(1);
  }
  return base1D[index];
}

int main()
{
  Array2D a(3, 4);
  // Инициализация и визуализация элементов массива
  for (int i = 0; i < 3; ++i)
  {
    for (int j = 0; j < 4; ++j)
    {
      a[i][j] = i + j;
      cout << setw(3) << a[i][j];
    }
    cout << endl;
  }
  return 0;
}
```

Результат работы программы:

```
  0  1  2  3
  1  2  3  4
  2  3  4  5
```

Как видим, после определения объекта класса ***Array2D*** все операции индексации массива ***a*** будут ограничены соответствующими диапазонами изменения индексов – первого от ***0*** до ***2***, а второго от ***0*** до ***3***.

➢ **Пример 57**

Очевидно, что с помощью вложенных proxy-классов без особых усилий можно создавать теперь любой защищенный многомерный массив. Например, чтобы создать защищенный трехмерный массив, нужно определить класс ***Array3D*** таким образом, чтобы его операторная функция ***operator*[]()** перехватывала бы индекс, выходящий за границы первой размерности массива, и возвращала бы ссылку на объект вложенного класса ***Array2D***. В свою очередь, операторная функция ***operator*[]()** вложенного класса ***Array2D*** перехватывала бы индекс, выходящий за границы второй размерности массива, и возвращала бы ссылку на объект вложенного класса ***Array1D***, а затем операторная функция ***operator*[]()** вложенного класса ***Array1D***, в свою очередь, уже перехватывала бы индекс, выходящий за границы третьей размерности массива, и возвращала бы ссылку на элемент массива. Очередное добавление вложенного класса помимо прочего влечет за собой еще добавление его конструктора по умолчанию.

Приведем определение класса ***Array3D*** для защищенного трехмерного массива, элементы которого принадлежат встроенному типу ***int***:

```
// C++ Абстрактный тип данных - защищенный трехмерный массив

#include <iostream>
#include <cstdlib>
#include <iomanip>

using namespace std;

class Array3D
{
  class Array2D
  {
    class Array1D
    {
      // Компонентные данные - все собственные (private)
      int* base1D;
      int size1D;
      public:
      // Компонентные функции - все общедоступные (public)
      // Конструктор объектов класса – конструктор по умолчанию
      Array1D();
      // Деструктор объектов класса
      ~Array1D()
      {
        delete[] base1D;
      }
      // Выделение свободной памяти для одномерного массива
      void allocate1D(int);
      // Индексация массива
      int& operator[](int);
    };
```

```
    // Компонентные данные - все собственные (private)
    Array1D* base2D;
    int size2D;
    public:
    // Компонентные функции - все общедоступные (public)
    // Конструктор объектов класса – конструктор по умолчанию
    Array2D();
    // Деструктор объектов класса
    ~Array2D()
    {
      delete[] base2D;
    }
    // Выделение свободной памяти для двумерного массива
    void allocate2D(int, int);
    // Индексация массива
    Array1D& operator[](int);
  };
  // Компонентные данные - все собственные (private)
  Array2D* base3D;
  int size3D;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Array3D(int, int, int);
  // Деструктор объектов класса
  ~Array3D()
  {
    delete[] base3D;
  }
  // Индексация массива
  Array2D& operator[](int);
};

// Конструктор объектов класса Array3D
Array3D::Array3D(int numberOfArrays2D, int columnSize,
                 int rowSize)
{
  base3D = new Array2D[numberOfArrays2D];
  for (int i = 0; i < numberOfArrays2D; ++i)
    base3D[i].allocate2D(columnSize, rowSize);
  size3D = numberOfArrays2D;
}

// Индексация массива
Array3D::Array2D& Array3D::operator[](int index)
{
  if (index < 0 || index > size3D - 1)
  {
    cout << "Первый индекс за границами диапазона!" << endl;
```

```
    exit(1);
  }
  return base3D[index];
}

// Конструктор объектов класса Array2D – конструктор по умолчанию
Array3D::Array2D::Array2D()
{
  base2D = 0;
  size2D = 0;
}

// Выделение свободной памяти для двумерного массива
void Array3D::Array2D::allocate2D(int columnSize, int rowSize)
{
  base2D = new Array1D[columnSize];
  for (int i = 0; i < columnSize; ++i)
    base2D[i].allocate1D(rowSize);
  size2D = columnSize;
}

// Индексация массива
Array3D::Array2D::Array1D& Array3D::Array2D::operator[](int index)
{
  if (index < 0 || index > size2D – 1)
  {
    cout << "Второй индекс за границами диапазона!" << endl;
    exit(1);
  }
  return base2D[index];
}

// Конструктор объектов класса Array1D – конструктор по умолчанию
Array3D::Array2D::Array1D::Array1D()
{
  base1D = 0;
  size1D = 0;
}

// Выделение свободной памяти для одномерного массива
void Array3D::Array2D::Array1D::allocate1D(int rowSize)
{
  base1D = new int[rowSize];
  size1D = rowSize;
}

// Индексация массива
int& Array3D::Array2D::Array1D::operator[](int index)
{
```

```
  if (index < 0 || index > size1D - 1)
  {
    cout << "Третий индекс за границами диапазона!" << endl;
    exit(1);
  }
  return base1D[index];
}

int main()
{
  Array3D a(2, 3, 4);
  // Инициализация и визуализация элементов массива
  for (int i = 0; i < 2; ++i)
  {
    for (int j = 0; j < 3; ++j)
    {
      for (int k = 0; k < 4; ++k)
      {
        a[i][j][k] = i + j + k;
        cout << setw(3) << a[i][j][k];
      }
      cout << endl;
    }
    cout << endl;
  }
  return 0;
}
```

Результат работы программы:

```
  0  1  2  3
  1  2  3  4
  2  3  4  5

  1  2  3  4
  2  3  4  5
  3  4  5  6
```

Как видим, после определения объекта класса ***Array3D*** все операции индексации массива ***a*** будут ограничены соответствующими диапазонами изменения индексов – первого от ***0*** до ***1***, второго от ***0*** до ***2***, а третьего от ***0*** до ***3***.

“Умные указатели”

➢ Пример 58

Перегрузка оператора разыменования **->** имеет значение для целого круга интересных задач. Например, одним из важных примеров являются итераторы стандартной библиотеки шаблонов. Итераторы поддерживают абстрактную модель данных как последовательность объектов – “нечто такое, что можно перебирать либо от начала к концу, либо от конца к началу, используя оператор получения следующего элемента”. Итератор – это абстракция понятия указателя на элемент последовательности.

Другим ярким примером являются так называемые “умные указатели” или интеллектуальные указатели (от слов ***smart pointers***) – объекты класса, которые ведут себя как встроенные указатели и, кроме того, выполняют некоторые действия, когда с их помощью осуществляется доступ к компонентам другого класса, на объекты которого эти указатели как раз и ссылаются.

Для класса, объекты которого будут интеллектуальными указателями на объекты другого класса должна быть определена не только операторная функция ***operator*->()**, но также еще одна операция разыменования ***operator**()** и операция индексации ***operator*[]()**, так как для встроенных указателей на объекты класса использование оператора **->** является синонимом некоторых применений операторов ***** и **[]**.

Интеллектуальными указателями можно пользоваться для доступа к компонентам другого класса так же, как и встроенными указателями на объекты этого класса.

Рассмотрим класс ***Complex***, на объекты которого ссылаются интеллектуальные указатели – объекты класса ***SmartPointer***:

```
// C++ Абстрактный тип данных – интеллектуальный указатель

#include <iostream>

using namespace std;

class Complex
{
  // Компонентные данные - все собственные (private)
  double re;
  double im;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Визуализация комплексного числа
  friend ostream& operator<<(ostream& stream, const Complex& c)
  {
    return stream << '(' << c.re << ", " << c.im << ')';
  }
```

```
  // Ссылка на вещественную часть комплексного числа
  double& get_re()
  {
    return re;
  }
  // Ссылка на мнимую часть комплексного числа
  double& get_im()
  {
    return im;
  }
};

class SmartPointer
{
  // Компонентные данные - все собственные (private)
  Complex* pointer;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  SmartPointer(Complex* p = 0) : pointer(p) {}
  // Конструктор объектов класса – конструктор копирования
  SmartPointer(SmartPointer& p)
  {
    pointer = p.pointer;
    p.pointer = 0;
  }
  // Копирующее присваивание
  SmartPointer& operator=(SmartPointer& p)
  {
    if (this != &p)
    {
      pointer = p.pointer;
      p.pointer = 0;
    }
    return *this;
  }
  // Разыменование интеллектуального указателя
  Complex* operator->()const
  {
    return pointer;
  }
  // Разыменование интеллектуального указателя
  Complex& operator*()const
  {
    return *pointer;
  }
  // Индексация интеллектуального указателя
  Complex& operator[](int index)const
  {
```

```
    return pointer[index];
  }
};

int main()
{
  Complex x(-1, 5);
  cout << x << endl;
  SmartPointer p(&x);
  cout << *p << endl;
  cout << p[0] << endl;
  p->get_re() = 10;
  p->get_im() = 7;
  cout << x << endl;
  cout << *p << endl;
  SmartPointer q = p;                  // новый владелец объекта
  cout << *q << endl;
  q->get_re() = -1;
  q->get_im() = 5;
  cout << x << endl;
  SmartPointer t;
  t = q;                               // новый владелец объекта
  cout << *t << endl;
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(-1, 5)
(-1, 5)
(10, 7)
(10, 7)
(10, 7)
(-1, 5)
(-1, 5)
```

Наследование классов

➢ Пример 59

Инициализация объектов, представляющих базовые и производные классы, может быть задана в их конструкторах, причем конструкторы выполняются в порядке наследования (конструктор базового класса выполняется раньше конструктора его производного класса), а деструкторы – в обратном (деструктор производного класса выполняется раньше деструктора базового класса). Здесь, как и ранее, там, где это необходимо для иллюстрации механизма вызова конструкторов и деструкторов классов, каждый их вызов будет сопровождаться своим идентифицирующим сообщением.

Для инициализации компонентных данных можно воспользоваться расширенной формой объявления конструкторов производных классов, позволяющей через список инициализации членов, в котором могут быть представлены явные вызовы конструкторов прямых базовых классов, передавать по цепочке наследования всем требуемым конструкторам косвенных базовых классов необходимые им аргументы вызова. Отметим, что при таком способе организации списка инициализации членов допустимо использование одних и тех же аргументов вызова для базового и производного классов. Для производного класса допустимо также игнорирование всех аргументов вызова, в этом случае необходима их передача напрямую в базовый класс. Если инициализация производится только в производном классе, аргументы вызова передаются обычным образом.

Производные классы, получая в наследство данные и методы своих базовых классов, могут пополняться собственными данными и методами.

Приведем пример открытого одиночного наследования компонентов базового класса ***Complex_based*** его производным классом ***Complex_derived***, который к наследуемым компонентам добавляет собственную компонентную функцию ***add*()**:

```
// C++ Одиночное наследование

#include <iostream>

using namespace std;

struct Complex_based
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов базового класса
  Complex_based(double r = 0.0, double i = 0.0) : re(r), im(i)
  {
    cout << "Конструктор базового класса" << endl;
  }
  // Деструктор объектов базового класса
  ~Complex_based()
  {
```

```
    cout << "Деструктор базового класса" << endl;
  }
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  double re;
  double im;
};

struct Complex_derived : Complex_based
{
  // Компонентные функции – все общедоступные (public)
  // Конструктор объектов производного класса
  Complex_derived(double r = 0.0, double i = 0.0)
  : Complex_based(r, i)
  {
    cout << "Конструктор производного класса" << endl;
  }
  // Деструктор объектов производного класса
  ~Complex_derived()
  {
    cout << "Деструктор производного класса" << endl;
  }
  // Сложение комплексных чисел
  Complex_derived& add(Complex_derived& a, Complex_derived& b)
  {
    re = a.re + b.re;
    im = a.im + b.im;
    return *this;
  }
};

int main()
{
  Complex_derived x1(-1, 5);
  Complex_derived x2(10, 7);
  Complex_derived x3;
  x1.print();
  x2.print();
  x3.print();
  x1.add(x1, x2);
  x1.print();
  return 0;
}
```

Результат работы программы:

```
Конструктор базового класса
Конструктор производного класса
Конструктор базового класса
Конструктор производного класса
Конструктор базового класса
Конструктор производного класса
(-1, 5)
(10, 7)
(0, 0)
(9, 12)
Деструктор производного класса
Деструктор базового класса
Деструктор производного класса
Деструктор базового класса
Деструктор производного класса
Деструктор базового класса
```

Как видим, отсутствие в производном классе собственных компонентных данных приводит к тому, что все операции с его объектами осуществляются только для наследуемых компонентных данных базового класса. Здесь конструкторы базового и производного классов используют одни и те же аргументы (что бывает не так часто).

В большинстве случаев конструкторы производных классов передают конструкторам базовых классов только те аргументы вызова, которые им требуются. Например, чтобы передать конструктору базового класса необходимые ему аргументы, конструктору производного класса следует передать все аргументы вызова, необходимые конструкторам обоих классов.

В производном классе наследуемые компоненты базового класса получают статус доступа ***private***, если производный класс определяется с помощью ключевого слова ***class***, и статус доступа ***public***, если производный класс определяется с помощью ключевого слова ***struct***. Кроме того, при объявлении производных классов можно явно изменить статус доступа к наследуемым компонентам базовых классов при помощи спецификаторов доступа (***public***, ***private*** и ***protected***), которые указываются непосредственно перед именами базовых классов.

Спецификаторы доступа при объявлении производных классов определяют, как компоненты базового класса наследуются производным классом.

➢ **Пример 60**

При наследовании компоненты базового класса могут быть переопределены в его производном классе. В этом случае они становятся скрытыми и тем самым недоступными из производного класса (теперь для доступа к ним необходимо использовать операцию разрешения области видимости). Для иллюстрации механизма доступа к компонентным данным базового и производного классов необходимо, чтобы их конструкторы использовали не одни и те же аргументы.

Приведем пример открытого одиночного наследования компонентов базового класса ***Complex_based*** его производным классом ***Complex_derived***, который переопределяет наследуемые компонентные данные и компонентную функцию ***print*()**, а также добавляет собственную компонентную функцию ***add*()**:

```
// C++ Одиночное наследование

#include <iostream>

using namespace std;

struct Complex_based
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов базового класса
  Complex_based(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  double re;
  double im;
};

struct Complex_derived : Complex_based
{
  // Компонентные функции – все общедоступные (public)
  // Конструктор объектов производного класса
  Complex_derived(double r = 0.0, double i = 0.0)
  : Complex_based(r), re(r), im(i) {}
  // Сложение комплексных чисел
  Complex_derived& add(Complex_derived& a, Complex_derived& b)
  {
    re = a.re + b.re;
    im = a.im + b.im;
    return *this;
  }
```

```
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  double re;
  double im;
};

int main()
{
  Complex_derived x1(-1, 5);
  Complex_derived x2(10, 7);
  Complex_derived x3;
  x1.Complex_based::print();
  x1.print();
  x2.Complex_based::print();
  x2.print();
  x3.Complex_based::print();
  x3.print();
  x1.add(x1, x2);
  x1.Complex_based::print();
  x1.print();
  return 0;
}
```

Результат работы программы:

```
(-1, 0)
(-1, 5)
(10, 0)
(10, 7)
(0, 0)
(0, 0)
(-1, 0)
(9, 12)
```

Как и следовало ожидать, теперь все операции с объектами производного класса осуществляются только для его собственных компонентных данных.

➢ **Пример 61**

Обратимся теперь к проблеме создания защищенных массивов на основе иерархии классов с открытым одиночным наследованием. Здесь, как и ранее, следует определить классы, позволяющие объявлять защищенные массивы, и разрешить доступ к их элементам только через перегруженный оператор **[]**. Отличие лишь в том, что благодаря иерархии классов теперь можно использовать общие средства для создания как одномерных, так и многомерных массивов.

Например, для создания защищенных двумерных массивов это можно сделать, определив производный класс ***Array2D*** таким образом, чтобы его операторная функция ***operator*[]()** перехватывала бы индекс, выходящий за границы первой размерности массива, и возвращала бы ссылку на объект прямого базового класса ***Array1D***. В свою очередь, операторная функция ***operator*[]()** базового класса ***Array1D*** перехватывала бы индекс, выходящий за границы второй размерности массива, и возвращала бы ссылку на элемент массива. Защищенный одномерный массив здесь создается обычным образом. Необходимо лишь помнить о том, что здесь следует избегать очевидной неоднозначности в именовании размерностей массивов.

Представим иерархию классов ***Array1D*** и ***Array2D*** для создания защищенных одномерных и двумерных массивов, элементы которых принадлежат встроенному типу ***int***:

```
// C++ Абстрактный тип данных - защищенный массив

#include <iostream>
#include <cstdlib>
#include <iomanip>

using namespace std;

class Array1D
{
  // Компонентные данные - все собственные (private)
  int* base1D;
  int size1D;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Array1D();
  // Конструктор объектов класса
  Array1D(int);
  // Деструктор объектов класса
  ~Array1D();
  // Выделение свободной памяти для одномерного массива
  void allocate1D(int);
  // Индексация массива
  int& operator[](int);
};
```

```
// Конструктор объектов класса Array1D – конструктор по умолчанию
Array1D::Array1D()
{
  base1D = 0;
  size1D = 0;
}

// Конструктор объектов класса Array1D
Array1D::Array1D(int rowSize)
{
  allocate1D(rowSize);
}

// Деструктор объектов класса Array1D
Array1D::~Array1D()
{
  delete[] base1D;
}

// Выделение свободной памяти для одномерного массива
void Array1D::allocate1D(int rowSize)
{
  base1D = new int[rowSize];
  size1D = rowSize;
}

// Индексация массива
int& Array1D::operator[](int index)
{
  if (index < 0 || index > size1D – 1)
  {
    cout << "Индекс за границами диапазона!" << endl;
    exit(1);
  }
  return base1D[index];
}

class Array2D : public Array1D
{
  // Компонентные данные - все собственные (private)
  Array1D* base2D;
  int size2D;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Array2D(int, int);
  // Деструктор объектов класса
  ~Array2D();
```

```
  // Выделение свободной памяти для двумерного массива
  void allocate2D(int, int);
  // Индексация массива
  Array1D& operator[](int);
};

// Конструктор объектов класса Array2D
Array2D::Array2D(int columnSize, int rowSize) : Array1D()
{
  allocate2D(columnSize, rowSize);
}

// Деструктор объектов класса Array2D
Array2D::~Array2D()
{
  delete[] base2D;
}

// Выделение свободной памяти для двумерного массива
void Array2D::allocate2D(int columnSize, int rowSize)
{
  base2D = new Array1D[columnSize];
  for (int i = 0; i < columnSize; ++i)
    base2D[i].allocate1D(rowSize);
  size2D = columnSize;
}

// Индексация массива
Array1D& Array2D::operator[](int index)
{
  if (index < 0 || index > size2D - 1)
  {
    cout << "Индекс за границами диапазона!" << endl;
    exit(1);
  }
  return base2D[index];
}

int main()
{
  Array1D a(5);
  int i, j;
  // Инициализация и визуализация элементов массива
  for (i = 0; i < 5; ++i)
  {
    a[i] = i;
    cout << setw(3) << a[i];
  }
  cout << endl << endl;
```

```
  Array2D b(3, 4);
  // Инициализация и визуализация элементов массива
  for (i = 0; i < 3; ++i)
  {
    for (j = 0; j < 4; ++j)
    {
      b[i][j] = i + j;
      cout << setw(3) << b[i][j];
    }
    cout << endl;
  }
  return 0;
}
```

Результат работы программы:

```
  0   1   2   3   4

  0   1   2   3
  1   2   3   4
  2   3   4   5
```

Как видим, после определения объектов класса ***Array1D*** и класса ***Array2D*** все операции индексации массива ***a*** будут ограничены диапазоном изменения индекса от ***0*** до ***4***, а массива ***b*** – диапазонами изменения индексов от ***0*** до ***2*** и от ***0*** до ***3***.

Отметим здесь, что операции индексации для индекса, выходящего за границы допустимого диапазона, как и ранее, должны приводить к аварийному завершению программы.

➢ **Пример 62**

Очевидно, что очередное добавление в иерархию классов с открытым одиночным наследованием каждого нового класса позволит теперь без особых усилий создавать любой защищенный многомерный массив.

Представим иерархию классов ***Array1D***, ***Array2D*** и ***Array3D*** для создания защищенных одномерных, двумерных и трехмерных массивов, элементы которых принадлежат встроенному типу ***int***:

```
// C++ Абстрактный тип данных - защищенный массив

#include <iostream>
#include <cstdlib>
#include <iomanip>

using namespace std;

class Array1D
{
  // Компонентные данные - все собственные (private)
  int* base1D;
  int size1D;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Array1D();
  // Конструктор объектов класса
  Array1D(int);
  // Деструктор объектов класса
  ~Array1D();
  // Выделение свободной памяти для одномерного массива
  void allocate1D(int);
  // Индексация массива
  int& operator[](int);
};

// Конструктор объектов класса Array1D – конструктор по умолчанию
Array1D::Array1D()
{
  base1D = 0;
  size1D = 0;
}

// Конструктор объектов класса Array1D
Array1D::Array1D(int rowSize)
{
  allocate1D(rowSize);
}
```

```
// Деструктор объектов класса Array1D
Array1D::~Array1D()
{
  delete[] base1D;
}

// Выделение свободной памяти для одномерного массива
void Array1D::allocate1D(int rowSize)
{
  base1D = new int[rowSize];
  size1D = rowSize;
}

// Индексация массива
int& Array1D::operator[](int index)
{
  if (index < 0 || index > size1D - 1)
  {
    cout << "Индекс за границами диапазона!" << endl;
    exit(1);
  }
  return base1D[index];
}

class Array2D : public Array1D
{
  // Компонентные данные - все собственные (private)
  Array1D* base2D;
  int size2D;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса – конструктор по умолчанию
  Array2D();
  // Конструктор объектов класса
  Array2D(int, int);
  // Деструктор объектов класса
  ~Array2D();
  // Выделение свободной памяти для двумерного массива
  void allocate2D(int, int);
  // Индексация массива
  Array1D& operator[](int);
};

// Конструктор объектов класса Array2D – конструктор по умолчанию
Array2D::Array2D()
{
  base2D = 0;
  size2D = 0;
}
```

```
// Конструктор объектов класса Array2D
Array2D::Array2D(int columnSize, int rowSize) : Array1D()
{
  allocate2D(columnSize, rowSize);
}

// Деструктор объектов класса Array2D
Array2D::~Array2D()
{
  delete[] base2D;
}

// Выделение свободной памяти для двумерного массива
void Array2D::allocate2D(int columnSize, int rowSize)
{
  base2D = new Array1D[columnSize];
  for (int i = 0; i < columnSize; ++i)
    base2D[i].allocate1D(rowSize);
  size2D = columnSize;
}

// Индексация массива
Array1D& Array2D::operator[](int index)
{
  if (index < 0 || index > size2D – 1)
  {
    cout << "Индекс за границами диапазона!" << endl;
    exit(1);
  }
  return base2D[index];
}

class Array3D : public Array2D
{
  // Компонентные данные - все собственные (private)
  Array2D* base3D;
  int size3D;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Array3D(int, int, int);
  // Деструктор объектов класса
  ~Array3D();
  // Выделение свободной памяти для трехмерного массива
  void allocate3D(int, int, int);
  // Индексация массива
  Array2D& operator[](int);
};
```

```
// Конструктор объектов класса Array3D
Array3D::Array3D(int numberOfArrays2D, int columnSize,
                 int rowSize) : Array2D()
{
  allocate3D(numberOfArrays2D, columnSize, rowSize);
}

// Деструктор объектов класса Array3D
Array3D::~Array3D()
{
  delete[] base3D;
}

// Выделение свободной памяти для трехмерного массива
void Array3D::allocate3D(int numberOfArrays2D,
                         int columnSize, int rowSize)
{
  base3D = new Array2D[numberOfArrays2D];
  for (int i = 0; i < numberOfArrays2D; ++i)
    base3D[i].allocate2D(columnSize, rowSize);
  size3D = numberOfArrays2D;
}

// Индексация массива
Array2D& Array3D::operator[](int index)
{
  if (index < 0 || index > size3D - 1)
  {
    cout << "Индекс за границами диапазона!" << endl;
    exit(1);
  }
  return base3D[index];
}

int main()
{
  Array1D a(5);
  int i, j, k;
  // Инициализация и визуализация элементов массива
  for (i = 0; i < 5; ++i)
  {
    a[i] = i;
    cout << setw(3) << a[i];
  }
  cout << endl << endl;
  Array2D b(3, 4);
  // Инициализация и визуализация элементов массива
  for (i = 0; i < 3; ++i)
  {
```

```
    for (j = 0; j < 4; ++j)
    {
      b[i][j] = i + j;
      cout << setw(3) << b[i][j];
    }
    cout << endl;
  }
  cout << endl;
  Array3D c(2, 3, 4);
  // Инициализация и визуализация элементов массива
  for (i = 0; i < 2; ++i)
  {
    for (j = 0; j < 3; ++j)
    {
      for (k = 0; k < 4; ++k)
      {
        c[i][j][k] = i + j + k;
        cout << setw(3) << c[i][j][k];
      }
      cout << endl;
    }
    cout << endl;
  }
  return 0;
}
```

Результат работы программы:

```
  0  1  2  3  4

  0  1  2  3
  1  2  3  4
  2  3  4  5

  0  1  2  3
  1  2  3  4
  2  3  4  5

  1  2  3  4
  2  3  4  5
  3  4  5  6
```

Как видим, и здесь после определения объектов класса ***Array1D***, класса ***Array2D*** и класса ***Array3D*** все операции индексации массива ***a*** будут ограничены диапазоном изменения индекса от ***0*** до ***4***, массива ***b*** – диапазонами изменения индексов от ***0*** до ***2*** и от ***0*** до ***3***, а массива ***c*** – диапазонами изменения индексов от ***0*** до ***1***, от ***0*** до ***2*** и от ***0*** до ***3***.

➢ Пример 63

При наследовании нередко требуется, чтобы выбор между одноименными компонентными функциями базового и производного классов осуществлялся по типам аргументов. Так как разрешение перегрузки не пересекает границ областей видимости классов, в этом случае в производном классе одноименные компонентные функции его базового класса объявляются при помощи ***using-объявлений***, что позволяет ввести эти функции в общую область видимости производного класса, чтобы затем на основе механизма разрешения перегрузки как раз и попытаться осуществить их вызов.

Приведем пример открытого одиночного наследования компонентов базового класса ***Complex_based*** его производным классом ***Complex_derived***, который переопределяет наследуемые компонентные данные, добавляет собственные компонентные функции ***add*()** и ***print*()**, а также вводит в свою область видимости компонентную функцию базового класса ***print*()** при помощи ***using-объявления***:

```
// C++ Одиночное наследование

#include <iostream>

using namespace std;

struct Complex_based
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов базового класса
  Complex_based(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Визуализация комплексного числа
  void print(Complex_based& c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  double re;
  double im;
};

struct Complex_derived : Complex_based
{
  using Complex_based::print;
  // Компонентные функции – все общедоступные (public)
  // Конструктор объектов производного класса
  Complex_derived(double r = 0.0, double i = 0.0)
  : Complex_based(r), re(r), im(i) {}
  // Сложение комплексных чисел
  Complex_derived& add(Complex_derived& a, Complex_derived& b)
  {
```

```
    re = a.re + b.re;
    im = a.im + b.im;
    return *this;
  }
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  double re;
  double im;
};

int main()
{
  Complex_derived x1(-1, 5);
  Complex_derived x2(10, 7);
  Complex_derived x3;
  x1.print(x1);
  x1.print();
  x2.print(x2);
  x2.print();
  x3.print(x3);
  x3.print();
  x1.add(x1, x2);
  x1.print(x1);
  x1.print();
  return 0;
}
```

Результат работы программы:

```
(-1, 0)
(-1, 5)
(10, 0)
(10, 7)
(0, 0)
(0, 0)
(-1, 0)
(9, 12)
```

Как видим, такой способ явного устранения неоднозначности вызова одноименных компонентных функций базового и производного классов является более удобным по сравнению с использованием операции разрешения области видимости.

➢ Пример 64

При наследовании чрезвычайно важна роль деструкторов базовых и производных классов, если конструируются объекты в свободной памяти с помощью указателей на базовый или его производный класс. Далее будет сказано, что деструкторы базовых классов обязательно должны быть виртуальными функциями, в противном случае возможны неожиданные побочные эффекты времени выполнения.

Ресурсы в виде свободной памяти, захваченной при конструировании объектов с помощью оператора ***new***, как правило, освобождаются без проблем, если только используются стандартные средства распределения свободной памяти. Проблемы начинаются, например, тогда, когда с помощью указателей на базовые классы конструируются динамические объекты производных классов, поскольку для операций по освобождению ресурсов с помощью оператора ***delete*** здесь важен тип указателя (полученный на этапе времени компиляции кода), а не его значение (полученное на этапе времени выполнения программы). Далее будет сказано, что лишь механизм виртуализации компонентных функций может обеспечить их вызов не по типу указателя, а по его значению (динамический полиморфизм). Поэтому так важен именно виртуальный деструктор базового класса.

Проиллюстрируем возможность получения такой подобной проблемы на примере открытого одиночного наследования компонентов базового класса ***Complex_based*** его производным классом ***Complex_derived***, который переопределяет наследуемые компонентные данные и компонентную функцию ***print*()**:

```
// C++ Одиночное наследование

#include <iostream>

using namespace std;

struct Complex_based
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов базового класса
  Complex_based(double r = 0.0, double i = 0.0) : re(r), im(i)
  {
    cout << "Конструктор базового класса" << endl;
  }
  // Деструктор объектов базового класса
  ~Complex_based()
  {
    cout << "Деструктор базового класса" << endl;
  }
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
```

```
  protected:
  // Компонентные данные - все защищенные (protected)
  double re;
  double im;
};

struct Complex_derived : Complex_based
{
  // Компонентные функции – все общедоступные (public)
  // Конструктор объектов производного класса
  Complex_derived(double r = 0.0, double i = 0.0)
  : Complex_based(r), re(r), im(i)
  {
    cout << "Конструктор производного класса" << endl;
  }
  // Деструктор объектов производного класса
  ~Complex_derived()
  {
    cout << "Деструктор производного класса" << endl;
  }
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  double re;
  double im;
};

int main()
{
  cout << sizeof(Complex_based) << endl;
  cout << sizeof(Complex_derived) << endl;
  Complex_based* p = new Complex_based(-1, 5);
  p->print();
  delete p;
  Complex_derived* q = new Complex_derived(10, 7);
  q->print();
  q->Complex_based::print();
  delete q;
  Complex_based* r = new Complex_derived(9, 12);
  r->print();
  ((Complex_derived*)r)->print();
  delete r;
  return 0;
}
```

Результат работы программы:

```
16
32
Конструктор базового класса
(-1, 5)
Деструктор базового класса
Конструктор базового класса
Конструктор производного класса
(10, 7)
(10, 0)
Деструктор производного класса
Деструктор базового класса
Конструктор базового класса
Конструктор производного класса
(9, 0)
(9, 12)
Деструктор базового класса
```

Как видим, при освобождении памяти, захваченной для динамического объекта производного класса, через указатель на его базовый класс вызывается только один деструктор, а именно деструктор базового класса. Тем самым налицо так называемая “утечка” памяти, так как на самом деле освобождается память не для всего объекта производного класса, а лишь для его наследуемого объекта базового класса.

Нелишним будет напомнить здесь, что компилятор определяет размер удаляемого объекта на основании вызываемого деструктора. Виртуальный деструктор класса обеспечивает правильность размера, передаваемого оператору ***delete***. Известно, что правильный размер будет передаваться при соблюдении любого из трех условий:

- деструктор является виртуальным;
- указатель ссылается на настоящий тип объекта в свободной памяти;
- тип, на который ссылается указатель, имеет тот же размер, что и настоящий тип.

Здесь, как видим, не соблюдается ни одно из перечисленных условий.

Множественное наследование и виртуальные базовые классы

➢ Пример 65

При множественном наследовании никакой класс не может больше одного раза использоваться в качестве прямого базового, однако класс может больше одного раза быть непрямым базовым классом, например:

```
class A { компоненты_класса_A };
class B : public A { компоненты_класса_B };
class C : public A { компоненты_класса_C };
class D : public B, public C { компоненты_класса_D };
```

Здесь непрямой базовый класс ***A*** дважды опосредованно наследуется классом ***D***, такое дублирование класса приводит к двукратному тиражированию объектов непрямого базового класса в производном объекте.

Кроме дублирования в производном классе ***D*** компонентных данных непрямого базового класса ***A*** здесь возможна также и неоднозначность доступа к ним, поэтому если из класса ***D*** необходим доступ к компонентным данным класса ***A***, ссылки на них должны быть явно квалифицированы.

Вначале приведем пример открытого множественного наследования компонентов непрямого базового класса ***A_based*** и двух прямых базовых классов ***B_derived*** и ***C_derived*** их производным классом ***D_derived***:

```
// C++ Множественное наследование

#include <iostream>

using namespace std;

struct A_based
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов базового класса
  A_based(int i = 0) : a_data(i)
  {
    cout << "Конструктор базового класса A_based" << endl;
  }
  // Деструктор объектов базового класса
  ~A_based()
  {
    cout << "Деструктор базового класса A_based" << endl;
  }
  // Визуализация компонента базового класса
  void print()
  {
    cout << a_data << endl;
  }
```

```
  protected:
  // Компонентные данные - все защищенные (protected)
  int a_data;
};

struct B_derived : A_based
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов производного класса
  B_derived(int i = 0, long l = 0) : A_based(i), b_data(l)
  {
    cout << "Конструктор производного класса B_derived" << endl;
  }
  // Деструктор объектов производного класса
  ~B_derived()
  {
    cout << "Деструктор производного класса B_derived" << endl;
  }
  // Визуализация компонентов классов
  void print()
  {
    cout << a_data << ' ';
    A_based::print();
    cout << b_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  long b_data;
};

struct C_derived : A_based
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов производного класса
  C_derived(int i = 0, float f = 0.0) : A_based(i), c_data(f)
  {
    cout << "Конструктор производного класса C_derived" << endl;
  }
  // Деструктор объектов производного класса
  ~C_derived()
  {
    cout << "Деструктор производного класса C_derived" << endl;
  }
  // Визуализация компонентов классов
  void print()
  {
    cout << a_data << ' ';
    A_based::print();
```

```
    cout << c_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  float c_data;
};

struct D_derived : B_derived, C_derived
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов производного класса
  D_derived(int i = 0, long l = 0, float f = 0.0, double d = 0.0)
  : B_derived(i, l), C_derived(i, f), d_data(d)
  {
    cout << "Конструктор производного класса D_derived" << endl;
  }
  // Деструктор объектов производного класса
  ~D_derived()
  {
    cout << "Деструктор производного класса D_derived" << endl;
  }
  // Визуализация компонентов классов
  void print()
  {
    cout << B_derived::a_data << ' ';
    B_derived::print();
    cout << b_data << endl;
    cout << C_derived::a_data << ' ';
    C_derived::print();
    cout << c_data << endl;
    cout << d_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  double d_data;
};

int main()
{
  cout << "sizeof(A_based) = " << sizeof(A_based) << endl;
  cout << "sizeof(B_derived) = " << sizeof(B_derived) << endl;
  cout << "sizeof(C_derived) = " << sizeof(C_derived) << endl;
  cout << "sizeof(D_derived) = " << sizeof(D_derived) << endl;
  D_derived d(1, 2, 3, 4);
  d.print();
  d.B_derived::print();
  d.C_derived::print();
  return 0;
}
```

Результат работы программы:

```
sizeof(A_based) = 4
sizeof(B_derived) = 8
sizeof(C_derived) = 8
sizeof(D_derived) = 24
Конструктор базового класса A_based
Конструктор производного класса B_derived
Конструктор базового класса A_based
Конструктор производного класса C_derived
Конструктор производного класса D_derived
1 1 1
2
2
1 1 1
3
3
4
1 1
2
1 1
3
Деструктор производного класса D_derived
Деструктор производного класса C_derived
Деструктор базового класса A_based
Деструктор производного класса B_derived
Деструктор базового класса A_based
```

Непрямой базовый класс ***A_based*** в соответствии с размером своего компонента данных имеет размер 4 байта (32-х разрядная архитектура). Размер производного класса ***B_derived*** равен сумме размера своего компонента данных (4 байта) и размера наследуемого базового класса ***A_based*** (4 байта). Размер производного класса ***C_derived*** равен сумме размера своего компонента данных (4 байта) и размера наследуемого базового класса ***A_based*** (4 байта). Размер производного класса ***D_derived*** равен сумме размера своего компонента данных (8 байт) и размеров наследуемых классов ***B_derived*** (8 байт) и ***C_derived*** (8 байт).

➢ **Пример 66**

Чтобы устранить многократное тиражирование компонентных данных какого-либо непрямого базового класса при множественном наследовании, этот базовый класс объявляют ***виртуальным*** с помощью спецификатора ***virtual***. Само наследование при этом нередко называют множественным виртуальным наследованием.

Например, непрямой базовый класс ***A*** будет виртуальным при таком определении:

```
class A { компоненты_класса_A };
class B : public virtual A { компоненты_класса_B };
class C : public virtual A { компоненты_класса_C };
class D : public B, public C { компоненты_класса_D };
```

Теперь объект производного класса ***D*** будет включать только один объект виртуального базового класса ***A***, доступ к которому равноправно разделяют его производные классы ***B*** и ***C***. Если из класса ***D*** необходим доступ к компонентным данным класса ***A***, то ссылки на них теперь уже могут быть и не квалифицированы.

Отметим, что если какой-либо непрямой базовый класс становится виртуальным, то инициализация его компонентных данных может быть выполнена только тем конструктором иерархии классов, объект которого как раз и создается. Поэтому без списков инициализации конструкторов производных классов здесь уже никак не обойтись. И здесь не так важно, какой именно конструктор иерархии классов будет использоваться для создания конкретного объекта.

Приведем пример открытого множественного наследования компонентов виртуального базового класса ***A_based*** и двух прямых базовых классов ***B_derived*** и ***C_derived*** их производным классом ***D_derived***:

```
// C++ Множественное виртуальное наследование

#include <iostream>

using namespace std;

struct A_based
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов базового класса
  A_based(int i = 0) : a_data(i)
  {
    cout << "Конструктор базового класса A_based" << endl;
  }
  // Деструктор объектов базового класса
  ~A_based()
  {
    cout << "Деструктор базового класса A_based" << endl;
  }
  // Визуализация компонента базового класса
  void print()
  {
```

```
    cout << a_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  int a_data;
};

struct B_derived : virtual A_based
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов производного класса
  B_derived(int i = 0, long l = 0) : A_based(i), b_data(l)
  {
    cout << "Конструктор производного класса B_derived" << endl;
  }
  // Деструктор объектов производного класса
  ~B_derived()
  {
    cout << "Деструктор производного класса B_derived" << endl;
  }
  // Визуализация компонентов классов
  void print()
  {
    cout << a_data << ' ';
    A_based::print();
    cout << b_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  long b_data;
};

struct C_derived : virtual A_based
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов производного класса
  C_derived(int i = 0, float f = 0.0) : A_based(i), c_data(f)
  {
    cout << "Конструктор производного класса C_derived" << endl;
  }
  // Деструктор объектов производного класса
  ~C_derived()
  {
    cout << "Деструктор производного класса C_derived" << endl;
  }
  // Визуализация компонентов классов
  void print()
  {
```

```
    cout << a_data << ' ';
    A_based::print();
    cout << c_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  float c_data;
};

struct D_derived : B_derived, C_derived
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов производного класса
  D_derived(int i = 0, long l = 0, float f = 0.0, double d = 0.0)
  : A_based(i), B_derived(i, l), C_derived(i, f), d_data(d)
  {
    cout << "Конструктор производного класса D_derived" << endl;
  }
  // Деструктор объектов производного класса
  ~D_derived()
  {
    cout << "Деструктор производного класса D_derived" << endl;
  }
  // Визуализация компонентов классов
  void print()
  {
    A_based::print();
    cout << a_data << ' ';
    B_derived::print();
    cout << b_data << endl;
    C_derived::print();
    cout << c_data << endl;
    cout << d_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  double d_data;
};

int main()
{
  cout << "sizeof(A_based) = " << sizeof(A_based) << endl;
  cout << "sizeof(B_derived) = " << sizeof(B_derived) << endl;
  cout << "sizeof(C_derived) = " << sizeof(C_derived) << endl;
  cout << "sizeof(D_derived) = " << sizeof(D_derived) << endl;
  A_based a(1);
  a.print();
  B_derived b(2, 3);
  b.print();
```

```
  C_derived c(4, 5);
  c.print();
  D_derived d(6, 7, 8, 9);
  d.print();
  return 0;
}
```

Результат работы программы:

```
sizeof(A_based) = 4
sizeof(B_derived) = 12
sizeof(C_derived) = 12
sizeof(D_derived) = 28
Конструктор базового класса A_based
1
Конструктор базового класса A_based
Конструктор производного класса B_derived
2 2
3
Конструктор базового класса A_based
Конструктор производного класса C_derived
4 4
5
Конструктор базового класса A_based
Конструктор производного класса B_derived
Конструктор производного класса C_derived
Конструктор производного класса D_derived
6
6 6 6
7
7
6 6
8
8
9
Деструктор производного класса D_derived
Деструктор производного класса C_derived
Деструктор производного класса B_derived
Деструктор базового класса A_based
Деструктор производного класса C_derived
Деструктор базового класса A_based
Деструктор производного класса B_derived
Деструктор базового класса A_based
Деструктор базового класса A_based
```

Наличие виртуального базового класса ***A_based*** привело к увеличению размеров производных классов ***B_derived*** и ***C_derived*** на 4 байта, которые необходимы для связи в иерархии виртуальных классов (указатель на разделяемый производными классами объект виртуального класса). Размер производного класса ***D_derived*** равен сумме размера наследуемого базового класса ***A_based*** (4 байта), размера своего компонента (8 байт) и размеров наследуемых классов ***B_derived*** (4 байта + 4 байта для связи класса) и ***C_derived*** (4 байта + 4 байта для связи класса).

При множественном виртуальном наследовании конструктору производного класса разрешается явно вызывать не только конструкторы его прямых базовых классов, но также и конструкторы их предшествующих классов в иерархии. Здесь, как видим, конструкторы производных классов с помощью списка инициализации передают конструкторам своих прямых базовых классов только те аргументы, которые им требуются. Однако не все так просто, как кажется при первом взгляде.

Так как при создании объектов производных классов конструкторы наследуемых виртуальных базовых классов иерархии всегда вызываются только один раз, то здесь, например, при создании объекта производного класса ***D_derived*** будут подавлены оба неявных вызова конструктора по умолчанию виртуального базового класса ***A_based*** при вызове конструкторов прямых базовых классов ***B_derived*** и ***C_derived***. Тем самым налицо очевидная избыточность при передаче аргументов конструкторам производных классов ***B_derived*** и ***C_derived***.

Это означает теперь, что при множественном виртуальном наследовании следует различать два состояния у производного класса: либо он является так называемым ближайшим производным классом в иерархии (когда создается его объект), либо промежуточным производным классом (когда создается не его объект). Далее будет сказано, как с помощью защищенных конструкторов по умолчанию решается проблема избыточности при передаче аргументов конструкторам производных классов, когда они выступают в роли промежуточных производных классов.

➢ Пример 67

Ромбовидное наследование недаром получило название “ужасного бриллианта наследования”, поскольку рассматриваемая проблема избыточности при передаче аргументов вызова конструкторам классов, выступающих в роли промежуточных производных классов, далеко не единственная. Одна из подобных проблем была затронута в связи с одиночным наследованием и ролью виртуальных деструкторов базовых классов. Ромбовидное наследование будет хорошо управляемым, если виртуальный базовый класс либо классы, производные непосредственно от него, являются абстрактными. Далее будет сказано, что класс становится абстрактным, если в нем объявляется хотя бы одна чисто виртуальная компонентная функция (например, деструктор).

Приведем теперь пример открытого множественного наследования компонентов виртуального базового класса ***A_based*** и двух прямых базовых классов ***B_derived*** и ***C_derived*** их производным классом ***D_derived***, где с помощью защищенных конструкторов по умолчанию решается проблема избыточности при передаче аргументов конструкторам классов, когда они выступают в роли промежуточных производных классов:

```
// C++ Множественное виртуальное наследование

#include <iostream>

using namespace std;

struct A_based
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов базового класса
  A_based(int i = 0) : a_data(i)
  {
    cout << "Конструктор базового класса A_based" << endl;
  }
  // Деструктор объектов базового класса
  ~A_based()
  {
    cout << "Деструктор базового класса A_based" << endl;
  }
  // Визуализация компонента базового класса
  void print()
  {
    cout << a_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  int a_data;
};
```

```
struct B_derived : virtual A_based
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов ближайшего производного класса
  B_derived(int i, long l) : A_based(i), b_data(l)
  {
    cout << "Конструктор производного класса B_derived" << endl;
  }
  // Деструктор объектов производного класса
  ~B_derived()
  {
    cout << "Деструктор производного класса B_derived" << endl;
  }
  // Визуализация компонентов классов
  void print()
  {
    cout << a_data << ' ';
    A_based::print();
    cout << b_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  long b_data;
  // Компонентные функции – все защищенные (protected)
  // Конструктор объектов промежуточного производного класса
  B_derived()
  {
    cout << "Конструктор производного класса B_derived*" << endl;
  }
};

struct C_derived : virtual A_based
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов ближайшего производного класса
  C_derived(int i = 0, float f = 0.0) : A_based(i), c_data(f)
  {
    cout << "Конструктор производного класса C_derived" << endl;
  }
  // Деструктор объектов производного класса
  ~C_derived()
  {
    cout << "Деструктор производного класса C_derived" << endl;
  }
  // Визуализация компонентов классов
  void print()
  {
    cout << a_data << ' ';
    A_based::print();
```

```
    cout << c_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  float c_data;
  // Компонентные функции – все защищенные (protected)
  // Конструктор объектов промежуточного производного класса
  C_derived(float f = 0.0) : c_data(f)
  {
    cout << "Конструктор производного класса C_derived*" << endl;
  }
};

struct D_derived : B_derived, C_derived
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов производного класса
  D_derived(int i = 0, long l = 0, float f = 0.0, double d = 0.0)
  : A_based(i), C_derived(f), d_data(d)
  {
    cout << "Конструктор производного класса D_derived" << endl;
    b_data = l;
  }
  // Деструктор объектов производного класса
  ~D_derived()
  {
    cout << "Деструктор производного класса D_derived" << endl;
  }
  // Визуализация компонентов классов
  void print()
  {
    A_based::print();
    cout << a_data << ' ';
    B_derived::print();
    cout << b_data << endl;
    C_derived::print();
    cout << c_data << endl;
    cout << d_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  double d_data;
};

int main()
{
  A_based a(1);
  a.print();
```

```
  B_derived b(2, 3);
  b.print();
  C_derived c(4, 5);
  c.print();
  D_derived d(6, 7, 8, 9);
  d.print();
  return 0;
}
```

Результат работы программы:

```
Конструктор базового класса A_based
1
Конструктор базового класса A_based
Конструктор производного класса B_derived
2 2
3
Конструктор базового класса A_based
Конструктор производного класса C_derived
4 4
5
Конструктор базового класса A_based
Конструктор производного класса B_derived*
Конструктор производного класса C_derived*
Конструктор производного класса D_derived
6
6 6 6
7
7
6 6
8
8
9
Деструктор производного класса D_derived
Деструктор производного класса C_derived
Деструктор производного класса B_derived
Деструктор базового класса A_based
Деструктор производного класса C_derived
Деструктор базового класса A_based
Деструктор производного класса B_derived
Деструктор базового класса A_based
Деструктор базового класса A_based
```

➢ Пример 68

При множественном наследовании, как и при одиночном, зачастую требуется, чтобы выбор между одноименными компонентными функциями различных базовых классов осуществлялся по типам аргументов. В этом случае, как и ранее, в производном классе одноименные компонентные функции базовых классов объявляются при помощи ***using-объявлений***, что позволяет ввести эти функции в общую область видимости производного класса, чтобы затем на основе механизма разрешения перегрузки как раз и попытаться осуществить их вызов.

Заметим при этом, что ***using-объявление*** в определении производного класса должно относиться к компонентам его базовых классов, ***using-объявление*** нельзя использовать для компонента класса вне этого класса, его производных классов или их компонентных функций.

В завершение обратимся к примеру открытого множественного наследования компонентов виртуального базового класса ***Complex_A*** и двух прямых базовых классов ***Complex_B*** и ***Complex_C*** их производным классом ***Complex_D***, где выбор между одноименными компонентными функциями ***add*()** прямых базовых классов осуществляется по типам аргументов вызова благодаря ***using-объявлениям*** этих функций в производном классе ***Complex_D***:

```
// C++ Множественное виртуальное наследование

#include <iostream>

using namespace std;

struct Complex_A
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов базового класса
  Complex_A(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  double re;
  double im;
};

struct Complex_B : virtual Complex_A
{
  // Компонентные функции – все общедоступные (public)
  // Конструктор объектов производного класса
  Complex_B(double r = 0.0, double i = 0.0) : Complex_A(r, i) {}
```

```
  // Сложение комплексных чисел
  Complex_B& add(Complex_B a, Complex_B b)
  {
    re = a.re + b.re;
    im = a.im + b.im;
    return *this;
  }
};

struct Complex_C : virtual Complex_A
{
  // Компонентные функции – все общедоступные (public)
  // Конструктор объектов производного класса
  Complex_C(double r = 0.0, double i = 0.0) : Complex_A(r, i) {}
  // Сложение комплексных чисел
  Complex_C& add(Complex_C a, double b)
  {
    re = a.re + b;
    return *this;
  }
};

struct Complex_D : Complex_B, Complex_C
{
  using Complex_B::add;
  using Complex_C::add;
  // Компонентные функции – все общедоступные (public)
  // Конструктор объектов производного класса
  Complex_D(double r = 0.0, double i = 0.0)
  : Complex_A(r, i), Complex_B(r, i), Complex_C(r, i) {}
  // Сложение комплексных чисел
  Complex_D& add(double a, Complex_D b)
  {
    re = b.re + a;
    return *this;
  }
};

int main()
{
  Complex_D x1(-1, 5);
  Complex_D x2(10, 7);
  x1.print();
  x2.print();
  x1.add(x1, x2);
  x1.print();
  x1.add(x1, 2);
  x1.print();
```

```
  x2.add(2, x2);
  x2.print();
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(9, 12)
(11, 12)
(12, 7)
```

В заключение отметим, что если какой-либо класс разработан с целью сделать его базовым (даже если он пока и не используется в качестве такового), он обязательно должен иметь открытый виртуальный деструктор. Такой класс не влияет на других пользователей иерархии классов, если они добавляют в нее новые производные от него классы, имеющих деструкторы. Отметим здесь также, что виртуальные базовые классы должны иметь конструкторы по умолчанию, в противном случае с такими классами очень сложно работать.

Виртуальные функции

➢ Пример 69

К механизму ***виртуальных функций*** обращаются в тех случаях, когда в базовый класс необходимо поместить компонентную функцию, которая должна по-разному выполняться в производных классах, т.е. в каждом производном классе требуется свой вариант такой функции. Виртуальные функции используются для поддержки ***динамического полиморфизма***. Основой виртуальных функций и динамического полиморфизма являются ссылки и указатели на объекты производных классов.

Напомним, что в C++ полиморфизм поддерживается тремя способами: при компиляции программы посредством перегрузки операторов и функций (статический полиморфизм) или посредством механизма шаблонов с использованием типа в качестве параметра при определении функций-шаблонов или классов-шаблонов (параметрический полиморфизм), а во время выполнения программы посредством виртуальных функций (динамический полиморфизм).

Рассмотрим поведение кода при наследовании невиртуальных компонентных функций с одинаковыми именами, типами возвращаемых значений и сигнатурами параметров. Если в базовом классе определена некоторая компонентная функция, то такая же функция (с тем же именем, того же типа и с той же сигнатурой параметров) может быть введена и в производном классе. Полные квалифицированные имена позволяют в теле какого-либо класса однозначно получить доступ к таким функциям. Если обращения к ним выполнены с помощью ссылок или указателей на объекты соответствующих классов, то выбор функции зависит только от типа ссылки или указателя, но не от их значения. Ранее с этой проблемой пришлось столкнуться при исследовании поведения невиртуального деструктора. Итак, выбор требуемой невиртуальной компонентной функции определяется уже при кодировании и не изменяется после компиляции (режим ***раннего*** или ***статического связывания***).

Начнем с примера открытого одиночного наследования компонентов базового класса ***A_based*** его производным классом ***B_derived***, который переопределяет невиртуальную нестатическую компонентную функцию базового класса ***print*()**:

```
// C++ Раннее (статическое) связывание

#include <iostream>

using namespace std;

struct A_based
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов базового класса
  A_based(int i = 0) : a_data(i) {}
  // Визуализация компонента данных
  void print()
  {
```

```
    cout << "A_based::data_member = " << a_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  int a_data;
};

struct B_derived : A_based
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов производного класса
  B_derived(int i = 0) : b_data(i) {}
  // Визуализация компонента данных
  void print()
  {
    cout << "B_derived::data_member = " << b_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  int b_data;
};

int main()
{
  A_based a(1);
  B_derived b(2);
  A_based* pointerA_based = &a;
  B_derived* pointerB_derived = &b;
  a.print();
  pointerA_based->print();
  b.print();
  pointerB_derived->print();
  // Доступ к наследуемому объекту A_based
  pointerA_based = &b;
  pointerA_based->print();
  A_based& referenceA_based = b;
  referenceA_based.print();
  return 0;
}
```

Результат работы программы:

```
A_based::data_member = 1
A_based::data_member = 1
B_derived::data_member = 2
B_derived::data_member = 2
A_based::data_member = 0
A_based::data_member = 0
```

➢ **Пример 70**

Наряду с режимом раннего или статического связывания существует и режим ***позднего*** (***отложенного***) или ***динамического связывания***, который предоставляется механизмом виртуальных функций. Любая нестатическая компонентная функция базового класса может стать виртуальной, если в ее объявлении используется спецификатор ***virtual***. В этом случае интерпретация вызова виртуальной функции через ссылку или указатель на базовый класс будет зависеть от значения ссылки или указателя, т.е. от типа объекта, для которого как раз и выполняется вызов. Этот процесс является реализацией принципа динамического полиморфизма. Говорят, что функция производного класса с тем же именем, того же типа и с той же сигнатурой параметров, что и виртуальная функция базового класса, замещает виртуальную функцию базового класса.

По существу, виртуальная функция реализует идею “один интерфейс, множество методов”, которая лежит в основе полиморфизма. Виртуальная функция в базовом классе задает свой ***интерфейс*** замещающим ее функциям, определенным в производных классах, а переопределение этой виртуальной функции в каждом производном классе определяет ее ***реализацию***, связанную со спецификой производного класса. Таким образом, переопределение виртуальной функции создает конкретный ***метод***. Классы, в которых объявлены виртуальные функции, называют ***полиморфными***. Чтобы объявление виртуальной функции работало в качестве интерфейса к функции производного класса, типы аргументов замещающей функции не должны отличаться от типов аргументов виртуальной функции базового класса, и только некоторые ослабления допускаются к типу возвращаемого значения.

Продолжая начатое исследование поведение кода, в базовом классе ***A_based*** теперь объявим виртуальной нестатическую компонентную функцию ***print*()**:

```
// C++ Позднее (динамическое) связывание

#include <iostream>

using namespace std;

struct A_based
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов базового класса
  A_based(int i = 0) : a_data(i) {}
  // Визуализация компонента данных
  virtual void print()
  {
    cout << "A_based::data_member = " << a_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  int a_data;
};
```

```
struct B_derived : A_based
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов производного класса
  B_derived(int i = 0) : b_data(i) {}
  // Визуализация компонента данных
  void print()
  {
    cout << "B_derived::data_member = " << b_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  int b_data;
};

int main()
{
  A_based a(1);
  B_derived b(2);
  A_based* pointerA_based = &a;
  B_derived* pointerB_derived = &b;
  a.print();
  pointerA_based->print();
  b.print();
  pointerB_derived->print();
  // Доступ к наследуемому объекту A_based
  pointerA_based = &b;
  pointerA_based->print();
  A_based& referenceA_based = b;
  referenceA_based.print();
  return 0;
}
```

Результат работы программы:

```
A_based::data_member = 1
A_based::data_member = 1
B_derived::data_member = 2
B_derived::data_member = 2
B_derived::data_member = 2
B_derived::data_member = 2
```

Как видим, вызовы компонентной функции ***print*()**, выполненные с помощью указателей на объекты классов ***A_based*** и ***B_derived***, а также ссылки на объект класса ***B_derived*** зависят теперь от значения указателя и ссылки, т.е. от типа объекта, для которого как раз и выполнялся вызов.

➢ **Пример 71**

Итак, при освобождении свободной памяти, захваченной для динамических объектов производных классов, лишь механизм виртуализации деструкторов может обеспечить их вызов не по типу указателя, а по его значению. Именно поэтому так важны виртуальные деструкторы базовых классов.

Проиллюстрируем теперь эту возможность на примере открытого одиночного наследования компонентов базового класса ***Complex_based*** его производным классом ***Complex_derived***, который переопределяет наследуемые компонентные данные и компонентную функцию ***print*()**:

```
// C++ Одиночное наследование

#include <iostream>

using namespace std;

struct Complex_based
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов базового класса
  Complex_based(double r = 0.0, double i = 0.0) : re(r), im(i)
  {
    cout << "Конструктор базового класса" << endl;
  }
  // Деструктор объектов базового класса
  virtual ~Complex_based()
  {
    cout << "Деструктор базового класса" << endl;
  }
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  double re;
  double im;
};

struct Complex_derived : Complex_based
{
  // Компонентные функции – все общедоступные (public)
  // Конструктор объектов производного класса
  Complex_derived(double r = 0.0, double i = 0.0)
  : Complex_based(r), re(r), im(i)
  {
```

```
    cout << "Конструктор производного класса" << endl;
  }
  // Деструктор объектов производного класса
  ~Complex_derived()
  {
    cout << "Деструктор производного класса" << endl;
  }
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  double re;
  double im;
};

int main()
{
  cout << sizeof(Complex_based) << endl;
  cout << sizeof(Complex_derived) << endl;
  Complex_based* p = new Complex_derived(-1, 5);
  p->print();
  ((Complex_derived*)p)->print();
  delete p;
  return 0;
}
```

Результат работы программы для платформы Windows:

```
24
40
Конструктор базового класса
Конструктор производного класса
(-1, 0)
(-1, 5)
Деструктор производного класса
Деструктор базового класса
```

Как видим, при освобождении памяти, захваченной для динамического объекта производного класса, через указатель на его базовый класс теперь вызываются оба деструктора. Тем самым так называемой “утечки” памяти здесь нет, так как теперь освобождается память для всего объекта производного класса, а не только лишь для его наследуемого объекта базового класса.

➢ **Пример 72**

Итак, чтобы объявление виртуальной функции работало в качестве интерфейса к функции производного класса, типы аргументов замещающей функции не должны отличаться от типов аргументов виртуальной функции базового класса, и только некоторые ослабления допускаются по отношению к типу возвращаемого значения. Например, если исходный тип возвращаемого значения был ***имя_базового_класса****, то тип возвращаемого значения замещающей функции может быть ослаблен до ***имя_производного_класса**** при условии, что класс ***имя_базового_класса*** является открытым базовым классом для класса ***имя_производного_класса***. Аналогично, вместо ***имя_базового_класса&*** тип возвращаемого значения может быть ослаблен до ***имя_производного_класса&***.

Проиллюстрируем одно из правил "ослабления типа" на примере так называемых "виртуальных конструкторов" – виртуальных компонентных функций класса, неявно вызывающих его конструкторы и возвращающих созданные объекты. Зачастую это необходимо для создания объектов, точный тип которых не известен. Представим иерархию классов с открытым одиночным наследованием, где "виртуальные конструкторы" ***newA_based*()** и ***cloneA_based*()** определяются в базовом классе ***A_based*** и переопределяются в производном от него классе ***B_derived***:

```
// C++ "Виртуальные конструкторы"

#include <iostream>

using namespace std;

struct A_based
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов базового класса
  A_based(int i = 0) : a_data(i)
  {
    cout << "Конструктор базового класса A_based" << endl;
  }
  // Деструктор объектов базового класса
  virtual ~A_based()
  {
    cout << "Деструктор базового класса A_based" << endl;
  }
  // "Виртуальный конструктор" базового класса
  virtual A_based* newA_based()
  {
    return new A_based;
  }
  // "Виртуальный конструктор копирования" базового класса
  virtual A_based* cloneA_based()
  {
```

```
    return new A_based(*this);
  }
  // Визуализация компонента данных
  virtual void print()
  {
    cout << a_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  int a_data;
};

struct B_derived : A_based
{
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов производного класса
  B_derived(float f = 0.0) : b_data(f)
  {
    cout << "Конструктор производного класса B_derived" << endl;
  }
  // Деструктор объектов производного класса
  ~B_derived()
  {
    cout << "Деструктор производного класса B_derived" << endl;
  }
  // "Виртуальный конструктор" производного класса
  B_derived* newA_based()
  {
    return new B_derived;
  }
  // "Виртуальный конструктор копирования" производного класса
  B_derived* cloneA_based()
  {
    return new B_derived(*this);
  }
  // Визуализация компонента данных
  void print()
  {
    cout << b_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  float b_data;
};

int main()
{
  A_based a(1);
  A_based* p = &a;
```

```
  p->print();
  A_based* q = p->newA_based();
  q->print();
  A_based* r = p->cloneA_based();
  r->print();
  delete q;
  B_derived b(2);
  B_derived* s = &b;
  s->print();
  B_derived* t = s->newA_based();
  t->print();
  B_derived* u = s->cloneA_based();
  u->print();
  delete t;
  return 0;
}
```

Результат работы программы:

```
Конструктор базового класса A_based
1
Конструктор базового класса A_based
0
1
Деструктор базового класса A_based
Конструктор базового класса A_based
Конструктор производного класса B_derived
2
Конструктор базового класса A_based
Конструктор производного класса B_derived
0
2
Деструктор производного класса B_derived
Деструктор базового класса A_based
Деструктор производного класса B_derived
Деструктор базового класса A_based
Деструктор базового класса A_based
```

Как видим, значения, возвращаемые “виртуальными конструкторами” ***B_derived*::*newA_based*()** и ***B_derived*::*cloneA_based*()**, имеют тип ***B_derived****, а не ***A_based****. Это как раз и позволяет создавать копии объектов класса ***B_derived*** без потери информации о типе.

➢ Пример 73

В заключение приведем пример реализации идеи “один интерфейс, множество методов”, который в дальнейшем станет основой для обобщения, связанного с переходом к абстрактным базовым классам.

Зададимся целью вычисления площади какой-либо геометрической фигуры по двум ее размерностям, например, прямоугольника или треугольника. Создадим базовый класс ***Area***, компонентные данные которого определяют две размерности фигуры. В базовом классе определим две компонентные функции: нестатическую функцию ***set*()** для задания двух размерностей фигуры и виртуальную функцию ***get_area*()**, которая при переопределении в производном классе возвращает значение площади фигуры, вид которой определяется производным классом. В этом случае определение ***get_area*()** в базовом классе задает интерфейс, а конкретные реализации, т.е. методы, определяются производными классами ***Rectangle*** и ***Triangle***, каждый из которых наследует базовый класс ***Area***.

Проиллюстрируем первую реализацию этой идеи с помощью иерархии классов с открытым одиночным наследованием, где виртуальная функция ***get_area*()** базового класса, являясь по сути своеобразной “заглушкой”, не будет выполнять каких-либо значимых действий, а будет служить лишь основой для замещающих ее функций в производных классах:

```
// C++ Реализация идеи "один интерфейс, множество методов"

#include <iostream>

using namespace std;

class Area
{
  protected:
  // Компонентные данные - все защищенные (protected)
  double dimension1;
  double dimension2;
  public:
  // Компонентные функции - все общедоступные (public)
  // Задание размерностей фигуры
  void set(double figureD1 = 1.0, double figureD2 = 1.0)
  {
    dimension1 = figureD1;
    dimension2 = figureD2;
  }
  // Интерфейс - площадь фигуры
  virtual double get_area()
  {
    return 0.0;
  }
};
```

```
class Rectangle : public Area
{
  public:
  // Компонентные функции - все общедоступные (public)
  // Метод - площадь прямоугольника
  double get_area()
  {
    return dimension1 * dimension2;
  }
};

class Triangle : public Area
{
  public:
  // Компонентные функции - все общедоступные (public)
  // Метод - площадь треугольника
  double get_area()
  {
    return dimension1 * dimension2 / 2;
  }
};

int main()
{
  Rectangle r;
  Triangle t;
  r.set(1.2, 3.4);
  t.set(1.2, 3.4);
  Area* p = &r;
  cout << "Площадь прямоугольника = " << p->get_area() << endl;
  p = &t;
  cout << "Площадь треугольника = " << p->get_area() << endl;
  return 0;
}
```

Результат работы программы:

```
Площадь прямоугольника = 4.08
Площадь треугольника = 2.04
```

Как видим, виртуальная функция, объявленная в базовом классе, может и не выполнять никаких значимых действий, конкретные действия выполняются лишь благодаря механизму переопределения виртуальной функции базового класса в каждом его производном классе. Это вполне обычная ситуация, базовый класс просто содержит базовый набор компонентов, для которых производный класс задает все недостающее. При этом, как видим, можно даже обойтись и без конструкторов, определяемых самим пользователем, полагаясь лишь на те, которые генерируются компилятором по умолчанию.

Абстрактные классы

➢ Пример 74

Итак, если в виртуальной функции базового класса отсутствует значимое действие, то хотя бы в одном из его производных классов эта функция обязательно должна быть переопределена. В C++ реализация этого принципа обеспечивается механизмом ***чисто виртуальных функций***. Класс, в котором объявляется хотя бы одна чисто виртуальная функция, становится ***абстрактным классом***, и его производные классы тоже станут абстрактными, если в них не будет переопределена эта функция.

Продолжая развивать реализацию идеи “один интерфейс, множество методов”, приведем пример открытого одиночного наследования компонентов абстрактного базового класса ***Area*** его производными классами ***Rectangle*** и ***Triangle***:

```
// C++ Абстрактный класс

#include <iostream>

using namespace std;

class Area
{
  protected:
  // Компонентные данные - все защищенные (protected)
  double dimension1;
  double dimension2;
  public:
  // Компонентные функции - все общедоступные (public)
  // Задание размерностей фигуры
  void set(double figureD1 = 1.0, double figureD2 = 1.0)
  {
    dimension1 = figureD1;
    dimension2 = figureD2;
  }
  // Интерфейс - площадь фигуры
  virtual double get_area() = 0;
};

class Rectangle : public Area
{
  public:
  // Компонентные функции - все общедоступные (public)
  // Метод - площадь прямоугольника
  double get_area()
  {
    return dimension1 * dimension2;
  }
};
```

```
class Triangle : public Area
{
  public:
  // Компонентные функции - все общедоступные (public)
  // Метод - площадь треугольника
  double get_area()
  {
    return dimension1 * dimension2 / 2;
  }
};

int main()
{
  Rectangle r;
  Triangle t;
  r.set(1.2, 3.4);
  t.set(1.2, 3.4);
  Area* p = &r;
  cout << "Площадь прямоугольника = " << p->get_area() << endl;
  p = &t;
  cout << "Площадь треугольника = " << p->get_area() << endl;
  return 0;
}
```

Результат работы программы:

```
Площадь прямоугольника = 4.08
Площадь треугольника = 2.04
```

Как видим, создание чисто виртуальных функций гарантирует, что производные классы обеспечат их переопределение. Чисто виртуальная функция недоступна для вызовов, она служит лишь основой для замещающих ее функций в производных классах. В терминологии объектно-ориентированного программирования такое “отложенное” решение о реализации функции называется ***отсроченным методом***.

➢ Пример 75

Механизм абстрактных классов разработан для представления общих понятий, которые в дальнейшем предполагается конкретизировать. Критерий общности, например, может отражать сходство наборов операций или сходство реализаций. Эти общие понятия невозможно использовать непосредственно, но на их основе можно построить производные классы, пригодные для описания конкретных объектов.

Очевидно, что рассматриваемый ранее способ перехода к иерархии с абстрактным классом является всего лишь одним из первых шагов на пути представления общих понятий. Следующим шагом на этом пути здесь вполне может стать реализация одного из критериев общности, отражающего, например, сходство набора операций инициализации компонентных данных с помощью конструкторов:

```
// C++ Абстрактный класс

#include <iostream>

using namespace std;

class Area
{
  protected:
  // Компонентные данные - все защищенные (protected)
  double dimension1;
  double dimension2;
  // Компонентные функции - все защищенные (protected)
  // Конструктор базового абстрактного класса
  Area(double figureD1, double figureD2)
  : dimension1(figureD1), dimension2(figureD2) {}
  public:
  // Компонентные функции - все общедоступные (public)
  // Интерфейс - площадь фигуры
  virtual double get_area() = 0;
};

class Rectangle : public Area
{
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов производного класса
  Rectangle(double figureD1 = 1.0, double figureD2 = 1.0)
  : Area(figureD1, figureD2) {}
  // Метод - площадь прямоугольника
  double get_area()
  {
    return dimension1 * dimension2;
  }
};
```

```
class Triangle : public Area
{
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов производного класса
  Triangle(double figureD1 = 1.0, double figureD2 = 1.0)
  : Area(figureD1, figureD2) {}
  // Метод - площадь треугольника
  double get_area()
  {
    return dimension1 * dimension2 / 2;
  }
};

int main()
{
  Rectangle r(1.2, 3.4);
  Triangle t(1.2, 3.4);
  Area* p = &r;
  cout << "Площадь прямоугольника = " << p->get_area() << endl;
  p = &t;
  cout << "Площадь треугольника = " << p->get_area() << endl;
  return 0;
}
```

Результат работы программы:

```
Площадь прямоугольника = 4.08
Площадь треугольника = 2.04
```

Локальные классы

➢ Пример 76

Класс может быть определен внутри блока, например, внутри определения функции. Такой класс называется ***локальным*** классом. Имя локального класса является локальным в окружающем контексте. Локальный класс имеет тот же доступ к именам вне функции, что и сама функция. Компонентные функции локального класса могут быть только встроенными. В локальном классе разрешено использовать из окружающего контекста только имена типов, статические переменные, внешние переменные, внешние функции и элементы перечислений.

Ранее отмечалось, что при объявлении массива объектов класса явное указание аргументов вызова для его конструктора возможно не только в случае использования списка инициализации в стиле C, но также и в случае использования локального класса. Например, при объявлении одномерного массива объектов локального производного класса ***B_derived*** из иерархии с открытым одиночным наследованием инициализация его элементов будет выполнена благодаря явному вызову конструктора базового класса ***A_based*** для указанных аргументов:

```
// C++ Локальный класс

#include <iostream>

using namespace std;

class A_based
{
  protected:
  // Компонентные данные - все защищенные (protected)
  int a_data;
  // Компонентные функции - все защищенные (protected)
  // Конструктор объектов базового класса
  A_based(int i = 0) : a_data(i)
  {
    cout << "Конструктор базового класса A_based" << endl;
  }
  // Деструктор объектов базового класса
  ~A_based()
  {
    cout << "Деструктор базового класса A_based" << endl;
  }
  // Визуализация компонента данных
  friend ostream& operator<<(ostream& stream, A_based& item)
  {
    return stream << item.a_data;
  }
};
```

```
const int size = 2;

void create()
{
  static int value;
  class B_derived : public A_based
  {
    public:
    // Компонентные функции - все общедоступные (public)
    // Конструктор объектов производного класса
    B_derived() : A_based(value++)
    {
      cout << "Конструктор производного класса B_derived" << endl;
    }
    // Деструктор объектов производного класса
    ~B_derived()
    {
      cout << "Деструктор производного класса B_derived" << endl;
    }
  };
  B_derived array[size];
  for (int i = 0; i < size; ++i) cout << array[i] << endl;
}

int main()
{
  create();
  return 0;
}
```

Результат работы программы:

```
Конструктор базового класса A_based
Конструктор производного класса B_derived
Конструктор базового класса A_based
Конструктор производного класса B_derived
0
1
Деструктор производного класса B_derived
Деструктор базового класса A_based
Деструктор производного класса B_derived
Деструктор базового класса A_based
```

➢ **Пример 77**

Подобным образом можно инициализировать также и динамические одномерные массивы объектов локального производного класса ***B_derived*** из иерархии с открытым одиночным наследованием.

Если раньше размер массива для функции ***create*()** определялся при помощи константы в глобальной области видимости, то теперь размер массива передадим этой функции в качестве ее аргумента вызова:

```
// C++ Локальный класс

#include <iostream>

using namespace std;

class A_based
{
  protected:
  // Компонентные данные - все защищенные (protected)
  int a_data;
  // Компонентные функции - все защищенные (protected)
  // Конструктор объектов базового класса
  A_based(int i = 0) : a_data(i) {}
  // Деструктор объектов базового класса
  virtual ~A_based() {}
  // Визуализация компонента данных
  friend ostream& operator<<(ostream& stream, A_based& item)
  {
    return stream << item.a_data;
  }
};

void create(int size)
{
  static int value;
  class B_derived : public A_based
  {
    public:
    // Компонентные функции - все общедоступные (public)
    // Конструктор объектов производного класса
    B_derived() : A_based(value++) {}
    // Деструктор объектов производного класса
    ~B_derived() {}
  };
  B_derived* base = new B_derived[size];
  for (int i = 0; i < size; ++i) cout << base[i] << endl;
  delete[] base;
}
```

```
int main()
{
  create(5);
  return 0;
}
```

Результат работы программы:

```
0
1
2
3
4
```

Отметим, что здесь даже нет необходимости в использовании виртуального деструктора, так как, с одной стороны, указатель ***base*** ссылается на настоящий тип объекта в свободной памяти, и с другой стороны, тип, на который ссылается указатель ***base***, имеет тот же размер, что и настоящий тип.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Шилдт Г. Теория и практика C++: Пер. с англ. – СПб.: BHV-Санкт-Петербург, 1996. – 416 с.
2. Паппас К., Мюррей У. Руководство программиста по C/C++. В 2 кн. Кн. I. – М.: “СК Пресс”, 1997. – 520 с.
3. Паппас К., Мюррей У. Руководство программиста по C/C++. В 2 кн. Кн. II. – М.: “СК Пресс”, 1997. – 452 с.
4. Сэвитч У. C++ в примерах: Пер. с англ. – М.: ЭКОМ, 1997. – 736 с.
5. Шилдт Г. Самоучитель C++: Пер. с англ. – СПб.: BHV-Санкт-Петербург, 1997. – 512 с.
6. Дейтел Х., Дейтел П. Как программировать на C++: Пер. с англ. – М.: ЗАО “Издательство БИНОМ”, 1998. – 1024 с.
7. Страуструп Б. Язык программирования C++, 3-е изд.: Пер. с англ. – СПб.; М.: “Невский Диалект” – “Изд-во БИНОМ”, 1999. – 991 с.
8. Топп У., Форд У. Структуры данных в C++: Пер. с англ. – М.: ЗАО “Издательство БИНОМ”, 1999. – 816 с.
9. Страуструп Б. Дизайн и эволюция C++: Пер. с англ. – М.: ДМК Пресс, 2000. – 448 с.
10. Дёмкин В.М. Основы объектно-ориентированного программирования в примерах на C++: Учебное пособие. – Н.Новгород: НФ ГУ-ВШЭ, 2005. – 148 с.

СОДЕРЖАНИЕ