Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет информационных технологий и управления

Кафедра информационных технологий автоматизированных систем

А. А. Навроцкий

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Допущено
Министерством образования Республики Беларусь
в качестве учебного пособия для студентов
учреждений высшего образования по специальностям
«Системы управления информацией»,
«Киберфизические системы»

Минск БГУИР 2025

УДК 004.42(075.8)
ББК 32.973я73
Н15

Р е ц е н з е н т ы:

кафедра программной инженерии
учреждения образования «Белорусский государственный
технологический университет»
(протокол № 3 от 29.10.2024);

заведующий кафедрой экономической информатики
учреждения образования «Белорусский государственный
экономический университет»
кандидат технических наук, доцент Л. В. Серебряная

Навроцкий, А. А.

Н15 Объектно-ориентированное программирование : учеб. пособие / А. А. Навроцкий. – Минск : БГУИР, 2025. – 176 с. : ил.
ISBN 978-985-543-821-3.

Содержит теоретические сведения об объектно-ориентированном программировании на языке C++.

Предназначено для студентов высших учебных заведений, изучающих дисциплину «Объектно-ориентированное программирование».

УДК 004.42(075.8)
ББК 32.973я73

ISBN 978-985-543-821-3

СОДЕРЖАНИЕ

Введение

Учебное пособие посвящено изучению объектно-ориентированного подхода (ООП) в программировании. Известно, что ООП не изменяет процесс выполнения программы, а является другим способом ее организации. Большая часть операторов идентична операторам процедурного языка, а методы классов идентичны функциям. Следовательно, для освоения ООП требуется наличие хороших навыков в процедурном программировании. В учебном пособии не содержатся сведения об элементарных конструкциях языка C++, т. к. предполагается, что они были изучены при освоении дисциплины «Основы алгоритмизации и программирования».

Материал учебного пособия предполагает последовательное освоение – от простых понятий к более сложным. Пропуск при изучении одной главы может исключить качественное освоение материала последующих глав. К сожалению, из-за сложности материала некоторые понятия в учебном пособии используются до приведения их детального описания, однако к концу пособия все станет на свои места.

Учебное пособие содержит большое количество примеров, в которых рассматриваются особенности использования изучаемых концепций и механизмов объектно-ориентированного программирования.

Все примеры программ были выполнены и проверены в Microsoft Visual C++.

1. Основные понятия объектно-ориентированного программирования

Основополагающей идеей объектно-ориентированного подхода является ***абстракция*** (*abstraction*) – выделение существенных для решения конкретной задачи признаков и исключение несущественных. Это позволяет разрабатывать программы, не вникая в сложность и детали нижележащего кода.

В основе ООП лежит три основных принципа: инкапсуляция, наследование и полиморфизм.

Инкапсуляция (*encapsulation*) – механизм, связывающий вместе данные и методы и защищающий их от произвольного внешнего доступа.

Описание инкапсулированного объекта называется ***классом***.

Инкапсуляция позволяет защитить данные от некорректного использования путем ограничения доступа. Чтение и изменение данных в этом случае осуществляется с помощью функций, доступных другим объектам: геттеров (от *get* – «получать»), предназначенных для получения данных объекта и передачи их наружу, и сеттеров (от *set* – «устанавливать»), которые позволяют менять значения этих данных. Таким образом данные защищаются от несанкционированного изменения со стороны других объектов.

Пример взаимодействия двух объектов представлен на рис. 1.1.

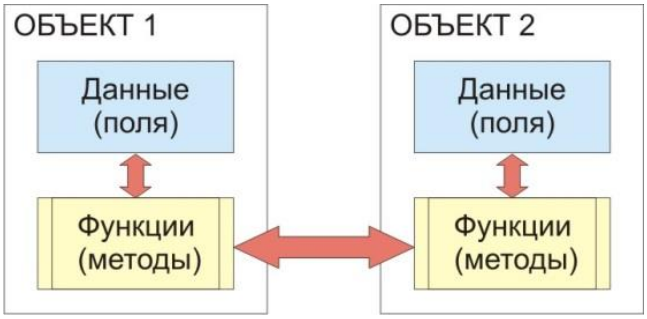

Рис. 1.1

На рис. 1.2 представлен пример взаимодействия студентов, преподавателей и деканата. Видно, что староста группы имеет доступ к журналу группы, но не имеет доступа к оценкам. Для того чтобы получить информацию об оценках, старосте необходимо обратиться к преподавателю, который имеет доступ к оценкам. Аналогичным образом необходимо обратиться к секретарю деканата за получением сведений о студентах. За доступ к данным и их модификацию отвечают конкретные лица (функции). Таким образом осуществляется защита данных.

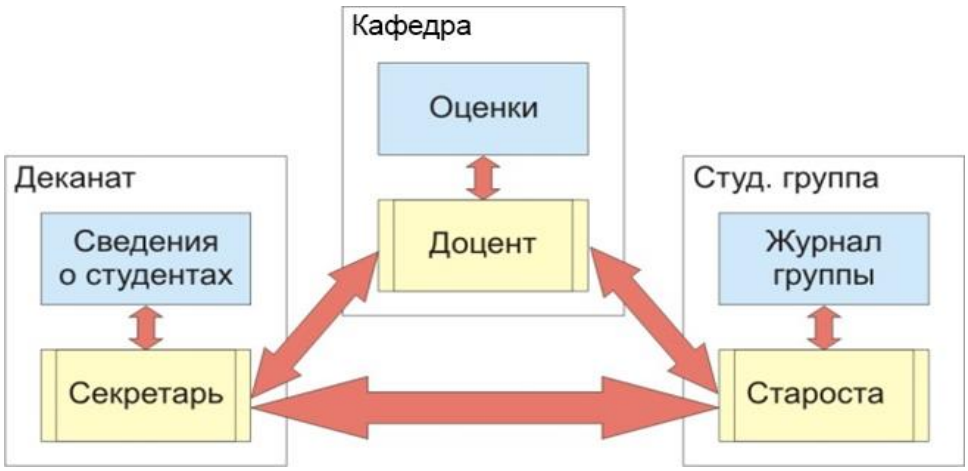

Рис. 1.2

Наследование (*inheritance*) – использование элементов (полей и методов) ранее определенных классов при создании новых классов.

Принцип, положенный в основу наследования, заключается в том, что каждый подкласс обладает как свойствами, присущими родительскому классу, так и собственными, уникальными, свойствами (рис. 1.3).

Рис. 1.3

Например, класс СТУДЕНТ определяет, что каждый конкретный студент (объект класса СТУДЕНТ) будет обладать определенным, заданным набором свойств (например, Ф. И. О., номер телефона, специальность, возраст).

На базе класса СТУДЕНТ можно создать новые классы, которые, не исключая имеющиеся свойства, будут обогащаться новыми свойствами, например, изучаемыми дисциплинами. Таким образом можно создать такие классы, как СТУДЕНТ 1 КУРСА, СУДЕНТ 2 КУРСА и т. д.

Класс, который порождает все остальные классы, называется ***базовым (родительским)*** классом. Классы, которые наследуют свойства базового класса, называются ***производными*** классами.

Наследование позволяет легко использовать уже написанный код, внося в него необходимые изменения. Все производные классы наследуют уже внесенные изменения, в то же время в каждый производный класс можно вносить свои изменения.

Полиморфизм (*polymorphizm*) – возможность использования одного имени для решения уникальных для каждого уровня иерархии производных объектов задач.

Обеспечивается это перегрузкой методов. В результате в объекте-родителе и объекте-потомке будут действовать одноименные методы с различными алгоритмами.

Выбор используемого метода осуществляется компилятором автоматически в зависимости от текущего состояния программы.

Различают три способа реализации полиморфизма:

- *статический полиморфизм* − реализуется на этапе компиляции путем перегрузки функций и операций;
- *динамический полиморфизм* − реализуется во время выполнения программы с использованием виртуальных функций;
- *параметрический полиморфизм* − реализуется на этапе компиляции с использованием механизма шаблонов.

Контрольные вопросы

1. Что такое инкапсуляция?
2. Что такое абстракция?
3. Что такое полиморфизм?
4. Укажите способы реализации полиморфизма.

2. Классы, объекты

2.1. Понятие класса

Класс – пользовательский тип данных, объединяющий данные и функции для отображения модели реального мира. Данные класса называются ***полями***, а функции класса – ***методами***. Поля и методы называются ***элементами*** класса.

Объявление класса не приводит к выделению памяти. Память выделяется для ***экземпляров (объектов)*** класса.

Описание простейшего класса:

```
class <имя_класса>
{
private: // По умолчанию, может отсутствовать
// Описание скрытых элементов
protected:
// Описание защищенных элементов
public:
// Описание общедоступных элементов
} ; // Точка с запятой в конце описания класса!
```

Модификаторы доступа определяют область видимости элементов класса:

- private (локальный) – элементы доступны только внутри класса. Данный вид доступа определен по умолчанию, поэтому ключевое слово private сразу после описания имени класса можно не указывать. Получить или изменить значения элементов с модификатором private можно только через обращение к соответствующим методам текущего класса;
- protected (защищенный) – доступ к элементам разрешен внутри текущего и производных классов;
- public (глобальный) – разрешен доступ к элементам из любого места программы.

Как правило, поля класса скрываются, а методы остаются общедоступными.

Действие любого модификатора распространяется до следующего модификатора или до конца класса. Разрешено задавать несколько одинаковых секций, а также менять порядок следования секций.

Поля класса:

- могут иметь любой тип, кроме типа своего класса (могут быть указателями или ссылками на этот класс);
- могут быть описаны со спецификатором const, при этом они инициализируются только один раз (с помощью конструктора) и не могут изменяться;
- могут быть описаны со спецификатором static.

Пример 2.1. Простейший класс для ввода-вывода поля *x*:

```
#include <iostream>
using namespace std;
class Cls {
private:
	int x;
	void f() {
	 cout << x << endl;
	}
public:
	void input(int a) {
		x = a;
	}
	void output() {
		cout << x << endl;
	}
};
int main() {
	Cls d, s;
// d.x = 1234; // Ошибка! Невозможно обратиться к private-элементу
	d.input(1234);
// d.f(); // Ошибка! Невозможно обратиться к private-элементу
	d.output(); // Выводит: 1234
	return 0;
}
```

Методы, описание которых находится внутри класса, называются ***встраиваемыми***.

Пример 2.2. Класс с невстраиваемыми методами:

```
class Cls {
private:
	int x;
public:
	void input(int a);
	void output();
};

void Cls::input(int a) {
	x = a;
}
void Cls::output() {
	cout << x << endl;
}
```

Символ «::» является знаком операции глобального разрешения, которая устанавливает взаимосвязь метода и класса, к которому относится этот метод.

2.2. Конструктор и деструктор класса

При создании каждого объекта выделяется память, достаточная для хранения всех его полей, для чего автоматически вызывается ***конструктор*** – специальный метод, выполняющий инициализацию. Имя конструктора должно полностью совпадать с именем класса (таким способом компилятор отличает конструктор от других методов класса).

Конструктор имеет ряд отличий от других методов:

- конструктор не имеет возвращаемого значения. Это объясняется тем, что конструктор автоматически вызывается системой и не имеет вызывающей программы, в которую можно вернуть значение. Результатом работы конструктора является созданный объект;
- класс может иметь несколько конструкторов, имеющих разные параметры (нужный конструктор выбирается исходя из соответствия количества и типов параметров);
- параметры конструктора могут иметь любой тип (кроме типа своего класса). Один из конструкторов может содержать параметры по умолчанию;
- при отсутствии в описании класса конструктора он создается компилятором автоматически. При наличии в описании класса неинициализированных констант и ссылок компилятор сгенерирует ошибку, т. к. автоматически создаваемый конструктор не может инициализировать их значения;
- конструктор не возвращает сообщение об ошибке инициализации (следует использовать механизмы обработки исключительных ситуаций);
- при описании конструктора нельзя использовать спецификаторы const, virtual и static;
- конструкторы глобальных объектов вызываются до вызова функции main. Конструкторы локальных объектов вызываются при вхождении в их зону видимости.

Конструкторы могут быть следующих основных типов:

- конструкторы по умолчанию (без параметров);
- параметризированные конструкторы;
- конструкторы копирования;
- конструкторы перемещения.

Конструктор по умолчанию используется в случае, если для создания объекта не требуется задание параметров.

В этом случае при создании объекта указывается только идентификатор переменной:

```
class Cls {
public:
	Cls() { }
};

void main() {
	Cls g;
}
```

Конструктором по умолчанию будет считаться и конструктор, у которого все параметры имеют значения по умолчанию:

```
class Cls {
private:
	int x, y;
public:
	Cls(int a = 0, int b = 3) {
		x = a;
		y = b;
	}
};

void main() {
	Cls g;
}
```

Пример 2.3. Класс, предназначенный для увеличения значения поля *x* на единицу. Конструктор по умолчанию инициализирует поле *x* значением 0.

```
class Cls
{
private:
	int x;
public:
	Cls() : x(0) {
	}
	void count() {
		x++;
	}
	void shows() {
		cout << x << endl;
	}
};
```

```
void main() {
        Cls d;
        d.count();
        d.shows(); // Выводит: 1
        d.count();
        d.shows(); // Выводит: 2
}
```

Инициализация поля расположена между заголовком конструктора и его телом и отделена от заголовка двоеточием. Инициализирующее значение помещается в скобки после имени инициализируемого поля. Если необходимо инициализировать сразу несколько полей класса, то они отделяются друг от друга запятыми:

```
Cls() : x(0), y(34), z(828)   {    }
```

Инициализация полей с помощью списка инициализации происходит до начала выполнения тела конструктора. Инициализация полей в теле конструктора возможна, но не рекомендуется.

Список инициализации конструктора используется и для задания начальных значений констант и ссылок:

```
…
private:
        const b;
public:
        Cls() : b(111)  {
                // b = 111;   // Ошибка!
        }
```

Если в классе нет объявленных конструкторов, то компилятор автоматически создает *неявный конструктор по умолчанию*.

Неявный конструктор предназначен только для создания объекта класса и не инициализирует значения его полей. В этом случае рекомендуется инициализировать элементы в определении класса:

```
class Cls {
private:
        int x{3};   // Инициализация поля x значением 3
        int y = 5; // Инициализация поля y значением 5
public:
        void shows() {
                cout << x <<" "<< y << endl;
        }
};
```

```
void main() {
        Cls d;
        d.shows();  // Выводит: 3 5
}
```

Если в объявлении класса имеются другие конструкторы, то конструктор по умолчанию автоматически не создается.

```
class Cls1 {
        int x;
};
class Cls2 {
        int x;
public:
        Cls2(int b) : x(b) { }
};
void main() {
        Cls1 p;
        //  Cls2 d; // Ошибка! Требуется конструктор по умолчанию
}
```

Для того чтобы разрешить компилятору автоматически создавать конструктор по умолчанию (при наличии других конструкторов), используется ключевое слово default. Например:

```
class Cls {
        int x;
public:
        Cls() = default;
        Cls(int b) : x(b) { }
};
void main() {
        Cls p;  // Будет создан конструктор по умолчанию
        Cls p(0);
}
```

В случаях, когда использование неявного конструктора по умолчанию приводит к появлению неинициализированных переменных, можно запретить создание такого конструктора, определив его как delete:

```
class Cls {
        int x;
```

```
public:
        Cls() = delete;
};

void main() {
// Cls d; // Ошибка! Запрещено ссылаться на конструктор по умолчанию
}
```

Параметризированный конструктор используется для инициализации полей объекта и для перегрузки конструкторов. Объявление параметров конструкторов аналогично объявлению параметров функций:

```
class Cls {
      int x, y;
   public:
      Cls(int a) : x(a) { }
      void shows() {
         cout << x << endl;
      }
       };
   void main() {
      Cls p(3);
      p.shows();   // Выводит: 3
   }
```

Объект класса может быть объявлен двумя способами:

```
Cls p(3);
```

или

```
Cls p = 3;
```

В последнем случае инструкция Cls p = 3 автоматически конвертируется в форму Cls p(3) .

Если использование операции присваивания снижает читабельность программы (например, указываемое значение не инициализирует переменную, а выделяет память), то можно запретить автоматическое конвертирование использованием спецификатора explicit, который запрещает неявное преобразование типов аргументов.

Конструктор, объявленный без спецификатора explicit, называют ***конструктором преобразования***. Конструктор преобразования неявно приводит типы аргументов к типам параметров конструктора.

Пример 2.4. Способы вызова перегруженных конструкторов:

```
class Cls {
   int x, y, z = 0;
public:
   Cls(int a) : x(a), y(5) { }       // Конструктор преобразования 1
   Cls(int a, int b) : x(a), y(b) { }  // Конструктор преобразования 2
explicit Cls(int a, int b, int c) : x(a), y(b), z(c) {} // Неконверт. конструктор
   void shows() {
      cout << x << " " << y << " " << z << endl;
   }
};
void main() {
   Cls p1 = 24; // Конструктор преобразования 1
   p1.shows(); // Выводит: 24 5 0
   Cls p2(30); // Конструктор преобразования 1
   p2.shows(); // Выводит: 30 5 0
   Cls p3{40}; // Конструктор преобразования 1
   p3.shows(); // Выводит: 40 5 0
   Cls p4 = (Cls)99; // Конструктор преобразования 1
   p4.shows(); // Выводит: 99 5 0
   Cls p5 = {2, 3};
   p5.shows(); // Выводит: 2 3 0
   Cls p6(7, 4); // Конструктор преобразования 2
   p6.shows(); // Выводит: 7 4 0
   Cls p7{8, 2}; // Конструктор преобразования 2
   p7.shows(); // Выводит: 8 2 0

  // Cls p8 = { 1, 2, 3 }; //  Ошибка!
   Cls p9(4, 5, 6);      // Неконвертирующий конструктор
   p9.shows(); // Выводит: 4 5 6
   Cls p10{7, 8, 9 };    // Неконвертирующий конструктор
   p10.shows(); // Выводит: 7 8 9
}
```

Конструктор копирования позволяет инициализировать поля создаваемого объекта значениями полей другого объекта этого же класса.

Конструктор копирования может создаваться автоматически. В этом случае он копирует все поля одного объекта в соответствующие поля другого объекта:

```
class Cls {
   int x;
```

```
  public:
    void init(int a) {
      x = a;
    }
    void shows() {
      cout << x << endl;
    }
};
void main() {
  Cls p1;
  p1.init(7);
  Cls p2(p1);  // или Cls p2 = p1;
  p2.shows();  // Выводит: 7
}
```

При необходимости конструктор копирования может быть описан явно:

```
Cls(Cls &d) {
      x = d.x;
    }
```

Различают ***поверхностное копирование*** (рассмотрено выше), которое поэлементно копирует данные, и ***глубокое копирование***, которое осуществляет копирование не только данных, но и динамических структур.

Необходимость использования глубокого копирования вызвана тем, что при поверхностном копировании будут скопированы все данные и все адреса, т. е. все объекты будут адресоваться к одним и тем же динамическим структурам.

Пример 2.5. Использование глубокого копирования при работе с динамическими массивами:

```
class Cls
 {
private:
      int* mas;
      int n;
public:
      Cls(int k) : n(k) {
            mas = new int[n];
            for (int i = 0; i < n; i++) mas[i] = 1;
      }
      Cls(const Cls& C1) : n(C1.n)   // Конструктор копирования
      {
            mas = new int[n];
            for (int i = 0; i < n; i++) mas[i] = C1.mas[i];
      }
```

```
        ~Cls() { // Деструктор
    if (mas) { for (int i = 0; i < n; i++)  cout << mas[i] << " "; cout << endl; }
           else cout << "No data" << endl;
                  delete[] mas;
           }
    };

    void main()
    {
           Cls x(5);
           Cls y(x);
            // Выводит: 1 1 1 1
            // Выводит: 1 1 1 1
    }
```

Несмотря на то что копируемый объект является внешним, копируются все, в том числе защищенные, поля. Это вызвано тем, что единицей защиты информации является не объект, а тип (поэтому для однотипных объектов можно использовать операцию присваивания).

Конструктор копирования может работать с динамическими объектами, например:

```
Cls* x = new Cls(7);
Cls y = *x;
```

В первом случае вызывается конструктор и создается динамический объект, адрес которого присваивается указателю *x*. Во втором случае для инициализации полей объекта *y* используются поля динамического объекта, связанного с указателем *x* (используется операция разадресации).

Аналогичным образом можно создавать временный объект для инициализации статического объекта:

```
Cls z = Cls(7);
```

Конструктор перемещения предназначен для передачи права собственности на данные другому объекту.

Пример 2.6. Использование конструктора перемещения для передачи ссылки на массив другому объекту:

```
class Cls
{
private:
       int* mas = nullptr;
       int n;
```

```
public:
        Cls(int k) : n(k)
        {
                mas = new int[n];
                for (int i = 0; i < n; i++) mas[i] = 1;
        }
        Cls(Cls& C1) : n(C1.n) // Конструктор перемещения
        {
          mas = C1.mas;
          C1.n = 0;
          C1.mas = nullptr;
        }
        ~Cls()  // Деструктор
        {
        if (mas) { for (int i = 0; i < n; i++)  cout << mas[i] << " "; cout << endl; }
                else cout << "No data" << endl;
                delete[] mas;
        }
};
void main()
{
        Cls x(5);
        Cls y(x);
         // Выводит: 1 1 1 1
         // Выводит: No data
}
```

Для перемещения данных можно использовать стандартную функцию move.

В последних стандартах языка C++ добавлены ***делегирующие конструкторы*** *(delegating constructors)*, которые предназначены для устранения дублирования кода при описании конструкторов.

Например, имеется класс, содержащий конструкторы для инициализации одного, двух или трех полей объекта:

```
class Cls
{
  int x = 0, y = 0, z = 0;
public:
        Cls(int a) { x = a; }
        Cls(int a, int b){ x = a; y = b; }
        Cls(int a, int b, int c) { x = a; y = b; z = c; }
```

```
        void shows() {
                cout << x << y << z << endl;
        }
};
```

Видно, что часть кода несколько раз дублируется. Для исключения повторного написания кода можно использовать ранее описанные конструкторы.

Пример 2.7. Использование делегирующих конструкторов для инициализации полей класса:

```
class Cls {
private:
        int x = 0, y = 0, z = 0;
public:
        Cls(int a) { x = a; }
        Cls(int a, int b) : Cls(a) { y = b; }
        Cls(int a, int b, int c) : Cls(a, b) { z = c; }
        void shows() {
                cout << x << y << z << endl;
        }
};

void main()
{
        Cls p1(1);
        p1.shows(); // Выводит: 100
        Cls p2(1, 2);
        p2.shows(); // Выводит: 120
        Cls p3(1, 2, 3);
        p3.shows(); // Выводит: 123
}
```

В примере 2.7 при создании объекта p3 вызывается конструктор для инициализации трех полей, который вызывает конструктор для инициализации двух полей, который в свою очередь вызывает конструктор для инициализации одного поля. Таким образом инициализируются все поля объекта.

Порядок вызова конструкторов при создании объекта:

- конструкторы базовых классов в порядке их указания в описании класса;
- конструкторы вложенных объектов класса в порядке их объявления;
- собственный конструктор класса.

Деструктор – особый метод класса, применяемый для освобождения памяти, занимаемой объектом. Имя деструктора соответствует имени класса и начинается с тильды (~).

Отличия деструктора от других методов:

- деструктор не имеет параметров и возвращаемого значения;
- в описании класса может быть только один деструктор;
- при описании деструктора нельзя использовать спецификаторы const, static и volatile;
- деструктор не наследуется;
- деструктор может быть виртуальным;
- деструктор может быть объявлен как чистый виртуальный метод;
- нельзя получить указатель на деструктор.

Деструктор можно явно вызвать по его полному имени. Автоматически деструктор вызывается в следующих случаях:

- для локальных объектов – при выходе из области видимости (блока, в котором они объявлены);
- для глобальных объектов – при завершении функции main;
- для объектов, заданных через указатели, – при выполнении оператора delete;

Если деструктор не описан в классе, то он создается компилятором автоматически. Явным образом деструктор описывается в случае необходимости освобождения динамически выделенной памяти.

Деструкторы вызываются в порядке, обратном вызову конструкторов:

- собственный деструктор класса;
- деструкторы вложенных объектов в порядке, обратном их объявлению;
- деструкторы базовых классов в порядке, обратном их описанию.

Пример 2.8. Использование деструктора:

```
class Cls {
private:
        int* p;
        int  n;
public:
        Cls(int a) : n(a) {
                p = new int[10];
         cout << "Create " << n << endl;
        }
        ~Cls() {
```

```
            delete[] p;
            cout << "Destruct " << n << endl;
        }
};
void main() {
        Cls ob1(1);  // Выводит: Create 1
        {
         Cls ob2(2);  );  // Выводит: Create 2
          Cls* ob3 = new Cls(3); // Выводит: Create 3
          delete ob3; // Выводит: Destruct 3
        }                  // Выводит: Destruct 2
        Cls ob4(4); // Выводит: Create 4
        // Выводит: Destruct 4       // Выводит: Destruct 1
}
```

При запуске программы объекты создавались последовательно. Удаление объекта ob3 происходит при вызове оператора delete; объекта ob2 – при выходе из блока, в котором он был объявлен; объектов ob1 и ob4 – при завершении функции main в порядке, обратном порядку их создания.

2.3. Конструкторы вложенных классов

Если объект является объектом другого класса, необходимо обеспечить правильную работу конструктора объекта включенного класса. Для этого в определение конструктора включающего класса после символа «:» записываются имена объектов включаемого класса с указанием в круглых скобках параметров их конструкторов. Конструктор включаемого объекта вызывается перед вызовом конструктора объекта включающего класса.

Пример 2.9. Класс с вложенным конструктором:

```
class Cls01 {
        int* x;
        double* y;
public:
        Cls01(int t, double r) {
                x = new int(t);
                y = new double(r);
        }
        void shows() {
                cout << "x = " << *x << " y = " << *y << endl;
        }
        ~Cls01() {
```

```
            delete x;   cout << "Delete x" << endl;
            delete y;   cout << "Delete y" << endl;
        }
};
class Cls02 {
        Cls01 ob1, ob2;
public:
        Cls02(int a, double b, int c, double d) : ob1(a, b), ob2(c, d)    { }
        void shows() {
          ob1.shows();
          ob2.shows();
        }
};
void main() {
        Cls02 s(10, 1.2, 20, 2.5);
        s.shows();  // Выводит:      x = 10 y = 1.2
                    //                x = 20 y = 2.5
// Выводит: Delete x
// Выводит: Delete y
// Выводит: Delete x
// Выводит: Delete y
}
```

2.4. Работа с объектами классов

Объекты класса могут использоваться в качестве параметров методов.

Пример 2.10. Нахождение суммы и разности целочисленных полей двух объектов класса и занесение результата в поля двух других объектов класса:

```
class Cls {
private:
        int k;
public:
        Cls() : k(0) {  }
        Cls(int n) : k(n) { }
        void shows() {
                cout << " k = " << k << endl;
        }
        void sum(Cls v1, Cls v2) {
                k = v1.k + v2.k;
        }
        Cls raz(Cls v2) {
```

```
            Cls tmp;
            tmp.k = k - v2.k;
            return tmp;
        }
};
void main() {
        Cls c1(8), c2(7), c3, c4;
        c1.shows();          // Выводит: k = 8
        c2.shows();          // Выводит: k = 7
        c3.sum(c1, c2);
        c3.shows();          // Выводит: k = 15
        c4 = c1.raz(c2);
        c4.shows();          // Выводит: k = 1
}
```

Размещение методов в памяти отличается от размещения полей. Поля различных объектов одного класса могут иметь разные значения и, следовательно, для их размещения будут выделены отдельные ячейки памяти. Методы различных объектов одного класса идентичны, поэтому они создаются и размещаются в памяти только один раз – при создании первого объекта класса.

В примере 2.10 для каждого из четырех объектов выделяется память для хранения данных в поле *k* и один набор методов для всех объектов (рис. 2.1).

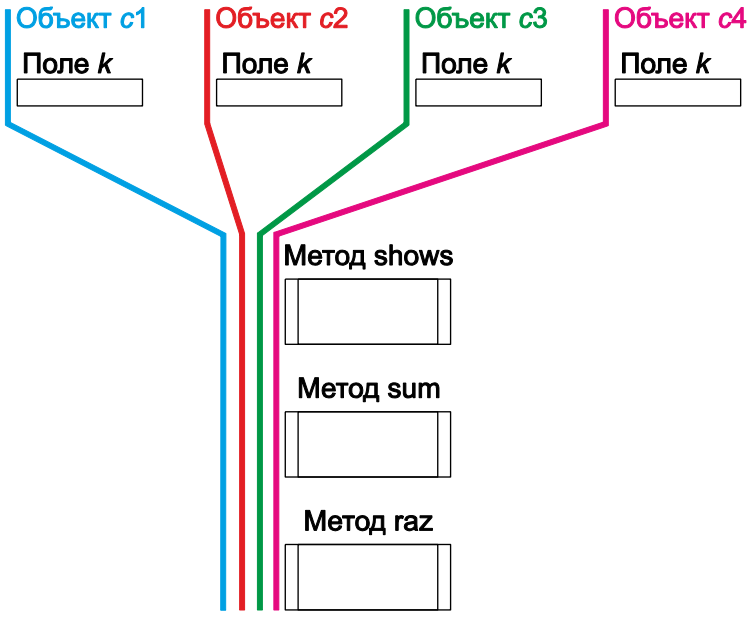

Рис. 2.1

Некоторые элементы класса должны быть представлены в единственном экземпляре для всех объектов класса (например, счетчик объектов класса). Такие

элементы называются ***статическими***. Для описания статических элементов используется ключевое слово static. Статические элементы не принадлежат конкретным объектам класса и размещаются в статической памяти.

Статическое поле аналогично статической переменной – область видимости ограничивается классом, в котором оно описано, а время жизни совпадает со временем жизни программы (продолжает существовать в случае, когда удалены все объекты класса). Статическое поле независимо от количества объектов будет размещено в единственном экземпляре в статической памяти и инициализировано один раз (по умолчанию нулями) (рис. 2.2).

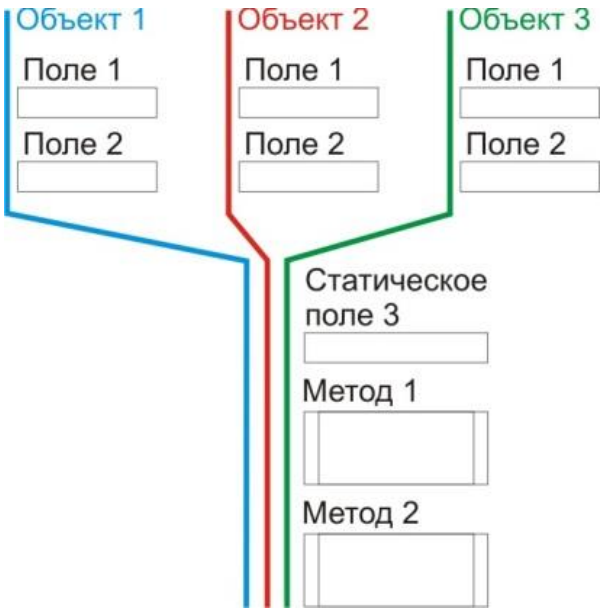

Рис. 2.2

Определение статического поля, как правило, располагается вне класса и зачастую представляет собой определение глобальной переменной.

Статические методы являются частью класса, а не объекта, поэтому обращение к статическому методу можно осуществлять как через обращение к имени объекта, так и используя имя класса. Статические методы, как правило, применяются для работы с глобальными объектами и статическими полями соответствующего класса.

Так как статические методы могут существовать без наличия объектов этого класса, то они не имеют доступа к нестатическим элементам класса. Обычные методы класса имеют доступ к статическим элементам класса.

Статический метод может создавать объекты своего и других классов.

При обращении к статическим полям и методам вне описания класса их имена дополняются слева именем класса (или объекта класса) и символами операции разрешения области видимости «::».

Пример 2.11. Программа для изменения значения статической переменной:

```
class Cls {
        static int k;                   // Статическое поле
        int n;                          // Нестатическое поле
public:
        static void adds() {            // Статический метод
         k++;
         // n++;   // Ошибка! Недопустимая ссылка на нестат. элемент
        }
    void shows() {                      // Нестатический метод
                cout << "k = " << k << endl;
        }
};
int Cls::k = 9;                 // Инициализация статического поля
void main() {
  Cls::adds();
 // Cls::shows();                // Ошибка! Недопустимый вызов
  Cls x, y;
   x.shows();                   // Выводит: k = 10
        x.adds();
        y.adds();
         x.shows();             // Выводит: k = 12
         y.shows();             // Выводит: k = 12
 }
```

Для установления методу запрета на изменение полей класса используется спецификатор const:

тип имя_метода() **const** { }

Если объявление и реализация метода разделены, то спецификатор const указывается в обоих случаях.

Константные методы могут изменять значение статических полей класса.

Если спецификатор const используется для объектов классов, то такие объекты не могут быть изменены и могут вызывать только константные методы.

Пример 2.12. Использование константных классов и методов:

```
class Cls {
        static int k;
        int n = 0;
public:
        void adds() const; // Константный метод
```

```
        void shows() {
                cout << k << " " << n;
        }
};
 int Cls::k = 0;
void Cls::adds() const {
  // n++; // Ошибка! Запрещено изменение через константный метод
    k++;
}
void main() {
        Cls b;
        b.adds();
        b.shows();  // Выводит: 1 0
        const Cls a;  // Константный класс
        a.adds();
      // a.shows(); // Ошибка!
}
```

2.5. Использование массивов объектов

При работе с массивами объектов происходит раздельный вызов конструктора и деструктора для каждого элемента массива.

Пример 2.13. Программа, работающая со статическим и динамическим массивами объектов классов:

```
class Cls {
        static int i;
        int k;
public:
        Cls() {
                k = ++i;
                cout << "Object " << k << " created" << endl;
        }
        ~Cls() {
                cout << "Object " << k << " destruction" << endl;
        }
};
int Cls::i = 0;

void main() {
        // Создание динамического массива объектов класса
        Cls* mas1 = new Cls[2]; // Выводит: Object 1 created
```

```
                                        // Выводит: Object 2 created
    // Освобождение памяти
      delete[] mas1;      // Выводит: Object 2 destruction
                          // Выводит: Object 1 destruction
    // Создание статического массива объектов класса
    Cls mas2[3];  // Выводит: Object 3 created
                  // Выводит: Object 4 created
                  // Выводит: Object 5 created
                  // Выводит: Object 5 destruction
                  // Выводит: Object 4 destruction
                  // Выводит: Object 5 destruction
}
```

Если в объявлении массива отсутствует список инициализации, то используется конструктор по умолчанию, иначе – конструктор для соответствующего списка инициализации.

Пример 2.14. Инициализация полей объектов массива объектов классов:

```
class Cls {
    int x, y;

public:
    Cls() : x(1), y(2) {}
    Cls(int a) : x(a), y(3) {}
    Cls(int a, int b) : x(a), y(b) {}
    void shows() {
     cout << x << " " << y << endl;
    }
};

void main() {
    Cls mas[4] = { Cls(4), Cls(5, 6) };
    for (int i = 0; i < 4; i++) mas[i].shows();
    // Выводит: 4 3
    // Выводит: 5 6
    // Выводит: 1 2
    // Выводит: 1 2
}
```

2.6. Дружественные функции класса

Принцип инкапсуляции предполагает запрет доступа к данным из функций и методов, не принадлежащих данному классу. Однако это не всегда удобно, поэтому определены дружественные функции.

Дружественная функция – функция, не являющаяся методом класса, но имеющая доступ ко всем элементам класса.

Прототип дружественной функции размещается в любом месте описания класса и начинается с ключевого слова friend. Тело дружественной функции может находиться в любом месте программы (ниже описания прототипа).

Особенности дружественных функций:

- при определении в классе им автоматически приписывается модификатор inline;
- не привязаны к объектам класса;
- не могут использовать указатель this;
- вызываются как обычные функции, а не как методы класса.

Пример 2.15. Использование дружественных функций:

```
class Cls
{
   int x;
   friend void Inc(Cls&);
public:
   Cls() : x(0) {}
   friend void shows(Cls& a) { cout << a.x << endl; }
};
void Inc(Cls& a) { a.x++; }
void main() {
  Cls b;
   Inc(b);
   shows(b);  // Выводит: 1
}
```

Использование дружественных функций позволяет организовать взаимодействие между классами.

Пример 2.16. Использование дружественной функции:

```
class Cl2;
class Cl1 {
        int x;
public:
        Cl1(int a) : x(a) {}
```

```
        int f(Cl2& obCl2);
};
class Cl2 {
        int y;
public:
        Cl2(int a) : y(a) {}
        friend int Cl1::f(Cl2& obCl2);
};
int Cl1::f(Cl2& obCl2) {
        return x + obCl2.y;
}
void main() {
        Cl1 m1(10);
        Cl2 m2(20);
        cout << m1.f(m2); // Выводит: 30
}
```

Метод f класса Cls1 объявлен в классе Cls2 как дружественный, поэтому получает доступ к полю y объекта класса Cls2.

Дружественные функции могут связывать элементы разных классов.

Пример 2.17. Использование дружественной функции для вычисления суммы значений полей объектов двух независимых классов:

```
class Cl2;
class Cl1
{
        int x;
public:
        Cl1(int a) : x(a) {}
        friend int sum(Cl1, Cl2);
};
class Cl2 {
        int y, z;
        friend int sum(Cl1, Cl2);
public:
        Cl2(int b, int c) : y(b), z(c) {}
};
int sum(Cl1 s, Cl2 f)
{
        return s.x + f.y + f.z;
}
```

```
void main() {
        Cl1 m1(10);
        Cl2 m2(20, 30);
        cout << sum(m1, m2) << endl; // Выводит: 60
}
```

Конструкторы и деструкторы не могут быть дружественными функциями.

Дружественными могут быть не только функции, но и классы. Дружественный класс получает доступ к закрытым и защищенным элементам другого класса.

Объявление

```
friend class имя_класса;
```

используется в случае предварительного описания класса (до его объявления).

Пример 2.18.

```
class Cls1 {
private:
   int x;
public:
   Cls1(int i) : x(i) {}
   void shows() {
      cout << x << endl;
   }
  friend class Cls2;
};
class Cls2 {
public:
   void addx(Cls1& C1, int k) { C1.x = k; }
};
void main() {
   Cls1 a(2);
    a.shows();   // Выводит: 2
    Cls2 b;
     b.addx(a, 5);
   a.shows();   // Выводит: 5
}
```

Объявление

```
friend имя_класса;
```

не создает новый класс, а используется в случае, если объявленный ранее класс необходимо определить как дружественный:

```
class Cls1 { };

class Cls2
{
        friend Cls1;
};
```

Дружественные отношения не наследуются и не являются переходящими (классы, дружественные одному классу, не являются дружественными для его дружественных классов).

2.7. Указатель this

Указатель this хранит адрес области памяти, выделенной под объект класса.

Особенности использования this-указателей:

- каждому объекту класса соответствует свой скрытый this-указатель;
- this-указатель содержит адрес области памяти, выделенной для объекта;
- тип this-указателя совпадает с типом объекта класса;
- this-указатель может использоваться только нестатическими методами или инициализаторами нестатических полей класса;
- this-указатель создается автоматически (не требуется дополнительное объявление);
- this-указатель является скрытым первым параметром всех методов класса (кроме статических);
- this-указатель является локальным и недоступен за пределами объекта;
- запрещено изменение значения this-указателя;
- обращаться к скрытому указателю можно this или *this.

Пример 2.19. Программа, суммирующая значения поля объекта класса и переданной переменной:

```
class Cls
{
        int x = 3;
        public:
        Cls(int n) : x(n) {}
        int sum(int x) {
                return this->x + x;
        }
};
```

```
void main()
{
	Cls a = 5;
	cout << a.sum(10) << endl;  // Выводит: 15
}
```

Так как имена поля класса и передаваемого параметра одинаковы, то для поля используется составное имя с указанием имени объекта или указателя на объект класса (this->x).

2.8. Указатель на элемент класса

Кроме адресации на объекты классов допустима адресация на отдельные элементы объектов классов.

Пример 2.20. Программа, использующая указатель на метод класса:

```
class Cls {
	int x;
public:
	Cls(int a) : x(a) {}
	int fsum(int a) {
		return x + a;
	}
	int fraz(int a) {
		return x - a;
	}
};
typedef int (Cls::* fun)(int);
void main() {
	Cls a = 5;
	fun f;
	// f = &a.fsum; // Ошибка: недопустимая операция
	f = &Cls::fsum;
	cout << (a.*f)(3) << endl; // Выводит: 8
	f = &Cls::fraz;
	cout << (a.*f)(3) << endl; // Выводит: 2
}
```

Доступ к адресу метода класса через объект невозможен, т. к. указатель на нестатический элемент класса определяется относительным адресом – смещением относительно базового адреса класса (а не объекта класса). При использо-

вании операции разадресации для указателя на нестатический метод класса доступ к методу осуществляется по относительному адресу, примененному к адресу указанного объекта.

При работе со статическими элементами класса идентификатор fun определяется как обычный указатель на функцию:

```
typedef int (* fun)(int);
```

Операция разадресации аналогична операции для обычной функции:

```
(a.*f)(3)
```

Контрольные вопросы

1. Какова область видимости элементов при использовании модификаторов доступа private, protected и public?
2. Какой тип могут иметь поля класса?
3. Назовите отличия конструктора от других методов.
4. В каких случаях используется конструктор по умолчанию?
5. В каких случаях требуется использование параметризированного конструктора?
6. Какой конструктор позволяет инициализировать поля создаваемого объекта значениями полей другого объекта этого же класса?
7. Чем отличается при использовании конструктора копирования поверхностное копирование от глубокого?
8. Какой конструктор позволяет передать права собственности на данные другому объекту?
9. Какие действия выполняет деструктор?
10. Каким образом размещаются поля и методы в памяти компьютера?
11. Для чего используются статические и константные методы класса?
12. Назовите особенности использования this-указателей.

3. Перегрузка операций

3.1. Перегрузка операций в C++

Перегрузка операций − это средство языка, позволяющее вводить операции для работы с пользовательскими типами данных.

Для перегрузки операций используется функция, содержащая ключевое слово operator, за которым следует знак переопределяемой операции:

```
тип_возвращаемого_значения operator знак_операции
                                        (список_параметров)
{
// Тело_функции
}
```

На этапе синтаксического анализа компилятор по значению символа «знак операции» выделит типы и имена операндов для выполнения соответствующей операции.

Перегрузить операцию можно:

- методом класса;
- дружественной функцией класса;
- обычной функцией, если имеется не менее одного аргумента, имеющего тип класса, указателя или ссылки на класс.

При перегрузке операций:

- количество аргументов, приоритеты и правила ассоциации (справа налево или слева направо) операций аналогичны операциям со стандартными типами данных;
- нельзя переопределять операции для базовых типов данных;
- нельзя использовать аргументы по умолчанию;
- функции-операции наследуются (за исключением операции присваивания);
- нельзя вводить новые операции;
- разрешено перегружать любые операции, за исключением «.», «.*», «?:», «::», «#», «##», «sizeof», «typeid».

3.2. Перегрузка унарных операций

Унарная операция – операция, которая применяется только к одному операнду.

По соглашению, принятому в C++, если компилятор встречает метод с параметром целого типа, он использует постфиксную форму операции, иначе – префиксную. Сам параметр при вызове метода не используется и является только признаком операции в постфиксной форме.

Пример 3.1. Программа с перегруженным оператором: инкремент в префиксной и постфиксной формах, изменяющий одновременно значения двух разнотипных полей:

```
class Icr {
        double s;
        int p;
public:
        Icr() : s(0), p(0)        { }
        void shows() {
                cout << s <<" "<< p << endl;
        }
        void operator++() {  // Префиксная форма оператора
          s += 3.4; p += 10;
        }
        void operator++(int) { // Постфиксная форма оператора
          s += 7.1; p += 5;
        }
};
void main() {
        Icr x;
        ++x;
          x.shows();  // Выводит:  3.4 10
        x++;
          x.shows();    // Выводит:  10.5 15
}
```

В примере 3.1 используется префиксная и постфиксная формы оператора, однако на приоритет выполнения операций такая запись влияния не оказывает.

Для сохранения стандартного приоритета выполнения операций в случае использования префиксной формы записи следует возвращать указатель на объект после того, как он был изменен, в случае использования постфиксной формы – после изменения.

Для реализации постфиксной формы записи инкремента или декремента создается временный объект, в который копируется исходное состояние изменяемого объекта, после этого изменяется состояние объекта и затем возвращается значение временного объекта.

Пример 3.2. Программа с перегруженным оператором: инкремент в префиксной и постфиксной формах:

```
class Icr {
        double s;
```

```
        int p;
public:
        Icr() : s(0), p(0)        { }
    friend ostream& operator<<(ostream&, const Icr&); // см. подразд. 3.5
        Icr& operator++() {  // Префиксная форма оператора
                s += 10; p += 5;
                return *this;
        }
        Icr operator++(int) { // Постфиксная форма оператора
                Icr temp = *this;
                ++* this;
                return temp;
        }
};
ostream& operator<<(ostream& ostr, const Icr& b)
{
 ostr << b.s << " " << b.p << " ";
return ostr;
}
void main() {
   Icr x;
        cout << x++ << " - " << x << endl; // Выводит: 0 0  - 10 5
        cout << ++x << " - " << x << endl; // Выводит: 20 10  - 20 10
}
```

3.3. Перегрузка бинарных операций

При перегрузке бинарной операции с помощью метода класса следует учитывать то, что число параметров будет на единицу меньше числа фактических аргументов операции, т. к. первым аргументом (неявно, по умолчанию) является указатель на объект перегружаемого класса (*this).

Пример 3.3. Программа с перегруженной операцией сложения для нуль-терминальных строк:

```
class CStr {
        char st[20];
public:
        CStr() {
                strcpy_s(st, "");
        }
        CStr(const char s[]) {
                strcpy_s(st, s);
        }
```

```
        void shows() {
                cout << st << endl;
        }
        CStr operator+ (const CStr st2) {
                CStr  tmp;
                strcat_s(tmp.st, st);
                strcat_s(tmp.st, st2.st);
                return tmp;
        }
        friend CStr& operator+= (CStr& st1, const CStr& st2);
        CStr& operator= (const CStr st2)
        {
                strcpy_s(st, st2.st);
                return *this;
        }
};

CStr& operator+= (CStr& st1, const CStr& st2)
{
        strcat_s(st1.st, st2.st);
        return st1;
}
void main() {
        CStr sr;
        CStr s1 = "one ";
        CStr s2 = "two ";
        CStr s3 = "three ";
        sr = s1 + s2;
        sr.shows();   // Выводит: one two
        sr = s1 + s2 + s3;
        sr.shows();  // Выводит: one two three
        s1 += "four";
        s1.shows();  // Выводит: one four
        sr = "five";
        sr.shows();  // Выводит: five
}
```

При обработке выражения sr = s1 + s2 левый операнд бинарной вызывает метод **operator+** (). Вызванный метод имеет прямой доступ к полям левого операнда. Правый операнд передается в метод в качестве аргумента. Результат передается в место вызова метода **operator+** ().

При перегрузке операции с использованием дружественной функции число параметров и аргументов функции совпадают. Например, перегрузка операции «+» выглядит следующим образом:

```
        friend CStr operator+ (const CStr, const CStr);
…
CStr operator+ (const CStr st1, const CStr st2)  {
        CStr  tmp;
        strcat_s(tmp.st, st1.st);
        strcat_s(tmp.st, st2.st);
        return tmp;
}
```

Перегрузка операции присваивания может быть осуществлена только нестатическим методом класса, а перегрузка операции присваивания, совмещенная с другой операцией (например, *=), может быть перегружена как нестатическим методом класса, так и дружественной функцией. Оператор присваивания не наследуется производными классами.

Пример 3.4. Класс, содержащий перегруженную операцию сравнения для нуль-терминальных строк:

```
class CStr {
        char st[20];
public:
        CStr(const char s[]) {
                strcpy_s(st, s);
        }
        bool operator>= (CStr st2)          {
                int k = strcmp(st, st2.st);
                if (k >= 0) return true;
                else return false;
        }
};
void main() {
        CStr s1 = "table";      CStr s2 = "book";
        if (s1 >= s2) cout << "Str 1 >= Str2";
                    else cout << "Str 1 < str2";
}
```

Функция для перегрузки операции может быть описана вне класса. Следует помнить, что функция имеет доступ только к открытым элементам класса.

Пример 3.5. Класс, использующий перегруженную операцию сложения строки и числа:

```
const int n = 20;
char s[n];
class Cls01 {
	int k;
public:
	Cls01(int m) : k(m) {}
	int rd() {
		return k;
	}
};
class Cls02 {
	char st[n];
public:
	Cls02(const char s[]) {
		strcpy_s(st, s);
	}
	char* rd() {
		return st;
	}
};
char* operator+ (Cls01 a, Cls02 b) {
	_itoa_s(a.rd(), s, 10);
	strcat_s(s, b.rd());
	return s;
}
char* operator+ (Cls02 b, Cls01 a) {
	char s2[n];
	strcpy_s(s, b.rd());
	_itoa_s(a.rd(), s2, 10);
	strcat_s(s, s2);
	return s;
}
void main() {
	Cls01 x(5);
	Cls02 y("abc");
	cout << x + y << endl; // Выводит: 5abc
	cout << y + x << endl; // Выводит: abc5
}
```

3.4. Перегрузка операций индексации и функционального вызова

Операция индексации [ ] обеспеспечивает доступ к элементу массива. Перегруженный метод имеет один параметр – номер элемента, и возвращает ссылку на этот элемент.

Операция функционального вызова ( ) предназначена для вызова функции. Перегруженный метод может содержать произвольное число параметров любого типа и возвращать результат любого типа. Классы, содержащие перегруженную операцию функционального вызова, называются ***функциональными***, а объекты – ***функциональными объектами (функторами)***.

Перегрузка операций индексации и функционального вызова осуществляется только методами класса.

Пример 3.6. Программа для подсчета суммы элементов, использующая переопределение операции ( ) и контролирующая выход данных за пределы массива переопределенной операцией [ ]:

```
const int n = 5;
class CMas {
        int a[n];
public:
        int& operator[] (int k) {
                if (k >= n || k < 0) {
                cout << endl << "Index came out the limits of the array!" << endl;
                exit(1);
                }
                return a[k];
        }

        int& operator()(int k) { // Функциональный оператор
          int sum = 0;
                for (int i = 0; i < k; i++) sum += a[i];
                return sum;
        }
};

void main() {
        CMas mas;
        for (int i = 0; i < n; i++) mas[i] = i * 10;
        cout << "Sum = " << mas(n) << endl; // Выводит: Sum = 100
        int i = 0;
        while (true) cout << mas[i++] << " "; // Выводит: 0 10 20 30 40
                                              // Index came out the limits of the array!
}
```

Так как операция может располагаться слева от операции присваивания, то перегруженный метод возвращает значение по ссылке.

3.5. Перегрузка потоковых операторов

Стандартный выходной поток cout является объектом класса ostream, а входной поток cin – объектом класса istream. Левый операнд операций извлечения из потока (<<) или помещения в поток (>>) является ссылкой на объект соответствующего класса. Правый операнд является ссылкой на объект, в который надо поместить или из которого необходимо извлечь данные. В качестве результата операторы возвращают ссылку на соответствующий поток (см. пример 3.2).

Операции << и >> являются бинарными операциями (используют два параметра). Если переопределять операции внутри класса, то у метода фактически будет не два, а три параметра: this и два объявленных параметра, что вызовет ошибку.

Операторы можно перегрузить:

- методом класса с одним явным параметром (единственный явный параметр будет выступать в качестве второго операнда, а первым операндом будет служить неявный параметр *this). В примере 3.2 такой подход вызовет ошибку, т. к. первым параметром должен быть указатель на поток. В случае если первый операнд операции не является объектом класса, операцию можно переопределить только дружественной функцией;
- функцией с двумя явными параметрами.

3.6. Перегрузка операций преобразования типов данных

В языке C++ определены операции явного и неявного преобразования типа для стандартных типов данных. Для пользовательских типов данных требуется написание собственных методов.

Пример 3.7. Перегрузка операции неявного приведения строки к значению целого типа:

```
class PStr {
        char str[20];
public:
        PStr(const char s[]) {
                strcpy_s(str, s);
        }
 operator int() const {
                return atoi(str);
        }
```

```
friend ostream& operator<<(ostream& os, const PStr& s) {
        os << s.str << endl;
        return os;
}
};
void main() {
        PStr st2 = "12";
        int sum = 10 + st2;
        cout << sum << endl;  // Выводит: 22
}
```

Контрольные вопросы

1. Что такое перегрузка операций?
2. Назовите особенности использования перегрузки операций.
3. Как описывается перегрузка унарных операций инкремента и декремента для префиксной и постфиксной форм записи?
4. Какие особенности имеет перегрузка операции индексации?
5. Как осуществляется перегрузка потоковых операторов?
6. Как осуществляется перегрузка операций преобразования типов данных?

4. Наследование

4.1. Производный класс

Наследование − механизм получения нового класса на основе класса, объявленного ранее. Объявленный ранее класс может быть дополнен или изменен для создания нового класса.

Объявленные ранее классы называются ***базовыми***, а новые – ***производными***. Производный класс наследует описание базового класса, которое можно изменить путем добавления новых и модификации существующих элементов, а также прав доступа к ним. Наследование позволяет повторно использовать ранее написанный код, что повышает производительность программиста и уменьшает вероятность ошибок.

Механизм наследования позволяет создавать иерархию классов. Наследуемые элементы не перемещаются в производный класс. Сообщения, обработку которых не могут выполнить методы производного класса, автоматически передаются в базовый класс. Поиск полей, отсутствующих в производном классе, будет вестись в базовом классе.

Методы и поля базового класса могут быть переопределены в производном классе. В этом случае соответствующие элементы базового класса становятся недоступными из производного класса. Для доступа к ним используется операция разадресации области видимости (::).

Допускается ***множественное наследование*** – использование для создания нового класса нескольких несвязанных базовых классов.

Синтаксис определения производного класса при одиночном наследовании:

```
class имя_класса : модификатор_доступа имя_базового_класса
                                {описание_элементов_класса};
```

Пример 4.1. Создание и использование производного класса:

```
class Cls {
protected:
        double s;
        int p;
public:
        Cls(double a, int b) : s(a), p(b) { }
        void get_sp() {
                cout << " s=" << s << " p=" << p << endl;
        }
```

```
        void operator++() {
                s++; p++;
        }
};

class PCls : public Cls {
public:
        PCls(double a, int b) : Cls(a, b) {}
                void operator--() {
                s--; p--;
        }
};

void main(){
        Cls x(2.4, 3);
        PCls y(2.4, 3);
                ++x; ++y;
        cout << " x:"; x.get_sp(); // Выводит для x: s = 3.4 p = 4
        cout << " y:"; y.get_sp(); // Выводит для y: s = 3.4 p = 4
        --y;
        cout << " y:"; y.get_sp(); // Выводит для y: s = 2.4 p = 3
}
```

Полученная иерархия классов представлена на рис. 4.1.

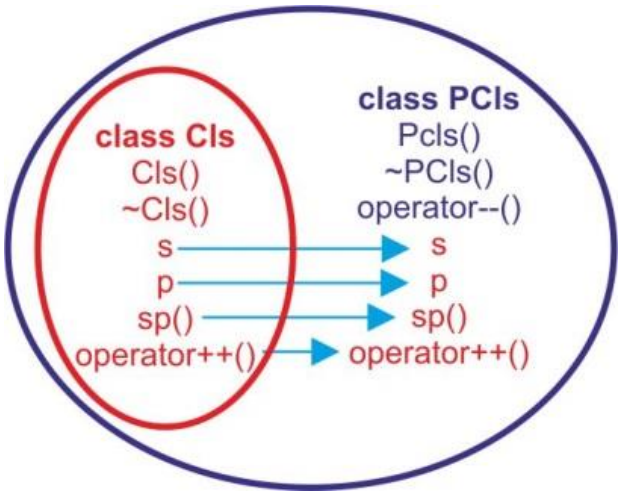

Рис. 4.1

По умолчанию в производном классе унаследованные компоненты получают модификатор доступа private. Явно изменить модификатор доступа при наследовании можно, указав новый модификатор перед именем базового класса (табл. 4.1).

Таблица 4.1

Модификатор доступа в базовом классе	Модификатор доступа, указанный при наследовании	Модификатор доступа, полученный в производном классе
public **protected** **private**	**public**	**public** **protected** недоступен
public **protected** **private**	**protected**	**protected** **protected** недоступен
public **protected** **private**	**private**	**private** **private** недоступен

Пример 4.2. Создание базового и производного классов с различными модификаторами доступа к полям:

```
class Cls
{
private:
    int a;
protected:
    int b;
public:
    int c;
};

class PCls : public Cls
{
private:
    int d;
protected:
    int f;
public:
    Cls::b;
    int g;
};

void main() {
    Cls x;
```

```
        x.a;    // Доступ запрещен - ошибка
        x.b;    // Доступ запрещен - ошибка
        x.c;    // Доступ разрешен

        PCls y;
        y.a;    // Доступ запрещен - ошибка
        y.b;    // Доступ разрешен
        y.c;    // Доступ разрешен
        y.d;    // Доступ запрещен - ошибка
        y.f;    // Доступ запрещен - ошибка
        y.g;    // Доступ разрешен
}
```

При работе с методами класса следует учитывать следующие особенности:

- скрытые методы класса недоступны при любом способе наследования;
- скрытые и защищенные методы недоступны для вызова вне класса, но могут быть использованы общедоступными методами класса;
- при скрытом наследовании защищенные и общедоступные методы доступны только внутри производного класса и недоступны извне.

Пример 4.3. Создание базового и производного классов с различными модификаторами доступа к полям и методам:

```
class BCls1
{
int x;
void f1() { cout << "BCls1 x = " << x; }
public:
        BCls1() : x(1) {}
        void f2() { cout << "BCls1 x = " << x;}

};

class PCls11 : protected BCls1
{
public:
//      void f3() { cout << "PCls11 x = " << x;}
// Синтаксическая ошибка - доступ к x запрещен
        void f4() { f2(); }
};
class PCls12 : public PCls11
{
```

```
public:
        // void f5(){ cout << "PCls12 x=" << x;}
        // Синтаксическая ошибка - доступ к x запрещен
        void f6() { f2(); }
};


class BCls2
{
protected:
        int x;
public:
        BCls2() : x(2) {}
        void f7() {cout << "BCls2 x=" << x;}
};

class PCls21 : BCls2
{
public:
        void f8() { f7(); }
        void f9() {cout << "PCls21 x=" << x;}
};

class PCls22 : public PCls21
{
        // void f10() {cout << "PCls22 x=" << x;}
        // Синтаксическая ошибка - доступ к x запрещен
};


void main()
{
        BCls1 x1;
                // x1.f1();       // Ошибка - доступ запрещен
                x1.f2();  // Выводит: BCls1 x = 1
        PCls11 x2;
                // x2.f2();  // Ошибка - доступ запрещен
                x2.f4();  // Выводит: BCls1 x = 1
        PCls12 x3;
                // x3.f2(); // Ошибка - доступ запрещен
                x3.f6();  // Выводит: BCls1 x = 1
```

```
    cout << endl;
    BCls2 y1;
        y1.f7();  // Выводит: BCls2 x = 2
    PCls21 y2;
        // y2.f7(); // Ошибка - доступ запрещен
        y2.f8(); // Выводит: BCls2 x = 2
        y2.f9();  // Выводит: PCls21 x = 2
    PCls22 y3;
        // y3.f7(); // Ошибка - доступ запрещен
}
```

Наследование, при котором общедоступные методы базового класса остаются общедоступными в производном классе, называется ***открытым наследованием*** (***наследованием интерфейса***).

Пример 4.4. Открытое наследование:

```
class BCls {
public:
  void f1() {  cout << "Base class";  };
};
class PCls : public BCls {  };
void main(void) {
PCls d;
d.f1();  // Выводит: Base class
}
```

Скрытым наследованием (***наследованием реализации***) называется наследование, при котором скрытые методы базового класса используются как подпрограммы в методах производного класса.

Пример 4.5. Скрытое наследование:

```
class BCls {
 public: void f1() { cout << "Base class"; };
};
class PCls : private BCls {
 public: void f2() { f1(); };
};
void main(void) {
PCls d;
// d.f1(); // Ошибка: нет доступа к "BCls::f1"
d.f2(); // Выводит: Base class
}
```

Для работы с объектами классов можно использовать указатели.

Пример 4.6. Создание базового и производного классов. Использование указателей для доступа к полям:

```
class BCls
{
public:
        int x;
        BCls() : x(10) {}
};


class PCls : public BCls
{
public:
        int y;
        PCls() : y(20) {}
};

void main()
{
        BCls a, *ua;
        PCls b, *ub;
        cout << a.x << endl;  // Выводит: 10
        cout << b.x << " " << b.y <<endl;  // Выводит: 10 20
        ua = &a;
        ub = &b;
        ub->x = 50;
        cout << (*ua).x << endl;  // Выводит: 10
        cout << (*ub).x << " " << (*ub).y << endl;  // Выводит: 50 20
         ua = ub;
        cout << (*ua).x << endl; // Выводит: 50
        // cout << (*ua).y << endl; // Ошибка:  ua не содержит элемент y
        ua->x = 70;
        // ub = ua; // Ошибка
        ub = (PCls*)ua;
        cout << (*ub).x << endl; // Выводит: 70
        cout << (*ub).y << endl; // Выводит: 20   // Опасная операция!
}
```

4.2. Конструкторы производных классов

Производные классы могут иметь конструкторы и деструкторы. Как правило, при конструировании объектов производного класса вызываются конструкторы базового класса.

Синтаксис вызова конструктора базового класса:

имя_производного_класса *(формальные параметры)* **:**
имя_базового_класса *(фактические параметры)*
{*тело конструктора*};

Если используется несколько базовых классов, то их конструкторы после символа «:» записываются в произвольном порядке.

При создании объекта класса сначала вызываются конструкторы базовых классов, а затем конструктор производного класса. Деструкторы вызываются в обратном порядке.

Пример 4.7. Создание базового и производного классов, использующих собственные методы для доступа к полям:

```
class Cls01  {
        int x;
public:
        Cls01(int a) : x(a) {}
        void outx()
        {
                cout << x << endl;
        }
};

class Cls02
{
        char st[10];
public:
        Cls02(const char s[])
        {
                strcpy_s(st, s);
        }

        void outst()
        {
                cout << st << endl;
        }
};
```

```
class PCls : public Cls01, public Cls02
{
public:
        PCls(int xx, const char ss[]) : Cls01(xx), Cls02(ss)
        {
        }
};

void main()
{
        PCls x(22, "aaa");
        x.outx();  // Выводит: 22
        x.outst(); // Выводит: aaa
}
```

Если в производном классе отсутствуют обращения к конструктору базового класса, то вызывается конструктор по умолчанию базового класса. Следует обратить внимание на то, что автоматически конструктор базового класса вызывается только в случае отсутствия описания в классе других конструкторов.

Пример 4.8.

```
class BCls {
public:
int x;
BCls() {  }
BCls(int a) :x (a) {}
};

class PCls : public BCls { };

void main(void) {
PCls d;  // Объект создан!
}
```

```
class BCls {
public:
int x;

BCls(int a) : x(a) {}
};


class PCls : public BCls { };

void main(void) {
PCls d;  // Ошибка!
}
```

4.3. Перегрузка методов

В случае, когда методы производного класса одноименны методам базового класса, имеет место перегрузка методов. Если метод существует в базовом и производном классе, то объект производного класса вызовет метод в производном классе. Объекты базового класса вызывают методы базового класса.

Пример 4.9. Создание базового и производного классов для расчета суммы полей:

```
const int n = 5;

class Cls01
{
int x;
protected:
 int sum1()
        {
                return x;
        }
public:
        Cls01(int a) : x(a) {}
        void outsum()
        {
                cout << "Sum = " << sum1() << endl;
        }
};

class Cls02
{
        int y, z;
protected:
int sum2()
        {
                return y + z;
        }
public:
        Cls02(int a, int b) : y(a), z(b) {}

        void outsum()
        {
                cout << "Sum = " << sum2() << endl;
        }
};

class PCls : private Cls01, private Cls02
{
        int mas[n];
public:
```

```
        PCls(int a, int b, int c, int p) : Cls01(a), Cls02(b, c)
        {
                for (int i = 0; i < n; i++) mas[i] = p * i;
        }

        void outsum()
        {
                int s = sum1() + sum2();
                for (int i = 0; i < n; i++) s += mas[i];
                cout << "Sum = " << s << endl;
        }
};

int main() {
        Cls01 g = 5;
                g.outsum();  // Выводит: Sum = 5
        Cls02 r(4, 6);
                r.outsum(); // Выводит: Sum = 10
        PCls w(2, 4, 6, 1); // Выводит: Sum = 22
                w.outsum();
        return 0;
}
```

4.4. Множественное наследование

Множественное наследование (***Multiple inheritance***) – это одновременное наследование от нескольких базовых классов. Базовые классы при множественном наследовании задаются с помощью списка. Порядок указания базовых классов может быть любым, за исключением случаев, когда необходимо установить строгий порядок вызова конструкторов и деструкторов. Вызов конструкторов выполняется в порядке указания классов в базовом списке, а деструкторов – в обратном порядке.

Пример 4.10. Ввод с клавиатуры и вывод на экран информации о работниках предприятия, среди которых могут быть как штатные сотрудники, так и студенты, проходящие практику:

```
class Fio {
        char fio[20];
public:
        void inp()
        {
                cout << "Enter the name" << endl;
```

```
		gets_s(fio);
	}
	void outp()
	{
		cout << "Name: " << fio << endl;
	}

};

class Otd
{
	int notd;  // Номер отдела
public:
	void inp() {
		cout << "Enter your department number" << endl;
		cin >> notd;
	}
	void outp() {
		cout << "Department: " << notd << endl;
	}
};

class Sotr : public Fio, public Otd  {
	char dolg[20];
public:
	void inp()  {
		Fio::inp();
		Otd::inp();
		cout << "Enter job title" << endl;
		gets_s(dolg);
	}

	void outp() {
		Fio::outp();
		Otd::outp();
		cout << "Job title: " << dolg << endl;
	}
};
class Stud : public Fio, public Otd
{
	int ngr;
```

```
public:
        void inp()      {
                Fio::inp();
                Otd::inp();
                cout << "Enter group number" << endl;
                cin >> ngr;
        }

        void outp()    {
                Fio::outp();
                Otd::outp();
                cout << "Group number " << ngr << endl;
        }
};

int main() {
        Stud stud;
        Sotr sotr;
        stud.inp();
        stud.outp();
        stud.Otd::outp();
        return 0;
}
```

При попытке использования метода outp возникает неопределенность, т. к. в обоих базовых классах присутствуют одноименные методы.

Компилятор проверяет в следующем порядке:

1. Если доступ к элементу неоднозначен (как в примере), то выводится сообщение об ошибке.

2. Если перегруженные элементы одинаковы, то доступ разрешается.

3. Если нарушаются правила доступа к элементу, выводится сообщение об ошибке.

Неопределенность в виде ромба возникает в случае, когда класс является производным от двух базовых классов, которые, в свою очередь, являются производными одного базового класса (рис. 4.2).

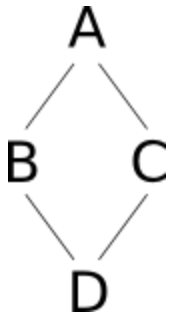

Рис. 4.2

При попытке обращения к элементам базового класса компилятор сообщает об ошибке. Это вызвано тем, что C++ для невиртуальных методов по умолчанию не создает ромбовидного наследования, и компилятор обрабатывает каждый путь наследования отдельно. В результате объекты *B* и *C* будут иметь свои подобъекты *A* и *B* (рис. 4.3).

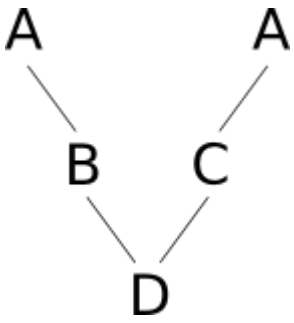

Рис. 4.3

Для использования таких объектов следует указывать путь наследования (*B*::*A* или *C*::*A*).

Пример 4.11. Обработка ситуации с множественным наследованием типа «ромб».

```
class A
{
public:
  int k;
};

class B : public A { };

class C : public A { };

class D : public B,  public C { };

…
```

```
D y;
 y.k;  // error C2385: неоднозначный уровень доступа "k"
 y.B::k;
```

Исключить неоднозначность иерархии классов можно также с помощью использования виртуальных базовых классов.

4.5. Включение классов в классы

Включение, не являясь наследованием, может выполнять аналогичные функции.

Пример 4.12. Программа для ввода-вывода информации о студенте:

```
class Fio
{
	char fio[20];
public:
	void inp() {
		cout << "Enter the name" << endl;
		gets_s(fio);
	}
	void outp() {
		cout << "Name: " << fio << endl;
	}
};

class Group
{
	int ngr;
public:
	void inp() {
		cout << "Enter group number:" << endl;
		cin >> ngr;
	}
	void outp() {
		cout << "Group number: " << ngr << endl;
	}
};

class Cstud
{
	int ngr;
	Fio fm;
```

```
        Group gr;
    public:
        void inp() {
            fm.inp();
            gr.inp();
            cout << "Enter your phone number " << endl;
            cin >> ngr;
        }

        void outp() {
            fm.outp();
            gr.outp();
            cout << "Phone number: " << ngr << endl;
        }
    };

    int main() {
        Cstud stud;
        stud.inp();
        stud.outp();
        return 0;
    }
```

Контрольные вопросы

1. Что такое наследование?
2. Какие классы называются производными?
3. Как можно изменить модификатор доступа при наследовании?
4. Что такое скрытое и открытое наследование?
5. В каком порядке вызываются конструкторы базового и производного классов?
6. Как осуществляется перегрузка методов?
7. Что такое множественное наследование?
8. Как исключить неопределенность в виде ромба?

5. Виртуальные методы

5.1. Взаимосвязь объектов производного и базовых классов

Объект производного класса хранит в памяти элементы базового класса. На этапе выполнения программы возможно выделение объекта базового класса из объекта производного класса, а также выделение объекта производного класса по его объекту базового класса. Для этого используется косвенное обращение к компонентам класса по указателям (ссылкам).

Указатели на объекты производных классов совместимы по типу с указателями на объекты базового класса, поэтому указателю на объект базового класса можно присваивать значение указателя на объект производного класса без явного преобразования типа. Присваивание указателю на объект производного класса значения указателя на объект базового класса неявным образом запрещено (приведение типа должно быть явным).

Пример 5.1. Организация доступа к методам посредством указателей:

```
class BCls {
public:
	void outp() {
		cout << "Base class" << endl;
	}
};

class PCls01 : public BCls {
public:
	void outp() {
		cout << "PClass 01" << endl;
	}
};

class PCls02 : public PCls01 {
public:
	void outp() {
		cout << "PClass 02" << endl;
	}
};

int main() {
	PCls01 p01;
	PCls02 p02;
```

```
    BCls   *ukb = &p01;
           ukb->outp();          // Выводит: Base class
    ukb = &p02;
            ukb->outp();         // Выводит: Base class
    PCls01 *ukp01 = &p02;
            ukp01->outp();       // Выводит: PClass 01
// ukp01 = ukb; // Ошибка: невозможно преобразовать BCls* в PCls01*
    ukp01 = (PCls01*) ukb;
           ukp01->outp();        // Выводит: PClass 01
    PCls02* ukp02 = (PCls02*) ukb;
           ukp02->outp();        // Выводит: PClass 02
    ukp02 = (PCls02*) ukp01 ;
           ukp02->outp();        // Выводит: PClass 02
    return 0;
}
```

При вызове оценивается **не тип объекта класса, а тип указателя**. По указателю на объект производного класса можно получить указатель на объект его базового класса. По указателю на объект базового класса (с использованием операции приведения типа) можно получить указатель на объект производного класса.

5.2. Виртуальные методы

Виртуальный метод – это метод класса, который может быть переопределен в производных классах таким образом, что конкретная реализация метода будет определяться во время выполнения программы.

Результат вызова виртуального метода (с использованием указателя или ссылки) зависит не от типа указателя, а от типа объекта класса, на который ссылается этот указатель.

Класс, содержащий хотя бы одну виртуальную функцию, называется ***полиморфным классом***, а объект такого типа – ***полиморфным объектом***.

При обращении к полиморфному объекту выбор метода осуществляется на этапе выполнения в зависимости от типа объекта, с которым ведется работа.

Для объявления виртуального метода используется ключевое слово virtual.

Если, например, в базовом классе BCls (см. пример 5.1) метод outp будет виртуальным:

```
virtual void outp()
```

то программа будет выполняться следующим образом:

```
int main() {
    PCls01 p01;
    PCls02 p02;
```

```
    BCls   *ukb = &p01;
           ukb->outp(); // Выводит: PClass 01
    ukb = &p02;
           ukb->outp(); // Выводит: PClass 02
    PCls01 *ukp01 = &p02;
           ukp01->outp(); // Выводит: PClass 02
    ukp01 = (PCls01*)ukb;
           ukp01->outp(); // Выводит: PClass 02
    PCls02* ukp02 = (PCls02*)ukb;
           ukp02->outp(); // Выводит: PClass 02
    ukp02 = (PCls02*)ukp01 ;
           ukp02->outp(); // Выводит: PClass 02
    return 0;
}
```

Вызов методов с использованием указателей или ссылок осуществляется следующим образом:

- вызов невиртуального метода происходит в соответствии с типом указателя или ссылки;
- вызов виртуального метода происходит в соответствии с типом объекта.

Пример 5.2. Совместное использование виртуальных и невиртуальных функций:

```
class BCls {
public:
   virtual void OutV() {
      cout << " Base Virtual" << endl;
 }
   void OutN() {
      cout << " Base Non-virtual" << endl;
   }
};
class PCls : public BCls {
public:
   virtual void OutV() {
      cout << " Derived Virtual" << endl;
   }
   void OutN() {
      cout << " Derived Non-virtual" << endl;
   }
};
```

```
int main() {
  PCls p;
  BCls* ubase = &p;
  PCls* uderived = &p;
    ubase->OutN();      // Выводит:  Base Non - virtual
    uderived->OutN();   // Выводит:  Derived Non - virtual
    ubase->OutV();      // Выводит:  Derived Virtual
    uderived->OutV();   // Выводит:  Derived Virtual
        return 0;
}
```

Особенности виртуальных методов:

- если объявлен виртуальный метод, то он останется виртуальным во всех производных классах (можно не использовать ключевое слово virtual).
- статический метод не может быть виртуальным;
- глобальные функции не могут быть виртуальными;
- виртуальные методы используются для обеспечения доступа к объектам базового и производных классов;
- как правило, базовый класс определяет способ работы с объектами, а производные классы предоставляют реализацию этого способа;
- виртуальные методы позволяют автоматизировать процесс классификации объектов в неоднородной иерархии классов.

Пример 5.3. Применение виртуальных методов для классификации объектов, помещаемых в контейнер:

```
class BCls {
public:
        virtual void outp () {
                cout << "Base_class ";
        }
};
class PCls01 : public BCls {
public:
        void outp() {
                cout << "Class_01 ";
        }
};

class PCls02 : public PCls01 {
public:
        void outp() {
```

```
        cout << "Class_02 ";
    }
};

void print_mas(BCls** a, int n) {
    for (int i = 0; i < n; i++) {
        a[i]-> outp();
    }
}

void main() {
    BCls* mas[3];
    mas[0] = new BCls;
    mas[1] = new PCls01;
    mas[2] = new PCls02;
    print_mas(mas, 3);
    // Выводит: Base_class Class_01 Class_02
}
```

Базовые классы и их виртуальные методы могут быть созданы и откомпилированы до определения производных классов.

Пример 5.4. Использование виртуальных методов:

```
class BCls {
public:
    int k;
    BCls(int a) : k(a) {};
    virtual void outp(BCls& x) {
        cout << "BClass: k = " << x.k << endl;
    }
    void operator ++() {
        ++(this->k);
        outp(*this);
    }
};
class PCls : public BCls {
public:
    PCls(int a) : BCls(a) {}
    void outp(BCls& x) {
        cout << "PClass: k = " << x.k << endl;
    }
};
```

```
void main() {
  PCls x(10);
      BCls* y = &x;
            y->outp(*y);   // Выводит: PClass: k = 10
            ++(*y);        // Выводит: PClass: k = 11
            x.outp(x);     // Выводит: PClass: k = 11
            ++x;           // Выводит: PClass: k = 12
}
```

При использовании виртуальных методов обязательно совпадение количества, типов и порядка следования параметров в методах базового и производных классов.

Для подавления виртуального обращения к методу можно использовать операцию разрешения области действия «::». Такое обращение позволяет избежать потерь ресурсов на динамическое определение типа в задачах, критичных к быстродействию.

Пример 5.5. Сформировать список товаров. Список должен содержать информацию о наименовании и цене товара. Для акционных товаров должен указываться размер скидки.

```
class CProd
{
protected:
      char* product;
      double cena;
public:
      CProd(const char* a, double b) : cena(b)        {
            int n = strlen(a) + 1;
            product = new char[n];
            strcpy_s(product, n, a);
      }

      ~CProd() { delete[]product; }

      virtual void outp() {
            cout << product << " : " << cena;
      }
};

class CDsc : public CProd
{
      int discount;
```

```
public:
	CDsc(const char* a, double b, int c) : CProd(a, b), discount(c) {}

	void outp() {
		CProd::outp();
		cout << " - " << discount << " %";
	}
};

struct tstk {
	tstk* a;
	void* inf;
	void add(tstk** sp, void* inform)
	{
		tstk* spt = new tstk;
		spt->a = *sp;
		spt->inf = inform;
		*sp = spt;
	}
};

class CShop {
	tstk* tovars;
public:
	CShop() : tovars(nullptr) { }

	void operator += (void* x)
	{
		tovars->add(&tovars, x);
	}

	void operator()() {
		cout << " List:" << endl;
		for (tstk* x = tovars; x != nullptr; x = x->a)
		{
			((CProd*)x->inf)->outp();
			cout << endl;
		}
	}
	~CShop() {
		tstk* spt;
```

```
            while (tovars != nullptr) {
                spt = tovars;
                tovars = tovars->a;
                delete spt;
            }
        }

};

void main() {
    CShop tov;
    tov += new CProd("Phone", 432.0);
    tov += new CDsc("Table", 126.3, 10);
    tov += new CDsc("TV", 692.3, 15);
    tov += new CProd("Chair", 63.4);
    tov();   // Выводит:
                // List:
                // Chair : 63.4
                // TV : 692.3 - 15 %
                // Table : 126.3 - 10 %
                // Phone : 432
}
```

Виртуальные методы позволяют достичь гибкости привязки функций к объектам за счет потерь памяти, уменьшения быстродействия и увеличения размера программы.

5.3. Раннее и позднее связывание

В C++ полиморфизм реализуется двумя способами:

- ***ранним (статическим) связыванием*** – на стадии компиляции посредством перегрузки функций и операторов;
- ***поздним (динамическим) связыванием*** − во время выполнения программы посредством виртуальных функций.

Основой динамического полиморфизма является возможность получения указателя на базовый класс, который можно использовать не только для обращения к объектам базового класса, но и к любым объектам производного класса.

Для каждого полиморфного типа данных компилятор создает таблицу адресов виртуальных методов (VTABLE) и встраивает в каждый объект скрытый указатель на нее (VPTR). Таблица виртуальных методов хранит адреса всех виртуальных методов текущего и базовых классов.

Количество таблиц виртуальных методов соответствует количеству полиморфных классов. При вызове виртуального метода из таблицы извлекается адрес соответствующего метода и происходит ее вызов. Объект класса содержит указатель на соответствующую таблицу.

Пример 5.6. Использование виртуальных методов:

```
class BCls {
public:
 BCls() { cout << "B : Constr" << endl; }
 virtual ~BCls() { cout << "B : Desrt" << endl; }
 virtual void M1() { cout << "B : M1" << endl; }
 virtual void M2() { cout << "B : M2" << endl; }
 };

class PCls : public BCls {
public:
  PCls() { cout << "P : Constr" << endl; }
  ~PCls() { cout << "P : Desrt" << endl; }
  virtual void M2() { cout << "P : M2" << endl; }
  virtual void M3() { cout << "P : M3" << endl; }
 };
int main()
{
 BCls *pBCls = new PCls;   // Выводит: B : Constr    P : Constr
   pBCls->M1();   // Выводит:  B : M1
   pBCls->M2();   // Выводит:  P : M2
 // pBCls->M3();   // Ошибка: M3: не является членом BCls
       delete pBCls;   // Выводит:  P : Desrt     B : Desrt
 return 0;
}
```

В программе (см. пример 5.6) имеется две таблицы виртуальных функций BCls (табл. 5.1) и PCls (табл. 5.2).

Таблица 5.1

void *vptr	
0	BCls : ~ BCls
1	BCls : M1
2	BCls : M2

Таблица 5.2

void *vptr	
0	BCls : ~ BCls
1	BCls : M1
2	PCls : M2
3	PCls : ~ PCls
4	PCls : M3

Для каждого класса создается таблица виртуальных методов. Каждому методу присваивается индекс (в порядке объявления методов), по которому определяется адрес этого метода в таблице (например, *vptr +1). Виртуальная таблица производного класса строится на основе виртуальной таблицы базового класса. Если виртуальный метод переопределен в производном классе, то изменяется указатель на этот метод в таблице производного класса. Если в производном классе появляется новый виртуальный метод, то виртуальная таблица производного класса расширяется.

В программе объявлен указатель на базовый класс BCls, которому присваивается адрес объекта производного класса PCls. Конструктор базового класса создает таблицу виртуальных функций и получает указатель на нее. После этого вызывается конструктор производного класса, который перезаписывает таблицу виртуальных методов. При вызове методов M1 и M2 компилятор через указатель VPTR получает фактический адрес метода. Для этого происходит обращение к методу по его номеру (M1 – *vptr + 1, M2 − *vptr + 2). Так как таблица виртуальных методов базового класса содержит только два метода, то при обращении к третьему методу (обращение *vptr + 4) возникает ошибка.

Размер полиморфного класса не увеличивается при увеличении количества виртуальных функций, т. к. хранит только указатель на таблицу VPTR, а не саму таблицу.

Пример 5.7. Определение размеров виртуального и невиртуального методов:

```
class ClsNV {
public:
        void f() {}
};
class ClsV1 {
public:
        virtual void f() {}
};
```

```
class ClsV2 {
public:
        virtual void f1() {}
        virtual void f2() {}
        virtual void f3() {}
        virtual void f4() {}
        virtual void f5() {}
        virtual void f6() {}
        virtual void f7() {}
        virtual void f8() {}
        virtual void f9() {}
};

void main() {
        cout << sizeof(ClsNV) << endl; // Выводит: 1
        cout << sizeof(ClsV1) << endl;  // Выводит: 8
        cout << sizeof(ClsV2) << endl;  // Выводит: 8
}
```

5.4. Виртуальные деструкторы

В отличие от конструкторов деструкторы могут быть виртуальными.

Пример 5.8. Работа деструктора в объекте производного класса (для краткости выделение и освобождение памяти в коде опущено).

```
class BCls
{
        double *x;
public:
 ~BCls() { cout << "BCls - Delete x" << endl; }
};

class PCls : public BCls
{
        char *z;
public:
 ~PCls() { cout << "PCls - Delete z" << endl; }
};

class VBCls
{
        double* x;
```

```
public:
        virtual ~VBCls() { cout << "VBCls - Delete x" << endl; }
};

class VPCls : public VBCls
{
        char* z;
public:
        ~VPCls() { cout << "VPCls - Delete z" << endl; }
};

void main()
{
        BCls *uk1 = new PCls;
        delete uk1;
        // Выводит: BCls - Delete x
        VBCls* uk2 = new VPCls;
        delete uk2;
        // Выводит: VPCls - Delete z
        // Выводит: VBCls - Delete x
}
```

При удалении объекта вызывается деструктор класса, соответствующего указателю на объект.

При удалении объекта класса PCls, доступ к которому осуществлялся через указатель на базовый BCls, вызывается только деструктор базового класса, следовательно, происходит утечка памяти.

При удалении объекта класса VPCls, доступ к которому осуществлялся через указатель на базовый VBCls, вызывается как виртуальный деструктор базового класса, так и деструктор производного класса.

Особенности:

- деструктор, генерируемый системой по умолчанию, является невиртуальным;
- если деструктор базового класса является виртуальным, то деструкторы производных классов также будут виртуальными.

5.5. Абстрактные классы и чисто виртуальные методы

Для запрета использования виртуальных методов базового класса используются ***чисто*** (***строго***) ***виртуальные методы***:

virtual *тип_результата <имя_метода>(<список_параметров)* **=** 0;

Любой класс, содержащий хотя бы один чисто виртуальный метод, называется ***абстрактным***.

Создание объектов абстрактного класса запрещено.

Абстрактный класс предназначен для использования в качестве базового для наследования невиртуальных элементов в производных классах.

Все чисто виртуальные методы должны быть переопределены в производном классе. Если в производном классе отсутствует переопределение чисто виртуального метода, то класс становится абстрактным!

Пример 5.9. Работа с абстрактными классами:

```
class BACls {
public:
   virtual void print() = 0;
};

class PCls01 : public BACls
{
public:

   virtual void print() { cout << "PCls01" << endl; }
};

class PCls02 : public BACls
{
public:
   virtual void printx() { cout << "PCls02" << endl; }
};
class PCls03 : public PCls02
{
public:

   virtual void print() { cout << "PCls03" << endl; }
};

void main() {
  // BACls b;
   // Ошибка: невозможно создать экземпляр абстрактного класса
   BACls * qb;
   PCls01 p1;
     qb = &p1;
      qb->print();   // Выводит: PCls01
```

```
    // PCls02 p2;
    // Ошибка: невозможно создать экземпляр абстрактного класса
      PCls03 p3;
      qb = &p3;
       qb->print();   // Выводит: PCls03
  }
```

Особенности абстрактных классов:

- нельзя объявлять объекты абстрактного класса;
- нельзя использовать имя абстрактного класса в качестве типа аргумента функции (метода);
- нельзя использовать имя абстрактного класса в качестве типа возвращаемого значения функции (метода);
- запрещено явное преобразование типа объекта к типу абстрактного класса;
- разрешено использование указателя или ссылки на абстрактный класс.

При создании производного класса возможны ошибки при переопределении абстрактного метода (например, ошибка в написании имени метода, отсутствие спецификатора const, неправильный тип параметра и др.). В этом случае абстрактный метод остается непереопределенным, а производный класс становится абстрактным.

Для исключения таких ошибок введен спецификатор override, который указывает, что данный метод является переопределением виртуального метода базового класса. Если переопределение неверное, то на этапе компиляции возникает ошибка.

Для запрета переопределения в производном классе виртуального метода, описанного в родительском классе, используется спецификатор final.

Пример 5.10. Написание абстрактного и производного классов для вычисления площади и периметра фигуры:

```
class BACls {
public:
   virtual double S() = 0; // Площадь
   virtual double P() = 0; // Периметр
   virtual void name() = 0;
   virtual void outp() final
   {
      name();
      cout << " square = " << S() << " perimeter = " << P() << endl;
   }
};
```

```
class Rect : public BACls {
private:
   double w;
   double h;
public:
   Rect(double a, double b) : w(a), h(b)
   { }

// double S(int x) override // Ошибка: метод со спецификатором "override"
// { }                      //  не переопределяет метод базового класса
   double S()  override
   {
      return w * h;
   }
   double P()  override
   {
      return 2 * (w + h);
   }
   void name() override
   {
      cout << "Rectangle";
   }
   // void outp() {}
   // Ошибка:  функцию, объявленную как "final", нельзя переопределить
};
class Crcl : public BACls
{
private:
   double r;
public:
   Crcl(double a) : r(a)
   {  }
   double S() override
   {
      return 3.14* r* r;
   }
   double P() override
   {
      return 2 * 3.14* r;
   }
```

```
    void name() override
    {
        cout << "Circle";
    }
};

int main()
{
    Rect rectangle(3, 4);
     rectangle.outp();   // Выводит: Rectangle square = 12 perimeter = 14
    Crcl circle(1);
     circle.outp();        // Выводит: Circle square = 3.14 perimeter = 6.28
    return 0;
}
```

Разрешена реализация чисто виртуальных методов. Метод в этом случае остается чисто виртуальным, а класс – абстрактным, однако в неабстрактном производном классе можно будет использовать описанную реализацию.

Пример 5.10. Использование реализации чисто виртуального метода:

```
class BCLs
{
public:
    virtual void prn() = 0
    { cout << "Base class"; }
};

class PCLs : public BCLs
{
public:
    void prn()
     { BCLs::prn(); }
};

void main()
{
  //  BCLs x;  // Ошибка! Невозможно создать экз. абстрактного класса
    PCLs y;
    y.prn();  // Выводит: Base class
}
```

Удобно использовать реализацию чисто виртуальных деструкторов, т. к. они всегда вызываются при уничтожении объектов классов. Использование чисто виртуального деструктора без реализации не разрешено.

Пример 5.11. Использование реализации чисто виртуального деструктора.

```
class BCLs{
public:
// virtual ~BCLs() = 0; // Ошибка! Ссылка на неразрешенный внешний символ
      virtual ~BCLs() = 0 {
          cout << " Destructor  - Base class";
      }
};
class PCLs : public BCLs
{
public:
   PCLs() {}
   ~PCLs() {}
};
void main() {
   PCLs y;
   // Выводит: Destructor  - Base class
}
```

Контрольные вопросы

1. Что такое виртуальный метод?
2. Какой класс называется полиморфным?
3. Назовите особенности виртуальных методов.
4. Как реализуется динамический полиморфизм?
5. Какие данные хранит таблица виртуальных функций?
6. Какое количество виртуальных методов создается при работе программы?
7. Какой класс называется абстрактным?
8. Для чего предназначены абстрактные классы?
9. Какие особенности имеют абстрактные классы?

6. Потоки и файлы

6.1. Потоковые классы

Поток – объект класса, предназначенный для переноса информации (последовательности байтов) от источника к приемнику. Преимуществами потоков являются простота использования (способы работы с данными не зависят от конкретного устройства) и возможность перегрузки операций помещения в поток и извлечения из потока.

Потоковые классы имеют сложную иерархическую структуру. Базовым классом является ios, который предназначен для отображения состояния процесса ввода-вывода и содержит общие для различных операций ввода-вывода константы и методы. Иерархия потоковых классов представлена на рис. 6.1.

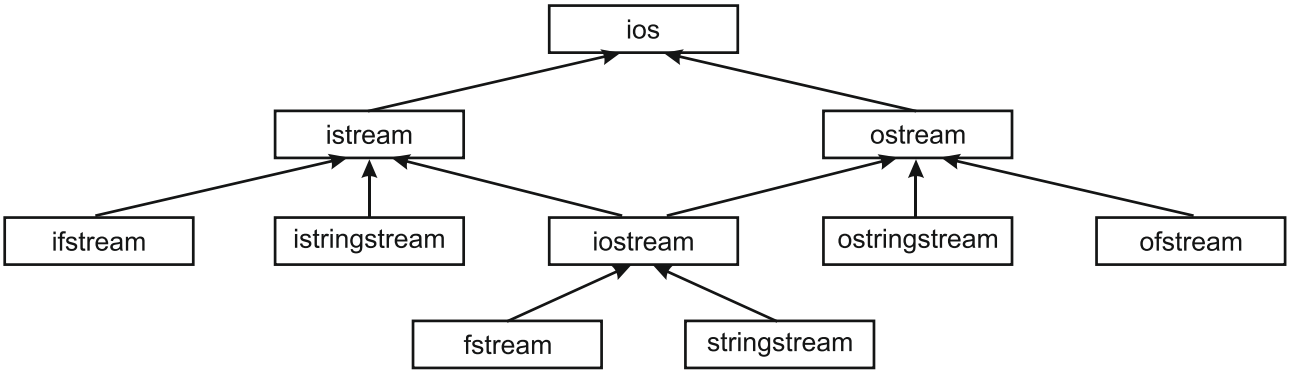

Рис. 6.1

Названия производных классов формируются с использованием следующих символов:

- направление передачи данных (*i* – ввод, *o* – вывод, *io* – ввод-вывод);
- вид источника данных (*f* – файл, *str* – строка символов).

Операция извлечения из потока является методом класса istream, а операция помещения в поток является методом класса ostream.

Базовый и большинство производных классов определены в библиотеках *iostream.lib*, которая отвечает за ввод данных с клавиатуры и вывода данных на экран, и *fstream.lib*, которая отвечает за ввод вывод файлов.

Класс ios

Все классы, предназначенные для работы с потоками, являются потомками класса ios. Класс ios содержит режимы работы с файлами, флаги форматирования и ошибок.

Флаги форматирования определяют форматы и способы ввода-вывода (табл. 6.1).

Таблица 6.1

Флаг	Значение
skipws	Пропуск пробелов при вводе
left	Выравнивание по левому краю
right	Выравнивание по правому краю
internal	Заполнение между знаком или основанием числа и самим числом
dec	Перевод в десятичную систему счисления
oct	Перевод в восьмеричную систему счисления
hex	Перевод в шестнадцатеричную систему счисления
boolalpha	Перевод логического 0 и 1 соответственно в *false* и *true*
showbase	Выводить индикатор основания системы счисления (0 для восьмеричной, *Ox* для шестнадцатеричной)
showpoint	Показывать десятичную точку при выводе
uppercase	Вывод в верхнем регистре (прописными) букв *X*, *E* и букв шестнадцатеричной системы счисления (*ABCDEF*) (по умолчанию – строчные) при выводе числовых значений
showpos	Показывать «+» перед положительными целыми числами
scientific	Экспоненциальный вывод чисел с плавающей запятой
fixed	Фиксированный вывод чисел с плавающей запятой
unitbuf	Сброс потоков после вставки
stdio	Сброс stdout, sterror после вставки

Обращение к флагу осуществляется через имя класса (например, ios::internal). Флаг устанавливается с помощью следующего метода:

```
cout.setf(ios::flag)
```

Флаг сбрасывается с помощью следующего метода:

```
cout.unsetf(ios::flag)
```

Пример 6.1. Использование флагов:

```
int main() {
        int x = 100;
        double y = 178.34;
```

```
cout.setf(ios::hex, ios::basefield); // Шестнадцатеричная
                                     // система счисления
cout.setf(ios::uppercase);           // Верхний регистр
cout << x << endl;                   // Выводит: 64
cout.setf(ios::showbase);            // Версия без маски
cout << x << endl;                   // Выводит: 0X64
cout.unsetf(ios::hex); // Отмена шестнадцатеричной
                       // системы счисления
cout << x << endl;                   // Выводит: 100
cout.setf(ios::showpos);             // Показывать "+"
cout.setf(ios::scientific, ios::floatfield); // Экспоненциальный формат
cout << y << endl;                   // Выводит: +1.783400E+02
cout.unsetf(ios::scientific); // Отмена экспоненциального формата
      cout << y << endl;             // Выводит: +178.34
      return 0;
}
```

Манипуляторы – это инструкции форматирования, передаваемые непосредственно в поток. Манипуляторы находятся в библиотеке *iomanip.lib*. Манипулятор определяет форматирование только для одного данного, находящегося за ним в потоке.

Важнейшие манипуляторы представлены в табл. 6.2.

Таблица 6.2

Манипулятор	Назначение
1	2
ws	Удаление из входного потока пробельных символов (пробел, символы табуляции, перевода строки, возврата каретки, символ перевода страницы)
dec	Перевод в десятичную систему счисления
oct	Перевод в восьмеричную систему счисления
hex	Перевод в шестнадцатеричную систему счисления
endl	Вставка разделителя строк и очистка выходного потока при выводе данных
ends	Вставка нулевого символа в выходную строку при вводе данных
flush	Очистка буфера выходного потока
lock	Закрыть дескриптор файла
unlock	Открыть дескриптор файла

Окончание табл. 6.2

1	2
setw(*ширина поля*)	Устанавливает ширину поля для вывода данных
setfill(*символ заполнения*)	Устанавливает символ заполнения (по умолчанию – пробел)
setprecision(*точность*)	Устанавливает точность (число выводимых знаков)
setiosflags(*флаги форматирования*)	Устанавливает указанные флаги форматирования
resetiosflags(*флаги форматирования*)	Сбрасывает указанные флаги форматирования

Пример 6.2. Создание пользовательского манипулятора:

```
ostream& Wrt(ostream &p) {
        p << setiosflags (ios :: fixed) << setw(8) << setprecision(3);
return p;
}

class myx {
  double x;
public:
  myx(double a) : x(a) {}
friend ostream& operator<<(ostream &p, myx ob)
{
  p << ob.x << endl;
return p;
}
};

int main()
{
        myx x = 12.21389;
cout << Wrt << x;
return 0;
}
```

Класс **istream**

Предназначен для извлечения данных из различных источников (чтение данных в оперативную память).

Большинство методов класса предназначены для работы с объектом cin. Некоторые методы для чтения данных из потока представлены в табл. 6.3.

Таблица 6.3

Метод (операция)	Назначение
>>	Операция извлечения данных из потока

get(ch)	Чтение одного символа (имеются перегруженные функции).
	Пример:
	cin.get(ch); // *или* (cin >> ws).get(ch);
	ch = cin.get(); // *или* ch = (cin >> ws).get();

get(str,n,ch)	Помещает в нуль-терминальную строку str(*n* - 1) символов строки или до символа ch (имеются перегруженные функции).
	Пример:
	cin.get(str ,7); // *Ввод в str 6 символов*
	cin.get(str, 10, '\n'); // *Ввод строки до символа '\n'*
	// *(нажатия клавиши Enter)*
	cin.get(str, 10, '0'); // *Ввод строки до символа '0'*
	// *или 9 символов*

ignore(n, ch)	Удаляет символы до ограничителя ch или до n-го символа включительно
peek(ch)	Читает один символ, оставляя его в потоке. Пример: ch = cin.peek();
putback(ch)	Вставляет обратно во входной поток последний прочитанный символ. Пример: cin.putback(ch);
gcount()	Возвращает число символов, считанных с помощью последнего вызова функций get(), getline(), read(). Пример: n = cin.gcount();

Класс ostream

Предназначен для вывода данных (помещения в поток).

Большинство функций класса предназначены для работы с объектом *cout*.

Некоторые методы представлены в табл. 6.4.

Таблица 6.4

Метод (операция)	Назначение
<<	Операция вставки в поток
put(ch)	Вставка символа ch в поток. Пример: cout.put(ch);
flush()	Очистка буфера Пример: cout. flush ();

Потоковый класс iostream

Класс создает объекты, предназначенные для чтения и записи в стандартные потоки.

При работе обычно используются четыре потоковых объекта:

- cin – используется для ввода с клавиатуры;
- cout – используется для вывода на экран;
- cerr – используется для вывода на экран сообщений об ошибках и программной диагностики. Объект немедленно выводится на экран, минуя буферизацию. Поток нельзя перенаправить;
- clog – используется для ведения журналов. Не может быть перенаправлен, но проходит буферизацию.

Как правило, определение операций ввода-вывода для пользовательских классов происходит путем переопределения операторов >> и << для подходящих библиотечных классов.

Пример 6.3. Определение операторов ввода-вывода для пользовательского класса:

```
class Cls {
public:
int s;
};

istream& operator>> (istream& input, Cls& m) {
cout << "Enter value: ";
input >> m.s;
return input;
}
```

```
ostream& operator<<(ostream& output, Cls& m) {
output << "Value: "<< m.s << endl;
return output;
}

void main() {
Cls x;
cin >> x;  // Выводит: Enter value: 454
cout << x;   // Выводит: Value: 454
}
```

Приведенный подход не требует вмешательства в определение библиотечных классов ввода-вывода и не зависит от версии таких классов.

6.2. Контроль исключительных ситуаций ввода-вывода

Информация о состоянии потока предоставляется элементами базового класса ios. Для получения информации о состоянии потока используется функция

```
int rdstate();
```

Возможные состояния потока представлены в табл. 6.5.

Таблица 6.5

Флаг	Значение	Описание
goodbit	0x00	Ошибок нет
eofbit	0x01	Достигнут конец файла
failbit	0x02	Операция не выполнена (без потери символов)
badbit	0x04	Недопустимая операция (потеря данных)

Обращение к флагам возможно посредством использования операции «::» (например, ios::goodbit).

Функции проверки флагов ошибок представлены в табл. 6.6.

Таблица 6.6

Функция	Возвращаемый результат
int eof()	Возвращает true, если достигнут конец файла (флаг eofbit)
int fail()	Возвращает true в случае неудачной операции (флаги failbit, badbit)
int bad()	Возвращает true, если имеется ошибка (флаги badbit)
int good()	Возвращает true, если ошибок не было
clear()	Снимает все флаги ошибок (при использовании без аргументов) или устанавливает указанный флаг (например, clear(ios::failbit))

После любой ошибки операции над потоком игнорируются до сброса битов ошибок.

Пример 6.4. Ввод данных с контролем исключительных ситуаций:

```
int main() {
int i;
  // cin.unsetf(ios::skipws);   // Не пропускать пробелы при вводе
while(true)
{
        cout << endl << "Enter an integer: ";
        cin >> i;
        if (cin.good()) break;  // Операция прошла без ошибок
        cout << "Data input error: ";
   switch (cin.rdstate()) {  // Получение информации о состоянии потока
        case ios::failbit : cout << "The operation failed "; break;
                                        // Операция не выполнена
        case ios::badbit : cout << "Error due to the failure"; break; // Сбой

   }
cin.clear(); // Сброс битов ошибок
cin.ignore(20,'\n');  // Очистка строки
}
cout << "The number entered: " << i << endl;
return 0;
}
```

Наличие ошибки можно проверить с использованием функций, например:

```
if (cin.fail()) cout << "Error";
```

В некоторых случаях требуется сгенерировать ошибку, для этого используется функция clear() с параметрами, например:

```
cin.clear (ios::failbit);
```

6.3. Связанные потоки

С потоком ввода может быть ассоциирован связанный поток вывода (tied stream), который позволяет выполнять логически корректные двунаправленные операции ввода-вывода. При связывании потоков их буферы синхронизируются таким образом, что содержимое буфера одного потока данных будет очищаться функцией flush() перед любой операцией в другом потоке. Каждый поток ввода может быть связан только с одним потоком вывода, а поток вывода – с несколькими потоками ввода.

Управление связыванием реализуется функциями:

- ostream* tie() – возвращает указатель на выходной поток, связанный с входным потоком;
- ostream* tie(ostream*) – связывает поток с заданным входным потоком и возвращает указатель на предыдущий связанный входной поток.

По умолчанию стандартный поток ввода связан со стандартным потоком вывода:

```
cin.tie(&cout);
```

Такая связь гарантирует, что при выполнении кода

```
cout << " Enter x: ";
cin >> x;
```

буфер, содержащий сообщение Enter x:, будет очищен перед выполнением операции ввода значения переменной x.

Вызов функции tie с нулевым параметром (cin.tie(0)) приводит к разрыву связи между потоками. В этом случае у потоков устанавливается асинхронный режим обмена данными. Асинхронный режим ускоряет работу программы, т. к. отменяется операция очистки потоковых буферов, но может приводить к ошибкам.

6.4. Потоковый ввод-вывод файлов

В библиотеке fstream описаны три класса, которые могут создавать объекты для работы с файлами:

- fstream – ввод-вывод;
- ifstream – чтение данных из файла;
- ofstream – запись данных в файл.

Объекты каждого из рассмотренных классов могут создавать свои потоки:

- имя_объекта() – создание потока без соединения с файлом. Для соединения с файлом используется метод open;
- имя_объекта.open(**const char** *имя_файла, **int** режим_открытия) – создание потока режима открытия, соединяемого с открываемым файлом с указанием режима открытия.

Конструктор или метод open (имеющий аналогичные параметры) может содержать следующие значения по умолчанию для различных потоков:

```
ofstream(const char *имя_файла, int mode = ios::out);
ifstream(const char *имя_файла, int mode = ios::in);
fstream(const char *имя_файла, int mode = ios::in | ios::out);
```

Наличие значений по умолчанию означает, что для конструирования потоков классов ifstream или ofstream достаточно указания только имени файла.

Параметр mode является перечислением и может принимать значения, представленные в табл. 6.7.

Таблица 6.7

Значение параметра mode	Возвращаемый результат
ios::in	Создается поток для чтения данных из файла
ios::out	Создается поток для записи данных в файл (режим по умолчанию)
ios::ate	При создании потока указатель перемещается в конец файла, запись ведется в текущую позицию
ios::app	При записи данные добавляются в конец файла
ios::trunc	При создании файла его содержимое уничтожается (режим по умолчанию)
ios::binary	Двоичный режим работы (по умолчанию – текстовый режим работы)

Для задания нескольких режимов можно использовать операцию ИЛИ.

Если файл открыт удачно, то связанный с ним объект возвращает значение true. Проверить успешность открытия файла можно также с помощью метода bool is_open().

Пример 6.5. Ввод строки в файл и чтение строки из файла:

```
int main() {
        ofstream ofl; // Создание потока для ввода данных в файл
        ofl.open("MyText.txt"); // Создание файла
        if (!ofl.is_open())  // Если файл не создан
        {
                cout << "Error creating file" << endl;
                return 1;
        }
          ofl << "File created";   // Ввод данных в файл
        ofl.close();    // Закрытие файла
        ifstream ifl; // Создание потока для вывода данных из файла
        ifl.open("MyText.txt"); // Открытие файла
        if (!ifl)  {
                cout << "Error reading file" << endl;
                return 2;
        }
```

```
    char str1[30], str2[30];
    ifl >> str1 >> str2;   // Чтение строки из файла
    cout << str1 <<" "<<str2 << endl;  ;   // Выводит: File created
        ifl.close(); //   Закрытие файла
    return 0;
}
```

Методы класса ifstream

Класс наследует методы класса istream. Некоторые методы класса *ifstream* представлены в табл. 6.8.

Таблица 6.8

Метод	Описание
getline(str, n, ch)	Читает символьную строку из текстового файла или данные из бинарного файла до указанного n – количества элементов или до символа-ограничителя ch
read (*st, n)	Считывает из файла в st заданное n число символов (байт) или до конца файла
eof()	Возвращает ненулевое значение (true), если указатель потока достиг конца файла
seekg(pos)	Устанавливает указатель в позицию pos от начала файла
seekg (pos, seek_dir)	Перемещает текущую позицию в файле на pos байтов от позиции, определяемой параметром seek_dir: ios::beg (от начала файла), ios::cur (от текущей позиции), ios::end (от конца файла)
tellg(pos)	Возвращает текущую позицию указателя от начала файла

Методы класса ofstream

Класс наследует методы класса ostream. Некоторые методы класса ofstream представлены в табл. 6.9.

Таблица 6.9

Метод	Описание
write(*st, n)	Записывает n символов (байт) из st в файл
seekp(pos)	Устанавливает указатель в позицию pos от начала файла
seekp(pos, seek_dir)	Перемещает текущую позицию в файле на pos байтов от позиции, определяемой параметром seek_dir: ios::beg (от начала файла), ios::cur (от текущей позиции), ios::end (от конца файла)
tellp()	Возвращает текущую позицию указателя от начала файла

Текстовый режим работы

При использовании текстового режима работы потоков следует обратить внимание на следующее:

1. Операция помещения в поток позволяет записывать в файл любые стандартные типы данных.

2. Числа записываются в виде последовательности символов (например, число 8,5 приводится к последовательности символов «8», «,», «5»). При чтении числа приводятся к своему обычному представлению (8,5). Поэтому текстовый режим менее эффективен при хранении чисел (например, для хранения числа -3,45433e-6 требуется в текстовом режиме 11 байт, а в двоичном – 8).

3. При записи в одну строку нескольких чисел их надо отделять пробелами, т. к. при извлечении числа нельзя будет разделить.

4. Файл можно не закрывать, т. к. его закроет деструктор в конце работы программы.

5. При чтении операция извлечения данных из файла читает строку до первого пробела, поэтому при записи необходимо устанавливать символ-ограничитель (обычно \n), а при чтении использовать метод getline().

Пример 6.6. Ввод в файл и вывод на экран данных различных типов:

```
int main() {
        ofstream ofl; // Создание потока для ввода данных в файл
        ofl.open("MyText.txt"); // Создание файла
        if (!ofl)  return 1; // Если файл не создан
        ofl << "Two words" << 234 << 'w '<< 55 << endl;
          ofl.close();    //  Закрытие файла

        ifstream ifl; // Создание потока для вывода данных из файла
        ifl.open("MyText.txt"); // Открытие файла
        if (!ifl) return 2; // Если файл не открыт

        char str[50];
          ifl >> str;   // Чтение строки из файла
        cout << str << endl;  // Выводит: Two
         ifl.seekg(0);  // Возврат указателя в начало файла
          ifl.getline(str, 50,'\n');
        cout << str << endl;  // Выводит: Two words234w55
        ifl.seekg(0);  // Возврат указателя в начало файла

        char ch, *uk;
                while (!ifl.eof()) {
```

```
        uk = str;
        ch = ifl.peek(); // Чтение одного символа без извлечения из потока
        if (isdigit(ch)) // Если прочитанный символ является цифрой
        {
      while (isdigit(*uk=ifl.get()) && !ifl.eof()) uk++;
              ifl.putback(*uk);
         *uk = '\0';
        cout << str << endl;  // Вывод числа
            }
        else  // Если прочитанный символ не является цифрой
            {
                int k = 1;
                str[0] = ifl.get();
            while (!isdigit(ifl.peek()) && !ifl.eof()) str[k++] = ifl.get();
                str[k]='\0';
      cout << str << endl;
            }
            }
                ifl.close(); //  Закрытие файла
        return 0;
}
```

Двоичный режим работы

Двоичный режим работы, как правило, используется для работы с данными одинаковой длины, что позволяет обращаться к ним по их номеру. Функция read в качестве первого аргумента использует значение типа *char, а write – const *char, поэтому необходимо преобразование используемых указателей в указанные выше типы. Для этого используется оператор явного преобразования типа reinterpret_cast:

```
istream& istream::read(reinterpret_cast<char *>(&st), streamsize n);
```

извлекает n символов из потока и помещает в st.

```
ostream& ostream::write(reinterpret_cast<const char *>(&st), streamsize n);
```

помещает n символов из st в поток.

Пример 6.6. Создание класса для хранения и сортировки информации в файле:

```
struct Tstud   // Информация о студенте
{
    char fio[30];  // Ф. И. О. студента
    int ngr;      // Номер группы
    int oop;      // Оценка по ООП
```

```
        Tstud input() {
                Tstud st;
                cout << endl << "Information about the students(fio, ngr, otc): ";
                cin >> st.fio >> st.ngr >> st.oop;
                return st;
        }
} stud;

int    nz = sizeof(Tstud); // Размер одной структуры

class CStud
{
        char flname[20];  // Имя файла
        fstream* fl;      // Указатель на поток
public:
        CStud(const char s[]) {
                strcpy_s(flname, s);
                fl = new fstream();  // Создание потока
        }

        ~CStud() {
                fl->close();
                delete fl;
        }
        bool Newfl();          // Создание файла
        bool Readfl(Tstud&);  // Чтение одного элемента из файла
        void Rewfl();         // Переход в начало файла
        void Addfl(Tstud);    // Запись одного элемента в файл
        void Sortfl();    // Сортировка по баллу
        const char* Getflname();  // Чтение имени текущего файла
};

bool CStud::Newfl()   // Создание нового файла для ввода-вывода
{
        fl->open(flname, ios::in | ios::out | ios::binary | ios::trunc);
                        // для чтения, записи, бинарный, новый
        if (!fl) return false;
        return true;
}
```

```
bool CStud::Readfl(Tstud & p)
                // Чтение одного элемента из текущей позиции файла
{
        fl->read(reinterpret_cast<char*>(&p), nz); // Чтение
        if (fl->good()) return true;
        fl->clear(); // Сброс битов ошибок
        return false;
}

void CStud::Rewfl()  // Перемещение указателей на начало потока
{
// Перемещение указателя на начало файла (для операций чтения)
        fl->seekg(0);
// Перемещение указателя на начало файла (для операций записи)
        fl->seekp(0);
}

void CStud::Addfl(Tstud stud)  // Запись одного элемента в файл
{
        fl->write(reinterpret_cast<char*>(&stud), nz);
        fl->flush();
}

void CStud::Sortfl() // Сортировка
{
        fl->seekp(0, ios_base::end);  // Переход в конец файла
        int nb = fl->tellp();     // Байт до конца файла
        int n = nb / nz;          // Количество студентов
        Tstud stud01, stud02;
        for (int i = 0; i < n - 1; i++)
                for (int j = i + 1; j < n; j++)
                {
                        fl->seekg(i * nz, ios::beg); // Сдвиг на позицию
                        fl->read(reinterpret_cast<char*>(&stud01), nz);
                        fl->seekg(j * nz, ios::beg); // Сдвиг на позицию
                        fl->read(reinterpret_cast<char*>(&stud02), nz);

                        if (stud01.oop > stud02.oop)
                        {
                                fl->seekp(i * nz, ios::beg); // Сдвиг на позицию
                                fl->write(reinterpret_cast<char*>(&stud02), nz);
```

```
                        fl->seekp(j * nz, ios::beg); // Сдвиг на позицию
                        fl->write(reinterpret_cast<char*>(&stud01), nz);
                    }
                }
        fl->flush();        // Выгрузка выходного потока в файл
}

const char* CStud::Getflname()    // Возврат имени файла
{
        return this->flname;
}

ostream& operator << (ostream & output, CStud & s)
{
        Tstud p;
        output << "File " << s.Getflname() << endl;
        s.Rewfl();
        while (s.Readfl(p))
                output << p.fio << ' ' << p.ngr << ' ' << p.oop << endl;
        return output;
}

void main()
{
        CStud myfile("stud.dat");
        if (!myfile.Newfl())
        {
                cout << "Error creating file";
                return 1;
        }
        int n;
        cout << endl << "Enter the number of students: ";
        cin >> n;
        for (int i = 0; i < n; i++)
                myfile.Addfl(stud.input());
        cout << myfile << endl;      // Просмотр файла
        myfile.Sortfl();  // Сортировка
        cout << myfile << endl;      // Просмотр файла
}
```

Разрешено использовать методы, записывающие объекты класса. Так как каждый объект находится в своей области памяти, то при записи и чтении объектов следует использовать указатель this.

Для хранения множества объектов обычно используются статические функции, которые применяются ко всему классу в целом. Для определения типа объекта во время выполнения программы используется оператор typeid (библиотека *typeinfo.lib*):

typeid *(объект)*

Перегруженные операторы == и != обеспечивают сравнение типов. Когда функция typeid применяется к указателю на базовый класс класса, она автоматически возвращает тип объекта, на который указывает указатель, в том числе любой производный класс.

Пример 6.7. Написание класса, хранящего в файле свои объекты и объекты порожденных классов. Адреса объектов должны храниться в массиве указателей, который является статической переменной.

```
#include <iostream>
#include <fstream>      // Библиотека потоковых файловых классов
#include <typeinfo>
using namespace std;

enum Clstype {Class01, Class02};
class BCls              // Класс BCls
{
private:
        double x;               // Число
        static int n;           // Количество введенных объектов
        static BCls* umas[]; // Массив указателей на объекты
public:
        BCls() { n = 0; }

        virtual void input()
        {
                cout << "Enter x: ";
                cin >> x;
        }
        virtual void output()
        {
                cout << endl << "x = " << x;
        }
```

```
        static void adds();          // Добавить работника
        static void shows();         // Вывести данные обо всех
        static void readfl();        // Чтение из файла
        static void addfl();         // Запись в файл
        virtual Clstype gettype();   // Получение типа файла
};

// Статические переменные
int BCls::n=0;
BCls* BCls::umas[20];

class Cls01 : public BCls
{
private:
        double y;     // Число
public:
virtual void input()
        {
                BCls::input();
                cout << "Enter y: ";
                cin >> y;
        }
Virtual void output()
        {
                BCls::output();
                cout << endl << "y = " << y;
        }
};

class Cls02 : public Cls01
{
private:
        double z;     // Число
public:
        void input()
        {
                Cls01::input();
                cout << "Enter z: "; cin >> z;
        }
```

```
        void output()
        {
                Cls01::output();
                cout << endl << "z = " << z;
        }
};

void BCls::adds()
{
        char ch;
        cout << "\n 'y' to enter x, y: "
                  "\n 'z' to enter x, y, z: "
                  "\n   SELECT: ";
                        cin >> ch;
        switch (ch)
        {       // Создание объекта указанного типа
        case 'y': umas[n] = new Cls01;  break;
        case 'z': umas[n] = new Cls02; break;
                default: cout << endl << "Error"; return;
        }
        umas[n++]->input();  // Ввод данных
}

Clstype BCls::gettype() // Возврат типа объекта
{
    if (typeid(*this) == typeid(Cls01)) return Class01;
else if (typeid(*this) == typeid(Cls02)) return Class02;
else {
         cerr << "Error" << endl; exit(1);
        }
}

void BCls::shows() {
        for (int i = 0; i < n; i++)
        {
                switch (umas[i]->gettype()) // Вывод типа
                {
                case Class01:  cout << "Class01:";  break;
                case Class02: cout << "Class02:"; break;
                 default: cout << "Error";
                }
```

```
            umas[i]->output();          // Вывод данных
            cout << endl;
        }
}

void BCls::addfl() // Запись в файл всех объектов
{
        int size;
        cout << "Writing to file" << endl;
        ofstream ouf;
        Clstype ctype;   // Тип объекта

        ouf.open("myfile.dat", ios::binary | ios::trunc);
        if (!ouf)
        {
                cout << "Error creating file"; return;
        }
        for (int j = 0; j<n; j++)
        {
        ctype = umas[j]->gettype(); // Получение типа объекта
        ouf.write(reinterpret_cast<char*>(&ctype), sizeof(ctype));
                                                        // Запись типа
        switch (ctype)        // Определение размера объекта
        {
        case Class01:  size = sizeof(Cls01); break;
        case Class02: size = sizeof(Cls02); break;
                        }
        ouf.write(reinterpret_cast<char*>(umas[j]), size);
                                                // Запись объекта в файл
        if (!ouf)
        {
                cout << "Error writing file" << endl; return;
        }
        }
}

void BCls::readfl() // Чтение данных из файла
{
        int size;
        Clstype ctype;    // Тип объекта
        ifstream inf;
```

```
        inf.open("myfile.dat", ios::binary);
        if (!inf) {
                cout << "Error open file" << endl; return;
        }
        n = 0;
        while (true)
        {
                inf.read(reinterpret_cast<char*>(&ctype), sizeof(ctype));
                                                // Определение типа объекта
                if (inf.eof())  break;  // Выход, если достигнут конец файла
                if (!inf)               // Если ошибка чтения типа
                {
                        cout << "Error reading file" << endl; return;
                }
                switch (ctype)  // Выделение памяти для хранения объектов
                {
                case Class01:
                        umas[n] = new Cls01;
                        size = sizeof(Cls01);
                        break;
                case Class02:
                        umas[n] = new Cls02;
                        size = sizeof(Cls02);
                        break;
                                default: cout << "Error type file"; return;
                }
        inf.read(reinterpret_cast<char*>(umas[n]), size);
                                                // Чтение объекта из файла
if (!inf) {
                        cout << "Error reading file" << endl; return;
                }
                n++;   // Увеличение количества объектов
        }
        cout << "Reading" << endl;
}
int main() {
        char ch;
        while (true)
        {
                cout << "\n'a' - Add"
                        "\n'p' - Show"
```

```
                "\n'w' - Write to file"
                "\n'r' - Read from file"
                "\n'q' -- Exit"
                "\n      SELECT: ";
        cin >> ch;      system("cls");
        switch (ch) {
        case 'a': BCls::adds(); break;        // Добавление объекта
        case 'p': BCls::shows(); break;       // Вывод информации
        case 'w': BCls::addfl(); break;       // Запись в файл
        case 'r': BCls::readfl(); break;      // Чтение файла
        case 'q': exit(0);                    // Выход
        default: cout << "Error" << endl;
        }
    }
    return 0;
}
```

6.5. Использование строковых потоков для работы с памятью

Область памяти может рассматриваться как поток. В этом случае в память можно записывать данные аналогично записи в файл. Поток ассоциируется со строкой символов некоторой длины.

В библиотеке *sstream.lib* описаны три специализированных строковых класса:

- stringstream – ввод-вывод;
- istringstream – ввод (чтение из строки);
- ostringstream – вывод (запись в строку).

Связывание потока с символьной строкой осуществляют конструкторы соответствующих классов. В отличие от буферизации данных файловой системой ввода-вывода, в этом случае существует возможность вмешательства в содержимое буфера, т. к. строка выводится после ее форматирования в памяти.

Классы наследуют методы класса istream и ostream, а также содержат свои методы, такие как:

- str() – возвращает копию содержимого потока в виде строки;
- str(string) – помещает в поток строковый аргумент.

Пример 6.7. Преобразование строковых данных в числовые:

```
#include <iostream>
#include <sstream>
using namespace std;
```

```
int main() {
        int k;
        double x, res;
        char s[10];
        istringstream st_in;
        st_in.str("50 12.44 Result = ");
        st_in >> k >> x >> s;    // Извлечение из потока
        res = x + k;
        ostringstream st_out;
        st_out << s << res << endl;
        cout << st_out.str();  // Выводит: Result = 62.44
        return 0;
}
```

6.6. Стандартный класс string

Класс string, расположенный в библиотеке *string.lib*, существенным образом упрощает работу со строками, кроме того, он более эффективен и защищен от неправильного использования.

Преимущества использования класса string:

- класс самостоятельно управляет памятью;
- использует перегруженные операции присваивания, сложения и логические операции;
- ведется контроль за выходом за границы строки.

Недостатки:

- отсутствие в классе встроенных средств для разбора строк (strtok);
- более медленная работа по сравнению с нультерминальными строками.

В памяти, кроме самой строки, хранится текущий размер строки, размер выделенного буфера и объединение, состоящее из указателя и массива символов. Буфер предназначен для обеспечения необходимого запаса выделенной памяти. Объединение позволяет хранить небольшие строки в собственном массиве либо адрес выделенного участка памяти, предназначенного для хранения строки большого размера (рис. 6.2).

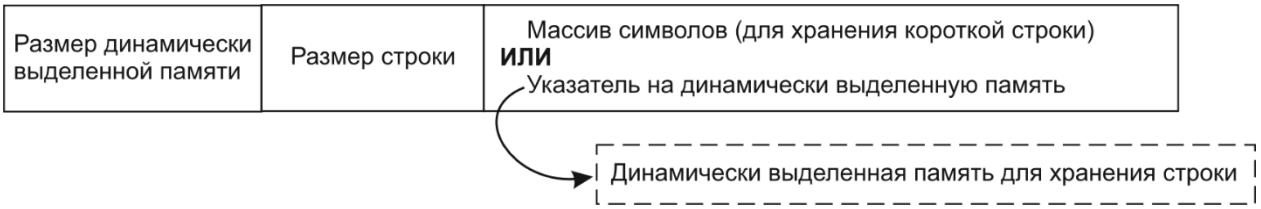

Рис. 6.2

Конструкторы:

```
string();
string(const char* строка);
string(const string& строка);
```

Пример 6.8. Использование стандартного класса для работы со строками:

```
int main()
{
        string st1;
        st1 = "Str1";
        string st2 = "Str2";
        string st3("Str3");
        cout << st1 << endl;  // Выводит: Str1
        cout << st2 << endl;  // Выводит: Str2
        cout << st3 << endl;  // Выводит: Str3
        st1 += " & " + st2 + " & " + st3;
        cout << st1 << endl;  // Выводит: Str1 & Str2 & Str3
        return 0;
}
```

Некоторые методы класса string для работы со строками представлены в табл. 6.10.

Таблица 6.10

Метод	Описание
1	2
const char* c_str()	Возвращает указатель на нуль-терминальную строку, аналогичную встроенной строке
const char* data()	Возвращает указатель на первый символ строки

size_t **copy**(**char** **подстрока*, size_t *количество_символов*, size_t *позиция* = 0)	Копирует из строки в нуль-терминальную строку указанное количество символов, начиная с указанной позиции. Возвращает количество скопированных символов.
	Пример:
	string s = "ABCDEFGH";
	char s1[20];
	size_t s1len = s.copy(s1, 2, 3);
	s1[s1len] = '\0'; // *s1 = DE*

Продолжение табл. 6.10

1	2

size_t **find**(**const** string &*подстрока*,
size_t *позиция* = 0)
	Возвращает номер позиции (начиная с указанной), в которой находится подстрока, передаваемая в качестве аргумента. Если указанная подстрока не найдена, то возвращается значение -1.
	Пример:
	string st = "ABCDCFCD";
	int k = st.find("CD"); // *k = 2*
	k = st.find("w"); // *k = -1*
	k = st.find("C",3); // *k = 4*
	k = st.find("C", 5); // *k = 6*
	Имеется ряд близких методов

string **substr**(size_t *позиция* = 0, size_t *длина = позиция_конца_строки*)	Возвращает подстроку, полученную из строки, начиная с указанной позиции и указанной длины (или до конца строки). Пример: string s = "ABCDEFGH"; string s1 = s.substr(2, 3); // *s1 = CDE*
string& **append**(**const** string& *строка2*)	Добавляет строку2 в конец строки. Имеются перегруженные методы
void push_back(**char** *символ*);	Добавляет символ в конец строки
string& **assign** (**const** string& *строка2*)	Заносит в строку значение строки2. Имеются перегруженные методы
string& **insert** (size_t *позиция*, **const** string &*строка2*)	Помещает в строку после указанной позиции строку2. Пример: string s = "ABCDCFCD"; s.insert(3,"wwww"); // *s="ABCwwwwDCFCD"* Имеются перегруженные методы
string& **erase**(size_t *позиция* = 0, size_t *количество_символов = размер_строки*)	Стирает из строки указанное число символов (или все символы до конца строки) начиная с указанной позиции. Пример: string s = "ABCDCFCD"; s.erase(4,2); // *s = ABCDCD*

Окончание табл. 6.10

1	2
void pop_back()	Удаляет последний символ строки. Пример: string s = "ABCD"; s.pop_back(); // *s = "ABC"*
size_t **size**() size_t **length** () **const**	Возвращает длину строки
size_t **max_size**()	Возвращает максимально возможную длину строки. Пример: size_t k = s.max_size(); // *k = 4294967294*
void clear()	Удаляет все содержимое строки

int compare(
size_type *начальная_позиция*,
size_type *количество_символов*, **const** string*& строка*)
	Сравнивает указанную строку с заданным количеством символов исходной строки начиная с указанной позиции. Функция возвращает -1, если заданный участок меньше указанной строки, 0 − если равны, 1 − если больше.
	Пример:
	string s1 = "abcdf";
	string s2 = "bcd";
	int res = s1.compare(s2); // *res = -1*
	res = s1.compare(0, 3, s2); // *res = 1*
	res = s1.compare(1, 3, s2); // *res = 0*
	res = s1.compare(2, 3, s2); // *res = 1*

Контрольные вопросы

1. Что такое поток?
2. Что определяют флаги форматирования потоков?
3. Для чего используются манипуляторы потоков?
4. Как контролируются исключительные ситуации ввода-вывода потоков?
5. Как связываются стандартные потоки ввода-вывода?
6. Какие потоки используются для организации работы с файлами?
7. Назовите основные методы работы с файлами.
8. Как размещается строка класса string в памяти?
9. Назовите основные методы работы со строками.

7. Обработка исключительных ситуаций

Исключительной ситуацией или ***исключением*** (*exception*) называется прерывание нормального хода выполнения программы в случае возникновения непредвиденного или аварийного события (например, деление на нуль). Кроме этого, исключительная ситуация может генерироваться функциями библиотеки API или специальным оператором.

Обработчик исключения позволяет распознать ошибку и запустить код, обрабатывающий ошибку, в противном случае будет остановлено выполнение программы.

В языке C++ имеется простой механизм обработки исключительных ситуаций, позволяющий корректно продолжить или завершить выполнение программы.

Существуют два уровня обработки исключений:

- обработка исключений C++, которая выполняется средствами языка C++;
- структурная обработка исключений (*structured exception handling, SEH*), которая выполняется операционной системой.

7.1. Обработка исключений средствами языка C++

В языке C++ практически любое действие можно заранее определить как особую ситуацию (исключение) и предусмотреть операции, которые нужно выполнить при ее возникновении.

Для реализации механизма обработки исключений в язык C++ введены следующие три ключевых слова:

- try – начало блока исключений;
- throw – генерация исключения;
- catch – начало блока, «ловящего» исключение.

Служебное слово try позволяет выделить контролируемый блок:

```
try
{
// Операторы
}
```

В качестве операторов могут использоваться как обычные операторы языка C++, так и специальные операторы генерации исключений:

throw *выражение_генерации_исключения*;

Выражение формирует исключение. Исключение является статическим объектом, тип которого определяется типом значения выражения_генерации_исключения. Местоположение оператора throw называют точкой выброса

исключения. Среди операторов контролируемого блока может быть любое количество операторов throw. Контролируемые блоки могут быть вложенными.

После формирования исключения исполняемый оператор throw автоматически передает управление непосредственно за пределы контролируемого блока, где обязательно должны находиться один или несколько обработчиков исключений:

```
catch (тип_исключения имя)
{
// Операторы блока обработки исключений
}
```

В случае использования нескольких обработчиков исключений они должны отличаться друг от друга типами исключений.

Тип исключения имеет три формы:

- catch (*тип_параметра имя_параметра*);
- catch (*тип_параметра*);
- catch (...).

Первая форма подобна спецификации формального параметра при определении функции. Имя этого параметра используется в операторах обработки исключения. С его помощью к ним передается информация из обрабатываемого исключения.

Вторая форма не предполагает использования значения исключения. Для обработчика важен только его тип и факт его получения.

В третьем случае (многоточие) обработчик реагирует на любое исключение независимо от его типа. Так как сравнение исключений со спецификациями обработчиков выполняется последовательно, то обработчик с многоточием в качестве спецификации следует помещать только в конце списка обработчиков. В противном случае все возникающие исключения «перехватит» обработчик с многоточием в качестве спецификации.

Блоки catch обрабатываются последовательно. После нахождения первого соответствующего catch последующие обработчики не проверяются. Нельзя, например, помещать обработчик для базового класса перед обработчиком для производного класса.

Если тип объекта исключения не соответствует указанному типу исключения, то проверяются типы, в которые можно преобразовать объект при помощи стандартных правил преобразования.

Если в контролируемом блоке исключение не порождено, то обработчики исключений не выполняются.

Пример 7.1. Создание массива, содержащего не более пяти целых чисел из диапазона от 10 до 100:

```
void main()  {
   int a[10], i, n;
try
{
  cout << "Vvedite n " << endl;      cin >> n;
  if (n > 5) throw "n is more than 5";
for (i=0; i<n; i++)
{
   cout << "Vvedite a["<< i <<"]:" << endl;
     cin >> a[i];
    if (a[i]<10 || a[i]>100) throw "Value is not within the range";
}
}
catch (const char s[20])
{
        cerr << "Error: " << s <<endl;
}
}
```

При порождении исключения выполняются следующие действия:

- создается копия объекта, представляющего выражение оператора trow;
- вызываются деструкторы объектов, порожденных от начала контролируемого блока до точки выброса исключения;
- осуществляется поиск подходящего блока обработки исключения, выбираемого по типу выражения в операторе throw, и передача управления выбранному блоку;
- после выполнения операторов блока обработки исключения управление передается первому оператору, расположенному за последним блоком catch активного контролируемого блока.

Если поиск подходящего блока обработки исключения в контролируемом блоке окончился неудачей, то поиск подходящего блока осуществляется в блоках верхнего уровня вложенности. Для перехода на предшествующий уровень вложенности используется оператор throw без параметров.

С одним блоком try может быть ассоциировано несколько блоков catch. Для того чтобы инструкция catch отвечала только за свое исключение, она должна иметь тип, отличный от типов других инструкций. Инструкции catch проверяются в порядке их размещения в блоке обработки. Выполняется только

инструкция catch, соответствующая типу исключения. Остальные блоки catch игнорируются.

Пример 7.2. Вычисление частного двух положительных чисел:

```
void main()  {
 double a, b;
 double s;
cout << "Vvedite a, b" << endl;
cin >> a >> b;
try
{
 if (a < 0) throw "a < 0";
 if (b < 0) throw "b < 0";
 if (b == 0) throw 0;
   s = a/b;
 cout << "Res =" << s <<endl;
}  // Конец контролируемого блока
catch (int) { cerr << "Error! Division by zero" << endl; }
catch (const char s[20]) { cerr << "Error! "<< s << endl; }
catch (...) { cerr << "Error! " << endl; }
}
```

Если генерируется исключение, для которого отсутствует подходящая инструкция catch, происходит аварийное завершение программы. В этом случае вызывается функция terminate(), которая по умолчанию вызывает функцию abort() для завершения выполнения программы. Функция terminate() может использовать функцию, назначенную в программе, для этого используется следующая конструкция:

```
typedef void ( *terminate_function )( );
…
terminate_function set_terminate( terminate_function termFunction );
```

Функция set_terminate может вызываться несколько раз, устанавливая новый обработчик и возвращая в качестве результата указатель на прежний обработчик. Функция set_terminate работает только вне отладчика. Для компиляции требуется установить /EHsc.

Пример 7.3. Программа, использующая set_terminate:

```
void MyErr()
{
   cerr << "Error processing" << endl;
   exit(1);
}
```

```
void main()
{
set_terminate(MyErr);
 try {
        throw 0;
    }
 catch(char)
        {
         cout<<"Error type - char";
    }
}
```

7.2. Структурное управление исключениями

При структурной обработке исключений (*structured exception handling, SEH*) компилятор генерирует специальный код на входах и выходах блоков исключений (*exception blorks*), создает таблицы вспомогательных структур данных и предоставляет системе возможность прохода по блокам исключений.

Структурное управление исключениями позволяет наряду с обработкой потока явно порождаемых программой исключений обрабатывать и исключения, порождаемые операционной системой в аварийных ситуациях.

Формат обработчика исключений:

```
__try
{
// Защищенный участок программы
}
__except (фильтр исключений)
{
// Обработчик исключений
}
__finally
{
// Операторы, выполняющиеся в любом случае
}
```

Блок __except активизируется только в момент порождения исключения, а блок __finally активизируется при любом выходе из контролируемого блока, что гарантирует обязательное выполнение указанных в блоке операторов (вне зависимости от того, была ли порождена исключительная ситуация).

Блок обработки исключения принимает решение о продолжении процесса на основании анализа вычисляемого после порождения исключения константного выражения. В качестве фильтра исключений могут использоваться три константных выражения:

EXCEPTION_EXECUTE_HANDLER (1) – исключение распознано. Управление передается обработчику исключений (__except ());

EXCEPTION_CONTINUE_EXECUTION (-1) – ошибка исправлена. Управление передается инструкции, которая вызвала исключение, поскольку предполагается, что в этот раз она не вызовет исключение;

EXCEPTION_CONTINUE_SEARCH (0) – исключение не распознано. Необходимо продолжение поиска подходящего обработчика.

Пример 7.4. Выведение на экран строки, используя указатель.

```
void main()  {
char *a=NULL, s[10]="SEH";

__try
 {
   cout << a <<endl; // Ошибка!
 }
__except(EXCEPTION_EXECUTE_HANDLER)
{
    a = s;
   cout << a <<endl;
}
}
```

Детальную информацию о причине возникновения исключения можно получить с помощью функции GetExceptionCode(), которую можно вызвать только из фильтра или из обработчика исключений.

Некоторые из возвращаемых функцией значений:

- EXCEPTION_ARRAY_BOUNDS_EXCEEDED – попытка обращения к элементу массива, индекс которого выходит за границы массива, если оборудование поддерживает такой тип контроля;
- EXCEPTION_**INT**_DIVIDE_BY_ZERO – попытка деления на нуль в операции с целыми числами;
- EXCEPTION_**INT**_OVERFLOW – переполнение при операции с целыми числами;
- EXCEPTION_FLT_DIVIDE_BY_ZERO – попытка деления на нуль в операции с вещественными числами;

- EXCEPTION_FLT_OVERFLOW – переполнение при операции с действительными числами;
- EXCEPTION_FLT_INVALID_OPERATION – ошибка при операции с плавающей точкой, для которой не предусмотрены другие коды исключения.

Для явного порождения исключения используется следующая функция:

```
VOID RaiseException (
DWORD dwExceptionCode,
DWORD dwExceptionFlags,
DWORD nNumberOfArguments,
const ULONG_PTR *lpArguments );
```

Здесь dwExceptionCode – код исключения (можно получить с помощью функции GetExceptionCode);

dwExceptionFlags – флаг возобновления исключения; указывает, можно ли возобновить выполнение программы с команды, следующей за RaiseException. Он может принимать значение 0 или EXCEPTION_NONCONTINUABLE (непродолжаемое исключение);

nNumberOfArguments – количество аргументов детализации описания исключения в массиве аргументов LpArguments.

Пример 7.5. Программа нахождения частного двух положительных чисел:

```
void main()
{
        int a, b;
        double s;
        cout << "Vvedite a, b" << endl;
        cin >> a >> b;
        __try
        {
                s = a / b;
                if (a < 0 || b < 0)
    RaiseException(EXCEPTION_PRIV_INSTRUCTION, 0, 0, NULL);
                cout << "Res = " << s << endl;
        }
        __except (EXCEPTION_EXECUTE_HANDLER)
        {
                int er = GetExceptionCode();
                if (er == EXCEPTION_INT_DIVIDE_BY_ZERO)
                        cerr << "Division by zero" << endl;
                else
                        if (er == EXCEPTION_PRIV_INSTRUCTION)
```

```
                    cerr << "Error: a < 0 || b < 0" << endl;
                else cerr << "Some other exception" << endl;
        }
    }
```

Выход из блока __try может быть осуществлен следующими способами:

- завершение блока без ошибок;
- выход из блока при помощи управляющей инструкции __leave;
- выход из блока при помощи операторов **return**, break, continue или goto (не рекомендуется!);
- передача управления обработчику исключения.

Любой выход из контролируемого блока приводит к выполнению операторов блока __finally. В блоке __finally можно проверить состояние контролируемого блока с помощью функции AbnormalTermination, которая возвращает true, если контрольный блок не завершен нормально.

Пример 7.6. Использование вложенных блоков:

```
  void main()
  {
   int x = 0;

  __try {
    cerr << "Begin block 1" << endl;
        __try {
          cerr << "Begin block 2" << endl;
                x /= 0;   // Ошибка
               }
        __finally {
         cerr << "End block 2" << endl;
         if (AbnormalTermination())  cerr << "Error in Block 2" << endl;
         else cerr << "Block 2 is completed without error" << endl;
                  }
        cerr << "End block 1" << endl;
        }
   __except(puts("Filter"),EXCEPTION_EXECUTE_HANDLER)
        {
        cerr << "Handling exceptions" << endl;
        }
   cout << "End of the program" << endl;
  }
```

Выводит:

```
Begin block 1
Begin block 2
Filter
End block 2
Error in Block 2
Handling exceptions
End of the program
```

Если бы в коде отсутствовала ошибка, то вывод был бы таким:

```
Begin block 1
Begin block 2
End block 2
Block 2 is completed without error
End block 1
End of the program
```

7.3. Обработка исключений в объектах классов

По сравнению с процедурным программированием в объектах классов обработка ошибок более сложная, т. к. они могут возникать и при отсутствии явных вызовов функции (например, при вызове конструктора). Также могут возникать ошибки и при использовании библиотек классов, реализация которых закрыта для программиста.

Пример 7.7. Класс с обработчиком ошибок:

```
class Cls {
        int x;
public:
        Cls(int a) : x(a) {}
        class MyErr {};
        void fdel()
        {
                if (x == 0) throw MyErr();
                x = 1 / x;
        }
};

void main() {
        try
        {
                Cls a(0);
```

```
                    a.fdel();
        }
        catch (Cls::MyErr)
        {
                cerr << "Division by zero";
        }
}
```

Класс MyErr не имеет полей и методов и предназначен только для связывания выражения генерации исключения с улавливающим блоком. Однако класс исключения может иметь свои поля и методы для детализации исключения.

Пример 7.8. Написание класса для ввода массива целых чисел:

```
  const int nmas = 10;
class Cls   {
        int mas[nmas];
        int n;
  public:
    class MyErr
       {
       public:
        int m;
        string st;
         MyErr(string a, int b) : m(b), st(a) {}
       };       // Конец описания класса MyErr

        Cls (int k)  {
                if (k > nmas) throw MyErr("Array index out of bounds", k);
                        n = k;
     for (int i = 0; i < n; i++) mas[i] = i*10;
       }

       void add_mas()
       {
                cout << "Enter the array: ";
      for (int i = 0; I < n; i++)
               {
                        cin >> mas[i];
                        if (mas[i] < 0) throw MyErr("Negative value", mas[i]);
               }
       }
  };    // Конец описания класса Cls
```

```
void main()
 {
 try
 {
  Cls a(5);
  a.add_mas();
 }
 catch(Cls::MyErr er)       // Enter the array: 3 -5 1 1 1
  {
cerr << "Error: " << er.st << " = " << er.m;    // Error: Negative value = -5
  }
}
```

Использование производных классов приводит к образованию сложной иерархической структуры объектов, методы которых могут стать источниками исключений. Схема управления исключениями остается неизменной, однако использование виртуальных функций позволяет организовать детализацию возникающих исключений.

Пример 7.9. Написание класса для ввода-вывода положительного ненулевого целого числа:

```
class BCls {
protected:
        int x;
public:
        BCls():x(0) {}
        virtual void MyErr() { cerr << "Error BCls" << endl; }
};

class Cls01: public BCls {
public:
        void input() {
                cout << "Input x: ";
                cin >> x;
                if (!cin.good()) throw *this;
         }
        virtual void MyErr() { cout << "Input error" << endl; }
};

class Cls02: public Cls01 {
public:
        void output() {
```

```
        if (x < 0) throw *this;
                cout << "Output x = " << x << endl;
                }
         virtual void MyErr() { cout << "Error: x<0" << endl; }
};

class Cls03: public Cls02 {
public:
        operator int() { return x != 0; }
    virtual void MyErr() {  cout << "Error: x == 0" << endl; }
};

void main()
 {
try {
Cls03 a;
 a.input();
 a.output();
  if (!a) throw a;
}
catch (BCls& err)
{
err.MyErr();
}
}
```

7.4. Спецификации исключений

Спецификации исключений – это механизм языка C++, определяющий способ обработки исключений в функции (или методе).

Исходя из этого можно выделить функции, не выбрасывающие исключения, и функции, которые могут выбросить исключение. Данная информация используется компилятором для оптимизации работы с функцией.

Спецификатор noexcept

Ключевое слово noexcept указывает на то, что функция не выбрасывает исключений:

тип имя_функции() **noexcept**;

Если в описании указан спецификатор noexcept(false) или спецификатор отсутствует, то функция может вызывать исключения любого типа.

Наличие спецификатора noexcept не запрещает выброс исключения и вызов функции, выбрасывающей исключение, однако обработка исключения в этом случае зависит от реализации и оптимизации.

Функции, не выбрасывающие исключения:

- конструкторы по умолчанию;
- конструкторы копирования и перемещения;
- деструкторы.

Функции, выбрасывающие исключения:

- обычные функции;
- пользовательские конструкторы;
- некоторые операторы (например, оператор new).

Оператор noexcept

Оператор noexcept принимает в качестве аргумента выражение и возвращает true или false в зависимости от того, считает ли компилятор, что выражение может или не может выбросить исключение.

Пример 7.10.

```
void f1() { throw 0; };
int f2() { return 0; };
int f3() noexcept { return 0; };
class Cls {};
void main()
{
   cout << noexcept(f1()) << endl;  // Выводит: false
   cout << noexcept(f2()) << endl;  // Выводит: false
   cout << noexcept(f3()) << endl;  // Выводит: true
   cout << noexcept(1 + 2) << endl;  // Выводит: true
   cout << noexcept(1 + f2()) << endl;  // Выводит: false
   cout << noexcept(1 + f3()) << endl;  // Выводит: true
   cout << noexcept(Cls()) << endl;    // Выводит: true
}
```

Оператор noexcept проверяется во время компиляции и не вычисляет входное значение. Проверка необходима для реализации гарантий безопасности.

7.5. Гарантии безопасности исключений

Обработчики исключений замедляют выполнение программы, а могут становиться источником ошибок. Для принятия правильного решения о необходимости обработки исключений Дейвом Абрахамсом были предложены ***гарантии***

безопасности исключений – договоренности о реализации поведения функций и классов в случае возникновения исключений.

Существует три уровня гарантии безопасности исключений:

1. **Базовая гарантия.** В случае выброса исключительной ситуации не происходит утечки ресурсов, но программа может изменить свое состояние.
2. **Строгая гарантия.** В случае выброса исключительной ситуации не происходит утечки ресурсов, программа остается в неизменном состоянии.
3. **Гарантия отсутствия исключений.** Функция всегда завершает работу без сбоев или аварийно, но без выбрасывания исключений. В случае аварийного завершения программа возвращает код ошибки или игнорирует проблему (пример – деструктор).

Контрольные вопросы

1. Что такое исключительная ситуация?
2. Какие существуют уровни обработки исключений?
3. Как реализован механизм обработки исключений в языке C++?
4. Какие действия выполняются при порождении исключения?
5. Как реализован механизм структурной обработки исключений?
6. Как получить детальную информацию о причине возникновения исключения?
7. Как организована обработка исключений в объектах классов?
8. Какой механизм языка C++ определяет способ обработки исключений в функции?
9. Какие существуют гарантии безопасности исключений?

8. Динамическая идентификация и приведение типа

8.1. Динамическая идентификация типа

Динамическая идентификация типа данных (*run-time type identification, RTTI*) – механизм, позволяющий определить тип данных переменной или объекта во время выполнения программы.

В процедурном программировании нет необходимости в получении информации о типе во время выполнения программы, т. к. тип каждого объекта известен во время компиляции (т. е. при написании программы).

ООП реализует полиморфизм посредством использования иерархии классов, виртуальных функций и указателей на базовые классы. Указатель на базовый класс можно использовать для ссылки как на объекты базового класса, так и на объекты производных классов. Виртуальные методы будут всегда работать корректно, т. к. будет вызываться метод, соответствующий типу вызвавшего его объекта. При использовании невиртуальных методов указатель на базовый класс содержит информацию только о методах, описанных в базовом классе. Приведение указателя такого типа к указателю на производный класс не является безопасным, поэтому для исключения ошибки требуется точная информация о типе объекта.

Для получения типа объекта во время выполнения программы используется оператор typeid (библиотека *typeinfo*).

```
const type_info& typeid(объект)
```

Некоторые методы класса type_info:

```
bool operator== (const typeinfo &ob) const;
bool operator!= (const typeinfo &ob) const;
const char *name() const;
```

Перегруженные операторы == и != обеспечивают сравнение типов. Метод name() возвращает указатель на имя типа.

При применении оператора typeid к неполиморфным объектам будет получена информация об их базовом типе.

Пример 8.1.

```
class BCls {
int x;
virtual void f() {};
};
class Cls01: public BCls {
int y;
};
```

```
class Cls02: public BCls {
int z;
};

void main() {
int k;
        cout <<"Type k - " << typeid(k).name() << endl;
                                        // Выводит: Type k - int
        cout <<"Type 7.56 - " << typeid(7.56).name() << endl;
                                        // Выводит: 7.56 - double
char s[10];
cout <<"Type s - " << typeid(s).name() << endl;
                                        // Выводит: Type s - char [10]
BCls a;
cout <<"Type a - " << typeid(a).name() << endl;
                                        // Выводит: Type a - class BCls
Cls01 *b = new Cls01;
cout <<"Type b - " << typeid(b).name() << endl;
                                // Выводит: Type b - class Cls01 * __ptr64
Cls02 c, d;
cout <<"Type c - " << typeid(c).name() << endl;
                                // Выводит: Type c - class Cls02
BCls &m = c;
cout <<"Type m - " << typeid(m).name() << endl;
                                        // Выводит: Type m - class Cls02
        if (typeid(a) == typeid(b)) cout << "Same data types" << endl;
                                else cout << " Different data types" << endl;
                                        // Выводит: Different data types
        if (typeid(d) == typeid(m)) cout << "Same data types " << endl;
                                else cout << "Different data types" << endl;
                                        // Выводит: Same data types
}
```

8.2. Приведение типа

Явное приведение типа, используемое в C++, имеет следующие недостатки:

- отсутствие запрета на использование неконтролируемых компилятором преобразований типов, которые могут породить ошибку на этапе выполнения программы;
- указатель на виртуальный базовый класс не может быть явно преобразован в указатель на производный класс;
- отсутствует проверка типов на совместимость.

Введенные в C++ операции приведения типа устраняют указанные недостатки путем использования:

- однозначно интерпретируемого синтаксиса операций;
- контроля ошибок как на этапе компиляции, так и этапе выполнения программы;
- разрешения преобразования указателей на базовый виртуальный класс в указатели на наследующий его производный класс.

Константное приведение типа

Правила языка C++ запрещают передачу константного указателя на место формального параметра, не имеющего модификатора const. Для обхода данного ограничения используется следующая операция:

тип **const_cast** *<тип> (выражение)*

Она приводит константу к неконстантной переменной указанного типа. Следует обратить внимание, что неконстантным становится результат операции но не сама константа.

Пример 8.2. Изменение значения константы:

```
void fun(char* s) {
        _strupr_s(s, 5);
}

void main() {
        const char str[] = "func";
        cout << str << endl; // Выводит: func
// fun(str); // Ошибка! Невозможно преобразовать из const char [5] в char*
fun(const_cast<char*>(str));
        cout << str; // Выводит: FUNC
// fun(str); // Ошибка! Невозможно преобразовать из const char [5] в char*
}
```

Пример 8.3. Увеличение значение поля объекта класса в 10 раз. Метод увеличения должен быть константным (не может изменять значения полей своего класса).

```
class Cls {
        int a;
public:
        Cls(int g) : a(g) {};
        void  fun() const {
                cout << "a = " << a << endl;
         // (this)->a *= 10; // Ошибка: "a" не может быть изменен,
```

```
    // поскольку доступ осуществляется через константный объект
            const_cast <Cls*> (this)->a *= 10;
            cout << "a = " << a << endl;
        }
};
void main() {
        Cls x(5);
        x.fun();
}
```

Динамическое приведение типа

Динамическое приведение типа используется для преобразования указателей **родственных полиморфных** классов иерархии с контролем корректности такого преобразования.

Формат операции:

dynamic_cast *<тип*> (параметр)*

или

dynamic_cast *<тип&> (параметр)*

Оператор преобразует параметр в указанный тип (если он является типом этого объекта). В случае ошибки возвращает nullptr (для указателей) или генерирует исключение bad_cast (для ссылок).

Преобразование из указателя на объект производного класса в указатель на объект базового класса называют ***повышающим*** (*upcast*), а преобразование указателя на объект базового класса в указатель на производный класс – ***понижающим*** (*downcast*). Приведение между производными классами одного базового или между базовыми классами одного производного называют ***перекрестным*** (*crosscast*) преобразованием.

Повышающее преобразование

Повышающее преобразование равносильно простому присваиванию:

```
class BCls { };
class Cls01 : public BCls { };
  …
Cls01* x = new Cls01;
BCls* y = dynamic_cast <BCls*>(x);
BCls *y = x;
```

Понижающее преобразование

Обычно операция dynamic_cast применяется для реализации понижающего преобразования.

Пример 8.4. Понижающее преобразование:

```
class BCls {
protected:
        int x = 1;
public:
    virtual ~BCls() {}
};
class PCls : public BCls {
public:
    void prn() { cout << x << endl;
    }
};
PCls* pntr(BCls* uk) {
    cout << "Po int er: ";
    PCls* pb = dynamic_cast<PCls*>(uk);
    if (pb) cout << "No error" << endl;
    else cout << "Error" << endl;
    return pb;
}
PCls& lnk(BCls* uk) {

        cout << "Link: ";
        PCls& rb = dynamic_cast<PCls&>(*uk);
        cout << "No error" << endl;
        return rb;
}
void main() {
    BCls* ub = new BCls;
    BCls* up = new PCls;
    // up->prn();          // Ошибка: "prn": не является членом "BCls"
    PCls* p = pntr(ub);         // Выводит: Pointer: Error
     if (p) p->prn();
            p = pntr(up);       // Выводит: Pointer: No error
     if (p) p->prn();           // Выводит: 1
     try {
        PCls& q = lnk(ub);
         q.prn();
     }
    catch (bad_cast& x) { cout << x.what() << endl; }
                                // Выводит: Link: Bad dynamic_cast!
```

```
        try {
                PCls& q = lnk(up); // Link: No error
                q.prn();        // Выводит: 1
            }
            catch (bad_cast& x) { cout << x.what() << endl; }
        delete ub; delete up;
    }
```

Понижающее преобразование не работает в следующих случаях:

- используется наследование со спецификаторами private или protected;
- классы не являются полиморфными (не содержат таблиц виртуальных функций);
- в случае возникновения неоднозначностей, связанных с перекрестным наследованием.

Перекрестное преобразование

Перекрестное приведение типа позволяет безопасно связывать производные классы, созданные в разные моменты времени.

Пример 8.5. Перекрестное преобразование:

```
class BCls01 {
public:
    virtualvoid output(const char* s) {
        cout << endl << s;
    }
};

class BCls02 {
protected:
    int x;
public:
    BCls02(int a) : x(a) {}
    virtual ~BCls02() {}
    friend ostream& operator<<(ostream& out, BCls02& p);
};

ostream& operator<<(ostream& out, BCls02& p) {
    BCls01* uk = dynamic_cast<BCls01*> (&p);
    if (uk) uk->output("BCls02: ");
    out << "x = " << p.x ;
    return out;
}
```

```
class Cls01 : public BCls02, public BCls01 {
public:
        Cls01(int a) : BCls02(a) {}
};
class Cls02 : public BCls01 {};
class Cls03 : public BCls02, public Cls02 {
public:
        Cls03(int a) : BCls02(a) {}
};
void main() {
        BCls02 ob01(1);
        cout << ob01;  // Выводит: x = 1
        Cls01 ob02(2);
        cout << ob02; // Выводит: BCls02: x = 2
        Cls03 ob03(3);
        cout << ob03; // Выводит: BCls02: x = 3
}
```

8.3. Другие способы приведения типа

Для невиртуальных типов данных может использоваться статическое приведение типа:

static_cast<*идентификатор_типа*>*(выражение)*

Выражение преобразуется к заданному типу на этапе компиляции. Статическое приведение типа не предполагает проверку безопасности преобразования на этапе исполнения.

Для приведения несовместимых типов используется:

reinterpret_cast<*идентификатор_типа*> *(выражение)*

Данный указатель является самым опасным в применении. Результат приведения в большинстве случаев будет неопределенным. Безопасно приведение значения обратно к исходному типу.

Пример 8.6.

```
class BCls1 { };
class BCls2 { };
class PCls21 : BCls2 { };
class PCls22 : BCls2 { };
void main() {
        int    k = 3,   *pk = &k;
        double b = 5.6, *pb = &b;
        char   ch = 'a';
```

```
    void *v = nullptr;
    BCls1 B1, *pB1 = &B1;
    BCls2 B2, *pB2 = &B2;
    PCls21 P1, *pP1 = &P1;
    PCls22 P2, *pP2 = &P2;

    // Приводим явно double к int
    // k = reinterpret_cast<int>(b);
            // Ошибка: невозможно преобразовать "double" в "int"
    // b = reinterpret_cast<double>(k);
            // Ошибка: невозможно преобразовать "int" в "double"
    // b = reinterpret_cast<double>(pb);
              // Ошибка: невозможно преобразовать "double" в "double"
    // pb = reinterpret_cast<double*>(b);
            // Ошибка: невозможно преобразовать "double" в "double *"
    ch = reinterpret_cast<char>(pk);
    v = reinterpret_cast<void*>(ch);
    pk = reinterpret_cast<int*>(v);
    pk = reinterpret_cast<int*>(pk);
    pB1 = reinterpret_cast<BCls1*>(pB2);
    pB2 = reinterpret_cast<BCls2*>(pP1);
    pP2 = reinterpret_cast<PCls22*>(pB2);
}
```

Контрольные вопросы

1. Какой механизм позволяет определить тип данных переменной или объекта во время выполнения программы?

2. Объясните механизм работы константного приведения типа.

3. Объясните механизм работы динамического приведения типа.

4. Что такое повышающее, понижающие и перекрестное преобразование при динамическом приведении типа?

5. Чем опасно использование reinterpret_cast для приведения типа?

9. Технологические аспекты объектно-ориентированного программирования

9.1. Использование статических элементов класса

Поля и методы класса, объявленные с модификатором доступа static, называются статическими членами класса. Доступ к статическим членам класса разрешен в пределах видимости класса. Независимо от количества существующих объектов в программе статические члены класса присутствуют в единственном экземпляре. Имя статической переменной квалифицируется именем класса с использованием операции разрешения области видимости (::), а не именем экземпляра класса.

Обычно статические элементы используют для нумерации и построения механизмов синхронизации процессов использования объектов.

Для использования статической переменной следует:

1. Объявить статическую переменную как член класса:

static *тип_переменной имя_переменной*;

2. Для выделения памяти ее следует повторно объявить в области объявления глобальных переменных:

тип_переменной имя_класса::*имя_переменной*;

3. Выполнить инициализацию статической переменной (по умолчанию статическая принимает нулевое значение):

имя_класса::*имя_переменной = значение_переменной*;

Инициализацию статической переменной можно выполнять одновременно с выделением памяти.

Пример 9.1.

```
class Cls {
public:
        static int MyStat;
};

int Cls::MyStat = 123;

void main() {
        cout << Cls::MyStat << endl;
}
```

Статические переменные и методы доступны без создания объекта класса. Имя статической переменной квалифицируется именем класса с использованием операции разрешения области видимости (::), а не именем объекта класса.

Статические методы могут вызывать и использовать только другие статические методы и статические переменные. Из статического метода нельзя выполнять вызов нестатического метода.

Для статических переменных могут использоваться указатели.

Например:

```
int *u = &Cls::MyStat;
```

Реализация статической функции записывается так же, как и реализация любого другого метода.

Пример 9.2. Реализация статического метода:

```
class Cls  {
public:
        static int fstat();
};

int Cls::fstat(void)  {
return 9;
}

void main() {
        std::cout << Cls::fstat();
}
```

9.2. Многофайловые программы

Для реализации принципа раздельного программирования используются многофайловые программы. В C++ широко используются библиотеки классов. Обычно библиотека классов состоит из интерфейса (*interface*) и реализации (*implementation*).

Интерфейс содержит различные определения, включая объявления классов, и находится в общедоступной части библиотеки (обычно в заголовочном файле с расширением *.h*). Реализация – содержимое классов (чаще всего поставляется в виде объектных (*.OBJ*) или библиотечных (*.LIB*) файлов).

В могофайловых программах осуществляется взаимодействие откомпилированных по отдельности, но скомпонованных вместе исходных файлов.

Правила межфайлового взаимодействия переменных, функций и классов различны.

Межфайловые переменные

Переменная может быть объявлена с выделением памяти (по умолчанию) или без выделения памяти (определение). Для определения переменной перед именем типа указывается зарезервированное слово extern.

Примеры, содержащие ошибку:

```
extern int k;

void f()
{
  k = 9;
}

void main() {
          f();
cout  << k << endl;
}
```

```
extern int k;

void f()
{
int k = 9;
}

void main() {
          f();
cout  << k << endl;
}
```

Программы содержат ошибку: «*error LNK2001: неразрешенный внешний символ*». Это вызвано тем, что определение переменной не приводит к выделению памяти.

Пример 9.3.

```
extern int k;

void f()
{
  k = 9;
}
int k;          // Выделение памяти
void main() {
        f();
cout  << k << endl;
}
```

Обычно определение используется для доступа к глобальным переменным, определенным в другом модуле, т. к. глобальная переменная может быть определена только один раз во всей программе независимо от количества файлов программы.

Пример объявления глобальных переменных в двух файлах модуля:

// Файл 1 **int** k;	*// Файл 1* **int** k;	*// Файл 1* **int** k;
// Файл 2 **int** k;	*// Файл 2* k = 9;	*// Файл 2* **extern int** k; k = 9;
Ошибка! Повторное объявление переменной	**Ошибка!** Переменная во втором файле не объявлена	**Правильно**

Определение переменной указывает на то, что объявление переменной находится в другом файле (в этом случает компилятор не считает ошибкой отсутствие оператора выделения памяти для переменной). Необходимая связь с объявленной глобальной переменной устанавливается на этапе компоновки.

Нельзя инициализировать переменную одновременно с определением переменной, т. к. в этом случае компилятор игнорирует спецификатор extern.

Для расширения области видимости констант до нескольких файлов проекта используются следующие объявления:

```
// Файл 1
    extern const int k = 9;    // В первом файле
// Файл 2
    extern const int k;        // Во втором файле
```

Компилятор различает объявление и определение константы по наличию или отсутствию инициализации.

Если в программах проекта необходимо использовать различные глобальные переменные с одинаковыми именами, то используется зарезервированное слово static. В этом случае область видимости глобальной переменной сужается до файла, в котором она определена:

// Файл 1 **int** k;	*// Файл 1* **static int** k;
// Файл 2 **int** k;	*// Файл 2* **static int** k;
Ошибка! Повторное объявление переменной	**Правильно**

Статическая глобальная переменная имеет внутреннее связывание, а нестатическая − внешнее связывание.

Ключевое слово static имеет несколько значений в зависимости от контекста. Для локальной переменной static изменяет время жизни, для классов – позволяет получать переменные, имеющие одно значение для всех объектов.

Межфайловые функции

Так как на этапе компиляции требуется только информация о прототипе функции, то возможно размещение тела функции в любом файле проекта (extern не требуется).

```
// Файл 1
    double f(int x, double y)  // Объявление функции
    {return a + b; }
// Файл 2
    double f(int, double);      // Прототип функции
    ...
    double s = f(4, 7.5);
```

Ключевое слово extern при работе с функциями не используется, т. к. компилятор отличает определение от объявления по наличию или отсутствию тела функции.

Для того чтобы одноименные функции в разных файлах не конфликтовали, используется зарезервированное слово **static** (аналогично переменным).

Межфайловые классы

Определение класса не резервирует память (резервирование памяти происходит только при создании объекта).

Для получения доступа к классу из любого файла проекта необходимо определять класс в каждом файле, т. к. компилятору необходимо знать тип компилируемых данных.

Заголовочные файлы

Заголовочные файлы (расширение *.h*) предназначены для открытия доступа к данным, находящимся в файле с расширением *.cpp*.

Пример 9.4.

Файл *mylib.h*:

```
extern int k;
int f(int, int);
```

Файл *mylib.cpp*:

```
int k;
int f(int x, int y)
{ return x + y; }
```

Файл *myfl.cpp*:

```
#include <iostream>
#include "mylib.h"
using namespace std;
void main() {
  k = f(4, 5);
 cout << k << endl;
}
```

В заголовочном файле располагаются определения переменных, функций, классов.

Пример 9.5.

Файл *fh.h*:

```
class Cls
{
  int k;
public:
  int f(int, int);
};
```

Файл *ff.cpp*:

```
#include "fh.h"
int Cls::f(int x, int y) {
        return x + y;
}
```

Файл *fm.cpp*:

```
#include "fh.h"
#include <iostream>
using namespace std;
void main() {
 Cls a;
 int s = a.f(4,5);
cout << s << endl;
}
```

Определение методов класса может находиться в любом исходном файле (компоновщик соединит все вместе).

9.3. Пространства имен

При написании библиотек классов удобно использовать короткие и понятные имена (например, *add, print*), однако высока вероятность того, что эти имена

уже используются в других программах проекта. Решить проблему совпадения имен позволяет использование пространств имен.

Пространство имен – именованная область программы, предназначенная для локализации имен идентификаторов. Определение пространства имен:

```
namespace имя_пространства_имен
{
идентификаторы
}
```

Операторы, находящиеся вне пространства имен, не имеют доступа к элементам пространства имен. Получить доступ к элементам пространства имен извне можно:

- указанием имени пространства имен перед идентификатором:

  имя_пространства_имен::*имя_элемента*

- использованием оператора using:

  using namespace *имя_пространства_имен*;

Директива using делает идентификаторы пространства имен доступными начиная с места ее объявления и до конца программы (если находится в области объявления глобальных переменных) или до конца функции (если объявлена в функции).

Можно сделать доступным только один элемент:

using *имя_пространства_имен*::*имя_элемента*;

Определение одного пространства имен может встречаться несколько раз, что позволяет использовать его для нескольких заголовочных файлов.

Пример 9.6.

Файл *fl01.h*:

```
namespace sp {
        void f1();
}
```

Файл *fl02.h*:

```
namespace sp  {
        void f2();
}
```

Файл *m.h*:

```
#include "fl01.h"
#include "fl02.h"
```

```
using namespace sp;
f1();
f2();
```

Объявление разрешено вне пространства имен, например:

```
using namespace sp
{
        int a;
}
int sp::b;
```

Допустимо использование вложенных пространств имен, например:

Файл *pr01.h*:

```
include <iostream>
namespace sp
{
   namespace sp01  {
     void f() { std::cout << "Namespace sp01"; }
   }
   namespace sp02  {
      void f() { std::cout << "Namespace sp02"; }
   }
  inline namespace sp03  {
       void f() { std::cout << "Namespace sp03"; }
   }
}
```

Файл *pr01.cpp*:

```
#include <iostream>
#include "pr01.h"
using namespace std;
void main()  {
   //  sp01::f();       // Ошибка: идентификатор не найден
   sp::sp01::f();      // Выводит: Namespace sp01
   sp::sp02::f();      // Выводит: Namespace sp02
   sp::sp03::f();      // Выводит: Namespace sp03
   sp::f();            // Выводит: Namespace sp03
   //  sp03::f();       // Ошибка: идентификатор не найден
}
```

В отличие от обычных вложенных пространств имен члены встроенного пространства имен обрабатываются как члены родительского пространства

имен. Это позволяет выполнять поиск перегруженных функций, имеющих перегрузки в родительском и вложенном пространстве имен, а также объявлять специализации в родительском пространстве имен для шаблонов, объявленных во встроенном пространстве имен.

Встроенные пространства используются в качестве механизма управления версиями в открытом интерфейсе библиотеки.

Каждое имя пространства имен должно быть уникальным, поэтому возрастает его длина (что неудобно в использовании). Допустимо использование псевдонимов пространства имен:

```
namespace very_long_namespace_name { class Cls {}; }
namespace P = very_long_namespace_name;
…
very_long_namespace_name::Cls C1;
      P::Cls C2;
```

Разрешено использовать неименованные пространства имен. В этом случае элементы пространства имен видны только из текущего файла, что позволяет избегать конфликтов использования одинаковых имен в разных файлах:

```
namespace
{
void f() { … }
}
```

Если идентификатор не объявлен явно в пространстве имен, то считается, что он находится в глобальном пространстве имен. В случае необходимости явного указания на идентификатор, находящийся в глобальном пространстве имен, используется операция разрешения области видимости:

```
::f();
```

Контрольные вопросы

1. Объясните особенности использования статических элементов класса.
2. Как объявляются и используются статические переменные?
3. Как организовано межфайловое взаимодействие переменных?
4. Как организовано межфайловое взаимодействие функций и классов?
5. Для чего создается пространство имен?
6. Как организуется доступ к элементам определенного пространства имен?

10. Шаблоны в языке C++

10.1. Шаблоны функций

Если алгоритм (например, сортировка) должен работать с различными типами данных, то приходится использовать перегруженные функции, что усложняет код программы.

В случае если необходимо написать несколько одинаковых алгоритмов, отличающихся только типом данных, используются шаблоны.

Шаблон (*template*) – средство языка, предназначенное для создания обобщенных алгоритмов без привязки к конкретным параметрам (например, типам данных, размерам буферов). Точное значение параметров передается на этапе компиляции. Таким образом, автоматически создается необходимое количество перегруженных функций.

Функция-шаблон имеет следующий вид:

```
template <typename список_параметров>
тип_результата имя_функции(список_параметров)
{
// Тело функции
}
```

Все шаблоны функций начинаются с ключевого слова template. Каждому параметру должно предшествовать зарезервированное слово class или typename.

Виды параметров шаблона:

- *типизированный параметр* – ключевое слово typename (class) и идентификатор;
- *нетипизированный параметр* – имя типа и идентификатор, определяющие константу.

Описанные параметры могут быть использованы в заголовке и теле функции.

Пример 10.1. Программа, находящая минимальное значение из двух элементов:

```
template <typename T> // Шаблон функции
T minx(T x, T y)
{
        return (x < y) ? x : y;
}
void main()
{
        int x = 4, y = 3;
```

```
        cout << minx(x, y) << endl;   // Выводит: 3
        double a = 4.563, b = 9.74;
        cout << minx(a, b) << endl;   // Выводит: 4.563
        char h = 's', f = 'a';
        cout << minx(h, f) << endl;   // Выводит: 'a'
    // cout << minx(a, h) << endl;
    // error C2782: T minx(T,T): в шаблоне параметр "T" неоднозначен
    }
```

Для шаблонов неявные преобразования типов не выполняются.

Порядок перечисления параметров в списке шаблона значения не имеет.

При использовании шаблона компилятор автоматически формирует экземпляр функции по набору типов аргументов.

Пример 10.2. Нахождение среднего арифметического значения элементов массива:

```
template <typename Tm, typename Tn>
double sred(const Tm* mas, Tn n)
{
        double s = 0;
        for (int i = 0; i < n; i++) s += mas[i];
        return s / n;
}
void main()
{
        const int n1 = 4;
        int mas1[n1] = { 1, 2, 3, 5 };
cout << " Average mas1 = " << sred(mas1, n1) << endl;  // Выводит: 2.75
        const signed char n2 = 3;
        double mas2[n2] = { 5.98, 6.2, 10.3 };
cout << " Average mas2 = " << sred(mas2, n2) << endl; // Выводит: 7.49333
}
```

Объявление шаблона не приводит к генерации кода. Сама генерация (реализация шаблона) происходит в месте вызова функции. При этом генерируется код с используемым в данном случае типом аргументов (при повторном вызове с аналогичными аргументами генерации не происходит). Если встречается другой, разрешенный тип аргументов, то происходит еще одна реализация шаблона. Таким образом, для примера 10.1 будет реализовано три шаблона (в программе будет существовать три перегруженных функции), а для примера 10.2 – два шаблона.

Использование шаблонов по сравнению с перегрузкой функции не уменьшает размер исполняемого модуля, но сокращает код программы и увеличивает ее надежность.

В списке параметров функции обязательно должны присутствовать типы, перечисленные в описании шаблона (список не может быть пустым).

Кроме параметров-типов допускается использование параметров-констант. В этом случае вместо имени параметра подставляется значение константы из определения шаблона. Для определения фактического типа и значения константы исследуются фактические аргументы, переданные при вызове функции. Процесс определения типов и значений аргументов шаблона по известным фактическим аргументам функции называется ***выведением*** (*deduction*) аргументов шаблона.

Пример 10.3. Нахождение суммы элементов массива:

```
template <typename T, int n>
T sum(T(&mas)[n])
{
        T s = 0;
        for (int i = 0; i < n; i++) s += mas[i];
        return s;
}
int main()
{
        int mas1[] = { 1, 2, 3, 4 };
        cout << sum(mas1) << endl;  // Выводит: 10
        double mas2[3] = { 2.7, 7.88, 12.5 };
        cout << sum(mas2) << endl;  // Выводит: 23.08
        return 0;
}
```

Вывод аргументов шаблона в примере 10.3 осуществляется следующим образом. Параметром функции в шаблоне sum является ссылка на массив элементов типа *T*. Для сопоставления с формальным параметром функции фактический аргумент также должен быть значением, представляющим тип массива.

При выводе аргументов шаблона принимается во внимание, что mas1 является параметром типа «указатель на массив из четырех элементов типа int».

Алгоритм выведения аргументов шаблона:

1. По очереди исследуется каждый фактический аргумент функции, чтобы выяснить, присутствует ли в соответствующем формальном параметре параметр шаблона.

2. Если параметр шаблона найден, то путем анализа типа фактического аргумента выводится соответствующий аргумент шаблона.

3. Если один и тот же параметр шаблона найден в нескольких формальных параметрах функций, то аргумент шаблона, выведенный по каждому из соответствующих фактических аргументов, должен быть одинаковым.

Недостаток подобного подхода состоит в том, что при использовании шаблона с массивами одного и того же типа, но разных размеров генерируются несколько экземпляров функции.

Разрешено явное задание аргументов шаблона.

Пример 10.4. Явное задание аргументов шаблона:

```
template <typename tx, typename ty, typename tz>
void fun(ty, tz)  {}
int main() {
        fun<int,char,double>(1,'2');        // tx - int, ty - char, tz - double
        fun<int,char>(1,'2');               // tx - int, ty - char, tz - char
        fun<int>(1,'2');                    // tx - int, ty - int, tz - char
// fun('1','2');
// error: void fun(ty,tz): не удается составить аргумент шаблона для "tx"
}
```

Если для некоторых типов данных требуется отличная от шаблона реализация, то можно описать обычную функцию, список типов аргументов и возвращаемого значения которой соответствует объявлению шаблона. Такая перегружающая шаблон функция называется ***специализацией шаблонной функции***.

Пример 10.5. Нахождение частного двух чисел:

```
template <typename  tp1, typename tp2>
double razn(tp1 x, tp2 y) {
 if (y == 0) exit(1);
   return x / y;
}
template <>
double razn<int, int>(int x, int y) {
 if (y == 0) exit(1);
   return static_cast <double>(x)/y;
}
void main() {
        cout << razn(5.2, 2) << endl; // Выводит: 2.6
        cout << razn(1, 4) << endl; // Выводит: 0.25
}
```

При наличии нескольких одноименных функций поиск функции для выполнения осуществляется в следующей последовательности:

- поиск нешаблонной функции с совпадающим списком параметров;
- поиск шаблона функции с точным соответствием списка параметров;
- поиск функции по условию совпадения списка параметров после возможных преобразований типов.

Анализ компилятором шаблонов функций не предполагает преобразования типов.

Пример 10.6. Выведение аргументов шаблона:

```
template <typename T>
void fun(T x, T y)
{ }
int main()
{
        fun(1, 1);              // T - int
        fun('s', 'd');          // T - char
        fun(5.2, 4.5);          // T - double
        // fun(5.2, 4);         // Ошибка: параметр шаблона неоднозначен
        fun <int> (5.2, 4);     // T - int
        // fun(1, '1');         // Ошибка: параметр шаблона неоднозначен
        fun <double> (1, '1');          // T - double
}
```

10.2. Шаблоны классов

Рассмотренный выше шаблонный принцип можно распространить на классы. Обычно это классы, являющиеся хранилищами данных.

Объявление шаблона класса:

```
template <typename параметр>
class имя_класса
{
// Тело класса
};
```

При включении шаблона класса в программу сам класс не генерируется до тех пор, пока не будет найден экземпляр шаблонного класса с указанным конкретным типом.

имя_класса <тип> имя;

или

*имя_класса <тип> *имя* = **new** *имя_класса <тип>*

Пример 10.7. Класс, позволяющий добавлять элементы в конец массива и извлекать элементы из конца массива:

```
template < typename tp>
class ShCls
{
  tp mas[100];
  int k;
public:
ShCls() : k(0) {}
void push(tp x){
        mas[++k] = x;
}
tp pop(){
        return mas[k--];
} };
void main()
{
ShCls <int> a;
ShCls <double> b;
a.push(1); // Ввод данных
a.push(4);
cout << a.pop() << " " << a.pop() << endl;  // Вывод данных
b.push(15.34);
b.push(48.3);
cout << b.pop() << " " << b.pop() << endl;
}
```

Если методы класса описываются вне класса, то обязательно добавление template перед заголовком каждого метода.

Если функции-элементы определяются вне определения класса, то синтаксис их определения должен иметь следующий вид:

```
template <список_параметров_шаблона>
тип имя_класса<параметры_шаблона>::имя_функции(параметры)
{
  // Тело функции
}
```

Например, для примера 10.7 при описании функции вне класса, она будет выглядеть так:

```
template <typename tp>
tp ShCls<tp>::pop()
```

```
{
        return mas[k--];
}
```

Шаблонный класс может использовать шаблонные функции.

Пример 10.8. Класс для работы со стеком:

```
template <typename st>
struct tstk         // Объявление стека tstk<st>
{
  st inf;
  tstk* a;
};
template <typename tp>
class StkCl    {   // Объявление класса StkCl<tp>
    tstk<tp> * sp;    // Указатель на вершину стека
  public:
         StkCl() : sp(NULL) {}
    void push(tp inf); // Добавление данных (одна ссылка)
 //   void outCls();       // Вывод всех ссылок
       ~StkCl();
  }
template <typename tp>
void StkCl<tp>::push(tp inf)  {   // Добавление данных
  tstk <tp> *spt = new tstk<tp>;
    spt->inf = inf;
    spt->a = sp;
  sp = spt;
  }
template < typename tp>           // Вывод и освобождение памяти
StkCl<tp>::~StkCl() {
tstk <tp> *spt;
  while(sp != NULL)
          {
    spt = sp;
        cout << sp->inf << endl;
     sp = sp->a;
   delete spt;
            }
return;
}
```

```
void main()
 {
 StkCl<double> s;
  s.push(101.1);
      s.push(202.2);
 StkCl<char> ch;
  ch.push('a');
      ch.push('b');
      ch.push('c');
 }
```

Каждая версия класса, создаваемая по шаблону, содержит одинаковый базовый код с измененными параметрами шаблона. Иногда для определенных типов данных требуется написание кода, отличного от базового. Для этого может быть предусмотрена специальная реализация отдельных методов шаблона либо полное переопределение (специализация) шаблона класса:

```
template <>
class имя_класса <имя_специализированного_типа> { }
```

Пример 10.9. Использование специализации:

```
template <typename T, int n> // Шаблон класса
class Cls
{
protected:
      T *mas;
public:
      Cls();
      ~Cls();
};
template <typename T, int n> // Конструктор
Cls<T, n>::Cls()
{
      mas = new T[n];   // Создание массива из n элементов типа T
      cout << endl << "Array of " << n << " elements of the size " << sizeof(T);
};
template <typename T, int n> // Деструктор
Cls<T, n>:: ~Cls()
{
      cout << endl << "Deallocation of " << n << " elements of the size "
                                                      << sizeof(T);
      delete[ ] mas;
}
```

```
// Частичная специализация класса
template <>
Cls<double, 12>::Cls()
{
	mas = new double[12];
	cout << endl << "Spartial specialization: array of " << 12
				<< " elements of the size " << sizeof(double);
}
// Полная специализация класса
const int k = 20;
template <>
class Cls <char, k> {
protected:
	char *mas;
public:
	Cls<char, k>() {
		mas = new char[k];
		mas[0] = '\0';
cout << endl << "Full specialization: array of " << k << " elements of char";
	}
	~Cls() {
		cout << endl << "Full specialization: deallocation of " << k
							<< " elements of char";
		delete[] mas;
	}
};
void main() {
	Cls<int, 10> x1; // Без специализации
					// Array of 10 elements of the size 4
	Cls<double, 12> x2; // Частичная специализация
		// Spartial specialization: array of 12 elements of the size 8
	Cls<int, 12> x3; // Без специализации
					// Array of 12 elements of the size 4
	Cls<char, 20> z2; // Полная специализация
			// Full specialization: array of 20 elements of char
	Cls<char, 30> z3; // Без специализации
				// Array of 30 elements of the size 1
	cout << endl;
	// Deallocation of 30 elements of the size 1
	// Full specialization: deallocation of 20 elements of char
	// Deallocation of 12 elements of the size 4
```

```
        // Deallocation of 12 elements of the size 8
        // Deallocation of 10 elements of the size 4
}
```

Для настройки шаблонных классов можно использовать функциональные объекты. ***Функциональным объектом*** (***функтором***) называется объект, для которого переопределена операция вызова функции **operator**().

Пример 10.10. Создание шаблонного класса для выбора минимального или максимального из полей класса. Стратегия выбора передается как параметр шаблона.

```
template <typename T>
struct Less {
 bool operator()(const T& x, const T& y)
        {
                return x < y;
        }
};
template <typename T>
struct More {
        bool operator()(T x, T y)
        {
                return x > y;
        }
};
template <typename T, typename Compare>
class Cls {
        T a, b;
public:
        Cls(T x, T y) : a(x), b(y) {}
        void ekstr()
        {
                cout << (Compare()(a, b) ? a : b) << endl;
        }
};
void main() {
        Cls<int, Less<int> > f1(22, 6);
        f1.ekstr();  // Выводит: 6
        Cls<double, More<double> > f2(24.2, 7.3);
        f2.ekstr();  // Выводит: 24.2
}
```

В шаблонах классов допускается использование статических данных и методов, дружественных функций и вложенных классов.

Дружественная функция, не использующая параметры шаблона, создается в единственном экземпляре. Если дружественная функция использует параметры шаблона, то она также будет шаблоном. Конкретная реализация такой функции соответствует реализации класса с совпадающими по типам фактическими аргументами.

Статические элементы создаются для каждого создаваемого класса.

Пример 10.11. Использование дружественной функции и статического поля в шаблоне класса:

```
template <typename T>
class Cls
{
        T x;
public:
        static T y;
        Cls(){}; // Конструктор по умолчанию
        Cls(const T& m) : x(m) {};
        void outp() {
                cout << x;
        }
        friend Cls<T>&  operator+= (Cls<T>& p1, const Cls<T>& p2)
        {
                p1.x += p2.x;
                return p1;
        }
        template <typename T1>
        friend Cls<T1>&  operator-= (Cls<T1>& p1, const Cls<T1>& p2);
};

template <class T> T Cls<T>::y = 9;
template <typename T1>
Cls<T1>&  operator-= (Cls<T1>& p1, const Cls<T1>& p2)
{
        p1.x -= p2.x;
        return p1;
}

void main() {
        Cls <int> a = 2, b = 3;
```

```
    Cls <double> d;
    a += b;  a.outp();      // Выводит: 5
    a -= b;  a.outp();      // Выводит: 2
    cout << a.y;            // Выводит: 9
    a.y = 7;
    cout << b.y;            // Выводит: 7
    cout << d.y;            // Выводит: 9
}
```

Шаблоны классов и функций могут содержать параметры по умолчанию. Параметры по умолчанию должны располагаться правее обычных параметров. Если в шаблоне все параметры имеют значения по умолчанию, используются пустые треугольные скобки.

Например:

```
template <typename T = int, typename P = double>
class Cls
{…}
…
Cls <char> a;  // T – char, P – double
Cls <> b;      // T – int,    P – double
```

10.3. Шаблоны с переменным числом параметров

Допустимо использование шаблонов классов и функций с произвольным числом аргументов.

Функция с переменным числом параметров объявляется таким образом:

```
template <typename ... Имя_пакета_аргументов>
тип_результата имя_функции(Имя_пакета_аргументов... args);
```

Класс с переменным числом параметров объявляется следующим образом:

```
template <typename ... Имя_пакета_аргументов>
class Имя_класса;
```

Если требуется задание не менее одного параметра, то этот параметр указывается отдельно:

```
template <typename Первый_параметр,
                    typename ... Имя_пакета_аргументов>
class Имя_класса;
```

В шаблонах с переменным числом аргументов используется оператор sizeof...(), который определяет число аргументов в пакете:

```
int n = sizeof ... (Имя_пакета_аргументов);
```

Пример 10.12. Вывод значения всех параметров функции на экран. Использовать шаблон с переменным числом.

```
// Шаблон для вывода одного параметра
    template <typename T>
    void outp(const T& p)
    {
     cout << p << endl;
    }
    // Шаблон двух и более параметров
    template <typename T, typename... Ar>
    void outp(const T& p, const Ar&... pp)
    {
            cout << p << " ";
            outp(pp...); // Рекурсивное движение по пакету
    }

void main() {
    outp(3, "absd");  // Выводит: 3 absd
    outp('d', 5.6, "xyz", 30); // Выводит: d 5.6 xyz 30
}
```

Достоинства шаблонов:

- высокая эффективность работы с различными типами объектов класса по сравнению с полиморфизмом;
- безопасное использование типов.

Недостатки шаблонов:

- увеличение размера исполняемого модуля программы из-за наличия представителей шаблона для каждого порожденного типа;
- реализация шаблона может оказаться оптимальной только для некоторого набора типов.

10.4. Умные указатели

Обычные указатели предоставляют программисту широкие возможности для управления данными, однако имеют ряд недостатков:

- нет информации о количестве объектов, на которые указывает указатель (на один или массив);
- нет информации о количестве указателей, которые указывают на один и тот же объект;

- нет информации о том, какой из указателей ответственен за освобождение памяти;
- нет информации о том, каким образом должна освобождаться память;
- нет информации о существовании объекта (он может быть уже уничтожен).

Отсутствие полной информации о связанном с указателем объекте приводит к опасности возникновения утечек памяти, ошибок, связанных с обращением к неинициализированной области и попыткой освобождения уже освобожденного участка памяти.

Для решения этих проблем предложена концепция *RAII* (*Resource Acquisition Is Initialization*), которая предлагает совмещение создания ресурса с инициализацией, а освобождение – с его уничтожением.

Для этой цели служат умные (интеллектуальные, *smart*) указатели, которые представляют собой шаблоны классов, которые объявляются в стеке и инициализируются с помощью указателя, указывающего на размещенный в куче объект. Умный указатель объявляется в стеке, поэтому в случае выхода из области видимости вызывается деструктор.

Наиболее часто используются три типа умных указателей: unique_ptr, shared_ptr и weak_ptr (библиотека *memory*).

Уникальный указатель (unique_ptr)

Создание объекта:

```
unique_ptr< тип > имя_указателя;
unique_ptr< тип > имя_указателя (nullptr);
unique_ptr< тип > имя_указателя (new тип);
unique_ptr< тип > имя_указателя (new тип(значение));
```

Уникальный указатель имеет непередаваемое исключительное право на динамический объект. Такой указатель нельзя скопировать, но можно передать значение указателя и права другому объекту.

Пример 10.13.

```
class Cls
{
public:
    Cls() { cout << "Constructor "; }
   ~Cls() { cout << "Destructor"; }
};
void main() {
```

```
    {
        unique_ptr<Cls> a(new Cls);    // Выводит: Constructor
    } // Выводит: Destructor
    cout << "End of program"; // Выводит: End of program
}
```

Использование new в некоторых случаях может приводить к утечкам памяти. Например, при вызове функции *f*:

```
void f(unique_ptr <int>a, unique_ptr <int> b) { }
void main() { f(unique_ptr<int>(new int), unique_ptr<int>(new int)); }
```

Так как компилятор самостоятельно определяет порядок действий при вызове функции, то может возникнуть ситуация, когда при возникновении исключительной ситуации в одном объекте память, выделенная для другого объекта, не освобождается.

Для решения этой проблемы был введен шаблонный класс make_unique:

```
template < class T, class ... Args >
unique_ptr < T > make_unique ( Args && ... args ); // Не массив
```

или

```
template < class T >
unique_ptr < T > make_unique ( std::size_t size ); // Массив
```

Пример 10.14.

```
class Cls {
    int x, y;
public:
Cls(int a = 0, int b = 0) : x(a), y(b) { }
friend ostream& operator<<(ostream& os, const Cls& p) {
    return os << p.x << " " << p.y << endl;
  }
};

void main()
 {
    unique_ptr<Cls> a = make_unique<Cls>();
    cout << *a; // Выводит: 0 0
    unique_ptr<Cls> b = make_unique<Cls>(7, 12);
   cout << *b; // Выводит: 7 12
    unique_ptr<Cls[]> mas = make_unique<Cls[]>(3);
    for (int i = 0; i < 3; i++)  cout << mas[i]; // Выводит: 0 0 / 0 0 / 0 0
    }
```

Основные методы класса unique_ptr:

- get() – возвращает обычный указатель, сохраняя право на владение объектом;
- release() – возвращает обычный указатель, передавая право на владение объектом. Новый указатель должен отвечать за освобождение памяти;
- swap() – меняет местами указатели на объекты;
- reset() – освобождает динамическую память и обнуляет указатель.

Так как объект unique_ptr нельзя скопировать, то запрещено использовать операцию присваивания. Для передачи права владения содержимым другому объекту используется функция перемещения move().

Для работы с умными указателями определены перегруженные операции * и ->.

Пример 10.15.

```
class Cls {
public:
   Cls() { cout << "Constructor" << endl; }
   void prn() { cout << "Available" << endl; }
   ~Cls() { cout << "Destructor" << endl; }
};
void main() {
   unique_ptr <Cls> p = make_unique<Cls>(); // Выводит: Constructor
     p.reset();  // Выводит:  Destructor
   unique_ptr <Cls> c = make_unique<Cls>(); // Выводит: Constructor
   auto a = make_unique<Cls>(); // Выводит: Constructor
      a->prn(); // Выводит: Available
      (*a).prn(); // Выводит: Available
 // unique_ptr<Cls> b(a); // Ошибка!
   unique_ptr <Cls> b;
  //   b = a;  // Ошибка!
   b = move(a);
   if (!a) cout << "a = nullptr" << endl; // Выводит: a = nullptr
   unique_ptr <Cls> g(b.release());
   if (!b) cout << "b = nullptr" << endl; // Выводит: b = nullptr
      g->prn(); // Выводит: Available
       Cls* h = g.release();
   h->prn();  // Выводит: Available
   delete h; // Выводит:  Destructor
   // Выводит: Constructor
}
```

Для передачи параметра unique_ptr в функцию по значению требуется использовать move, т. к. копирование запрещено.

Пример 10.16.

```
class Cls {
   int x;
public:
   Cls(int x = 0) : x(x) { }
   friend ostream& operator<<(ostream& os, const Cls& p) {
      return os << p.x;
   }
   unique_ptr<Cls> f(unique_ptr<Cls> p) {
      p->x = 10;
      return p;
   }
};
void main() {
  unique_ptr<Cls> a = make_unique<Cls>(5);
   cout << *a; // Выводит: 5
   unique_ptr<Cls> b = a->f(move(a));
   cout << *b; // Выводит: 10
}
```

Указатель unique_ptr предназначен только для перемещения, поэтому занимает мало ресурсов. Разрешено преобразование указателя unique_ptr в указатель shared_ptr.

Указатель unique_ptr также можно использовать при работе с динамическими массивами, однако для этого лучше использовать array, vector или string.

Совместно используемый указатель (shared_ptr)

В отличие от уникального указателя совместно используемый указатель позволяет совместное владение объектом. Деструктор выполняется при уничтожении последнего владельца. Указатель, кроме адреса объекта, хранит указатель на управляющий блок и другую необходимую информацию (рис. 10.1). Память для размещения управляющего блока выделятся динамически.

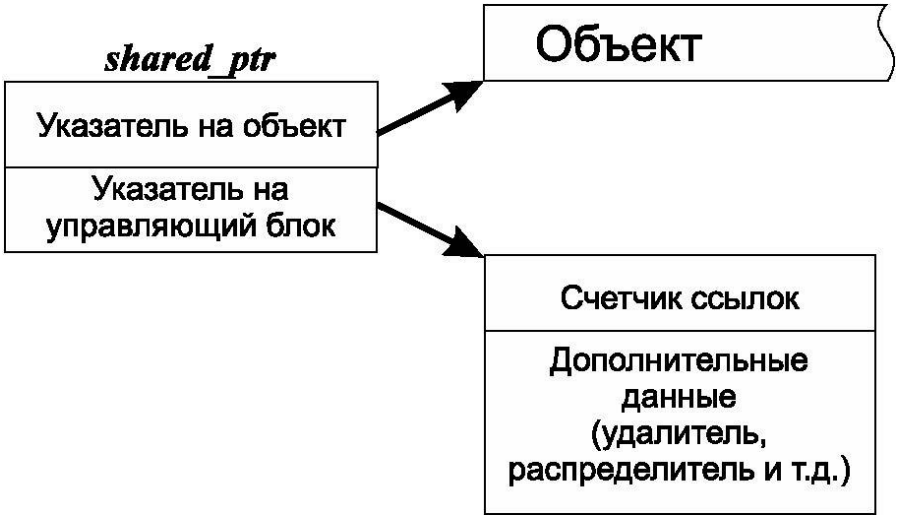

Рис. 10.1

Для удаления объекта, как правило, используется функция delete, однако разрешено использовать пользовательский удалитель, информация о котором помещается в управляющий блок.

Если указатель shared_ptr создается из указателя unique_ptr или обычного указателя, то он создает управляющий блок.

Основные методы класса shared_ptr:

- use_count() – возвращает количество указателей, совместно использующих объект;
- unique() – возвращает true, если у объекта только один владелец, иначе – false;
- get() – возвращает обычный указатель на объект;
- swap() – меняет местами указатели на объекты;
- reset() – уменьшает значение счетчика ссылок на 1. Если значение счетчика становится равным 0, то освобождает динамическую память и обнуляет указатель.

Указатель unique_ptr может быть конвертирован в указатель shared_ptr с использованием конструктора shared_ptr. Обратное преобразование небезопасно.

Пример 10.17.

```
class Cls
{
public:
   Cls() { cout << "Constructor " << endl; }
   void prn() { cout << "Available " << endl; }
```

```
    ~Cls() { cout << "Destructor" << endl; }
};

void main()
{
    shared_ptr<Cls> a( new Cls );    // Выводит: Constructor
    a->prn(); // Выводит: Available
    (*a).prn(); // Выводит: Available
    shared_ptr<Cls> b;
    b = a;
    shared_ptr<Cls> c(b);
    cout << c.use_count(); // Выводит количество владельцев: 3
    a.reset();
    b.reset();
    cout << c.use_count(); // Выводит количество владельцев: 1
    c->prn(); // Выводит: Available
      unique_ptr<Cls> m( new Cls );    // Выводит: Constructor
    shared_ptr<Cls> n(m.release());
   //  unique_ptr<Cls> v(n);  // Ошибка!
     cout << n.use_count(); // Выводит количество владельцев: 1
   n.reset();   // Выводит:  Destructor
} // Выводит:  Destructor
```

Слабый указатель (weak_ptr)

Недостатком совместно используемых указателей (shared_ptr) является возможность получения циклических зависимостей, приводящих к утечкам памяти.

Например, в программе

```
class Cls {
public:
    shared_ptr<Cls> q;
    Cls() { cout << "Constructor " << endl; }
    ~Cls() { cout << "Destructor" << endl; }
};

void main()
{
    {shared_ptr<Cls> a(new Cls);    // Выводит: Constructor
    a->q = a;
    }
}
```

указатель a при выходе из области видимости, не уничтожает объект, т. к. им одновременно владеет и указатель q. Так как q никогда не выйдет из области видимости, то деструктор никогда не будет выполнен.

Для решения этой проблемы введен weak_ptr (слабый указатель), который позволяет работать с объектом, не являясь его владельцем (не увеличивая счетчик ссылок).

Например, изменив в предыдущем примере указатель, получим

```
class Cls
 {
public:
   weak_ptr<Cls> q;
   Cls() { cout << "Constructor " << endl; }
   ~Cls() { cout << "Destructor" << endl; }
};
void main()
{
   {shared_ptr<Cls> a(new Cls);    // Выводит: Constructor
   a->q = a;
   } // Выводит: Destructor
}
```

Основные методы класса weak_ptr:

- expired() – возвращает true, если объект, на который ссылается указатель, удален;
- lock() – возвращает объект типа shared_ptr, владеющий указанным ресурсом;
- reset() – разрывает связь с объектом;
- swap() – меняет местами указатели на объекты;
- use_count() – возвращает количество указателей shared_ptr, совместно использующих объект.

Для указателя weak_ptr не определена операция «->», поэтому для доступа к данным его необходимо конвертировать с помощью метода lock() в указатель shared_ptr:

```
class Cls {
public:
   void prn() { cout << "Available " << endl; }
};

void main()
{
```

```
    auto b = weak_ptr<Cls>();
   //  b->prn(); // Ошибка!
    b.lock()->prn();
}
```

Контрольные вопросы

1. Что такое шаблон функции?
2. Какие имеются типы параметров шаблонов?
3. Как описывается шаблон функции?
4. Что такое выведение аргументов шаблона?
5. Каков алгоритм выведения аргументов шаблона?
6. Какая функция называется специализацией шаблонной функции?
7. Укажите последовательность поиска функции для выполнения при использовании шаблонов.
8. Назовите особенности использования шаблонных классов.
9. Что такое функциональный оператор?
10. Как объявляется шаблонная функция с переменным числом параметров?
11. Для какой цели служат умные указатели?
12. Объясните работу уникального, совместно используемого и слабого указателей.

11. Стандартная библиотека шаблонов

Стандартная библиотека шаблонов STL (*Standard Template Library*) является частью стандарта языка C++, содержит средства для хранения и обработки данных. Для работы определена иерархия шаблонов классов и функций.

Основными сущностями STL являются:

- контейнеры (определяют структуру размещения данных);
- алгоритмы (функции обработки данных);
- итераторы (аналог указателей на данные);
- аллокаторы (распределители памяти);
- функциональные объекты (функторы);
- лямбда-функции.

11.1. Контейнеры

Контейнер – шаблон класса, предназначенный для хранения коллекций объектов одного типа. При объявлении указывается тип элементов, которые будут храниться в контейнере. Разрешено использовать списки инициализации. Контейнеры содержат методы для работы с элементами контейнера. Доступ к элементам контейнера осуществляется с помощью итераторов. Итераторы для всех контейнеров имеют общий интерфейс, однако каждый контейнер использует свои специализированные итераторы.

Контейнеры можно разделить на три основные категории: последовательные контейнеры, ассоциативные контейнеры и контейнеры-адаптеры.

Последовательные контейнеры. Данные расположены последовательно, каждый элемент данных имеет свою позицию и соседствует с другими элементами. Их можно вставлять в любое место контейнера.

array (#include <array>) – последовательность фиксированного размера. Каждый элемент имеет фиксированное положение относительно других элементов. Аналог обычного статического массива (размер массива задается на этапе разработки программы).

Описание:

```
template <class T, size_t N>
class array;
```

vector (#include <vector>) – структура, аналогичная динамическому массиву, автоматически размещаемая в памяти, имеющая возможность изменения размера, добавления и удаления данных. Поддерживает доступ к элементам с помощью оператора индексации.

Описание:

```
template<class T, class Allocator = allocator<T> >
class vector;
```

deque (#include <deque>) – двусторонняя очередь (*ouble-ended queue*). Динамическая структура, которая позволяет получать доступ к элементам с двух сторон контейнера. Данные размечаются не последовательно, а в виде связанных динамических элементов. Поддерживает доступ к элементам с помощью оператора индексации. Осуществляет эффективную вставку и удаление элементов в контейнер.

Описание:

```
template <class T, class Allocator =allocator<T> >
class deque
```

list (#include <list>) – двусвязный линейный список. Элементы упорядочены в строгой последовательности друг относительно друга без прямого доступа. Наличие ссылки на предыдущий элемент позволяет быстрее проводить операции перемещения элементов. Позволяет осуществлять эффективную вставку и удаление из начала и в конец контейнера. Произвольный доступ к элементам запрещен.

Описание:

```
template <class T, class Allocator = allocator<T>>
class list
```

forward_list (#include <forward_list>) – односвязный линейный список. В отличие от list имеет одну связь, что экономит ресурсы, но замедляет доступ к элементам.

Описание:

```
template <class T, class Allocator = allocator<T>>
class forward_list
string, wstring
```

Ассоциативные контейнеры. Данные размещаются в упорядоченном (отсортированном) виде (имеются также неупорядоченные ассоциативные контейнеры). Поиск ведется в дереве поиска по значению ключа. Главное преимущество таких контейнеров – скорость доступа.

set (#include <set>) – упорядоченное по возрастанию множество элементов. Реализовано на основе самобалансирующегося двоичного дерева поиска. Поиск ведется по ключу (ключом является значение элемента). Изменение значений элементов запрещено, добавление и извлечение – разрешено. Множество может хранить только уникальные (неповторяющиеся) элементы. Для хранения

повторяющихся элементов используется множество multiset. Имеются неупорядоченные версии unordered_set (#include <unordered_set>) и unordered_multiset (#include <unordered_multiset>), основанные на хеш-таблицах.

Описание:

```
template <class Key,
    class Traits = less<Key>,
    class Allocator = allocator<Key>>
class set
```

map (#include <map>) – упорядоченный ассоциативный контейнер, состоящий из пар «ключ – значение». Ключ используется для сортировки и индексации данных и должен быть уникальным. Контейнер multimap допускает наличие одинаковых ключей. Имеются неупорядоченные версии unordered_map (#include <unordered_map>) и unordered_multimap (#include <unordered_multimap>), основанные на хеш-таблицах.

Описание:

```
template <class Key,
    class Type,
    class Traits = less<Key>,
    class Allocator = allocator<pair <const Key, Type>>>
class map;
```

Контейнеры-адаптеры. Для хранения данных используются последовательные или ассоциативные контейнеры, но свой ограниченный набор методов. Контейнеры-адаптеры не поддерживают использование итераторов, поэтому их нельзя использовать с алгоритмами стандартной библиотеки.

stack (#include <stack>) – стек, контейнер, в котором добавление и удаление элементов осуществляется с одного конца.

Описание:

```
template <class Type, class Container= deque <Type>>
class stack
```

queue (#include <queue>) – очередь, контейнер, с одного конца которого можно добавлять элементы, а с другого – извлекать.

Описание:

```
template <class Type, class Container = deque <Type>>
class queue
```

priority_queue (#include <queue>) – очередь, в которой максимальный элемент всегда стоит на первом месте.

Описание:

```
template <class Type, class Container= vector <Type>,
                  class Compare= less <typename Container :: value_type>>
class priority_queue
```

В стандарте C++23 появились четыре новых ассоциативных контейнера: flat_map, flat_multimap, flat_set и flat_multiset, которые являются полноценной заменой соответствующих ассоциативных контейнеров. Отличаются более высокой производительностью и низким расходом памяти.

Методы работы с данными

Каждый контейнер имеет набор методов для работы с данными. Часть методов используется только в определенных контейнерах, другая часть одинакова для большинства контейнеров.

Общие для контейнеров методы:

- iterator begin() – возвращает итератор, указывающий на первый элемент последовательности;
- reverse_iterator rbegin() – возвращает обратный итератор, указывающий на последний элемент последовательности для обозначения начала обратной последовательности;
- iterator end() – возвращает итератор, указывающий на следующий элемент после конца последовательности;
- reverse_iterator rend() – возвращает обратный итератор, указывающий на элемент, стоящий перед первым элементом последовательности для обозначения конца обратной последовательности;
- size_type max_size() – возвращает длину самой длинной последовательности, которая может контролироваться объектом;
- size_type size() – возвращает количество элементов в контейнере;
- bool empty() – возвращает true, если в контейнере нет элементов;
- iterator erase(iterator first, iterator last) – удаляет элементы последовательности от первого элемента до стоящего перед последним (необязательный параметр). Возвращает итератор на элемент, стоящий после последнего удаленного элемента (иначе end);
- void clear() – удаляет все элементы последовательности (очищает контейнер);

Для большинства контейнеров также определены следующие методы:

- at(size_type *позиция*) – возвращает ссылку на элемент в указанной позиции;

- insert(*позиция*, *значение*) – добавляет один элемент (или несколько элементов, или диапазон элементов) в указанное место последовательности;
- push_front(*значение*) – добавляет элемент в начало последовательности;
- pop_front () – удаляет первый элемент последовательности;
- push_back(*значение*) – добавляет элемент в конец последовательности;
- pop_back() – удаляет последний элемент последовательности.

Пример 11.1. Работа с использованием последовательных контейнеров:

```
#include <iostream>
#include <array>
#include <vector>
#include <deque>
#include <list>
using namespace std;
void main()
{
	// Работа с array
	array <int, 4> mas = { 1, 2, 8 };
	mas[2] = 3;
	mas[3] = 4;
	// mas[4] = 5; // Ошибка!
	for (auto m : mas) cout << m << ' '; // Выводит: 1 2 3 4
	// Работа с вектором
	vector<double> vec = { 6.3, 9.25, 8.4, 7.2 };
	for (int i = 0; i < vec.size(); i++) cout << vec[i] << ' ';
                                        // Выводит:  6.3, 9.25, 8.4 7.2
	vec.pop_back();
	vec.push_back(100);
	// vec.push_front(9); // Ошибка!
	vec[0] = 1;
	// vec[20] = 1; // Ошибка времени выполнения!
	vec.insert(vec.begin() + 3, 55);
	vec.erase(vec.begin() + 1, vec.end() - 2);
	for (auto v : vec) cout << v << ' '; // Выводит:  1 55 100
	vec.clear();
	if (vec.empty()) cout << "Vector is empty";

	vector<vector<double> > vec2 = {{1, 2},{3, 4},{5, 6}};
		for (int i = 0; i < vec2.size(); i++) {
		for (int j = 0; j < vec2[i].size(); j++)
			cout << vec2[i][j] << " ";
```

```
            cout << endl;
        }
        // Выводит:  1 2
        //           3 4
        //           5 6

        vector<vector<int>> vec3 = { {1},{2, 3},{4, 5, 6} };
        for (vector<int>  row : vec3)
        {
            for (int v : row)
                cout << v << " ";
            cout << endl;
        }
        //      Выводит:    1
        //                  2 3
        //                  4 5 6
    //  Работа со списком
    list <int> lst = { 1, 4, 4 };
    lst.push_back(4);
    lst.push_back(8);
    lst.push_front(0);
    // lst[0] = 90;  // Ошибка! Оператор [] не определен!
    lst.unique();   // Удаление соседних повторяющихся элементов
    for (auto l : lst) cout << l << ' '; // Выводит: 0 1 4 8
    //  Работа с очередью
    deque <char> dq = { 'z', 'w', 'c' };
    dq.push_back('d'); // Добавление в конец очереди
    dq.push_front('a'); // Добавление в начало очереди
    dq[1] = 'b';
    dq.erase(dq.begin() + 2);
    for (auto d : dq) cout << d << ' '; // Выводит: a b c d
}
```

Пример 11.2. Работа с использованием ассоциативных контейнеров:

```
#include <iostream>
#include <set>
#include <map>
using namespace std;
void main() {
    set<int> st = { 8, 4, 1 };
    st.insert(5);
```

```
    st.insert(5);
    for (auto t : st) cout << t << ' '; // Выводит: 1 4 5 8
    map<int, string> mp = { {6, "Petrov"}, {8, "Ivanov"},  {10, "Sidorov"}};
    mp.try_emplace(7, "Volkov");
    mp.try_emplace(8, "Sokolov");
    cout << mp.at(8);  // Выводит: Ivanov
    for (auto p : mp) cout << p.first << " " << p.second << " / ";
    // Выводит: 6 Petrov / 7 Volkov / 8 Ivanov / 10 Sidorov /
}
```

Пример 11.3. Подсчет количества различных символов в строке:

```
string st = "one lesson";
    set<char> s;
        for (auto ch = st.begin(); ch != st.end(); ch++) s.insert(*ch);
    cout << "Total - " << st.size();  // Выводит: Total - 10
    cout << "Non-repeating - " << s.size();  // Выводит: Non-repeating - 6
```

11.2. Итераторы

Итератор – объект класса iterator, предназначенный для обеспечения единообразного способа обращения с элементами контейнеров независимо от их типа. Итератор позволяет перебирать элементы в контейнере и предоставлять доступ к ним. По отношению к контейнеру итераторы выполняют ту же роль, что и указатели по отношению к массиву. Итераторы не включаются в контейнер и полностью независимы от него.

Подключение итератора:

тип_контейнера <тип_элементов>::**iterator** имя_итератора;

Работа с итераторами похожа на работу с указателями. Допустимо использовать в зависимости от типа итератора следующие операторы:

- «*» – получение доступа (чтение, запись) к элементу;
- «++» – перемещение к следующему элементу контейнера;
- «- - » – перемещение к предыдущему элементу контейнера;
- «[ ]» – индексированный доступ к элементам;
- swap – обмен;
- «+», «+=», «-», «-=» – перемещение на указанное число элементов;
- «==», «!=», «>», «<», «>=», «<=» – сравнение операторов;
- «=» – задание новой позиции.

Для задания позиции все контейнеры содержат следующие методы:

- begin() – указывает положение первого элемента контейнера;
- end() – указывает положение после последнего элемента контейнера;

- cbegin() – возвращает константный итератор на первый элемент контейнера;
- cend() – возвращает константный итератор на элемент, следующий за последним элементом контейнера.

В библиотеке STL введено шесть основных категорий итераторов. Каждая категория имеет свой набор возможностей:

1. Итератор ввода (input). Определены операции разадресации (только для чтения содержимого элемента), инкремента, сравнения на равенство, обмена. Используется для структур данных istream.

2. Итератор вывода (output). Определены операции разадресации (только для записи содержимого элемента): записывается только один раз, после этого итератор увеличивает свое значение), инкремента, сравнения на равенство, обмена. Используется для структур данных ostream.

3. Однонаправленный итератор (forward). Определены операции чтения, записи, инкремента, сравнения на равенство, обмена.

4. Двунаправленный итератор (bidirectional). Определены операции чтения, записи, инкремента, декремента, сравнения на равенство, обмена. Используется при работе с list, set, multiset, map и multimap.

5. Итератор с произвольным доступом (random access). Определены операции чтения, записи, индексированного доступа «[ ]», инкремента, декремента, сложения, вычитания, сравнения, обмена. Используется при работе с vector, deque, string и array.

6. Непрерывный итератор (contiguous iterator). В отличие от итератора с произвольным доступом требует, чтобы элементы, смежные в контейнере, были смежными в памяти. Используется при работе с vector, string и array.

Алгоритмы могут использовать итераторы с уровнем не ниже указанного в описании.

Пример 11.4. Работа с итератором:

```
void main()
{
 const int n = 5;
 list <int> lst(n);
 list <int>::iterator iter = lst.begin();
 for(int i = 0; i < n; i++)  {
  *iter = i + 1;
     iter ++;
 }
  for(iter = lst.begin(); iter != lst.end(); iter++)
```

```
        cout << *iter << " "; // Выводит: 1 2 3 4 5
}
```

Для расширения возможностей итераторов используют ***вспомогательные итераторы*** (***адаптеры итераторов***), которые позволяют получить модификации обычного итератора. Некоторые адаптеры описаны ниже.

Обратный итератор (reverse_iterator) позволяет проходить контейнеру в обратном направлении.

Пример 11.5. Использование обратного итератора:

```
void main()
{
  const int n = 5;
 list <int> lst;
 for(int i = 0; i < n; i++) lst.push_back(i + 1);
 list <int> :: reverse_iterator rit;
   for( rit = lst.rbegin(); rit != lst.rend(); rit++)
        cout << *rit << " ";  // Выводит: 5 4 3 2 1
}
```

Потоковые итераторы (ostream_iterator и istream_iterator) позволяют интерпретировать файлы и потоки ввода-вывода как итераторы.

```
void main()
{
   vector <int> a(10);
   cout << "Enter integer values: " << endl;
    // Ввод данных
   // 1 параметр: потоковый итератор для чтения целых чисел из cin
   // 2 параметр: итератор конца потока (Ctrl+Z)
    copy(istream_iterator<int>(cin), istream_iterator<int>(), a.begin());
     cout << endl;
   //параметр: итератор вывода (в поток cout). Разделитель - пробел
   copy(a.begin(), a.end(), ostream_iterator<int>(cout, "  "));
}
```

Итераторы вставки (back_inserter, front_inserter, inserter) используются для добавления элементов в любое место контейнера, если соответствующие функции вставки не работают:

- back_inserter – создает итератор, позволяющий вставлять элементы с обратной стороны контейнера;
- front_inserter – создает итератор, позволяющий вставлять элементы с передней стороны контейнера;

- inserter – вставляет элементы в последовательность.

Пример 11.6. Использование итераторов вставки:

```
int main(void)
{
int mas0[] = {0, 0};
int mas1[] = {1, 1};
int mas2[] = {2, 2, 2};
int mas3[] = {3, 3};
 list<int> a(2);
   copy(mas0, mas0 + 2, a.begin()); // Выводит: 0 0
   copy(mas1, mas1 + 2, front_inserter(a)); // Выводит: 1 1 0 0
   copy(mas3, mas3 + 2, back_inserter(a)); // Выводит: 1 1 0 0 3 3
    list<int>::iterator it = a.begin();
        advance(it, 3); // Сдвиг итератора на три позиции
   copy(mas2, mas2 + 3, inserter(a, it));
       copy(a.begin(), a.end(), ostream_iterator<int>(cout, " "));
       // Выводит: 1 1 0 2 2 2 0 3 3
 return 0;
}
```

11.3. Алгоритмы

Алгоритмы – независимые шаблонные функции, производящие действия над контейнерами. Находятся в библиотеке *algorithm*. Алгоритмы используются для любых типов контейнеров. Каждый алгоритм представляет собой шаблон или набор шаблонов функций.

Все алгоритмы (всего имеется более 100 алгоритмов) можно разделить на три группы.

1. Немодифицирующие алгоритмы. Предназначены для извлечения информации из контейнера.

Некоторые часто встречающиеся алгоритмы:

- all_of(first, last, f) – возвращает true, если функция *f* возвращает значение true для всех элементов из диапазона [first, last), и в false противном случае;
- copy(first, last, d) – копирует элементы последовательности итераторов [first, last) в точку назначения *d*;
- count(first, last, n) – возвращает количество элементов из диапазона [first, last), соответствующих значению *n*;
- count_if(first, last, f) – возвращает количество элементов из диапазона [first, last), значения которых соответствуют заданному условию *f*.

Пример 11.7. Определение наличия в массиве отрицательных элементов, а также количества элементов, равных 3, и количества положительных элементов:

```
#include <list>
#include <algorithm>
#include <iostream>
using namespace std;
bool f(int x) {
    return !(x < 0);}
void main() {
   list<int> a{ 1, -2, 3, -5, 3 };
    if (all_of(a.begin(), a.end(), f))
      cout << "No negative items found";
     else  cout << "Negative items found";  // Выводит: Negative items found
    cout <<endl<< "The number of 3s: "
       <<count(a.begin(), a.end(), 3); //  Выводит: The number of 3s: 2
    cout << endl << "Number of positive elements: "
       << count_if(a.begin(), a.end(), f);
                                  // Выводит: Number of positive elements: 3
}
```

- for_each(first, last, f) – применяет функцию *f* к каждому элементу контейнера в диапазоне итераторов [first, last). Значения элементов контейнера не изменяются;
- find(first, last, value) – возвращает значение первого итератора из диапазона [first, last), для которого значение элемента контейнера равно value. Если такой итератор не найден, возвращается значение last.

Пример 11.8.

```
void f(int x) { cout << x + 10 << endl;}
void main() {
        vector<int> a{ 1, -5, 3, 9 };
        for_each(a.begin(), a.end(), f);  // Выводит: 11, 5, 13, 19
        vector<int>::iterator it;
        it = find(a.begin(), a.end(), 3);
        cout << "position " << it - a.begin();   //Выводит: position 2
}
```

2. Модифицирующие алгоритмы. Используются для модификации содержимого контейнера.

Некоторые часто встречающиеся алгоритмы:

- remove(first, last, value*)* – удаляет из диапазона итераторов [first, last), все элементы, для которых значение элемента контейнера равно value;

- unique(first, last*)* – удаляет все подряд повторяющиеся элементы в диапазоне итераторов [first, last). Алгоритм возвращает прямой итератор на новый конец последовательности.

3. Алгоритмы сортировки и поиска:

- sort(first, last) – сортирует элементы в промежутке итераторов [first, last) в порядке возрастания;
- count(first, last, value) – возвращает количество вхождений значения value в промежутке итераторов [first, last).
- lower_bound(first, last, value) – находит позицию первого элемента в промежутке итераторов [first, last), значение которого больше или равно value (используется двоичный поиск).

Пример 11.9.

```
vector <int> a = { 1, 8, 8, 0, 22, 7, 8, 1 };
        sort(a.begin(), a.end());
        copy(a.begin(), a.end(), ostream_iterator<int>(cout, " "));
                                                    // Выводит: 0 1 1 7 8 8 8 22
                int k = count(a.begin(), a.end(), 8);
                cout << endl << k << endl;  // Выводит: 3
                vector <int>::iterator it = unique(a.begin(), a.end());
    copy(a.begin(), it, ostream_iterator<int>(cout, " ")); // Выводит: 0 1 7 8 22
```

Для последовательного изменения содержимого контейнера используется алгоритм transform(). Результат помещается в исходный или новый контейнер. Для реализации изменений используется пользовательская функция, возвращающая результат, тип которого должен соответствовать типу элементов контейнера.

Пример 11.10. Создание списка студентов. Сортировка списка по фамилии и по оценке. Проверка наличия неуспевающих студентов в группе, если имеются, то после их фамилий добавляется слово *weak*.

```
struct st {
        string fio;
        int otc;
};

vector <st> mas = { { "Ivanov", 8 },{ "Aktanov", 2 },{ "Bistrov", 10 } };
        bool cmp1(st s1, st s2) { return s1.fio < s2.fio; }
        bool cmp2(st s1, st s2) { return s1.otc > s2.otc;}
        void outp(st s) { cout << s.fio << " " << s.otc << endl; }
        bool pp(st s) { return s.otc == 2; }
        st w(st s)      {
```

```
        if (s.otc < 4) s.fio += " - weak";
        return s;
}

void main()   {
        vector <st>::iterator it;
         sort(mas.begin(), mas.end(), cmp1);

         for (it = mas.begin(); it != mas.end(); it++)
                        cout << it->fio << it->otc << endl;
        // Выводит: Aktanov 2
        // Выводит: Bistrov 10
        // Выводит: Ivanov 8
        cout << endl;
        sort(mas.begin(), mas.end(), cmp2);
                for_each(mas.begin(), mas.end(), outp);
        // Выводит: Bistrov 10
        // Выводит: Ivanov 8
        // Выводит: Aktanov 2

        it = find_if(mas.begin(), mas.end(), pp);
        if (it == mas.end())
                cout << "Not weak student" << endl;
        else {
                cout << "There are weak students " << endl;

                transform(mas.begin(), mas.end(), mas.begin(), w);
                for_each(mas.begin(), mas.end(), outp);
                // Выводит: Bistrov 10
                // Выводит: Ivanov 8
                // Выводит: Aktanov - weak 2
                        }
}
```

Некоторые алгоритмы в качестве дополнительного параметра могут использовать функциональный объект (перегруженная операция «( )») или функцию. Например, sort в качестве третьего параметра может использовать функцию для сравнения.

Некоторые алгоритмы (например, find) имеют версии с окончанием _if (find_if), которые также содержат дополнительный параметр, являющийся функциональным объектом или функцией. В этом случае алгоритм применяет функцию или функциональный объект к каждому элементу контейнера.

11.4. Диапазоны

Диапазон (*ranges*) – специализированный класс, предназначенный для работы с итерируемыми последовательностями библиотеки STL.

При работе с алгоритмами библиотеки STL обычно необходимо указание на начало и конец используемой части коллекции, что является дополнительной работой в случае, если требуется использование всей коллекции.

Использование диапазонов (библиотека *ranges*) позволяет исключить передачу итераторов диапазона. Например, алгоритм сортировки

```
vector <int> a = {1, 5, 4, 0, 9, 3};
sort(a.begin(), a.end());
copy(a.begin(), a.end(), ostream_iterator<int>(cout, " "));
```

можно переписать следующим образом:

```
vector <int> a = {1, 5, 4, 0, 9, 3};
ranges::sort(a);
ranges::copy(a, ostream_iterator<int>(cout, " "));
```

Библиотека диапазонов является абстракцией, работающей поверх:

- итераторов всего диапазона [begin, end) (например, ranges::sort);
- итераторов заданного начала и конца диапазона, например:

```
vector <int> a = {1, 5, 4, 0, 9, 3};
for (int x : views::counted(a.begin() + 2, 4)) cout << x << ' ';
// Выводит: 4 0 9
```

- итераторов, диапазон перебора которых ограничивается заданным условием, например:

```
vector <int> a = {1, 5, 4, 0, 9, 3};
for (auto x : a | views::take_while([](int x) {return x > 0;})) cout << x << ' ';
// Выводит: 1 5 4
```

- итератора генерируемой последовательности элементов, например:

```
ranges::iota_view<int, int> s{3, 8};
 for (auto x : s)  cout << x << ' ';
 // Выводит: 3 4 5 6 7
  vector <int> a = {1, 5, 4, 0, 9, 3};
  auto begin = ranges::find(a, 5);
```

```
auto end = ranges::find(a, 9);
  ranges::iota_view r{begin, end};
for (auto x : r)  cout << *x << ' ';
// Выводит: 5 4 0
```

В классе определены методы доступа к диапазону.

В библиотеку *ranges* добавлены алгоритмы, которые предназначены для работы непосредственно с диапазонами (аналогичные обычным алгоритмам работы с контейнерами).

Пример 11.11. Программа из примера 11.9 с использованием *ranges*:

```
vector <int> a = {1, 8, 8, 0, 22, 7, 8, 1};
        ranges::sort(a);
        ranges::copy(a, ostream_iterator<int>(cout, " "));
        // Выводит: 0 1 1 7 8 8 8 22
        int k = ranges::count(a, 8);
        cout << endl << k << endl;  // Выводит: 3
        const auto [f, l] = ranges::unique(a);
        a.erase(f, l);
ranges::copy(a, ostream_iterator<int>(cout, " ")); // Выводит: 0 1 7 8 22
```

11.5. Аллокаторы (распределители памяти)

Аллокатор (*allocator*) – специализированный класс, предназначенный для выделения и освобождения памяти в библиотеке STL.

Все контейнеры библиотеки STL (кроме array) имеют параметр allocator<тип_контейнера>. Например, класс list объявляется следующим образом:

```
template <class Type, class Allocator = allocator<Type>>
class list
```

Реализованный в библиотеке STL распределитель по умолчанию (минимальный распределитель) достаточен для решения большинства задач. Распределитель по умолчанию использует методы **new** и **delete** для выделения и освобождения памяти.

Основные методы распределения памяти:

- pointer allocate(size_type n) – выделяет область памяти, достаточную для хранения *n* элементов;
- construct(pointer p, **const** type& t) – создает объект по адресу *p* и инициализирует его значением *t*;
- **void** destroy(pointer ptr) – вызывает деструктор объекта без освобождения памяти;

- **void** deallocate(pointer p, size_type n*)* – освобождает *n* объектов из памяти, начиная с позиции *p*.

Пример 11.12.

```
allocator<int> myAllocator;
int* a = myAllocator.allocate(3);
 myAllocator.construct(a, 1);
 myAllocator.construct(a + 1, 4);
 myAllocator.construct(a + 2, 3);
  for (int i = 0; i < 3; i++) cout << a[i];   // Выводит: 1 4 3
 myAllocator.destroy(a + 1);
 myAllocator.construct(a + 1, 2);
  for (int i = 0; i < 3; i++) cout << a[i];    // Выводит: 1 2 3
 myAllocator.destroy(a);
 myAllocator.destroy(a + 1);
 myAllocator.destroy(a + 2);
 myAllocator.deallocate(a, 3);
  vector <int> b = {1, 3, 5, 7};
  vector <int>::allocator_type ba;
                                  // Тип класса allocator для объекта вектор
  allocator<int>::pointer al;
   for (auto s : b) cout << s << " ";  // Выводит: 1 3 5 7
int x = 5, y = 100;
   al = ba.address(*find(b.begin(), b.end(), x));
 ba.destroy(al);
 ba.construct(al, y);
  for (auto s : b) cout << s << " ";  // Выводит: 1 3 100 7
```

11.6. Лямбда-выражения

Некоторые алгоритмы библиотеки STL обрабатывают элементы коллекции путем вызова функции. Проблемы использования функций при работе с алгоритмами библиотеки STL:

- передаваемые в функцию данные могут быть описаны в другой области видимости, поэтому требуется написание дополнительного кода для передачи этих данных;
- часто функция описывается для однократного использования.

Лямбда-выражение – способ определения анонимной функции непосредственно в месте вызова.

Лямбда-выражения имеют несколько вариантов синтаксиса, например:

[*список_захвата***] (***список_параметров***)**
спецификаторы
-> *тип_возвращаемого_значения*
{*тело_выражения*}

Список захвата определяет, к каким переменным из внешнего окружения получает доступ лямбда-выражение. Список может содержать следующие значения:

- [] – доступ к переменным из внешней области видимости запрещен;
- [=] – доступ ко всем внешним переменным осуществляется по значению;
- [&] – доступ ко всем внешним переменным осуществляется по ссылке.

Доступ к переменным с префиксом с амперсандом (&) осуществляется по ссылке, а к переменным без префикса – по значению. Элементы списка отделяются друг от друга запятыми.

Пример списков захвата:

- [x, &y] – *x* захвачено по значению, *y* – по ссылке;
- [&, x] – *x* захвачено по значению, остальные по ссылке;
- [=, ***this**] – захват переменных и объекта по значению.

Если указан общий доступ по значению или по ссылке, то запрещено давать такой же доступ отдельным элементам.

```
[&, &x]     // Ошибка!
[=, this]   // Ошибка!
```

Глобальные переменные лямбда-выражение может использовать без указания в списке захвата.

Список параметров аналогичен списку параметров функции и является необязательным (пустые скобки можно не указывать).

Спецификаторы (необязательные параметры). Допускается использование следующих спецификаторов:

- mutable – разрешает изменение захваченных по значению объектов (по умолчанию они являются константными в теле выражения);
- noexcept – указывает на то, что лямбда-выражение не создает исключений;
- constexpr – определение константного лямбда-выражения, значение которого будет вычислено на этапе компиляции.

Типом возвращаемого значения может быть любой допустимый тип (разрешено использовать тип auto). Если тип возвращаемого результата явно не

указывается, то он выводится автоматически по типу выражения в операторе return (если он один), в противном случае возвращаемый тип – void.

Тело выражения описывается аналогично описанию тела функции. Операторы функции имеют доступ к данным аналогично функциям и дополнительно доступ, полученный согласно списку захвата.

Пример 11.13. Использование лямбда-выражений:

```
double x = 7, y = 3;
    double s = 0;

    [=, &s]() {return s = x / y;}();
    cout << s << endl;   // Выводит: 2.33333

    [&](double x, double y) {return s = x / y;}(15, 3);
    cout << s << endl;   // Выводит: 5

    auto w = [=, &s](double x) {return x / y;};
    cout << w(9) << endl; // Выводит: 3
    cout << w(14) << endl; // Выводит: 4.66667

int k = 0;
    auto inck([&k] { ++k;});
    inck(); cout << k; // Выводит: 1
    inck(); cout << k; // Выводит: 2
    inck(); cout << k; // Выводит: 3

    auto incn = [n = 0]() mutable { return ++n;};
    cout << incn(); // Выводит: 1
    cout << incn(); // Выводит: 2
    cout << incn(); // Выводит: 3

    auto plus([](auto a, auto b) { return a + b;});
    auto pl([=](int x) { return plus(x, 3);});
    cout << pl(6);  // Выводит: 9
```

Пример 11.14. Переписанная программа из примера 11.10 с использованием лямбда-выражений:

```
struct st {
    string fio;
    int otc;
};
```

```
vector <st> mas = {{"Ivanov", 8 },{ "Aktanov", 2 },{ "Bistrov", 10}};

int main() {
        vector <st>::iterator it;
        sort(mas.begin(), mas.end(), [](st s1, st s2) {return s1.fio < s2.fio;});

        for (it = mas.begin(); it != mas.end(); it++)
                cout << it->fio << it->otc << endl;
        cout << endl;
        sort(mas.begin(), mas.end(), [](st s1, st s2) {return s1.otc > s2.otc;});
        auto prn = [](st s) { cout << s.fio << " " << s.otc << endl; };
        for_each(mas.begin(), mas.end(), prn);

        it = find_if(mas.begin(), mas.end(), [](st s) {return s.otc == 2;});
        if (it == mas.end())
                cout << "Not weak student" << endl;
        else {
                cout << "There are weak students " << endl;
                transform(mas.begin(), mas.end(), mas.begin(), [](st s)
                        {if (s.otc < 4) s.fio += " - weak";  return s;});
                for_each(mas.begin(), mas.end(), prn);
        }
        return 0;
}
```

11.7. Базовые исключения стандартной библиотеки

Для обработки ошибок стандартной библиотеки определен базовый класс exception и несколько производных классов. От класса exception наследуются все исключения, генерируемые стандартной библиотекой. Метод what возвращает описание исключения.

Класс elogic_error предназначен для обработки ошибок, вызванных неправильной логикой программы, которые можно предотвратить.

Класс runtime_error предназначен для обработки исключений, вызванных ошибками, которые можно обнаружить только при выполнении программы.

Пример 11.15. Контроль ошибок выхода за пределы диапазона и превышения размера объекта:

```
void main() {
   try {
     vector<int> mas(5);
```

```
        mas.at(10) = 9;      // Выход за пределы диапазона
    }
    catch (const out_of_range& err) {
        cerr << "Error: " << err.what() << '\n';
    }
    catch (const length_error& err){
        cerr << "Error: " << err.what() << endl;
    }
}
```

Контрольные вопросы

1. Назовите основные сущности стандартной библиотеки шаблонов.
2. Какие типы контейнеров определены?
3. Как размещаются данные в последовательных, ассоциативных контейнерах и контейнерах-адаптерах?
4. Для чего используются итераторы?
5. Какие категории итераторов используются в стандартной библиотеке шаблонов?
6. Как работают обратный итератор, итератор вставки и потоковый итератор?
7. Что такое алгоритм в стандартной библиотеке шаблонов?
8. В чем различие модифицирующих и немодифицирующих алгоритмов в стандартной библиотеке шаблонов?
9. Что такое аллокатор?
10. Объясните синтаксис лямбда-выражения.

Список использованных источников

1. Лафоре, Р. Объектно-ориентированное программирование в C++. Классика Computer Science / Р. Лафоре. – 4-е изд. – СПб. : Питер, 2021. – 928 с.

2. Павловская, Т. А. C/C++. Структурное и объектно-ориентированное программирование : практикум / Т. А. Павловская, Ю. А. Щупак. – СПб. : Питер, 2021. – 325 с.

3. Павловская, Т. А. C/C++. Программирование на языке высокого уровня / Т. А. Павловская. – СПб. : Питер, 2021. – 464 с.

4. Страуструп, Б. Язык программирования C++ / Б. Страуструп ; пер. с англ. – М. : Бином, 2011. – 1136 с.

5. Шилдт, Г. Полный справочник по C++ / Г. Шилдт ; пер. с англ. – 4-е изд. – М. : Вильямс, 2006. – 800 с.

6. Лаптев, В. В. C++. Объектно-ориентированное программирование : учеб. пособие / В. В. Лаптев. – СПб. : Питер, 2008. – 464 с.

7. Мэтт, В. Объектно-ориентированный подход / В. Мэтт. – 5-е изд. – СПб. : Питер, 2020. – 256 с

8. Прата, С. Язык программирования C++. Лекции и упражнения / С. Прата ; пер. с англ. – 6-е изд. – М. : Вильямс, 2012. – 1248 с.

9. Галовиц, Я. C++17 STL. Стандартная библиотека шаблонов / Я. Галовиц. – СПб. : Питер, 2018. – 432 с.

10. Липпман, С. Язык программирования C++. Базовый курс / С. Липпман, Б. Му, Ж. Лажойе. – 5-е изд. – М. : Вильямс, 2008. – 1120 с.

11. Саттер, Г. Новые сложные задачи на C / Г. Саттер ; пер. с англ. – М. : Вильямс, 2008. – 272 с.

12. Нобак, М. Объекты. Стильное ООП / М. Нобак. – СПб. : Питер, 2023. – 304 с.

13. Мартин, Р. Чистый код: создание, анализ и рефакторинг. Библиотека программиста / Р. Мартин. – СПб. : Питер, 2013. – 464 с.

14. Вайсфельд, М. Объектно-ориентированное мышление / М. Вайсфельд. – СПб. : Питер, 2014. – 304 с.

Св. план 2024, поз. 3

Учебное издание

Навроцкий Анатолий Александрович

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

УЧЕБНОЕ ПОСОБИЕ

Редактор *А. Ю. Шурко*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *Е. Г. Бабичева*

Подписано в печать 08.10.2025. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 10,35. Уч.-изд. л. 9,5. Тираж 100 экз. Заказ 27.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
Ул. П. Бровки, 6, 220013, г. Минск