Московский государственный университет им. М.В.Ломоносова
Факультет Вычислительной Математики и Кибернетики

И.А.Волкова, А.В.Иванов, Л.Е.Карпов

Основы объектно-ориентированного программирования. Язык программирования C++.

Учебное пособие для студентов 2 курса

Москва

2011

УДК
ББК

Печатается по решению Редакционно-издательского совета факультета вычислительной математики и кибернетики МГУ им. М. В. Ломоносова

Рецензенты:
проф., д.ф.-м.н. Машечкин И. В.
доцент, к.ф.-м.н. Терехин А. Н.

Волкова И. А., Иванов А. В., Карпов Л. Е.
Основы объектно-ориентированного программирования. Язык программирования С++. Учебное пособие для студентов 2 курса. – М.: Издательский отдел факультета ВМК МГУ (лицензия ИД № 05899 от 24.09.2001), 2011 – 112 с.
ISBN-13
978-5-89407-439-9

Настоящее учебное пособие является дополнением к ранее выпущенным пособиям по курсу «Системы программирования»: Волкова И.А., Вылиток А.А., Руденко Т.В. «Формальные грамматики и языки. Элементы теории трансляции» и Волкова И.А., Головин И.Г., Карпов Л.Е. «Системы программирования». Пособие представляет собой конспект соответствующих лекций.

В учебном пособии кратко рассматривается объектно-ориентированный подход к программированию на примере языка программирования С++.

Учебное пособие предназначено для студентов второго курса факультета ВМК МГУ им. М.В.Ломоносова, а также может быть рекомендовано студентам отделения второго высшего образования при изучении курса «Языки программирования», читаемого по аналогичной программе.

УДК
ББК

ISBN-13
978-5-89407-439-9

© Издательский отдел факультета вычислительной математики и кибернетики МГУ им. М. В. Ломоносова, 2011
© Волкова И.А., Иванов А.В., Карпов Л.Е., 2011

Оглавление

1. Объектно-ориентированное программирование (ООП) ... 5
   1.1. Краткий обзор основных парадигм программирования ... 5
   1.2. Основные принципы ООП ... 7
   1.3. Абстрактные типы данных ... 9
2. Отличия C и C++ ... 9
   2.1. Работа с динамической памятью ... 9
   2.2. Описания, значения параметров по умолчанию ... 10
   2.3. Тип *bool* ... 10
   2.4. Ссылки ... 11
   2.5. Стандартная библиотека C++, стандартный ввод-вывод ... 12
   2.6. Пространства имен, пространство имен *std. Операция* `'::'` ... 14
3. Классы и объекты C++ ... 16
   3.1. Синтаксис описания класса ... 16
   3.2. Управление доступом к членам класса ... 17
   3.3. Классы и структуры C++ ... 18
   3.4. Манипуляции с состоянием объекта ... 19
   3.5. Класс как область видимости ... 19
   3.6. Объявление и определение методов класса. Спецификатор *inline* ... 20
   3.7. Указатель `this` ... 21
   3.8. Указатель на член класса ... 21
4. Конструкторы и деструкторы ... 23
   4.1. Конструктор умолчания ... 25
   4.2. Конструктор преобразования и конструкторы с двумя и более параметрами ... 25
   4.3. Конструктор копирования ... 26
   4.4. Спецификатор explicit ... 29
   4.5. Конструктор копирования и операция присваивания ... 29
   4.6. Автоматическая генерация конструкторов и деструкторов ... 30
   4.7. Список инициализации ... 30
   4.8. Порядок вызова конструкторов и деструкторов ... 32
5. Статические члены класса ... 35
6. Константные члены класса. Модификатор const ... 37
7. Друзья классов ... 39
8. Статический полиморфизм ... 41
   8.1. Перегрузка бинарных операций ... 41
   8.2. Перегрузка унарных операций ... 45
   8.3. Перегрузка функций ... 47
   8.4. Алгоритм поиска оптимально отождествляемой функции для одного параметра ... 49
9. Виды отношений между классами ... 56
10. Одиночное наследование ... 59
    10.1. Правила наследования ... 59
    10.2. Преобразования указателей ... 60
    10.3. Правила видимости при наследовании. ... 61
    10.4. Закрытое (private) наследование ... 64
    10.5. Перекрытие имен ... 66
    10.6. Наследование и повторное использование кода ... 68
11. Динамический полиморфизм, механизм виртуальных функций ... 72
    11.1. Виртуальные деструкторы ... 73

11.2. Реализация виртуальных функций ... 75
11.3. Абстрактные классы. Чистые виртуальные функции ... 77
12. Средства обработки ошибок, исключения и обработка исключений ... 77
12.1. Правила выбора обработчика исключения ... 78
12.2. Стандартные исключения. ... 80
12.3. Последовательность действий при возникновении исключительной ситуации ... 80
13. Множественное наследование, интерфейсы ... 81
13.1. Видимость при множественном наследовании. ... 81
13.2. Виртуальные базовые классы ... 82
13.3. Интерфейсы ... 83
14. Динамическая информация о типе (RTTI). ... 84
15. Параметрический полиморфизм ... 89
15.1. Параметры шаблона ... 89
15.2. Шаблоны функций. ... 91
15.3. Специализация шаблонной функции ... 92
15.4. Алгоритм поиска оптимально отождествляемой функции (с учетом шаблонов) ... 93
15.5. Шаблонные классы ... 94
15.6. Эквивалентность типов ... 96
16. Стандартная Библиотека шаблонов STL ... 97
16.1. Контейнеры ... 97
16.2. Распределители памяти ... 98
16.3. Итераторы ... 99
16.4. Алгоритмы ... 102
16.5. Достоинства и недостатки STL-подхода ... 103
16.6. Контейнер вектор ... 104
16.7. Контейнер список ... 108
16.8. Пример решения задачи с использованием контейнеров STL ... 110
17. Литература ... 112

1. Объектно-ориентированное программирование (ООП)

Объектно-ориентированная технология (**парадигма**) программирования наиболее распространена и востребована в настоящее время. При объектно-ориентированном подходе к программированию программа представляет собой совокупность взаимодействующих между собой данных – объектов. Функциональную возможность и структуру объектов задают классы – типы данных, определенные пользователем.

Изучение ООП целесообразно начать на примере объектно-ориентированнного языка программирования C++ ([9], [14]), как наиболее теоретически выдержанного в этой части. Другие языки, поддерживающие идеи ООП, такие, как Object Pascal [4], Java [7], разрабатывались в первую очередь с учетом удобства программирования задач в соответствующих предметных областях.

Язык C++, унаследовав язык C (стандарт C90 – см. [1]), был разработан его автором Б. Страуструпом [12] с наиболее возможной реализацией теоретических концепций ООП. Язык C++ вобрал в себя не только некоторые концепции языка C, а также и некоторых других языков. Например, концепция классов взята автором C++ из языка Simula [10], а концепция наследования – из языка Smalltalk.

Следует отметить, что развитие языка C++, а также сопряженное с ним развитие технологии программирования, в частности, развитие COM (Component Object Model) технологии [5] приводит к некоторому отступлению от строгой теории ООП. Так, в COM-технологии требуется наличие базового надкласса, что отсутствует в теории ООП.

Некоторые отступления также имеются в визуальных реализациях C++: Visual C++ [6] в рамках пакета Microsoft Visual Studio, C++ Builder [3] фирмы Borland.

В данном учебном пособии C++ рассматривается в строго теоретическом аспекте.

1.1. Краткий обзор основных парадигм программирования

В соответствии с концепцией *фон-Неймана* – основателя теоретической концепции компьютерной техники, процессор обрабатывает данные, выполняя инструкции (команды), которые находятся в той же оперативной памяти, что и данные.

Таким образом, можно выделить две основные сущности процесса обработки информации: **код**, как совокупность инструкций, и **данные**. Все программы в соответствии с выбранной технологией программирования концептуально организованы вокруг своего кода или вокруг своих данных.

Рассмотрим основные на сегодняшний день парадигмы программирования:

1) **Процессно-ориентированная парадигма**, при которой программа представляет собой ряд последовательно выполняемых операций – модель фон-Неймана. При этом **код воздействует на данные**. Языки, реализующие эту парадигму, называются **процедурными** или **императивными**. Такими языками являются, например, C, Pascal и др.

2) **Объектно-ориентированная парадигма**, при которой программа рассматривается как совокупность фрагментов кода, обрабатывающих отдельные совокупности данных – **объекты**. Эти объекты взаимодействуют друг с другом посредством так называемых **интерфейсов**. При этом данные управляют доступом к коду.

При повышении сложности алгоритма процессно-ориентированная парадигма сталкивается с существенными проблемами. Переход к объектным принципам программирования позволяет значительно улучшить внутреннюю организацию программы, в результате чего повышается производительность при разработке программных комплексов.

Наряду с двумя вышеизложенными основными в настоящее время парадигмами программирования используются еще две парадигмы:

3) **Аппликативная** или **функциональная** парадигма. Основная идея данного подхода заключается в формализованном определении функции, которую выполняет программа. Таким образом, вместо определения последовательности состояний, через которые должен пройти компьютер, чтобы получить требуемый результат, необходимо определить функцию, при применении которой к исходным данным получается требуемое решение:

$$\overline{y} = f(\overline{x})$$

Разработка программы при этом подходе сводится к конструированию сложной функции из имеющихся стандартных простых функций:

$$\overline{y} = f_1(f_2(f_3(...), f_4(...),...))$$

Языками, поддерживающими такую парадигму, являются, например, языки LISP и ML. Данные при таком подходе, так же, как и код, представляются списками одинаковой структуры, значит, программа, работая под управлением интерпретатора, может обрабатывать свой собственный код, как данные. В этом случае стирается грань между кодом и данными. Поэтому одной из важных областей применения данной парадигмы являются системы **искусственного интеллекта (ИИ)**.

Примечание. Обрабатывать коды, как обрабатывают данные, можно и при использовании процессно-ориентированного подхода, однако, при этом программирование должно производиться в среде низкого уровня – на языке ассемблера.

4) Парадигма, основанная на использовании **системы правил** (**парадигма логического программирования)**. При этом подходе операторы программы выполняются не в той последовательности, в которой они написаны, а на основе анализа **разрешающих условий (РУ)**.

Программа при такой парадигме состоит из списка пар:

$$PУ_1 \to D_1$$
$$PУ_2 \to D_2$$
$$\ldots\ldots\ldots\ldots$$
$$PУ_N \to D_N$$

Здесь $D_1, D_2, ..., D_N$ – действия, выполняемые в случае истинности соответствующих разрешающих условий $PУ_1, PУ_2, ..., PУ_N$.

Выполнение программы заключается в циклической проверке разрешающих условий и выполнения действий, соответствующих разрешающим условиям, в случае истинности последних.

Примером языка логического программирования является язык PROLOG.

Структура программы при логическом программировании концептуально связана с теоретической концепцией **нормальных алгоритмов Маркова**, представляющей алгоритм преобразования информации в виде совокупности подстановок:

$$T_{11} \to T_{12}$$
$$T_{21} \to T_{22}$$
$$\ldots\ldots\ldots$$
$$T_{N1} \to T_{N2}$$

Операторы с разрешающими условиями и подстановки просматриваются циклически до обнаружения завершающего условия.

В данном курсе рассматривается парадигма ООП.

1.2. Основные принципы ООП

Центральной идеей ООП является реализация понятия "**абстракция**". Смысл абстракции заключается в том, что *сущность* произвольной сложности можно рассматривать, а также производить определенные действия над ней, как над **единым целым**, не вдаваясь в детали внутреннего построения и функционирования.

При создании программного комплекса необходимо разработать определенные абстракции.

Пример: Задача составления расписания занятий.

Необходимые **абстракции**: студент, курс лекций, преподаватель, аудитория.
Операции:

- Определить студента в группу
- Назначить аудиторию для группы
- . . . .

Одним из основных способов создания абстракции является использование концепции **иерархической классификации**. Ее суть заключается в том, что сложные системы разбиваются на более простые фрагменты.

Практически все сложные системы иерархичны, и уровни их иерархии отражают различные уровни абстракции. Для каждой конкретной задачи рассмат-

ривается соответствующий уровень. Выбор низшего уровня абстракции достаточно произволен. Выбранный уровень в одном случае в качестве низшего уровня может оказаться уровнем достаточно высокой абстракции в другом проекте.

Различают **типовую** иерархию и **структурную** иерархию, которые далее мы будем называть соответственно **структурой классов** и **структурой объектов**.

Во всех объектно-ориентированных языках программирования реализованы следующие **основные механизмы (постулаты) ООП**:

- **Инкапсуляция**
- **Наследование**
- **Полиморфизм**

Все эти механизмы важны для разработки и использования абстракций.

1) Инкапсуляция – механизм, связывающий вместе код и данные, которыми он манипулирует, и одновременно защищающий их от произвольного доступа со стороны другого кода, внешнего по отношению к рассматриваемому. Доступ к коду и данным жестко контролируется интерфейсом.

Основой инкапсуляции при ООП является **класс**.

Механизма инкапсуляции позволяет оставлять скрытыми от пользователя некоторые детали реализации класса (то есть инкапсулировать их в классе), что упрощает работу с объектами этого класса.

fork - exec

2) Наследование – механизм, с помощью которого один объект (производного класса) приобретает свойства другого объекта (родительского, базового класса). При использовании наследования новый объект не обязательно описывать, начиная с нуля, что существенно упрощает работу программиста. Наследование позволяет какому-либо объекту наследовать от своего родителя общие атрибуты, а для себя определять только те характеристики, которые делают его уникальным внутри класса.

Наследование есть очень важное понятие, поддерживающее концепцию иерархической классификации.

3) Полиморфизм – механизм, позволяющий использовать один и тот же интерфейс для общего класса действий.

Пример: Имеются 3 типа стека для хранения:

- целых чисел
- чисел с плавающей точкой
- символов

Вместо трех подпрограмм управления в объектно-ориентированной программе требуется всего одна подпрограмма (один интерфейс)

Общая концепция полиморфизма: **один интерфейс – много методов**.

Выбор конкретного действия (метода) применительно к конкретной ситуации возлагается на компилятор. Программисту же достаточно запомнить и применить один интерфейс, вместо нескольких, что также упрощает работу.

Различаются **статический** (реализуется на этапе компиляции с помощью перегрузки функций и операций), **динамический** (реализуется во время выполнения программы с помощью механизма виртуальных функций) и **параметрический** (реализуется на этапе компиляции с использованием механизма шаблонов) полиморфизм.

Примечание. Рассмотренные понятия абстракции, инкапсуляции, наследования, полиморфизма присущи не только парадигме ООП. Так, выполнение арифметических операций над целыми числами и числами с плавающей точкой осуществляются в процессоре по разным алгоритмам. Однако в данном случае полиморфизм проявляется неявно.

1.3. Абстрактные типы данных

Типы данных, создаваемые пользователем (программистом), называются **пользовательскими типами данных**. Пользовательский тип данных с полностью скрытой (инкапсулированной) внутренней структурой называется **абстрактным типом данных (АТД).**

В С++ АТД реализуется с помощью классов, в которых нет открытых членов-данных, то есть **вся** структура этих классов скрыта от внешнего пользователя.

2. Отличия С и С++

Объектно-ориентированный язык программирования С++ унаследовал типы данных, операции и управляющие конструкции процедурного языка С (стандарт С90).

Кроме реализации принципов объектно-ориентированного программирования в язык С++ по сравнению с С внесены и некоторые другие изменения, которые сделали его лучше и удобнее, чем С. Рассмотрим основные их них.

2.1. Работа с динамической памятью

В языке С++ для работы с динамической памятью введены операции ***new*** и ***delete***, которыми можно пользоваться наряду с функцией стандартной библиотеки С ***malloc***.

Операция ***new*** используется как для выделения памяти для одного объекта (при этом возможна инициализация выделенной памяти передаваемым значением), так и для массива однородных объектов. Операция ***new*** возвращает адрес начала выделенной динамической памяти соответствующего типа.

Ее синтаксис: **new** *тип*;
new *тип* (*выражение-инициализатор*);
new *тип* [*выражение_размерность_массива*];

Пример:

```
int * p;
int * q;
p = new int(5);  // выделение памяти и инициализация
                 // значением 5
q = new int[10]; // выделение памяти для массива из 10
                 // элементов
```

Операция **delete** освобождает распределенную операцией ***new*** память. Ее синтаксис:

delete *указатель_на_объект*;
delete [ ] *указатель_на_массив_объектов*;

Первая форма используется, если операцией ***new*** размещался единичный (скалярный) объект. Векторная форма используется**,** если операцией ***new*** создан массив объектов, при удалении которого для каждого из объектов необходим вызов деструктора (деструкторы описываются далее). Такими объектами являются объекты пользовательского типа.

2.2. Описания, значения параметров по умолчанию

В функциях, написанных на языке С++, в отличие от функций на С, описания локальных объектов и операторы располагаются в произвольном, удобном для программиста порядке. Остается лишь единственное ограничение: любое вводимое программистом имя должно быть описано до его первого использования.

В частности, в С++ можно описать переменную, используемую в первом выражении заголовка цикла ***for*** непосредственно в этом выражении:

```
for ( int i = 0, i < 10, i++ ) {…}
```

В С++ разрешено задавать априорные значения формальных параметров функции. Для этого после описания формального параметра в заголовке функции ставится знак = и соответствующее значение параметра по умолчанию.

Если в заголовке функции есть параметры с априорными значениями и без них, то все параметры, значения которых необходимо задавать явно при обращении к функции, располагаются в начале списка формальных параметров, а затем описываются формальные параметры с априорными значениями.

Пример:

```
void f ( int a, int b, int c = 1, int d = 2, int e = 3 ){…}
```

При этом обратиться к функции f можно как к функции с двумя, тремя, четырьмя или пятью параметрами:

```
f(5, 5);             // a = 5, b = 5, c = 1, d = 2, e = 3
f(1, 2, 3);          // a = 1, b = 2, c = 3, d = 2, e = 3
f(1, 1, 1, 1);       // a = 1, b = 1, c = 1, d = 1, e = 3
f(7, 7, 7, 7, 7);    // a = 7, b = 7, c = 7, d = 7, e = 7
```

В С++ отменены правила определения типов по умолчанию. Например, тип возвращаемого функцией `main` результата надо указывать явно: `int main () {…}`.

2.3. Тип *bool*

В С++ введен логический тип данных ***bool*** и введены в употребление логические константы ***true*** и ***false***.

2.4. Ссылки

В C++ введен ссылочный тип данных.

Описание ссылки вводит идентификатор, который будет *псевдонимом* (*alias*) для объекта, указанного при инициализации. Ссылка с точки зрения реализации является адресом объекта. Отличие ссылочного типа от типа «указатель» заключается во множестве операций, применимых для этого типа. Ссылке сопоставляется некоторый адрес единственный раз при ее инициализации. Проинициализированная ссылка в дальнейшем не может быть изменена. Так же как и указатель, ссылка является типизированной. Синтаксис определения ссылки:

тип& *идентификатор* = *инициализатор*;

```
int a;
int & b = a;
```

Инициализация ссылки, как обычно, записывается с помощью знака операции присваивания. После инициализации идентификатор ссылки используется так же, как и переменная–инициализатор ссылки. Таким образом, дальнейшее присваивание значений переменной-ссылке приведет к изменению не ее значения, а значения переменной-инициализатора. Ссылка, по сути, является синонимом имени переменной-инициализатора.

Пример:

```
int a = 5;
int & aa = a;           // ссылка обязана иметь начальное
                        // значение!
                        // a и aa ссылаются на одно и то же
                        // целое число
a = aa + 1;
cout << a << aa;       // будет выведено "6 6", поскольку
                        // a и aa – одна и та же сущность
if (& aa == & a) { . . . }; // адрес ссылки равен адресу
                        // самого объекта!
                        // можно инициировать ссылку
                        // константным значением, если
                        // сама ссылка тоже константная:
const int & aa1 = 1;   // ссылка на константу 1
double b[10];
double & bb = b[9];    // псевдоним для b[9] – последнего
                       // элемента массива b
```

Ссылка может быть определена и для динамического объекта:

```
int & x = * new int;
```

В этом случае, несмотря на то, что, как было отмечено выше, ссылка с точки зрения реализации является адресом объекта, формат операции ***delete*** будет следующий:

```
delete & x;
```

При описании формального параметра функции с использованием ссылки этот параметр передается **по ссылке.**

При передаче параметра **по ссылке** в функцию передается его адрес, и все изменения, происходящие с формальным параметром, происходят и с соответствующим фактическим параметром.

При передаче параметра **по значению** создается локальная копия объекта-параметра, которая инициализируется значением соответствующего фактического параметра, при этом соответствующий фактический параметр не может изменить своего значения в теле функции.

Ссылка инициализируется при передаче параметров в функцию и при передаче возвращаемого значения в виде псевдонима объекта, который продолжает существовать после выхода из функции (например, ссылка на текущий объект при выходе из метода).

Примечание. Текущий объект может быть возвращен при выходе из метода не только в виде ссылки на текущий объект, но и в виде объекта. Однако, это менее эффективно, так как в этом случае создается копия текущего объекта.

Пример:

```
int & imax(int * m){
      int i;
      …
      return m[i];
}
int main () {
      int A[10];
      …
      imax(A) = 0;
      …
}
```

В данном примере в результате вызова функции *imax()* будет возвращено значение ссылки на максимальный элемент массива А. По данной ссылке этому элементу вектора может быть присвоено новое значение.

2.5. Стандартная библиотека С++, стандартный ввод-вывод

Для языка С++ создана своя стандартная библиотека, которая отличается от стандартной библиотеки С. Изменения затронули, например, функции ввода-вывода, введен раздел, посвященный шаблонам (STL – стандартная библиотека шаблонов, описанная ниже).

Тем не менее, стандартная библиотека С целиком включена в стандартную библиотеку С++.

В C++ введен новый формат задания заголовочных файлов стандартной библиотеки: без `.h.`, при этом все используемые в библиотеке имена погружены в стандартное пространство имен std (см. раздел 2.6).

Например, в C++ файл заголовков ввода-вывода называется `iostream`, и для его подключения необходимо использовать директиву

```
#include <iostream>
```

Чтобы подключить к программе на C++ заголовочный файл стандартной библиотеки C в новом формате, надо добавить в начало названия соответствующего файла букву `c`. Например,

```
#include <cstdio>
```

В файле стандартной библиотеки C++ `<iostream>` введены классы, соответствующие стандартным (консольным) потокам ввода (класс **`istream`**) и вывода (класс **`ostream`**), а также объекты этих классов: **`cin`** – объект класса `istream`, **`cout`**, **`cerr`** – объекты класса `ostream`. Через эти объекты для базовых типов данных доступны операции ввода `>>` из стандартного потока ввода, например,

```
cin >> x;
```

и вывода `<<` в стандартный поток вывода (`cout`) или стандартный файл ошибок (`cerr`), например,

```
cout << "String" << S << endl;
```

При этом явно не указываются никакие форматирующие элементы.

`endl` – константа, обозначающая конец строки, она так же определена в файле `<iostream>`.

Стандартная библиотека C++:

1) обеспечивает поддержку свойств языка, например, управление памятью (функции, основанные на использовании операций ***`new`*** и ***`delete`***)
2) предоставляет информацию о типах во время выполнения программы (RTTI)
3) обеспечивает поддержку обработки исключений (стандартные исключения)
4) предоставляет информацию о зависящих от реализации аспектах языка (например, максимальное значение ***`float`***)
5) предоставляет программисту общеупотребительные математические и некоторые другие функции (например, *`sqrt()`*, генератор случайных чисел и т. д.)
6) предоставляет программисту некоторые нетривиальные и машинно-зависимые средства, что позволяет писать переносимые программы (например, списки, функции сортировки, потоки ввода/вывода).

С подробным описанием стандартной библиотеки C++ можно познакомиться в [12], часть III «Стандартная библиотека».

2.6. Пространства имен, пространство имен *std*. *Операция* '::'

В С++ так же, как и в языке программирования С, используется обычный механизм разрешения видимости имен по принципу локализации.

Кроме того, в С++ введено понятие **пространства имен**. Пространство имен является **областью видимости** и позволяет локализовать имена, описанные внутри этого пространства, что бывает полезно, например, для устранения коллизий имен, которые могут возникнуть при написании одного программного продукта разными программистами.

Пространства имен задаются следующим образом:

```
namespace имя_пространства_имен {
    . . . описания . . .
}
```

Пример:

```
namespace A {
    int x;
    void f ();
      …
}
```

В С++ также введена **операция разрешения области видимости имен** '`::`', которая расширяет имя, определяя из какого пространства имен (или области видимости) данное имя выбирается.

Если используется имя из текущей области видимости или имя из объемлющей области видимости, не совпадающее ни с одним именем из текущей области , оно записывается обычным образом, если же нужно совпадающее имя из объемлющей области или имя из другой области видимости, это имя можно записать следующим образом:

```
имя_области_видимости :: имя
```

Идентификатор области видимости слева от операции '`::`' называется **квалификатором**.

С++ унаследовал от языка программирования С единое глобальное пространство имен. Если необходимо использовать глобальную переменную, находящуюся вне пространств имен, то для устранения неоднозначности используется операция '`::`' без квалификатора.

Пример:

```
char c;
namespace x{
   char c;
   void f(char e){
         ::c = c = e;
   }
}
```

Пространства имен могут быть вложенными.

Пример:

```
namespace ns1 (
   int n;
   . . .
   namespace ns2 {
          int sq(){
               return n*n;
          }
          void print_ns();
      . . .
   }
   void ns2::print_ns(){
          cout << "namespace ns2" << endl;
   }
}
```

При обращении к объекту, находящемуся во вложенном пространстве имен, например, из глобального пространства имен используются несколько квалификаторов:

```
ns1::ns2::print_ns();
```

Для сокращения записи имен используется объявление имени из пространства имен (***using***-объявление), например:

```
using namespace ns1;
```

В этом случае идентификаторы из указанной области можно использовать без квалификаторов вплоть до нового объявления пространства имен (***using***). Подробно об использовании ***using***-объявления см. в [12, стр. 211-216, 924-926].

Введение понятия пространства имен является одним из видов **статического полиморфизма**: одинаковым идентификаторам **придается разный смысл**.

Имена стандартной библиотеки С++ находятся в пространстве имен **std**. Поскольку функции, типы и константы стандартной библиотеки используются практически в каждой программе, рекомендуется для удобства использования имен стандартной библиотеки в начало программы вставлять ***using***-объявление:

using namespace std;

3. Классы и объекты С++

Центральным понятием ООП является **класс**. Класс используется для описания типа, на основе которого создаются **объекты** (переменные типа **класс**).

Класс, как и любой тип данных, характеризуется множеством значений, которые могут принимать объекты класса, и **множеством функций**, задающих операции над объектами.

Пример: Имеется множество пар чисел (a,b).

Если для данного множества определить арифметические операции следующим образом:

$$(a,b)+(c,d)=(ad+bc,bd)$$

$$(a,b)-(c,d)=(ad-bc,bd)$$

$$(a,b)*(c,d)=(ac,bd)$$

$$(a,b)/(c,d)=(ad,bc),$$

то это множество можно рассматривать как множество рациональных дробей: $(a,b)\to\frac{a}{b}$

Если же арифметические операции определить по-другому:

$$(a,b)+(c,d)=(a+c,b+d)$$

$$(a,b)-(c,d)=(a-c,b-d)$$

$$(a,b)*(c,d)=(ac-bd,ad+bc)$$

$$(a,b)/(c,d)=\left(\frac{ac+bd}{c^2+d^2},\frac{bc-ad}{c^2+d^2}\right)$$

это же множество пар чисел можно рассматривать как множество комплексных чисел: $(a,b)\to a+bi$.

Класс полноценно определяет тип данных как совокупность множества значений и набора операций над этими значениями.

3.1. Синтаксис описания класса

class *Имя_класса* { *определение_членов_класса* };

Члены класса можно разделить на информационные члены и функции-члены (методы) класса. Информационные члены описывают внутреннюю структуру информации, хранящейся в объекте, который создается на основе класса. Методы класса описывают алгоритмы обработки этой информации.

Данные, хранящиеся в информационных членах, описывают **состояние** объекта, созданного на основе класса. Состояние объекта изменяется на основе изменения хранящихся данных с помощью методов класса. Алгоритмы, заложенные в реализации методов класса, определяют **поведение** объекта, то есть реагирование объекта на поступающие внешние воздействия в виде входных данных.

3.2. Управление доступом к членам класса

Принцип **инкапсуляции** обеспечивается вводом в класс **областей доступа**:

- **private** (закрытый, доступный только собственным методам)
- **public** (открытый, доступный любым функциям)
- **protected** (защищенный, доступный только собственным методам и методам производных классов)

Члены класса, находящиеся в закрытой области (***private***), недоступны для использования со стороны внешнего кода. Напротив, члены класса, находящиеся в открытой секции (***public***), доступны для использования со стороны внешнего кода. При описании класса каждый член класса помещается в одну из перечисленных выше областей доступа следующим образом:

```
class  Имя_класса  {
 private:
      определение_закрытых_членов_класса
 public:
      определение_открытых_членов_класса
 protected:
      определение_защищенных_членов_класса
 …
};
```

Порядок следования областей доступа и их количество в классе – произвольны.

Служебное слово, определяющее первую область доступа, может отсутствовать. **По умолчанию** эта область считается **private**.

В закрытую (private) область обычно помещаются информационные члены, а в открытую (public) область – методы класса, реализующие интерфейс объектов класса с внешней средой. Если какой-либо метод имеет вспомогательное значение для других методов класса, являясь подпрограммой для них, то его также следует поместить в закрытую область. Это обеспечивает логическую целостность информации.

После описания класса его имя можно использовать для описания объектов этого типа.

Доступ к информационным членам и методам объекта , описанным в открытой секции, осуществляется через объект или ссылку на объект с помощью **операции выбора члена класса '.' .**

Если работа с объектом выполняется с помощью указателя на объект, то доступ к соответствующим членам класса осуществляется на основе **указателя на член класса '->'**:

```
class X {
 public:
      char c;
      int f(){…}
};
```

```
int main () {
      X x1;
      X & x2 = x1;
      X * p = & x1;
      int i, j, k;
      x1.c = '*';
      i = x1.f();
      x1.c = '+';
      j = x2.f();
      x1.c = '#';
      k = p -> f();
      …
}
```

Объекты класса можно определять совместно с описанием класса:

```
class Y {…} y1, y2;
```

3.3. Классы и структуры C++

Синтаксис класса в C++ совпадает с синтаксисом структуры C++:

`struct` *Имя_структуры* { *определение_членов_структуры* };

Класс C++ отличается от структуры C++ **только** определением по умолчанию первой области доступа в их описании (а также определением по умолчанию способа наследования, см. раздел 10.3):

- для структур умолчанием является открытый доступ (**`public`**)
- для классов умолчанием является закрытый доступ (**`private`**).

Различия в умолчаниях связаны с различиями целей создания таких конструкций. Структуры создавались для объединения и совместного использования разнородных типов данных, например записей файлов. Класс предназначен для определения полноценного типа данных.

В C++ объекты можно создавать также на основе структур и объединений (***`union`***).

Структуру можно рассматривать как прообраз понятия **класс**. Изначально концепция структуры (например, в Си) служила исключительно объединению разнородных данных в единой конструкции для совместной обработки, то есть в понятии структуры присутствовала исключительно информационная составляющая. В C++ время в структуре могут быть и функции-члены, в том числе и специальные члены – **конструкторы** и **деструкторы**, о которых будет рассказано в следующих разделах. Структуры наравне с классами можно использовать для полноценного описания типов данных, включающего описания операций, применяемых к описываемому типу данных. Однако наличие информационных членов в открытой секции нарушает один из основных принципов ООП – **принцип инкапсуляции**.

3.4. Манипуляции с состоянием объекта

Для доступа к внутренним информационным членам объекта, созданного на основе класса (чтение/запись), необходимо использовать специальные методы класса, называемые модификаторами (setters) и селекторами (getters). Они осуществляют подконтрольное считывание и изменение внутренних информационных членов. Так, если изменяется внутреннее информационное поле *size* объекта класса *stack*, описывающее максимальный размер стека, то необходимо осуществить ряд действий по согласованному изменению других информационных членов (выделение дополнительной памяти и т. д.):

```
class stack {
      int* c1;
      int top, size;
   public:
      . . .
      int putnewsize(int ns){
         if (top > ns) return 1;
         int* nc1 = new int[ns];
         if (top > 0)
            for (int i = 0; i < top; i++)
               nc1[i] = c1[i];
         delete c1;
         c1 = nc1;
         size = ns;
         return 0;
      }
};
```

Таким образом, изменение информационных полей объекта должно осуществляться специальными методами, производящими изменение требуемого информационного поля согласованно с одновременным изменением других информационных полей. Такие методы обеспечивают согласованность внутренних данных объекта.

3.5. Класс как область видимости

Класс является областью видимости описанных в нем членов класса. Идентификатор члена класса локален по отношению к данному классу. Классы могут быть вложенными. Одноименные идентификаторы членов класса закрывают **видимость** соответствующих внешних идентификаторов.

Операция `'::'` позволяет получить **доступ** к одноименным объектам, внешним по отношению к текущей области видимости, в частности, к глобальным функциям и переменным, следующим образом:

имя_класса :: имя_члена_класса или

:: имя - для имен глобальных функций и переменных.

Пример:

```
int ia1;
void f1(int b1) {
     ia1 = ia1 + b1;
}
class x {
      int ia1;
   public:
     x(){ia1 = 0;}
     void f1(int b1){
          ::f1(b1);  // вызов глобальной функции
     }
};

int main(){
   x a2;
   a2.f1(2);
   return 0;
}
```

3.6. Объявление и определение методов класса. Спецификатор *inline*

Каждый метод класса, должен быть **определен** в программе. Определить метод класса можно либо непосредственно в классе (если тело метода не слишком сложно и громоздко), либо вынести определение вне класса, а в классе только **объявить** соответствующий метод, указав его прототип.

При определении метода класса вне класса для указания области видимости соответствующего имени метода используется операции `'::'`:

Пример:

```
class x {
      int ia1;
   public:
      x(){ia1 = 0;}
      int func1();
};

int x::func1(){ … return ia1; }
```

Это позволяет повысить наглядность текста, особенно, в случае значительного объема кода в методах. При определении метода вне класса с использованием операции `'::'` прототипы объявления и определения функции должны совпадать.

Метод класса и любую функцию, не связанную ни с каким классом, можно определить со спецификатором ***inline***:

```
inline int func1();
```

Такие функции называются *встроенными*.

Спецификатор ***inline*** указывает компилятору, что необходимо по возможности генерировать в точке вызова код функции, а не команды вызова функции, находящейся в отдельном месте кода модуля. Это позволяет уменьшить время выполнения программы за счет отсутствия команд вызова функции и возврата из функции, которые кроме передачи управления выполняют действия соответственно по сохранению и восстановлению *контекста* (содержимого основных регистров процессора). При этом размер модуля оказывается увеличенным по сравнению с программой без спецификаторов ***inline***. Следует отметить, что спецификатор ***inline*** является рекомендацией компилятору. Данный спецификатор неприменим для функций со сложной логикой. В случае невозможности использования спецификатора для конкретной функции компилятор выдает предупреждающее сообщение и обрабатывает функции стандартным способом.

По определению методы класса, определенные непосредственно в классе, являются ***inline***-функциям

3.7. Указатель *this*

В классах С++ неявно введен специальный указатель ***this*** – указатель на текущий объект. Каждый метод класса при обращении к нему получает данный указатель в качестве **неявного параметра**. Через него методы класса могут получить доступ к другим членам класса.

Указатель ***this*** можно рассматривать как локальную константу, имеющую тип *X**, если *X* – имя описываемого класса. Нет необходимости использовать его явно. Он используется явно, например, в том случае, когда выходным значением для метода является текущий объект.

Данный указатель, как и другие указатели, может быть разыменован.

При передаче возвращаемого значения метода класса в виде ссылки на текущий объект используется разыменованный указатель ***this***, так как ссылка, как уже было указано, инициализируется непосредственным значением.

Пример:

```
class x {
   . . .
   public:
      x& f(. . .){
           . . .
           return *this;
      }
};
```

3.8. Указатель на член класса

Кроме адресации областей памяти, содержащих информационные объекты, указатели могут содержать адреса членов класса. Технологию создания и использования таких указателей легче описать с использованием конкретного примера, операторы которого подробно прокомментированы:

```
class X {
```

```
        int i;
    public:
        X(){i = 1;}
        int f1(int j){
            cout << "print i" << i << "\n";
            return j;
        }
        int f2(int j){
            cout << "reset i \n"; i = 1;
            return j;
        }
        int f3(int j){
            cout << "set i \n"; i = j;
            return j;
        }
};

typedef int (X::* pf)(int);  // см. комментарий 1)

int main(){
    int k, sw, par;
    x b;
    pf ff;                      // см. комментарий 2)
    . . .
    switch (sw){
        case 1: ff=&x::f1;      // см. комментарий 3)
                break;
        case 2: ff=&x::f2;
                break;
        case 3: ff=&x::f3;
    };
    k = (b.*ff)(par);            // см. комментарий 4)
  . . .
    return 0;
}
```

Комментарии:

1) ключевое слово **typedef** вводит новое имя для типа:
   `typedef int (X::* pf)(int);` - `pf` – тип указателя на метод класса `X` с одним входным параметром типа ***int*** и типом возвращаемого значения – ***int***.

2) `pf ff;` – создание объекта *ff*, имеющего введенный тип *pf*.

3) `ff = &X::f1;` – указателю *ff* присваивается адрес одного из методов класса. Доступ к этому методу по данному указателю через какой-либо объект невозможен (оператор *ff = &b.f1;* – неверен). Дело в том, что -указатель на член класса представляется для нестатических членов не

абсолютным, а относительным адресом, то есть смещением относительно базового адреса класса (указатель на статический член класса представляет собой истинный адрес).

4) `k =(b.* ff)(par);` – разыменование указателя на нестатический метод класса дает доступ к коду по относительному адресу, который применяется к базовому адресу конкретного объекта (в данном случае – объекта *b*).

Примечание. В случае объявления методов статическими членами (см. раздел «Статические члены класса») идентификатор *pf* необходимо объявить обычным указателем на функцию:

```
typedef int (* pf)(int);
```

Разыменование объекта такого типа представляется обычным разыменованием указателя на функцию:

```
k = (*ff)(par);
```

Применение техники разыменования указателя на метод класса является проявлением **динамического полиморфизма**, когда исполняемый код для одного и того же оператора (`k = (b.*ff)(par)`) определяется на этапе исполнения, а не компиляции. В большей мере динамический полиморфизм реализуется **виртуальными функциями**, описываемыми в следующих разделах.

4. Конструкторы и деструкторы

Конструкторы и **деструкторы** являются специальными методами класса

Конструкторы вызываются при создании объектов класса и отведении памяти под них.

Деструкторы вызываются при уничтожении объектов и освобождении отведенной для них памяти.

В большинстве случаев конструкторы и деструкторы вызываются автоматически (неявно) соответственно при описании объекта (в момент отведения памяти под него) и при уничтожении объекта. Конструктор (как и деструктор) может вызываться и явно, например, при создании объекта в динамической области памяти с помощью операции ***new***.

Так как конструкторы и деструкторы неявно входят в интерфейс объекта, их следует располагать в открытой области класса.

Примечание. Конструкторы и деструкторы могут располагаться и в закрытой области для блокирования возможности неявного создания объекта. Но в этом случае явное создание объекта возможно только при использовании статических методов, являющихся частью класса, а не конкретного объекта. Статические методы описываются далее.

Отличия и особенности описания **конструктора** от обычной функции:

1) Имя конструктора совпадает с именем класса
2) При описании конструктора не указывается тип возвращаемого значения

Следует отметить, что и обычная процедура может не возвращать значения, а только перерабатывать имеющиеся данные. В этом случае при описании соответствующей функции указывается специальный тип возвращаемого значения ***void***.

В описании конструктора тип возвращаемого значения не указывается не потому, что возвращаемого значения нет. Оно как раз есть. Ведь результатом работы конструктора в соответствии с его названием является **созданный объект** того типа, который описывается данным классом. Страуструп отмечал, что конструктор – это то, что область памяти превращает в объект.

Конструкторы можно классифицировать разными способами:

1) по наличию параметров:

   – без параметров,
   – с параметрами;

2) по количеству и типу параметров:

   – конструктор **умолчания**,
   – конструктор **преобразования**,
   – конструктор **копирования**,
   – конструктор **с двумя и более параметрами.**

Набор и типы параметров зависят от того, на основе каких данных создается объект.

В классе может быть несколько конструкторов. В соответствии с правилами языка C++ все они имеют одно имя, совпадающее с именем класса, что является одним из проявлений статического **полиморфизма**. Компилятор выбирает тот конструктор, который в зависимости от ситуации, в которой происходит создание объекта, удовлетворяет ей по количеству и типам параметров. Естественным ограничением является то, что в классе не может быть двух конструкторов с одинаковым набором параметров.

Деструкторы применяются для корректного уничтожения объектов. Часто процесс уничтожения объектов включает в себя действия по освобождению выделенной для них по операциям ***new*** памяти.

Имя деструктора: ***~имя_класса***
У деструкторов нет параметров и возвращаемого значения.

В отличие от конструкторов деструктор в классе может быть только один.

Пример: Описание класса.

```
class box{
      int len, wid, hei;
   public:
      box(int l, int w, int h){
           len = l; wid = w; hei = h;
      }
```

```
        box(int s){
             len = wid = hei = s;
        }
        box(){
             len = 2; wid = hei = 1;
   }
        int volume(){
             return len * wid * hei;
        }
  };
```

4.1. Конструктор умолчания

Конструктор без параметров называется **конструктором умолчания**.

Если для создания объекта не требуется каких-либо параметров, то используется конструктор умолчания. При описании таких объектов после имени класса указывается только идентификатор переменной:

```
class X{ … };
X x1;
```

Замечание: роль конструктора умолчания может играть конструктор, у которого все параметры имеют априорные значения, например:

```
box (int l = 24, int w = 12, int h = 6);
```

4.2. Конструктор преобразования и конструкторы с двумя и более параметрами

Если для создания объекта необходимы параметры, то они указываются в **круглых скобках** после идентификатора переменной:

```
box b2(1,2,3);
box b3(5);
```

Указываемые параметры являются параметрами конструктора класса. Если у конструктора имеется **ровно один** входной параметр, который **не** представляет собой ссылку на свой собственный класс, то соответствующий конструктор называется **конструктором преобразования**. Этот конструктор называется так в связи с тем, что в результате его работы на основе объекта одного типа создается объект другого типа (типа описываемого класса).

Если уже описан класс *T* и описывается новый класс *X*, то его конструкторы преобразования могут иметь любой из следующих прототипов:

```
X(T);
X(T&);
X(const T&);
```

Последний прототип служит для защиты от изменения передаваемого фактического параметра в теле конструктора, так как при получении ссылки на фактический параметр используется собственно передаваемый объект, а не его локальная копия.

Примечание. Выделение в отдельную группу конструкторов с двумя и более параметрами, независимо от их типа, является в некотором смысле, условным. Так, например, если есть два класса: *Vector* и *Matrix*, то для создания соответствующих объектов:

```
Vector v1(10);
Matrix m1(10,15);
```

используется в первом случае один параметр, а во втором случае – два параметра. Таким образом, в первом случае объект создается с помощью конструктора преобразования, а во втором случае, с формальной точки зрения, с помощью конструктора с двумя параметрами, хотя в обоих случаях фактически выполняется одна и та же процедура: создание объекта на основе заданных числовых параметров.

Как уже было отмечено, если у параметра конструктора преобразования имеется априорное значение, и при описании объекта явно не задается фактический параметр, этот конструктор играет роль конструктора умолчания.

Пример:

```
class X {
      int x1;
   public:
      X(int px1 = 0}
};
```

Для такого класса будут верны следующие объявления объектов:

```
int main(){
… X x1, x2(1); …
}
```

4.3. Конструктор копирования

При создании объекта его информационные члены могут быть проинициализированы значениями полей другого объекта этого же типа, то есть объект создается как копия другого объекта.

Для такого создания объекта используется **конструктор копирования**.

Инициализация может быть выполнена аналогично инициализации переменных встроенных типов с использованием операции присваивания совместно с объявлением объекта:

```
box b5(2,4,6); // создание объекта типа box с
               // использованием числовых данных
box b6 = b5;   // создание объекта b6 – копии объекта b5
```

Если инициализация производится объектом такого же типа, то объект-инициализатор также может быть указан в круглых скобках после идентификатора создаваемого объекта:

```
box b7(b5);
```

Свод ситуаций, в которых используется конструктор копирования, описаны ниже.

Если класс не предусматривает создания внутренних динамических структур, например, массивов, создаваемых с использованием операции **new,** то в конструкторе копирования достаточно предусмотреть **поверхностное копирование**, то есть почленное копирование информационных членов класса.

Конструктор копирования, осуществляющий поверхностное копирование, можно явно не описывать, он сгенерируется **автоматически**.

Если же в классе предусмотрено создание внутренних динамических структур, использование только поверхностного копирования будет ошибочным, так как информационные члены-указатели, находящиеся в разных объектах, будут иметь одинаковые значения и указывать на одну и ту же размещенную в динамической памяти структуру. Автоматически сгенерированный конструктора копирования в данном классе не позволит корректно создавать объекты такого типа на основе других объектов этого же типа.

В подобных случаях необходимо **глубокое копирование**, осуществляющее не только копирование информационных членов объекта, но и самих динамических структур. При этом, естественно, информационные члены-указатели в создаваемом объекте должны не механически копироваться из объекта-инициализатора, а указывать на вновь созданные динамические структуры нового объекта.

Пример: Для класса `stack` конструктор копирования может быть определен следующим образом:

```
class stack {
      char* c1;
      int top, size;
 public:
      stack(int n = 10){
           size = n;
           top = 0;
           c1 = new char[size];
      }
      stack(stack & s1);
           . . .
};

stack::stack(stack & s1){
      size = s1.size;
      top = s1.top;
      c1 = new char[size];
      for (int i = 0; i < size; i++)
           c1[i] = s1.c1[i];
}
```

Замечания по работе конструктора копирования:

1) Входной параметр является внешним объектом по отношению к созда-

ваемому объекту. Тем не менее, имеется возможность прямого обращения к закрытым членам этого внешнего объекта. Это возможно только потому, что входной параметр имеет тип, совпадающий с типом создаваемого в результате работы конструктора копирования объекта. Если бы на вход конструктора поступал бы объект другого типа (например, в конструкторе преобразования класса *vector* входным параметром был бы объект, созданный на основе класса *matrix*), то для доступа к закрытым членам объекта-параметра необходимо было бы применять специальные средства. Это связано с тем, что **единицей защиты является не объект, а тип**, то есть методы объекта могут обращаться к закрытым членам не только данного объекта, но и к закрытым членам любого объекта данного типа.

2) В момент описания конструктора копирования класс, как тип данных, еще не описан до конца. Тем не менее, идентификатор класса уже используется в качестве полноценного типа данных при описании входного параметра конструктора копирования. Такая технология схожа с описанием рекурсивной функции, когда тело описываемой функции содержит вызов этой же функции.

В отличие от конструктора преобразования, входной параметр конструктора копирования имеет тип, описываемый данным классом. Таким образом, если описывается класс *X*, то его конструктор копирования может иметь один из следующих прототипов:

```
X(X&);
X(const X&);
```

Объект, создаваемый с использованием конструктора копирования, может инициализироваться не только именованными объектами, но и временно созданными объектами.

Пример:

```
box* b4 = new box(2,3,5); // Явный запуск конструктора
                          // с тремя параметрами. Адрес
                          // динамически созданного
                          // объекта типа box
                          // присваивается переменной
                          // b4.
box b5 = * b4;       // Разыменование указателя на объект,
                     // т.е. получение доступа к
                     // информации, хранящейся в нем, и
                     // использование ее для инициализации
                     // создаваемого объекта.
box b6 = box(4,7,1); // Создание временного объекта и
                     // инициализация именованного
                     // объекта.
```

4.4. Спецификатор explicit

Если инициализация создаваемого объекта производится объектом другого типа, то автоматически производится вызов соответствующего **конструктора преобразования** для преобразования инициализирующего значения к типу объявленного объекта.

Например, для приведенного выше класса *stack* описание объекта класса с инициализацией может быть таким:

```
stack st1 = 15;
```

оно эквивалентно следующему:

```
stack st1 = stack(15);
```

Таким образом, если пользователь класса *stack* предполагал создать объект *st1* типа *stack* с максимальной глубиной, задаваемой по умолчанию (10 элементов), и поместить в его вершину значение 15, то он ошибся, поскольку реальным результатом сделанного объявления будет создание пустого стека с максимальной глубиной в 15 элементов.

Для подавления неявного вызова конструктора преобразования, если такое действие может привести к ошибке, конструктор преобразования необходимо объявлять с использованием ключевого слова ***explicit*** (явный):

```
explicit stack(int n = 10){. . .}
```

В этом случае при объявлении переменной:

```
stack st1 = 15;
```

будет выдана ошибка компиляции: невозможно преобразовать целочисленное значение в тип класса *stack*.

4.5. Конструктор копирования и операция присваивания

Если объект уже создан, то операция присваивания **'='** осуществляет не инициализацию создаваемого объекта, а копирование данных, то есть передачу данных между существующими объектами.

Пример:

```
box b3(4,1,1); // создание объекта b3
box b2;        // создание объекта b2
b2 = b3;     // операция присваивания: копирование объекта
             // b3 в существующий объект b2.
```

В операции присваивания, так же, как и в конструкторе копирования, по умолчанию осуществляется **поверхностное копирование**. Если требуется **глубокое копирование**, то необходимо **перегрузить** (описать нужный алгоритм) операцию присваивания. Перегрузка операций рассматривается в следующих разделах.

4.6. Автоматическая генерация конструкторов и деструкторов

Автоматически могут генерироваться только конструкторы умолчания, конструкторы копирования и деструкторы.

Если в классе явно **не описано ни одного конструктора**, то автоматически генерируется **конструктор умолчания** с пустым телом.

Если в классе явно описан хотя бы один конструктор, например, конструктор копирования, то конструктор умолчания **не будет** автоматически генерироваться, даже, если он необходим в соответствии с постановкой задачи.

В случае отсутствия в классе явно описанного конструктора копирования он **всегда** генерируется автоматически и обеспечивает **поверхностное копирование**.

Если в классе **не описан деструктор**, то **всегда** автоматически **генерируется** деструктор, который не производит никаких действий.

Таким образом, даже если в классе не описаны конструкторы и деструктор, они все равно неявно присутствуют в нем.

4.7. Список инициализации

При инициализации информационных членов класса, являющихся объектами, создаваемыми на основе других классов, вызываются их конструкторы умолчания. Если внутренние объекты должны быть созданы с использованием параметров (то есть должны быть использованы конструктор преобразования, копирования или конструктор с двумя и более параметрами), то для этого может быть использован **список инициализации**. В списке инициализации также могут быть инициализированы внутренние объекты встроенных типов.

Список инициализации указывается в описании конструктора основного класса.

Прототип конструктора со списком инициализации:

Имя_класса (*параметры*) : *список_инициализации* **{** *тело_класса* **}**

Список инициализации – последовательность разделенных запятой инициализируемых переменных встроенных типов и конструкторов для переменных, представляющих объекты типов описанных классов (пользовательских типов данных), а также необходимых конструкторов преобразования или конструкторов с двумя и более параметрами для базовых классов, от которых унаследован описываемый класс.

В качестве инициализаторов переменных встроенных типов и параметров конструкторов могут выступать формальные параметры конструктора описываемого класса. В случае использования списка инициализации тело конструктора может оказаться пустым. Но и в этом виде оно обязательно должно присутствовать.

Пример:

```
class Z0{
	int z01;
public:
	Z0(){
		z01 = 0;
```

```
        }
        Z0(int pz0){
             z01 = pz0;
        }
};
class Point{
        int x, y;
    public:
        Point(): x(0), y(0){}
        Point(int x1, const int y1): x(x1), y(y1){}
        Point(const point& p1) : x(p1.x), y(p1.y){}
};

class Z1{
        Point p, p2, p3;
        int z;
    public:
        Z1(int z2, int x2, int y2, int pz01):
           p2(x2,y2), p(pz01,0), p3(p2), z(z2){}
};

int main(){
       z1* z3 = new z1(1,2,3,4);
       return 0;
}
```

Недопустимо использование конструкторов с параметрами (даже константами) при описании подобъектов классов (членов-данных, являющихся объектами других классов). Если подобъект класса необходимо инициализировать с использованием констант, то это можно сделать только в строке инициализации. Так, класс *Z1* из приведенного только что примера не может быть описан следующим образом:

```
class z1{
        point p;
        point p2(2,3);  // ОШИБКА!
        point p3;
        int z;
    public:
        z1(int z2): p3(p2),z(z2){}
};
```

Правильное описание класса:

```
class z1{
        point p, p2, p3;
        int z;
    public:
        z1(int z2): p2(2,3),p3(p2),z(z2){}
};
```

Если бы в классе `z1` были описаны несколько конструкторов, то в строку инициализации каждого из них необходимо было бы поместить вызов конструктора с двумя и более параметрами для поля `p2`.

4.8. Порядок вызова конструкторов и деструкторов

Итак, конструкторы можно классифицировать по набору входных параметров следующим образом:

1) `x()` – конструктор умолчания
2) `x(x&), x(const x&)` – конструктор копирования
3) `x(t), x(t&), x(const t&)` – конструктор преобразования
4) конструктор с двумя и более параметрами

В классе может быть описано несколько конструкторов преобразования и конструкторов с двумя и более параметрами. Единственное требование к ним: не может быть двух конструкторов с одинаковым (по типам) набором параметров, в частности, не может быть двух конструкторов умолчания.

При создании объекта конструкторы вызываются в следующем порядке:

1) Конструкторы базовых классов, если класс для создаваемого объекта является наследником других классов в порядке их появления в описании класса. Если в списке инициализации описываемого класса присутствует вызов конструктора преобразования (или конструктора с двумя и более параметрами) базового класса, то вызывается конструктор преобразования (или конструктор с двумя и более параметрами), иначе вызывается конструктор умолчания базового класса.

2) **Конструкторы умолчания** всех вложенных информационных членов, которые не перечислены в списке инициализации, и конструкторы преобразования, копирования и конструкторы с двумя и более параметрами всех вложенных информационных членов, которые перечислены в списке инициализации. Все перечисленные в данном пункте конструкторы (умолчания, преобразования, копирования, с двумя и более параметрами) вызываются в порядке описания соответствующих информационных членов в классе.

3) Собственный конструктор.

Такая последовательность вызова конструкторов логически обосновывается тем, что в момент выполнения собственного конструктора все информационные поля должны быть уже проинициализированы.

Деструкторы вызываются в обратном порядке:

1) Собственный деструктор. В момент начала его работы поля класса еще не очищены, и их значения могут быть использованы в теле деструктора.

2) Деструкторы вложенных объектов в порядке, обратном порядку их описания.

3) Деструкторы базовых классов в обратном порядке их задания.

Свод ситуаций, при которых вызываются конструкторы и деструкторы

Обычный конструктор (не копирования) вызывается:

1) при создании объекта (при обработке описания объекта);
2) при создании объекта в динамической памяти (с использованием операции ***new***). При этом в динамической памяти предварительно отводится необходимый по объему фрагмент памяти;
3) при композиции объектов;
4) при создании объекта производного класса.

Замечание. Если в программе создается не единичный (скалярный) объект, а массив объектов, то в соответствующем классе (структуре) должен быть конструктор умолчания (явно или неявно сгенерированный). Действительно, при объявлении массива объектов или создании массива в динамической памяти указывается размерность массива, что несовместимо с заданием параметров для конструктора преобразования или конструктора с двумя и более параметрами:

```
class X{ . . .};
X* x1 = new X[10];
```

Как уже было указано, если в классе явно описан хотя бы один конструктор, то конструктор умолчания не генерируется системой неявно. Для удаления такого массива должна применяться **векторная форма** операции ***delete***, при использовании которой деструктор вызывается для каждого элемента массива:

```
delete[] x1;
```

Конструктор копирования вызывается:

1) при инициализации создаваемого объекта:

```
box a(1,2,3);     // вызов конструктора с тремя параметрами
box b = a;        // инициализация
```

2) при инициализации временным объектом:

```
box c = box(3,4,5);
```

3) при передаче параметров-объектов в функцию **по значению**:

```
int f(box b);
```

4) при возвращении результата работы функции в виде объекта.

```
box f ();
```

Примечание 1. Если используется **оптимизирующий** компилятор, то при обработке инициализации вида:

```
box c = box(3,4,5)
```

временный объект не создается, и вместо конструктора копирования используется кон-

структор с тремя параметрами:

(*box c(3,4,5)*).

Примечание 2. Если при возвращении результата работы функции в виде объекта тип возвращаемого значения не совпадает с типом результата работы функции, то вызывается не конструктор копирования, а конструктор преобразования или функция преобразования (описывается ниже). Данное преобразование выполняется, если оно однозначно. Иначе фиксируется ошибка.

Пример:

```
class Y;
class X{
   . . .
   public:
      X(const Y& y1);
      . . .
};

class Y{
   . . .
   public:
     . . .
     X f(){
        . . .
        return *this;
     }
};

X::X(const Y& y1)
      {. . .}
```

Деструктор вызывается:

1) при свертке стека – при выходе из блока описания объекта, в частности, при обработке исключений (при выходе из ***try***-блока по оператору ***throw***, ***try***-блоки описываются далее), завершении работы функций;

2) при уничтожении временных объектов – сразу, как только завершится конструкция, в которой они использовались;

3) при выполнении операции ***delete*** для указателя, получившего значение в результате выполнения операции ***new***. После выполнения деструктора освобождается выделенный для объекта участок памяти;

4) при завершении работы программы при уничтожении глобальных и статических объектов.

Примечание. Все правила описания и использования конструкторов и деструкторов применимы и для структур.

5. Статические члены класса

Информационные члены класса, которые могут быть представлены в единственном экземпляре для всех объектов данного типа, в случае такого представления называются статическими членами. Они не являются частью объектов этого класса и размещаются в статической памяти. Для их описания используется служебное слово ***static***.

Для всех объектов, созданных на основе класса, содержащего статический член, существует только одна копия этого члена. Примером такого статического члена является счетчик числа созданных объектов данного класса. Такой счетчик может существовать только в отрыве от всех экземпляров объектов данного класса, в то же время работать с ним обычно приходится одновременно с вызовом обычных методов, например, конструкторов объектов, поэтому описывать счетчик удобнее именно как элемент класса.

Статическими могут быть не только информационные члены класса, но и его методы. Статический метод не может использовать никакие нестатические члены класса, так как, являясь частью класса, а не объекта, он не имеет неявного параметра ***this***. Поскольку статический метод не использует специфического содержимого конкретного объекта, то обращение к нему может осуществляться не только с использованием идентификатора объекта, но и с использованием идентификатора класса:

имя_класса :: *имя_функции (фактические_параметры);*

Одно из очевидных применений статических методов – манипуляция глобальными объектами и статическими полями данных соответствующего класса.

Примечание 1. Статические методы класса не могут вызывать нестатические, так как последние имеют доступ к данным конкретных объектов. Обратное допустимо: нестатические методы могут вызывать статические методы.

Примечание 2. Статический метод класса может создавать объекты данного и любого другого класса. Это можно использовать, в частности, если необходимо в программе запретить создание объектов простым объявлением или с использованием операции ***new***. В этом случае конструктор и деструктор помещаются в закрытую область класса, а для создания и уничтожения объекта используются специальные статические методы, которые можно вызывать, не имея ни одного объекта.

Примечание 3. Статические методы класса не могут быть виртуальными и константными (`inline`–функциями быть могут).

Пример:

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

class X{
      X(){}
     ~X(){}
   public:
      static X& createX(){
```

```
            X* x1 = new X;
            cout << "X created" << endl;
            return *x1;
         }
        static void destroyX(X& x1){
            delete &x1;
            cout << "X destroyed" << endl;
         }
};

int main(){
   X& xx1 = X::createX();
   . . .
   X::destroyX(xx1);
   return 0;
}
```

Статические информационные члены класса, даже находящиеся в закрытой области (а это характерно для информационных членов класса в соответствии с принципом инкапсуляции), необходимо объявить дополнительно вне класса (с возможной инициализацией):

тип_переменной имя_класса :: *идентификатор* = *инициализатор*;

Это связано с тем, что память для статического объекта должна быть выделена до начала работы программы. В то же время при обработке описания класса до создания конкретных объектов никакие области памяти не отводятся. В дальнейшем прямое обращение к статическим информационным членам, находящимся в закрытой секции, недопустимо. Если инициализация не нужна, то все равно необходимо дополнительное объявление статического члена вне класса для резервирования памяти для статического члена. В противном случае на этапе сборки исполняемого модуля будет выдана ошибка о неразрешенной внешней ссылке.

Пример:

```
class B {
        static int i;  //  статический информационный член
                       //  класса
   public:
      static void f(int j){ // статический метод
           i = j;
      }
};

int B::I = 10;     // дополнительное внешнее определение
                   // статической переменной с
                   // инициализацией статического
                   // информационного члена класса b.
```

```
int main(){
   B a;
   . . .
   B::f(1);    // вызов статической функции-члена класса.
   . . .
   return 0;
}
```

6. Константные члены класса. Модификатор const

Все информационные члены класса, не являющиеся статическими информационными членами, можно представлять, как данные, доступные методу класса через указатель ***this*** (в случае необходимости этот указатель можно употреблять явно).

Если необходимо запретить методу изменять информационные члены объектов класса, то при его описании используется дополнительный модификатор ***const*:**

$$\begin{matrix}\textit{Тип_возвращаемого} \\ \textit{значения}\end{matrix} \quad \begin{matrix}\textit{Имя} \\ \textit{функции}\end{matrix} \left(\begin{matrix}\textit{формальные} \\ \textit{параметры}\end{matrix} \right) \ \textbf{\textit{const}} \ \{\textit{тело_функции}\}$$

Описанные таким образом методы класса называются **константными.**

Тем не менее, статические члены класса могут изменяться такой функцией, так как они являются частью класса, но не объекта. Если же статические информационные члены класса имеют дополнительный модификатор **const,** то они не могут изменяться никакими методами класса.

Примечание. В некоторых изданиях (см., например, [9] стр. 144, п.5.8. Функции-члены типа ***static*** и ***const***) данные, доступные через указатель ***this***, рассматриваются как неявные аргументы метода класса. Конечно, можно рассматривать глобальные переменные также в качестве неявных параметров для всех функций, а не только методов класса. Тем не менее, если дать определение неявных аргументов метода класса как данных, доступных через указатель ***this,*** то вышеописанное можно сформулировать следующим образом: константные методы не могут изменять свои неявные аргументы.

Таким образом, если объект типа описанного класса является **константным объектом**, то есть он объявлен с модификатором **const**, это означает, что изменение его состояния недопустимо. В таком случае все применяемые к этому объекту методы (кроме конструкторов и деструктора) **должны иметь** модификатор ***const*.** Данное требование является обязательным **независимо от наличия или отсутствия** информационных членов в классе.

Для защиты от изменения передаваемых фактических параметров в теле функции соответствующие формальные параметры также объявляются с модификатором ***const***:

$$\textbf{\textit{const}} \ \begin{matrix}\textit{Тип} \\ \textit{параметра}\end{matrix} \ \textit{идентификатор}$$

Объявление методов класса и формальных параметров с модификатором ***const*** называется **контролем постоянства**.

Если необходимо запретить изменение объекта в пределах его области видимости, то при объявлении объекта используется ключевое слово ***const***, например:

```
const X x2 = x1;
```

Примечание. При объявлении объекта с модификатором ***const*** объект должен быть обязательно инициализирован. Объект пользовательского типа может быть инициализирован неявно (например, с помощью конструктора класса), если в описании типа объекта указаны параметры, принимаемые по умолчанию при отсутствии в объявлении объекта явных параметров.

Пример:

```
#include <iostream>
using namespace std;
class A {
      static int i;
      void f() const { // модификатор const, запрещающий
                       // изменять неявные аргументы,
                       // необходим в связи с тем, что
                       // имеется объект данного класса с
                      // описателем const
           if (i < 0) g(i);
           cout << "f()" << endl;
      }
public:
     void g(const int & n) const {
                         // модификатор const для
                         // параметра-ссылки необходим в
                         // связи с использованием
                         // числовой константы 2 при вызове
                         // данного метода для объекта:
                         // a.g(2)
           i = n;
           f();
           cout << "g()" << endl;
      }
};

int A::i = 1;        // инициализация статической переменной

int main(){
   const A a = A();
   a.g(2);
   return 0;
}
```

7. Друзья классов

Имеется ряд ситуаций, когда объекту одного класса необходимо иметь прямой доступ к закрытым членам объекта другого класса без использования методов-селекторов. Для этого в языке С++ введена концепция друзей и специальное ключевое слово ***friend***..

Друг класса – это функция, не являющаяся членом класса, но имеющая доступ к его закрытым и защищенным членам.

Друзья класса объявляются в самом классе с помощью служебного слова ***friend***. в любой области доступа.

Другом класса может быть обычная функция, метод другого класса или другой класс (при этом каждый его метод становится другом класса).

Пример:

```
class B;   // предварительное объявление идентификатора
           // b как идентификатора типа данных
class X {
     int ia1;
public:
     X(){
          ia1 = 0;
     }
     int func1(b& bb);
};

class B {
      int b1;
   public:
      friend int X::func1(B & bb);
      B(){
         b1 = 1;
      }
};

int X::func1(B & bb){
   ia1 = ia1 + bb.b1;
   return ia1;
}

int main(){
   int i1;
   B b2;
   X a2;
   i1 = a2.func1(b2);
   return 0;
}
```

Примечание. Несмотря на предварительное объявление идентификатора *B,* его можно использовать в описании класса *X*, находящемся перед описанием класса *B*,

только в описании формального параметра в прототипе функции (`func1`). Саму функцию `func1` необходимо описывать вне класса `X` после описания классов `B` и `X`, используя операцию разрешения области видимости **`'::'`** с квалификатором `X`. Неправильным будет следующее описание функции `func1`:

```
class B;

class X {
      int ia1;
   public:
      X(){
           ia1 = 0;
      }
      int func1(B & bb){
         ia1 = ia1 + bb.b1;  // ОШИБКА!
         return ia1;
      }
};

class B {
      int b1;
   public:
      friend int X::func1(B & bb);
      B(){
           b1 = 1;
      }
};

int main(){
   int i1;
   B b2;
   X a2;
   i1 = a2.func1(b2);
   return 0;
}
```

Другом можно объявить и весь класс: ***`friend class X;`***

Другом класса может быть не только метод другого класса, но и внешняя функция. Кроме того, возможна дружественность сразу для нескольких классов. Это необходимо, например, в случае организации взаимодействия нескольких объектов разных классов, когда функция, обеспечивающая взаимодействие, должна иметь доступ к закрытым компонентам одновременно нескольких объектов. Объявить функцию методом одновременно нескольких классов невозможно, поэтому в стандарте языка С++ предусмотрена возможность объявлять внешнюю по отношению к классу функцию дружественной данному классу. Для этого необходимо в теле класса объявить некоторую внешнюю по отношению к классу функцию с использованием ключевого слова ***`friend`***:

`friend` *имя_функции* (*список_формальных_параметров*);

Пример:

```
class B;

class D {
   int x;
   . . .
   friend void func(B &, D &); //функция дружественна
классу D
   . . .
};

 class B {
   int y;
   . . .
   friend void func(B &, D &); // функция дружественна
                               // классу B
   . . .
 };

 void func(B & b1, D & d1) {
   cout << d1.x + b1.y; // дружественная функция имеет
                        // доступ к закрытым
                        // компонентам обоих классов
 }
```

8. Статический полиморфизм

Статический полиморфизм реализуется с помощью **перегрузки** функций и операций. Под перегрузкой функций в С++ понимается описание **в одной области видимости** нескольких функций с одним и тем же именем. О перегрузке операций в С++ говорят в том случае, если в некоторой области видимости появляется описание функции с именем **operator <обозначение_операции_С++>**, задающее еще одну интерпретацию заданной операции.

8.1. Перегрузка бинарных операций

Для перегрузки операций используется ключевое слово **operator.** Прототип перегруженной операции:

$$\begin{matrix}\textit{Тип_возвращаемого} \\ \textit{значения}\end{matrix}\ \ \textbf{operator}\ \begin{matrix}\textit{Символ} \\ \textit{оператора}\end{matrix}(\textit{операнды})\{\textit{тело_функции}\};$$

Перегружать операции можно с помощью:

- функции-члена;
- функции-друга;
- глобальной функции (как правило, менее эффективно).

Можно перегружать любые операции языка С++, кроме следующих:

- .   операция выбора члена класса
- ::   операция разрешения области видимости
- ? :   условная операция (например, `j = i>0 ? 1 : 0;`)
- .*   операция разыменования указателя на член класса
- #   директива препроцессора
- ***sizeof***
- ***typeid***

При перегрузке операции с помощью метода число формальных параметров оказывается на единицу меньше числа фактических операндов операции. В этом случае первый операнд операции соответствует объекту типа класса, в котором перегружается операция. В случае бинарной операции входной параметр соответствует второму операнду перегружаемой операции.

При перегрузке операции с помощью функции-друга число формальных параметров совпадает с числом операндов операции, так как в этом случае операнды операции, представленные формальными параметрами, являются внешними объектами для такой функции.

Тип выходного параметра является встроенным типом или типом, определенным пользователем (то есть классом).

Если при перегрузке операции методом класса результатом применения операции является изменение первого (или единственного) операнда, то рекомендуется объявлять выходной параметр в виде ссылки на текущий объект. Это необходимо для оптимизации использования результата операции в других операциях, совмещенных в одном операторе, например: `z = x += y;`

Если при перегрузке операции функцией-членом результатом применения перегружаемого операции является вычисление значения, не изменяющего первый операнд, а также при перегрузке операции функцией-другом, выходной параметр не может быть ссылкой (если выходной параметр требуется). Это связано с тем, что вычисляемое значение помещается во временный объект, который уничтожается при завершении работы алгоритма перегруженной операции и выходе из области видимости этого временного объекта.

Пример: Перегрузка операции **'+'** методом класса:

```
class complex {
      double re, im;
   public:
      complex(double r=0, double i=0):re(r),im(i){}
      complex operator +(const complex& y);
};

complex complex::operator +(const complex & y){
   complex t(re + y.re, im + y.im);
   return t;
}
```

Пример: Перегрузка операции `'+'` функцией-другом:

```
class complex {
     double re, im;
  public:
     complex(double r = 0, double i = 0):re(r),im(i){}
     friend complex operator +(const complex & x,
                               const complex & y);
};

complex operator +(const complex& x, const complex& b){
   complex t(x.re + b.re, x.im + b.im);
   return t;
}
```

Перегрузка операции присваивания может быть произведена только методом класса и не может быть перегружена функцией-другом.

В отличие от операции присваивания **операция `'+='`** (и другие подобные операции) **может быть перегружена как методом класса, так и функцией-другом**.

Прототип перегрузки операции присваивания:

X & ***operator*** = (***const*** X &);

или X & ***operator*** = (X&);

Пример:

```
class vector{
     int* p;
     int size;
  public:
     . . .
     vector& operator=(const vector& v1);
     friend vector & operator+=(vector & v1,
                              const vector & v2);
};

vector& vector::operator =(const vector& v1){
   if (size != v1.size){
      delete p;
      size = v1.size;
      p = new int[size];
   };
   for (int i = 0; i < size; i++)
       p[i] = v1.p[i];
   return *this;       // возвращается ссылка на текущий
                      // объект.
}
```

```
vector & operator +=(vector& v1, const vector& v2){
    int j;
    j = v1.size;
    if (j > v2.size)
        j = v2.size;
    int i;
    for (i= 0; i < j; i++)
        v1.p[i] = v1.p[i] + v2.p[i];
    return v1;          // возвращается значение первого
                        // параметра.
}
```

Операцию следует перегружать функцией членом того класса, который является типом первого операнда. Если первый операнд имеет встроенный или библиотечный тип, в описание которого невозможно вставить описание дружественной функции, то такую операцию можно перегружать только функцией-другом класса, к которому относится второй операнд.

Пример: Перегрузка операции вывода.

В файле внешней стандартной библиотеки *iostream* стандартная операция языка ***'<<'***, осуществляющая побитный сдвиг, перегружена в классе ostream как операция вывода. Операция ***'<<'*** перегружена для вывода объектов стандартных типов: ***int***, ***char***, ***double***, ***char**** и других встроенных типов. Формат использования данной операции:

cout << *переменная_стандартного_типа*;

Таким образом, первый операнд операции ***'<<'*** должен иметь тип *ostream*. Если необходимо перегрузить данную операцию для структурированного вывода объекта пользовательского типа, то, как было пояснено, это можно сделать только функцией-другом разработанного класса. Например, для класса комплексных чисел операция ***'<<'*** может быть перегружена так:

```
class complex {
      double re,im;
   public:
        complex(double re2, double im2):re(re2),im(im2){}
        friend ostream& operator<<(ostream & out,
                                   const complex par);
        . . .
};

ostream& operator<<(ostream& out, const complex par){
    out << par.re << "+" << par.im << "i";
    return out;
}
```

Здесь операция *'<<'* получает в качестве первого параметра ссылку на существующий объект типа *ostream*. Данный объект дополняется необходимой для вывода информацией и ссылка на него возвращается во внешнюю среду. Благодаря этому можно в одном операторе программы осуществить вывод ряда значений:

```
cout << c1 << "    " << c2;
```

Примечание. Операцию вывода можно перегружать и методом соответствующего класса. Но это будет выглядеть не совсем естественно. Действительно, в этом случае первый операнд будет тип текущего класса. Поэтому, например, для класса комплексных чисел перегрузка операции вывода методом класса будет выглядеть следующим образом:

```
ostream& operator<<(ostream& out){
    out << re << "+" << im << "i";
    return out;
}
```

а вызов операции вывода будет выглядеть так:

`c1 << cout;` либо так: `c1.operator << (cout);`

Конечно, и в том, и в другом случае операция вывода выглядит не совсем привычно. Для того, чтобы вид операции вывода при перегрузке методом остался привычным, данную операцию необходимо перегрузить непосредственно в классе *ostream* (если, это целесообразно и возможно, поскольку этот тип определен в библиотеке, произвольно менять которую обычно не рекомендуется).

8.2. Перегрузка унарных операций

Если для унарной операции имеется только одна форма, то ее перегрузка реализуется по общим описанным выше правилам. При этом, как уже было описано, для оптимизации использования результата операции в других операциях, совмещенных в одном операторе с данной операцией, рекомендуется объявлять выходной параметр в виде ссылки на текущий объект.

Специфика перегрузки операций инкремента и декремента, операции индексации

При перегрузке унарной операции в том случае, если для нее в языке определены две формы – **префиксная** и **постфиксная**, имеются особенности.

Для того, чтобы отличать постфиксную форму от префиксной, при перегрузке операции в постфиксной форме в списке формальных параметров указывается дополнительный, неиспользуемый в алгоритме операции, параметр (точнее, тип параметра).

Примечание. Компилятор корректно обрабатывает перегруженную операцию и в случае явного указания дополнительного, неиспользуемого в алгоритме операции, параметра.

Пример: Для класса complex перегрузим операцию *'++'* в префиксной и постфиксной формах со следующей семантикой:

```
complex c1, c2;
. . .
c1 = ++c2; // c2 = c2 + 1; c1 = c2;
c1 = c2++; // c1 = c2; c2 = c2 + 1;

complex & operator++(){    // Префиксная форма
   ++re;
   return *this;
}

complex operator++(int){ // Постфиксная форма
   complex tmp(*this);   // Во временном объекте
                         // запоминается состояние
                         // текущего объекта.
   re++;                 // Изменяется текущий объект.
   return tmp;           // Во внешнюю среду выдается
                         // запомненное состояние в
                         // виде объекта, а не ссылка.
                         // Во внешнюю среду не может
                         // быть выдана ссылка на
                         // текущий объект, т.к. он
                         // изменил свое состояние.
}
```

Примечание. Приведенная форма перегрузки префиксной операции *'++'* позволяет корректно выполнять следующую операцию:

c1 = ++ ++c2;

с семантикой:

c2 = c2 + 2; c1 = c2;

Однако, приведенная форма перегрузки постфиксной операции *'++'* не позволяет корректно выполнить операцию

c1 = c2++ ++;

с семантикой

c1 = c2; c2 = c2 + 2;

Дело в том, что первое исполнение операции *'++'* передает во внешнюю среду неизмененный объект *c2*, а второе исполнение операции *'++'*, принимая на входе неизмененное значение объекта, передает его объекту *c1*, параллельно изменяя его на 1, а не на 2.

Операция индексирования является бинарной: ее операнды – объект с нумерованными элементами (массив, вектор и т. д.) и целое число – индекс элемента.

Примечание. В некоторых пособиях операция индексирования ошибочно рассматривается как унарная, хотя явно имеются два вышеуказанных операнда.

При перегрузке операции индексирования объявление в качестве выходного параметра ссылки на элемент объекта позволяет присваивать ее результату новые значения.

Пример:

```
class vector {
      int* p;
      int size;
   public:
      . . .
      int & operator[](int i){return p[i];}
};

int main(){
   vector v1(10);
   v1[1] = 5;
}
```

Здесь оператор ***return*** возвращает значение выбранного элемента вектора, который инициализирует выходной объект, как было описано, по реализации адресом этого элемента.

Примечание. Перегрузка операций (как бинарных, так и унарных) позволяет не только описывать для стандартных операций необходимую семантику, но и блокировать исполнение нежелательных операций над объектами описываемого типа. Для этого необходимо перегрузку операции описывать в открытой области**.** Естественно, что такая перегрузка описывается методом класса, а не функцией-другом.

8.3. Перегрузка функций

Имеется возможность описывать разные алгоритмы для одного и того же идентификатора функции при разных количествах и наборах типов входных параметров. Такое описание разных алгоритмов в одной зоне описания (класс, пространство имен) называется **перегрузкой функций** (если описание разных алгоритмов для одного и того же имени осуществляется в разных зонах, то говорят о **перекрытии**).

При вызове функции для выбора подходящей перегруженной функции выполняется следующий алгоритм:

Алгоритм поиска оптимально отождествляемой функции

1) Отбираются функции с необходимым количеством формальных параметров.

Примечание. Возможно описание функции с переменным числом параметров. Для этого используются символы `'. . .'` в конце списка формальных параметров, обозначающих произвольное количество дополнительных неименованных параметров, типы которых будут определяться непосредственно при вызове функции. Пример прототипа функции с переменным числом параметров:

```
void f1(int i1, . . .);
```

В этом случае внутри функции необходимо иметь специальные средства получения значений таких дополнительных параметров, не имеющих собственных имен. При обработке списка формальных параметров компилятор не имеет информации, необходи-

мой для выполнения стандартной проверки и преобразования типов неименованных параметров. Поэтому средства получения таких параметров могут использовать только информацию, недоступную компилятору. Для облегчения работы с этими параметрами в файле *stdarg.h* стандартной библиотеки имеются описания структуры *va_list* и функций: *va_start()*, *va_arg()*, *va_end()*. Описания, содержащиеся в библиотечном файле <*cstdarg*>, становятся доступными после его подключения директивой препроцессора ***#include*** <*cstdarg*>.

Функции с переменным числом параметров рекомендуется использовать в исключительных случаях, когда типы параметров действительно неизвестны. В большинстве случаев можно использовать функции с аргументами по умолчанию или функцией с двумя параметрами следующего вида: первый параметр – целое число, равное количеству содержательных параметров, второй аргумент – указатель на массив указателей на фактические параметры. Такой метод используется при передаче списка строковых параметров из командной строки вызова программы на исполнение:

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

Здесь *args* – количество строковых параметров в командной строке вызова программы на исполнение, включая идентификатор программы.

2) Для каждого фактического параметра вызова функции строится множество функций, оптимально отождествляемых по этому параметру (best matching)

3) Находится пересечение этих множеств

4) Если полученное множество состоит из одной функции, то вызов разрешим. Если множество пусто или содержит более одной функции, то генерируется сообщение об ошибке.

Пример:

```
class x{
   . . .
   public:
      x(int i1){. . . }
      . . .
};

class y{. . .};

void f(x x1, int i1){. . .}
void f(x x1, double d1){. . .}
void f(y y1, double d1){. . .}
void g(){. . . f(1,1) . . .}  // вызов первой реализации
f(x, int)
```

Пример:

```
class x{
   . . .
   public:
```

```
        x(int i1){. . . }
        . . .
};

void f(x x1, int i1){. . .}
void f(int i1, x x1){. . .}
void g(){. . . f(1,1) . . .}  // ошибка: пересечение
                              // множеств – пусто.
```

8.4. Алгоритм поиска оптимально отождествляемой функции для одного параметра

Если функция имеет один параметр, то выполняется следующая последовательность шагов для поиска оптимально отождествляемой функции.

Такая же последовательность шагов выполняется для каждого параметра функции с несколькими параметрами на втором этапе ранее описанного алгоритма поиска оптимально отождествляемой функции с несколькими параметрами.

1). **Точное** отождествление.

2). Отождествление с помощью **расширений.**

3). Отождествление с помощью **стандартных преобразований.**

4). Отождествление с помощью **преобразований пользователя.**

5). Отождествление по **'...'**.

Данные шаги определяют приоритет метода отождествления параметра. В рамках каждого шага разные виды преобразований являются равно приоритетными.

Развернутое описание пунктов 1). – 5).

1). Точное отождествление

1.1). **Точное** совпадение

1.2). Совпадение с точностью до **typedef**

1.3). Тривиальные преобразования:

```
T[] <-> T*
T   <-> T&
T    -> const T
```

Пример:

```
void f(float);
void f(double);
void f(int);
void g(){ . . . f(1.0F); . . . f(1.0); . . . f(1);}
```

2). Отождествление с помощью **расширений**

2.1). Целочисленные расширения

$$\left.\begin{cases} char \\ unsigned\ short \\ signed\ short \\ bool \\ enum \end{cases}\right\} \rightarrow \text{int}$$

Примечание. Если фактический параметр невозможно преобразовать к типу ***signed int*** **без потери информации**, то осуществляется его преобразование к типу ***unsigned int***. Дело в том, что на три основных целочисленных типа – ***short***, ***int*** и ***long***, в самых распространенных компиляторах отводится два основных размера – 2 и 4 байта. При этом, в разных компиляторах одинаковый размер имеют разные целочисленные типы. Так, в компиляторе Borland C++ 3.1 типы ***short*** и ***int*** имеют размер 2 байта, а ***long*** – 4 байта. В компиляторах Borland C++ 5.02 и Microsoft Visual C++ 6.0 тип ***short*** имеет размер 2 байта, а типы ***int*** и ***long*** – 4 байта. Поэтому, если типы ***short*** и ***int*** имеют одинаковый размер, то преобразование ***unsigned short*** -> ***signed int*** – невозможно.

2.2). Расширения с плавающей точкой.

float -> **double**

3). Отождествление с помощью **стандартных преобразований**

3.1). Остальные стандартные целочисленные и вещественные преобразования

Примечание. При этом тип ***char*** **-** целочисленное значение размером в 1 байт:

Пример:

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

void f(char c){
   int i;
   union U {
      char c1;
      unsigned b:8;
   };
   U u1;
   u1.c1 = c;
   cout << u1.b << '\n';
   i = c;
   cout << i << '\n';
}
```

```
int main() {
    f(-1);
    return 0;
}
```

В выходной поток будет выдано:

255 // 8-битное представление числа -1, преобразованное в беззнаковое целое
-1 // расширенный до целого дополнительный код, представляющий -1

3.2). Преобразование указателей

Константа 0 при преобразовании параметров рассматривается не только как число, но и как нулевой указатель. Нулевой указатель может быть преобразован в любой указатель. Любой указатель может быть преобразован в так называемый **обобщенный (свободный) указатель *void****. При этом обратное преобразование должно выполняться явно.

Пример:

```
int main() {
    int i;
    void* p;                    // Обобщенный указатель
    int *ip = &i;               // Указатель на int,
                                // инициализируемый адресом
                                // целочисленной переменной
    int* ip2;                   // Указатель на int
    *ip = 15;                  // Разыменование указателя на
                               // int
    p = ip;              // Допустимое неявное преобразование
                         // указателя
    ip2 = (int*)p;            // Явно заданное преобразование
                              // указателя
    . . .
    return 0;
}
```

В связи с тем, что тип ***char****, используемый для работы с динамически размещаемыми строками, является указателем, то при наличии перегруженных функций с прототипами:

```
int f(char* c1);
int f(double d1);
```

вызов `i = f(0);` будет неоднозначным, так как при отождествлении с помощью стандартных преобразований следующие преобразования:

0 -> **double** (**int** -> **double**)
0 -> **char*** (пустой указатель -> указатель на **char**)

равноправны.

Указатель на объект производного класса может быть преобразован в указатель на объект базового класса. О наследовании классов будет подробно рассказано далее.

Пример:

```
class X {
      int x1;
   public:
      X():x1(0){}
};

class Y: public X {
      int y1;
   public:
      Y(): y1(0){}
};

void f(X * x2){}

int main() {
   Y * y2;
   f(y2); // фактический параметр – указатель на объект
          // производного класса, а формальный параметр
          // – указатель на объект базового класса
   return 0;
}
```

Замечание 1. Через указатель на объект базового класса доступны только члены производного класса, унаследованные от базового класса. Это связано с тем, что, как было отмечено, указатели являются типизированными, то есть, в указателе хранится не только адрес объекта, но и информация о структуре объекта.

Замечание 2. Как будет описано далее, неявное преобразование указателей возможно только при открытом (***public***) – наследовании.

4). Преобразования пользователя

4.1). Конструктор преобразования

Конструктор преобразования может быть использован в качестве неявного преобразования пользователя только в том случае, если он объявлен без ключевого слова ***explicit***

4.2). Функция преобразования (**операция преобразования**)

Функция преобразования – это метод класса с прототипом:

operator *тип*();

Здесь *тип* – тип возвращаемого значения (встроенного или пользовательского).

Тело данной функции определяет возвращаемое значение. Тело функции должно содержать оператор ***return***, возвращающее данное значение. При этом возвращаемое значение преобразуется к указанному типу. Если тип является пользовательским, то создается временный объект данного типа. При этом вызывается конструктор преобразования, определенный в этом классе. Возвращаемое функцией значение будет входным параметром для этого конструктора.

Функция преобразования выполняет действие, обратное действию конструктора преобразования: если конструктор преобразования создает объект типа описываемого класса на основе объекта другого типа, то функция преобразования возвращает объект другого типа на основе объекта (внутренней информации, хранящейся в объекте) описываемого класса.

Сходством функции преобразования и конструктора преобразования является то, что у функции преобразования так же, как и у конструктора преобразования нет типа возвращаемого значения. Тип возвращаемого значения у функции преобразования определяется типом, указанным после ключевого слова ***operator***.

Кроме функции преобразования ключевое слово ***operator*** используется также при перегрузке операций, о чем уже было рассказано.

При использовании преобразования пользователя строятся все возможные цепочки преобразований параметра, которые позволяют применить одну из реализаций перегруженной функции. При этом выбирается та реализация, для которой в цепочке преобразований шаг преобразования с минимальным приоритетом имеет максимальный приоритет по сравнению с другими цепочками. Так, если в одной цепочке преобразований шагом с минимальным приоритетом будет шаг 4 (преобразование пользователя), а в другой цепочке – шаг 3 (стандартные преобразования), то будет выбрана реализация функции, для которой цепочка преобразований параметра содержит шаг 3.

Пример:

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

class S {
      long ss1;
   public:
      S():ss1(0){}
      S(long p):ss1(p){
         cout << "constructor S -> ";
      }
      operator int(){
         cout << "S.operator int() -> ";
         return ss1;
      }
};

void f(long p){
   cout << "f(long)" << '\n';
}
```

```
void f(char* p){
   cout << "f(char*)" << '\n';
}

void g(S p){
   cout << "g(S)" << '\n';
}

void g(char* p){
   cout << "g(char*)" << '\n';
}

void ex(S& a) {
   f(a);
   g(1);
   g(0);
}

int main(){
   S s1;
   ex(s1);
   return 0;
}
```

Последовательности преобразований параметров при вызове функций *f* и *g* в функции *ex()*:

```
f(a): s.operator int(): шаг 4. Преобразование пользователя
       int -> long  : шаг 3. Стандартное преобразование
       f(long)

g(1): int -> long  :  шаг 2. Расширение (целочисленное)
      s.constructor(long): шаг 4. Преобразование пользователя
      g(s)

g(0):  0 -> char*  :  шаг 3. Стандартные преобразования
                          (преобразования указателей – константа 0
                          преобразуется в указатель)
       g(char*)
```

Замечание 1. Пользовательские преобразования применимы неявно только в том случае, если они однозначны.

Пример:

```
class B {
   int i1;
   public:
      B(int p):i1(p){}
      operator int(){ return i1; }
      B operator+(const B & pb){
```

```
                B tmp(0);
                tmp.i1 = i1 + pb.i1;
                return tmp;
        }
};

int main(){
   B x(0);
   x = x + 1; // неоднозначность:  возможно
              // x.operator int()+1 либо
              // x.operator+(B(1))
   return 0;
}
```

Замечание 2. Допустимо не более одного пользовательского преобразования **для одного параметра.**

Пример:

```
class X {
      int x1;
   public:
      operator int(){
          return x1;
      }
      X(int px):x1(px){}
};

class Y {
      int y1;
   public:
      operator X(){
         X tmp(y1);
         return tmp;
      }
      Y():y1(0){}
};

int main(){
    Y a;
    int b;
    b = a;     // ошибка: требуется
               // a.operator X().operator int()
   return 0;
}
```

5). Отождествление по `'...'`

Использование символов `'...'` в конце списка формальных параметров в описании функции с переменным числом параметров для обозначения оставшихся неявных параметров было описано ранее.

Пример:

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

class X {
      double x1;
   public:
      X(double px):x1(px){}
};

void f(int i1, X x2){
   cout << "f(int, X)" << '\n';
}

void f(int i1, ...){
   cout << "f(int, ...)" << '\n';
}

int main(){
   f(1,1);
   f(1,"Test");
   return 0;
}
```

Использование символов `'. . .'` может привести к неоднозначности. Например,

```
void f(int i1);
void f(int i1, . . .);
```

9. Виды отношений между классами

Ассоциация представляет смысловую связь между сущностями (объектами), создаваемыми на основе классов. *Ассоциация (association)* определяется некоторой связью между классами. Когда в системе создаются представители ассоциированных классов, они связываются так, как определяет данная ассоциация.

Ассоциации между классами разрабатываются в процессе так называемого **семантического моделирования**: моделирования структуры данных исходя из их смысла. Для этого полезно использовать **ER-диаграммы** (Entity – Relationship: Сущность – Связь).

Примечание. ER-диаграммы используются в разных аспектах проектирования сложных программных комплексов, причем, не только комплексов, разрабатываемых с использованием объектно-ориентированной парадигмы, но и при разработке баз данных и во многих других приложениях.

Основными понятиями, используемыми при построении ER-диаграмм, являются:

1. **Сущность** – класс однотипных объектов, информация о которых должна быть учтена в модели. Сущность в рамках ООП представляется классом.
2. **Экземпляр сущности** – объект, создаваемый на основе класса.
3. **Атрибут сущности** – именованная характеристика. В ООП – информационный член класса
4. **Ключ сущности** – совокупность атрибутов, однозначно определяющих объект.
5. **Связь** – ассоциация между сущностями.

Типы связей:
- один к одному
- один ко многим
- многие ко многим

Пример связи: группа – студенты. Связь может иметь одну из двух модальностей:
- может (может быть, а может и не быть)
- должен

При разработке ER-модели определяется следующая информация:
- Список сущностей
- Список атрибутов
- Описание связей

Связи между сущностями реализуются с помощью механизмов наследования, агрегирования, использования.

Наследование – отношение между классами, при котором один класс повторяет структуру и поведение другого класса (*одиночное наследование)* или других (*множественное наследование*) классов.

Класс, поведение и структура которого наследуется, называется базовым (родительским) классом, а класс, который наследует – производным классом.

В производном классе структура и поведение базового класса (информационные члены и методы), дополняются и переопределяются. В производном классе указываются только дополнительные и переопределяемые члены класса. Производный класс является уточнением базового класса:

```
class z: public y{. . . };
```

Агрегация – это отношение между классами типа целое/часть. Агрегируемый класс в той или иной форме является частью агрегата. Объект класса-агрегата может хранить объект агрегируемого класса, или хранить ссылку (указатель) на него.

Пример:

```
class node { . . .};// агрегируемый класс, описывающий
                    // вершину дерева
class tree {       // класс-агрегат, описывающий дерево.
```

```
      node* root;  // единственным информационным членом
                   // является указатель на выделенную
                   // вершину – корень дерева
   public:
      tree(){root = 0;}
      . . .
};
```

Композиция является специальным видом агрегирования (так называемое сильное агрегирование). Композиция объектов заключается в использовании объектов типов разработанных классов в качестве информационных членов при описании других классов.

Пример:

```
class point{
        int x,y;
    public:
        point(){ . . .}
        point(int x1, int y1){ . . .}
    . . .
};

class z1{
        point p;
        int z;
    public:
        z1(int z2){ . . . }
    . . .
};

z1* z3 = new z1(1);
```

Использование – отношение между классами, при котором один класс в своей реализации использует в той или иной форме реализацию объектов другого класса.

Использование одним классом объектов другого класса может проявляться одним из следующих образов:

- Имя одного класса используется в профиле метода другого класса
- В теле метода одного класса создаётся локальный объект другого класса
- Метод одного класса обращается к методу другого класса (не совсем частный случай предыдущего способа использования, так как при вызове статических членов классов локальный объект не создаётся).

10. Одиночное наследование

10.1. Правила наследования

Наследование является одним из трех основных механизмов ООЯП. В результате использования механизма наследования осуществляется формирование иерархических связей между описываемыми типами. Тип-наследник уточняет базовый тип.

Прототип объявления типа-наследника:

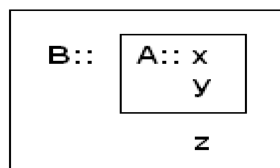
$$\begin{bmatrix} class \\ struct \end{bmatrix} \text{ имя типа} : \begin{bmatrix} public \\ protected \\ private \end{bmatrix} \text{имя базового типа} \begin{Bmatrix} \text{описание} \\ \text{членов класса} \end{Bmatrix}$$

Пример:

```
struct A {
   int x,y;
};

struct B: A {
   int z;
};

A a1;
B b1;
b1.x = 1;
b1.y = 2;
b1.z = 3;
a1 = b1;
```

Объект типа `B` наследует свойства объекта типа `A`.

Таким образом, объект типа-наследника содержит внутри себя члены базового типа:



При наследовании наследуются не только информационные члены, но и методы.

Не наследуются:

- Конструкторы
- Деструктор
- Операция присваивания

Как уже было описано, единственный способ использования конструктора базового класса – **список инициализации**. Он записывается при описании конструктора производного класса:

$$\begin{matrix} Имя_производ- \\ ного_класса \end{matrix} \begin{pmatrix} формальные \\ параметры \end{pmatrix} : \begin{matrix} Имя_базового \\ класса\ с\ парам \end{matrix} \begin{pmatrix} фактические \\ параметры \end{pmatrix} \{\ldots\}$$

При создании объекта производного типа *B* сначала будет вызван конструктор базового типа *A*. При этом если конструктору базового типа нужны параметры, то **его необходимо вызывать явно в списке инициализации**. Затем будет вызван конструктор производного типа *B*.

Деструкторы вызываются в обратном порядке. При разрушении объекта производного типа сначала будет вызван деструктор этого типа, а затем деструктор базового типа.

Допустимо присвоение объекту базового типа объекта производного типа. При этом объекту базового типа будет присвоена та часть объекта производного типа, которая структурно совпадает с базовым типом.

10.2. Преобразования указателей.

Безопасным приведением является приведение указателя на объект типа-наследника к указателю на объект базового типа.

Пусть *A* и *B* – типы из предыдущего примера. Тогда

```
A a1;
A* pa;
B b1;
B* pb;
pb = &b1;
pa = pb;  // Указателю pa присваивается адрес объекта
          // b. Т.к. указатель pa описан как указатель
          // на объект типа A, то с его помощью видна
          // только та часть объекта b, которая
          // структурно соответствует типу A.
pa = & a1;
pb = (B*)pa; // Допустимое, но небезопасное явное
             // преобразование указателя на объект
             // базового типа к указателю на объект
             // производного типа. Через указатель на
             // объект типа B можно обращаться к
             // информации, которая присутствует в
             // типе B. Однако, в структуре объекта a1
             // базового типа A отсутствуют
             // дополнительные члены производного
             // типа.
// pb -> z;  // Ошибочное обращение к таким членам после
             // указанного преобразования (ошибка во
             // время исполнения программы).
```

10.3. Правила видимости при наследовании.

Наследование свойств и поведения могут контролироваться с помощью **квалификаторов доступа**, задаваемых при наследовании: ***public***, ***protected***, ***private***. Названия квалификаторов доступа совпадают с названиями ранее описанных областей доступа в классах и структурах. Квалификаторы доступа ограничивают видимость полностью или частично для полностью или частично открытых членов. Закрытые члены всегда остаются закрытыми. При наследовании можно уменьшить видимость членов, но не расширить их видимость.

Кратко вид доступа в типе-наследнике для членов базового типа можно представить таблицей:

Квалификатор доступа \ Область доступа базового типа	**public**	**protected**	**private**
public	**public**	**protected**	**private**
protected	**protected**	**protected**	**private**
private	**private**	**private**	**private**

Замечание. Данная таблица показывает вид доступа для членов в типе наследнике для типа наследника следующего уровня. Закрытый вид доступа в типе-наследнике для закрытых членов базового типа имеет особый статус, описываемый далее.

Если не указан тип наследования, то тип наследования по умолчанию определяется описанием типа наследника . Если тип-наследник описывается классом, то тип наследования – закрытый (***private***), если же это структура, то наследование по умолчанию будет открытым (***public***).

Пример:

```
struct A {
   int x;
};

class C: A {};
int main(){
   C c;
   // c.x = 1; // ошибка: в классе C из-за закрытого по
              // умолчанию наследования поле x
              // становится закрытым.
   return 0;
}
```

Если тип-наследник описывается структурой, то наследование по умолчанию становится открытым.

Пример:

```
class A {
   public:
      int x;
   private:
      int y;
};

struct C: A {};

int main(){
   C c;
   c.x = 1;    // ошибки нет, т.к. наследование – открытое.
   return 0;
}
```

При необходимости открытого наследования членов базового типа, если тип-наследник описывается с использованием класса, следует явно указывать квалификатор ***public***:

```
class C: public A {
   int z;
};
```

Замечание. В других системах программирования, связанных, например, с языком Java, не предусмотрен квалификатор доступа при наследовании, так как в языке Java предусмотрен только открытый способ наследования.

Защищенный вид доступа (**protected**) означает, что члены базового типа в типе-наследнике доступны только для методов своего (базового) типа, а также для методов производного типа. Во всех остальных случаях они ведут себя так же, как члены с закрытым видом доступа (***private***).

Пример:

```
struct A {
   int x,y;
};

class B: protected A {
      int z;
  public:
      void putx(int ap) {
              x = ap;
      }
};
```

```
int main(){
   B b1;
   b1.putx(1);
   . . .
   return 0;
}
```

Ограничение видимости при наследовании ограничивает манипуляции с членами базового типа только в объектах типа-наследника и его потомках. Поэтому при преобразовании указателя типа-наследника к указателю на объекты базового типа работа с объектом осуществляется в соответствии с правилами видимости для базового класса.

Пусть указатель на объект типа-наследника при защищенном наследовании преобразован к указателю на объекты базового типа. Тогда работа с объектом типа-наследника с использованием указателя на объекты базового типа происходит в соответствии с правами доступа для базового типа (как уже было указано, через такой указатель виден не весь объект типа-наследника, а только его часть, соответствующая базовому типу):

```
struct A {
   int x; int y;
};

struct B: A {
   int z;
};

class C: protected A {
   int z;
};

int main(){
   A a;
   A* pa;
   B b;
   C c;
   C* pc = &c;
   b.x = 1;
   //pc -> z;   // ошибка: доступ к закрытому полю
   b.y = 2;
   //pc -> x;   // ошибка: доступ к закрытому полю
   b.z = 3;
   pa = (A*)pc; // см. примечание далее.
   a = b;
   pa -> x=4;   // правильно: поле A::x - открытое
   return 0;
}
```

Примечание. Так как в данном примере наследование – защищенное, то при присвоении указателя производного типа указателю базового типа требуется явное

преобразование (pa=(A*)pc;). При открытом наследовании возможно простое присвоение указателей (pa=pc). Это связано с тем, что указатель кроме адреса содержит информацию об объекте. При защищенном наследовании изменяется не только состав членов класса, но и права доступа.

Далее вопросы наследования будут рассмотрены на основе классов. При необходимости данные сведения могут быть переработаны для структур с учетом области доступа и квалификатора доступа при наследовании в структурах по умолчанию.

10.4. Закрытое (private) наследование

Закрытые члены базового класса недоступны напрямую с использованием дополнительных методов класса-наследника (при любом способе наследования). Работа внутри класса-наследника с такими получаемыми закрытыми членами базового класса возможна только с использованием открытых и защищенных методов базового класса.

Закрытые и защищенные получаемые методы недоступны для манипулирования с объектом вне класса. Они могут использоваться как подпрограммы другими методами класса.

При закрытом наследовании открытые и защищенные члены базового класса (любые) доступны только внутри производного класса и недоступны извне (через объекты производного класса), как и его собственные закрытые члены.

Таким образом, приведенная таблица показывает вид доступа для членов в типе наследнике для типа наследника следующего уровня. Но, для текущего типа-наследника доступность зависит от вида доступа в базовом типе.

Пример:

```
class X1{
      int ix1;
   public:
      int f1(){ . . . }
      . . .
};

class Y1: protected X1 {
. . .
};

class Z1: public Y1 {
. . .
};

class X2{
   protected:
      int ix2;
   public:
      int f2(){ . . . }
      . . .
};
```

```
class Y2: X2 {. . .};
class Z2: public Y2 {. . .};
```

В классе *Y1* переменная *ix1* недоступна непосредственно, так как она в базовом классе *X1* находится в закрытой области. Однако она может быть использована в функции *f1()*. В то же время функция *f1()* не может использоваться в качестве внешнего метода по отношению к объекту, созданному на основе класса *Y1*, так как она находится в защищенной области класса *Y1*. Это же остается справедливым и для класса *Z1*. В классе *Y2* переменная *ix2*, в отличие от переменной *ix1* в классе *Y2*, доступна непосредственно. В классе *Z2* переменная *ix2* становится недоступной для непосредственного использования, так же, как и переменная *ix1* в классе *Z1*. Другие отличия классов *Z1* и *Z2*: в отличие от функции *f1()* в классе *Z1*, функция *f2()* в классе *Z2* доступна в классе *Z2* качестве внутренней подпрограммы, только для функций, унаследованных из класса *Y2*, так как в классе *Y2* она находится в закрытой области.

Закрытое наследование целесообразно в том случае, когда меняется сущность нового объекта.

Пример: Базовый класс описывает фигуры на плоскости и имеет методы вычисления площади фигур, а класс-наследник описывает объемные тела, например, призмы с основанием – плоской фигурой, описываемой базовым классом. В этом случае объем тела, описываемого классом-наследником, вычисляется умножением площади основания на высоту. При этом не имеет значения, каким образом получена площадь основания. Кроме того, методы работы с объемными объектами отличны от методов работы с плоскими объектами. Поэтому в данном случае не имеет смысла наследование методов базового класса для работы с объектами, описываемыми классом-наследником:

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

class twom {
    double x,y;
public:
    twom(double x1=1, double y1=1): x(x1), y(y1) {}
    double sq(){
          return x*y;
    }
};

class thm: private twom {
    double z;
public:
    thm(double x1 = 1, double y1 = 1,
        double z1 = 1):twom(x1,y1), z(z1){}
    double vol(){return sq()*z;}
};
```

```
int main(){
    thm t1(1,2,3);
    double d1;
    d1 = t1.vol();
    cout << "vol=  " << d1 << '\n';
    return 0;
}
```

Таким образом, закрытое наследование несколько напоминает **композицию** объектов, когда подобъект находится в закрытой области. Все же необходимо помнить, что наследование – это совсем другая концепция ассоциирования классов, по многим своим свойствам отличная от агрегации, даже в ее строгом варианте (композиции).

10.5. Перекрытие имен

В производном классе могут использоваться имена членов класса, перекрывающие видимость таких же имен в базовом классе (**overriding**). При перекрытии имен при работе с объектом через указатель будет исполняться тот метод, который содержится в классе, используемом в объявлении указателя, независимо от типа объекта, на который указывает указатель.

Пример:

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

class A {
   public:
      void f(int x){
           cout << "A::f" << '\n';
      }
};

class C: public A{
   public:
      void f(int x){
           cout << "C::f" << '\n';
      }
};

int main(){
   A a1;
   A* pa;
   C c1;
   C* pc;
   pc =  &c1;
   pc -> f(1);     // C::f
   pa = pc;
   pa -> f(1);   // A::f – несмотря на то, что pa указывает
                 // на объект c1 типа класс C.
```

```
    pc = (C*) & a1; // Небезопасное преобразование указателя
                    // на объект базового класса к указателю
                    // на объект производного класса (данное
                    // преобразование должно объявляться
                    // явно, иначе - ошибка).
    pc -> f(1);     // C::f - несмотря на то, что pa указывает
                    // на объект a1 типа класс A. В общем
                    // случае такой вызов некорректен.
    return 0;
}
```

Члены базового класса с именами, совпадающими с именами членов производного класса, доступны в производном классе. Для доступа к ним необходимо указывать квалификатор (имя базового класса) с использованием операции *'::'*, так как данные члены находятся в доступной области видимости, которая не совпадает с текущей областью видимости. Также метод базового класса доступен через указатель класса-наследника при условии использования квалификатора.

Пример:

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

 class A {
    public:
        void f(int x){cout<<"A::f"<<'\n';}
 };

 class C: public A{
    public:
        void f(int x){
            cout << "C::f" << '\n';
        }
        void g(){
            f(1);
            A::f(1);
        }
 };

int main(){
    C c1;
    C* pc;
    pc=&c1;
    pc->A::f(1); // вызов метода базового класса с
                 // использованием указателя
                 // класса-наследника.
    pc->f(1);
    pc->g();
    return 0;
}
```

Таким образом, при перекрытии методы базового класса не «затираются» в классе-наследнике. Они доступны через квалификатор.

10.6. Наследование и повторное использование кода

Повторное использование кода предполагает выделение некоего фрагмента кода в процедуру, которая может вызываться из различных модулей программы.

Наследование – наиболее удобный механизм для повторного использования кода. Суть наследования состоит в том, что в иерархии классов выделяется базовый класс, реализующий функциональность, общую для всех классов-наследников. Классы-наследники наследуют эту функциональность, а также при необходимости реализуют некоторые дополнительные функции, специфические для этих классов-наследников.

В случае открытого (***public***) наследования открытые методы базового класса, остаются открытыми методами классов-наследников, таким образом, интерфейс базового класса расширяется в классе-наследнике. В случае закрытого (***private***) наследования методы базового класса не могут использоваться в классе-наследнике в качестве методов интерфейса, но могут использоваться как подпрограммы для выполнения некоторых базовых действий (их можно вызывать из других доступных методов производного класса).

Поэтому **закрытое наследование** называется **наследованием реализации** в противоположность **наследованию интерфейса** (открытое наследование).

Наследование реализации можно эффективно применять при построении схем отношений в сложных системах. Например, совокупность базовых классов может описывать обобщенное двоичное отсортированное дерево с обобщенными вершинами:

```
class bnode {      // класс, описывающий обобщенную вершину
   friend class gen_tree;
   friend void out(bnode* n); // функция-друг для вывода
                              // данных, хранящихся в
                              // вершине
   bnode* left;
   bnode* right;
   void* data;       // обобщенный указатель на информацию
                    // в вершине
   int count;           // счетчик повторений значения во
                        // входном потоке
   bnode(void* d, bnode* l, bnode* r):
                    data(d), left(l), right(r), count(1){}
};

class gen_tree{                     // класс, описывающий
                                    // обобщенное дерево
   protected:
      bnode* root;                  // ссылка на корень
      void* find(bnode* r, void* d) const;
      void print(bnode* r) const;
   public:
```

```
        gen_tree(){root=0;}
        void insert(void* d); // Ввод данных в вершины
                                 // дерева.
                        // От реализации этой функции
                        // зависит упорядоченность данных
                        // в дереве.
                        // Эта функция в своей работе
                        // использует функцию find для
                        // поиска данных в дереве
        void* find(void* d) const {
                return (find(root, d));
        }
        void print() const {print(root);} // Вывод данных.
                        // От реализации этой функции
                        // зависит вывод данных,
                        // хранящихся в дереве.
};
```

Примечание. Возможная реализация методов описанных классов:

```
int comp(void* a, void* b);
     // объявление функции для сравнения данных,
     // хранящихся в вершинах. Конкретная реализация
     // функции зависит от типа данных, хранящихся
     // в конкретном дереве, класс для которого является
     // наследником класса обобщенного дерева.

void gen_tree::insert(void * d){
   bnode* temp = root;
   bnode* old;
   if (root == 0){
      root = new bnode (d, 0, 0);
      return;
   };
   while (temp != 0){
      old = temp;
      if (comp (temp -> data, d) == 0){
         (temp -> count)++;
         return;
      };
      if (comp (temp -> data, d) > 0)
         temp = temp -> left;
      else
         temp = temp -> right;
   };
   if (comp (old -> data, d) > 0)
      old -> left  = new bnode (d, 0, 0);
   else
      old -> right = new bnode (d, 0, 0);
}
```

```
void* gen_tree::find (bnode * r, void* d) const {
   if (r == 0)
      return 0;
   else if (comp ( r -> data, d) == 0)
      return r -> data;
   else if (comp (r -> data, d) > 0)
      return find (r -> left,  d);
   else
      return find (r -> right, d);
}

void gen_tree::print (bnode * r) const {
   if (r != 0) {
      print (r -> left);
      out (r);
      print (r -> right);
   };
}
```

На основе базового класса обобщенного дерева можно создать класс-наследник для хранения данных конкретного типа. При этом основные алгоритмы будут реализованы в базовом классе. В классе-наследнике эти алгоритмы используются путем вызова соответствующих методов, реализованных в базовом классе, с передачей им необходимых параметров, соответствующих данным конкретного вида.

Приведем представление класса для хранения данных типа символьных строк, а также реализацию функций *comp()* для сравнения таких данных и *out()* для вывода данных конкретного типа, хранящихся в вершине дерева:

```
class s_tree: private gen_tree {
   public:
      s_tree () {}
      void insert (char * d) {gen_tree::insert (d);}
      char * find (char * d) const
      {
         return (static_cast<char *>(
                                 gen_tree::find (d)));
      }
      void print() const {gen_tree::print ();}
};

int comp (void * i, void * j){
   return (
           strcmp (static_cast<char*> (i),
                   static_cast<char*> (j)));
}

void out (bnode * r) {
   cout << static_cast<char*> (r -> data) << " : ";
   cout << r -> count << "   ";
}
```

Примечание. *strcmp()* – стандартная процедура сравнения символьных данных, содержащаяся в библиотечном файле *string.h*.

При так описанной реализации методов *insert()* и *print()* в базовом классе данные из дерева выводятся в отсортированном виде. Так, при следующем наборе операций ввода в дерево:

```
s_tree s1;
s1.insert("4");
s1.insert("2");
s1.insert("1");
s1.insert("3");
s1.insert("6");
s1.insert("5");
s1.insert("7");
s1.print();
```

В выходной поток будет выдана следующая строка:

```
1 : 1    2 : 1    3 : 1    4 : 1    5 : 1    6 : 1    7 : 1
```

Если реализация методов *print()* будет следующей:

```
int lev;

void gen_tree::print0 () const {
   lev=0;
   print0 (root);
}

void gen_tree::print (bnode* r) const {
   int pp;
   if (r != 0){
      out (r);
      lev ++;
      print0 (r -> right);
      cout << '\n';
      for (pp = 0; pp < lev; pp ++)
         cout << "        ";
      print0 (r -> left);
      lev --;
   };
}
```

То дерево будет выведено в виде иерархической структуры, повернутой на 90°:

```
4 : 1    6 : 1    7 : 1
                  5 : 1
         2 : 1    3 : 1
                  1 : 1
```

11. Динамический полиморфизм, механизм виртуальных функций

В C++ введено понятие **виртуальных функций (методов)**. Механизм виртуальных методов заключается в том, что, результат вызова виртуального метода с использованием указателя или ссылки зависит не от того, на основе какого типа создан указатель, а от типа объекта, на который указывает этот указатель.

Тип данных (класс), содержащий хотя бы одну виртуальную функцию, называется **полиморфным типом (классом)**, а объект этого типа – **полиморфным объектом**.

Таким образом, при вызове виртуальной функции через указатель на полиморфный объект осуществляется динамический выбор тела функции в зависимости от текущего тела объекта, а не от типа указателя. Тело функции в таком случае выбирается на этапе выполнения, а не компиляции. В этом и проявляется динамический полиморфизм.

Замечание. В языке C++ виртуальные методы классов существуют наряду с невиртуальными методами. В некоторых объектно-ориентированных языках программирования, например, в языке Java, все методы в иерархиях классов являются виртуальными.

Виртуальная функция объявляется описателем **virtual**. Во всех классах-наследниках наследуемая виртуальная функция остается таковой (виртуальной). Таким образом, все типы-наследники полиморфного типа являются полиморфными типами.

Пример:

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

class A {
   public:
      virtual void f (int x) {
           cout << "A::f" << '\n';
      }
};

class C: public A{
   public:
      void f (int x) {
           cout << "C::f" << '\n';
      }
};

int main(){
   A a1;
   A* pa;
   C c1;
```

```
    C* pc;
    pc = & c1;
    pc -> f (1);        // C::f
    pa = pc;
    pa -> f (1);        // C::f
    pc = (C*) & a1;
    pc -> f (1);        // A::f
    return 0;
}
```

11.1. Виртуальные деструкторы

Виртуальный деструктор – важная часть аппарата динамического полиморфизма. Дело в том, что, если указатель типа базового класса указывает на объект производного класса, то при удалении объекта с использованием данного указателя в случае невиртуальности деструкторов сработает деструктор того типа, который был использован при объявлении указателя. При описании конструкторов и деструкторов уже было указано, что при удалении объекта во вторую очередь срабатывает деструктор базового типа, удаляя информационные члены базового типа, унаследованные в типе-наследнике, а сначала срабатывает деструктор текущего объекта, удаляя дополнительные члены типа-наследника. Таким образом, деструктор базового типа применяется к объектам производного типа. Но при его локальном срабатывании не будут удалены дополнительные члены типа-наследника.

Таким образом, при создании и удалении объектов производных типов с использованием указателей необходимо описывать деструкторы как виртуальные, если типы-наследники в своем составе имеют динамические структуры. При этом:

1) Виртуальный деструктор необходим и для объекта без динамических структур в случае наличия динамических структур у типа-наследника, так как деструктор, автоматически генерируемый системой по умолчанию, является невиртуальным.

2) Несмотря на то, что имя деструктора производного класса отличается от имени деструктора базового класса, достаточно объявления деструктора виртуальным только в базовом классе.

3) Конструктор, в отличие от деструктора, нельзя описывать как виртуальный, так как всегда срабатывает конструктор именно того типа, который используется при создании объекта, и только после создания объекта его адрес передается для присвоения указателю.

Пример:

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

class A {
      int* pa;
```

```
        int ia;
    public:
        A(int par1 = 10) {
            ia = par1;
            pa = new int [par1];
            cout << "A()  ";
        }
        virtual ~A () {
            delete [] pa;
            cout << "~A()  ";
        }
        virtual int sa () {
             return ia;
        }
};

class C: public A{
        double* pc;
        int ic;
    public:
        C (int par2 = 10, int par1 = 10): A (par1) {
            ic = par2;
            pc = new double [par2];
            cout << "C()  ";
        }
        ~ C () {
            delete [] pc;
            cout << "~C()  ";
        }
        virtual int sc () {
             return ic;
        }
};

int main(){
    A* pp1 = new C( 5);
    // . . .
    cout << '\n' << "size int = " << pp1 -> sa ();
    cout << "     size double = ";
    cout << ((C*) pp1) -> sc () << '\n';
    delete pp1;
    return 0;
}
```

Примечание. Комментарий по работе программы.

1) В начале работы создается объект типа класса *C* (при этом через объявленный указатель будут доступны только члены, унаследованные от базового класса *A*). При создании объекта сначала срабатывает конструктор базового класса, создавая целочисленный массив. Так как для этого массива фактического параметра нет, то в качестве

размера массива берется значение по умолчанию, заданное в конструкторе базового класса. Затем срабатывает конструктор класса *C*, создавая дополнительный вещественный массив.

2) В конце работы удаляется созданный объект. Так как деструктор базового класса объявлен виртуальным, то сначала срабатывает деструктор текущего объекта, удаляя вещественный массив, а затем – деструктор базового класса, удаляя целочисленный массив. В результате работы данной программы будет выдана следующая информация:

```
A()   C()

size int = 10  size double = 5

~C()   ~A()
```

3) Если бы деструктор базового класса не был объявлен виртуальным, то при удалении объекта в соответствии с типом указателя *pp1* для объекта был бы вызван только деструктор базового класса. В результате вещественный массив остался бы неудаленным, и была бы выдана информация:

```
A()   C()

size int = 10  size double = 5

~A()
```

4) Несмотря на то, что метод *sc()* в классе *C* является виртуальным, он недоступен напрямую через указатель *pp1*, так как этого метода нет в структуре класса *A*. Поэтому для вызова этого метода для созданного объекта через указатель *pp1* требуется преобразование:

```
((C*)pp1)->sc()
```

С другой стороны, если бы в классе *C* был описан метод с прототипом: **int** *sa()*, то он был бы виртуальным и по операции *pp1 -> sa()* сработал бы его алгоритм, а не алгоритм метода *sa()*, объявленного в базовом классе.

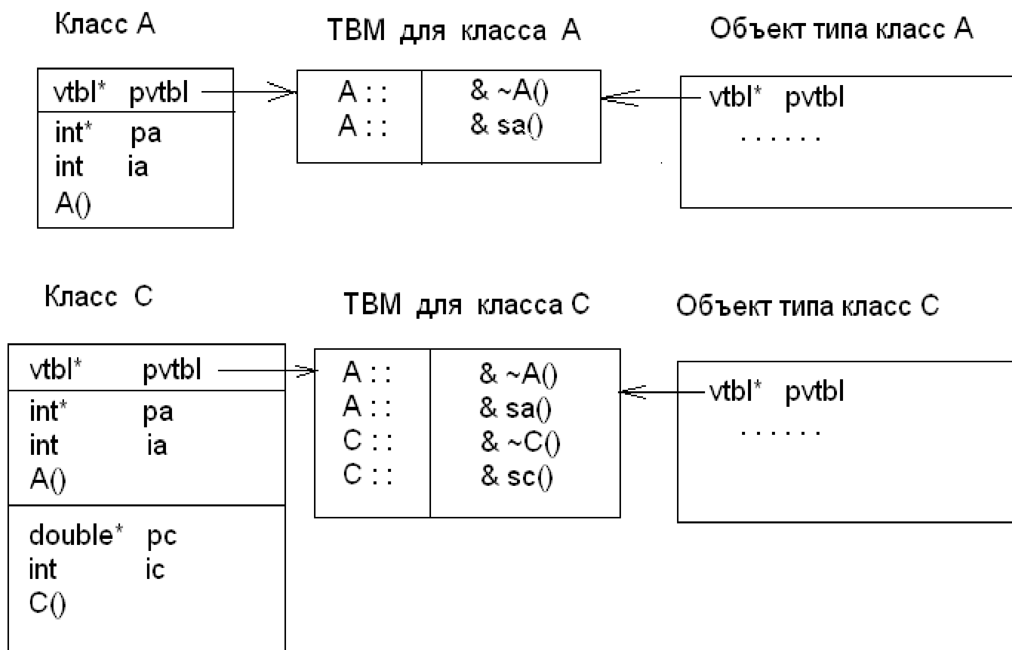
11.2. Реализация виртуальных функций

Для реализации механизма виртуальных функций используется специальный, связанный с полиморфным типом, массив указателей на виртуальные методы класса. Такой массив называется **Таблицей Виртуальных Методов (ТВМ).** В каждый полиморфный объект компилятор неявно помещает указатель, условно обозначаемый как

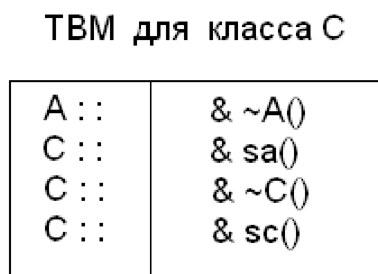
```
vtbl* pvtbl ,
```

на соответствующую ТВМ, хранящую адреса виртуальных методов.

Для указанного выше примера с виртуальными функциями будут созданы следующие структуры:



В ТВМ типа-наследника имеющиеся адреса одинаковых методов замещаются, а новые – дописываются в конец. Так, если бы в классе *C* был описан метод с прототипом ***int*** *sa()*, то ТВМ для класса *C* имела бы вид:



Так как указатель на ТВМ находится в самом начале объекта, то он доступен всегда, каким бы ни был тип указателя на объект. Конечно, при этом из ТВМ могут быть выбраны только те методы, которые имеются в структуре указателя (входят в так называемый интерфейс), То есть, как показано в примере, если объект производного типа обрабатывается через указатель базового типа, то из ТВМ данного объекта можно вызывать только виртуальные методы, перечисленные в базовом типе. Естественно, при этом будет выполняться алгоритм, определенный для данного объекта в соответствии с его типом.

Издержками при использовании виртуальных функций является дополнительная память для неявного хранения указателя на ТВМ в каждом полиморфном объекте.

11.3. Абстрактные классы. Чистые виртуальные функции

Чистая виртуальная функция – функция вида:

virtual *Тип_возвращаемого значения* *Имя функции* (*формальные параметры*) = 0;

Такая форма записи функции означает, что данная функция (точнее – метод класса) не имеет тела, описывающего ее алгоритм.

Абстрактный класс – это класс, содержащий хотя бы одну чистую виртуальную функцию.

Нельзя создавать объекты на основе абстрактных классов, так как последние, имея в своем составе чистые виртуальные функции, не являются полноценными типами данных. Однако указатели на абстрактные классы создавать можно.

Несмотря на то, что абстрактный класс не является полноценным типом, ТВМ для него создается. При этом в ТВМ перечисляются все виртуальные функции, в том числе и чистые виртуальные функции.

В классе-наследнике чистая виртуальная функция может быть переопределена обычной виртуальной функцией с соответствующей заменой пустого значения на адрес данной функции в ТВМ класса-наследника.

Класс-наследник абстрактного класса может также быть абстрактным классом, если в нем осталась (или была дополнительно введена) хотя бы одна чистая виртуальная функция.

12. Средства обработки ошибок, исключения и обработка исключений

Аппарат обработки ошибок позволяет создавать надежно работающие программы, которые не завершаются аварийно в случае возникновения неожиданных аварийных ситуаций. Такими аварийными ситуациями являются, например, отсутствие необходимого файла, блокировка записи информационной базы и т. д.

Для обработки исключительных ситуаций используются ключевые слова ***try***, ***catch*** и ***throw***.

Часть программы, при выполнении которой необходимо обеспечить обработку исключительных ситуаций, помещается в ***try***-блок. В программе может быть произвольное количество произвольно вложенных друг в друга ***try***-блоков.

Если ошибка возникла внутри некоторого внутреннего ***try***-блока (либо в вызванной из него функции), то с помощью инструкции ***throw*** возбуждается исключительная ситуация. Сигнал о таком возбуждении будет перехвачен и обработан в теле функции-обработчика, расположенном после ключевого слова ***catch***. Обработчики помещаются сразу после ***try***- блока. При возникновении исключительной ситуации управление передается в подходящий обработчик и в ***try***-блок не возвращается (действительно, при возникновении подобной ошибки продолжение исполнения части алгоритма, содержащей данную ошибку, нецелесообразно, а часто, и просто бессмысленно).

Основные формы использования ***try***, ***catch***, ***throw***:

```
try {
    . . .
    throw исключ_ситуация;
    . . .
}
catch(type)       { /* . . . throw . . . */}
   . . .
catch(type arg)   { /* . . . throw . . . */}
   . . .
catch(. . .)      { /* . . . throw . . . */}
```

Параметром инструкции ***throw*** является **исключительная ситуация** – объект некоторого типа, в частности, встроенного. Так, в качестве исключительной ситуации может быть диагностическое сообщение (то есть, строка) или число.

Если обработчик не может до конца обработать исключительную ситуацию, то инструкция ***throw*** без параметров передает обработку объемлющему ***try***-блоку (так называемый **перезапуск** перехваченного исключения).

12.1. Правила выбора обработчика исключения

При возникновении исключительной ситуации обработчики catch просматриваются в порядке их следования. Срабатывает тот обработчик ***catch***, тип параметра которого соответствует типу исключительной ситуации в инструкции ***throw***.

Обработчик с параметром базового типа перехватывает исключительную ситуацию типа-наследника. Поэтому обработчик с параметром типа-наследника должен быть объявлен раньше объявления обработчика с параметром базового типа.

Аргумент *arg* при начале работы обработчика получает значение исключительной ситуации, указываемой в инструкции ***throw***. Его можно использовать в теле обработчика. **Обработчик *catch(...)*** перехватывает **все** исключительные ситуации. Поэтому он должен быть последним в списке обработчиков.

Если для исключительной ситуации не описан подходящий обработчик после соответствующего блока ***try***, то данная ситуация перехватывается обработчиком для объемлющего ***try***-блока.

Если подходящий обработчик не найден во всех объемлющих ***try***-блоках, то выполняется стандартная функция *terminate()*, которая по завершении работы вызывает стандартную функцию *abort()*, аварийно завершающую работу программы.

При описании функции можно указать типы исключительных ситуаций (**спецификация исключений**), которые может возбуждать функция:

Тип_возвращаемого значения *Имя функции* (*формальные параметры*) ***throw*** (*список типов*);

Если список после ключевого слова ***throw*** пуст, то функция не может возбуждать исключительных ситуаций с помощью инструкции ***throw*** (ни прямо, ни косвенно).

Пример:

```
void f1() throw(int, over_flow);

void noexcp(int i) throw();
```

Если в прототипе функции отсутствует ключевое слово ***throw***, то функция может возбуждать любое исключение.

При нарушении спецификации исключений, выполняется стандартная функция *unexpected()*, аварийно завершающая программу.

Блок ***try*** может содержать фрагмент программы, в котором объявляются объекты. При этом в конструкторах классов для данных объектов могут возбуждаться исключительные ситуации в случае невозможности корректного создания объекта по передаваемым параметрам. В этом случае в подходящем обработчике для данного ***try***-блока возможно корректно обработать ситуацию невозможности создания объекта.

Пример:

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

class vect{
      int* p;
      int size;
   public:
      vect (int n):size (n){
         if (n < 1)
               throw n;
         p = new int [n];
         if (p == 0)
               throw "no free memory";
      };
      ~vect () {
               delete[] p;
      }
     //  . . .
};

int g(int m){
   try {
         vect a(m);
   }
   catch (int n){
      cout << "error of size " <<n << '\n';
```

```
        return 1;
    }
    catch (const char* er){
        cout << er << '\n';
        return 2;
    };
    //  . . .
    return 0;
}

int main(){
    int ierr;
    int m;
    //  . . .
    ierr = g (5);
    . . .
    ierr = g (0);
    //  . . .
    ierr = g (m);
    //  . . .
    return 0;
}
```

12.2. Стандартные исключения.

Стандартные исключения – типы данных (классы или структуры), описывающие некоторые предопределенные исключительные ситуации, в частности, они могут составлять иерархию типов.

Стандартные исключения могут быть включены в состав компиляторов C++ или поставляться вместе со стандартной библиотекой. Обычно, соответствующие классы и структуры содержатся в текстовых заголовочных файлах, например, в *<exception>*, *<excpt.h>* и других аналогичных, которые подключаются явно или неявно директивой препроцессора ***#include***.

Стандартные исключения, описанные классом или структурой, содержат внутренние информационные члены, которые могут быть проанализированы, а также собственные конструкторы, деструкторы и необходимые методы.

12.3. Последовательность действий при возникновении исключительной ситуации.

1) Создание временного объекта – копии исключительной ситуации.

2) Уничтожение объектов, созданных в ***try***-блоке, с запуском для них необходимых деструкторов, освобождающих динамическую память (**свертка стека**).

3) Выход из ***try***-блока.

4) Подбор и выполнение обработчика для данного **try**-блока в соответствии с типом исключительной ситуации (**статическая ловушка**).

5) Если необходимый обработчик для данного ***try***-блока не найден или в обработчике имеется инструкция ***throw*** без параметров, сигнализирующая о незавершенности обработки исключительной ситуации, то происходит выход в объемлющий ***try***-блок с повторением пунктов 2-4 (**динамическая ловушка**).

6) Если исключительная ситуация осталась необработанной после выхода из всех объемлющих ***try***-блоков, то вызывается функция *terminate()*.

13. Множественное наследование, интерфейсы

Множественное наследование возникает, когда имеется несколько базовых типов и один тип – наследник. При множественном наследовании появляется возможность моделирования более сложных отношений между типами.

```
class X { . . . };

class Y { . . . };

class Z: public X, public Y { . . . };
```

При описании производного класса каждый базовый класс имеет свой собственный описатель типа наследования (явно указанный или неявно предполагаемый) .

13.1. Видимость при множественном наследовании.

При множественном наследовании возникает проблема неоднозначности из-за совпадающих имен в базовых классах.

Пример:

```
struct X {
   int i1;
   int jx;
};

struct Y {
   int i1;
   int jy;
};

struct Z: X, Y {
   int jz;
};

int main(){
   Z z1;
   z1.i1 = 5;  // ошибка - неоднозначность: член i1
               // наследуется как из базового типа X, так
```

```
                    // и из базового типа Y.
    return 0;
}
```

Тем не менее, данная неоднозначность проявляется не при объявлении объекта типа-наследника, а при использовании его членов, имеющихся в нескольких базовых типах. Эту неоднозначность можно обойти при помощи операции разрешения области видимости:

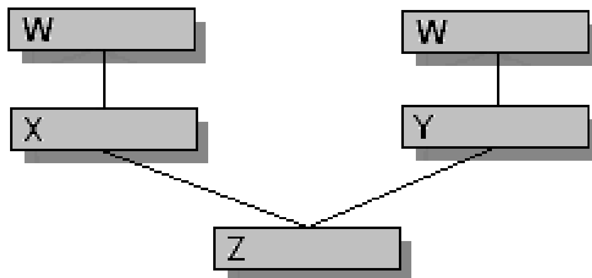
```
z1.X::i1 = 5
```

Таким образом, в тип-наследник попадают все члены базовых типов. Но повторяющиеся имена необходимо сопровождать квалификатором базового типа.

13.2. Виртуальные базовые классы

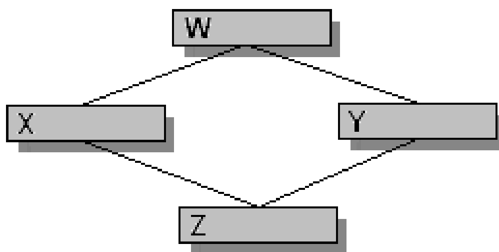
При многоуровневом множественном наследовании базовые классы могут быть получены от общего предка. В этом случае итоговый производный класс будет содержать несколько подобъектов общего предка:

```
class W
{ . . .};

class X: public W
{ . . . };

class Y: public W
{ . . . };

class Z: public X, public Y
{ . . . };
```



Если необходимо, чтобы общий предок присутствовал в итоговом производном классе в единственном экземпляре (например, если необходимо, чтобы функции классов *X* и *Y* в классе *Z* использовали общие информационные члены класса *W*, или для экономии оперативной памяти), то наследование базовых классов от общего предка описывается с использованием виртуального наследования:

```
class W
{ . . .};

class X: public virtual W
{ . . . };

class Y: public virtual W
{ . . . };

class Z: public X, public Y
{ . . . };
```



13.3. Интерфейсы

Указанная неоднозначность при множественном наследовании отсутствует, если все базовые классы являются абстрактными классами без информационных членов и содержат только открытые чистые виртуальные функции. Такие базовые классы называются **интерфейсами**. Действительно, если с объектом работать через указатель такого класса, то набор чистых виртуальных функций данного класса определяет, какие методы объекта доступны через этот указатель.
Класс-наследник классов-интерфейсов, если он не является абстрактным классом (то есть, не содержит ни одной чистой виртуальной функции), называется **классом реализации**. На основе класса реализации создаются конкретные объекты. Работа с такими объектами осуществляется с использованием указателей типа классов-интерфейсов.

14. Динамическая информация о типе (RTTI).

RTTI (Run Time Type Identification) – механизм безопасного преобразования типов объектов. Этот механизм включает:

- **dynamic_cast** – операция преобразования типа указателя или ссылки на полиморфные объекты
- **typeid** – операция определения типа объекта
- **type_info** – структура, содержащая информацию о типе объекта (данная структура содержится в библиотечном файле `typeinfo.h`).

Точнее, `type_info` – это класс, описывающий тип данных, значениями которого является информация о типе исследуемого объекта. Операция ***typeid*** возвращает ссылку на объект типа ***const*** *type_info*. Таким образом, ***typeid*** является встроенной операцией, но для его корректного использования в программу должна быть подгружена библиотечный файл `typeinfo.h`.

Для текстового представления имени типа объекта в классе `type_info` имеется функция `name()`.

Примечание. Так как операция ***typeid*** возвращает значение типа ***const*** *type_info &*, то нецелесообразно явно создавать объект такого типа. Обычно такой объект используется неявно, то есть, используется создаваемый временный объект.

Синтаксис операции динамического приведения типа (***dynamic_cast***):

dynamic_cast < *тип_результата* > (*выражение*);

тип_результата – указатель или ссылка.

Выражение – указатель, если *тип_результата* является указателем, или объект (или ссылка на объект), если *тип_результата* является ссылкой.

Пример:

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

class A{
   public:
      virtual void fx (){
           cout << "A::fx" << '\n';
      }
};

class B: public A{
   public:
      void fx (){
           cout << "B::fx" << '\n';
      }
};
```

```
void f2(A* ptr){
   B* dptr = dynamic_cast<B*> (ptr);
   cout<<"type of pointer dptr:  ";
   cout<<typeid(dptr).name()<<'\n';
}

int main(){
   A* p1 = new A;
   f2 (p1);
   return 0;
}
```

В результате работы данной программы будет выведена следующая строка:

```
type of pointer dptr: class B *
```

Примечание. Для обеспечения возможности использования механизма RTTI в компиляторе, как правило, необходимо указать специальный параметр, так как обычно в целях повышения эффективности работы программы механизм RTTI по умолчанию отключен. Так, в компиляторе Microsoft Visual C++ 6.0 таким параметром является /GR.

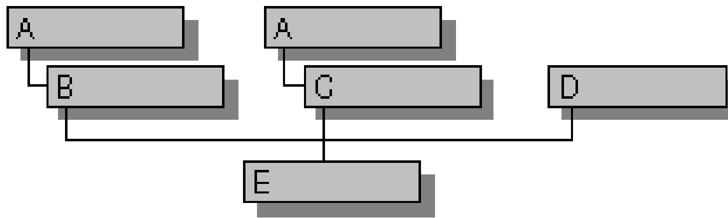
Динамическое приведение типов с помощью операции динамического приведения типа (***dynamic_cast***) возможно только для объектов родственных полиморфных классов, относящихся к одной иерархии классов. Указатель может иметь нулевое значение, поэтому при динамическом приведении указателя в случае возникновения ошибки может возвращаться это нулевое значение, которое затем может быть проверено в программе. Ошибка при приведении ссылки всегда приводит к возбуждению исключительной ситуации ***bad_cast***, так как никакого выделенного значения для ссылок не существует. Проверка правильности динамического приведения ссылок всегда выполняется перехватом исключительной ситуации**.** ***Bad_cast*** – класс, описывающий исключительную ситуацию. Так же, как и класс ***type_info***, он содержится в библиотечном файле **<*typeinfo*>**.

Пример:

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

class A{
   public:
      virtual void fx () {
           cout << "A::fx" << '\n';
      }
};

class B: public A{
   public:
      void fx () {
           cout << "B::fx" << '\n';
```

```
        }
};

class C: public A{
    public:
        void fx () {
            cout << "C::fx" << '\n';
        }
};

void f (A* p, A& r){
        if (B* pp = dynamic_cast<B*> (p)){
            cout << "using of pp" << '\n';
                /* использование указателя pp */
        }
        else {
            cout << "NULL" << '\n';
        /* указатель pp не принадлежит нужному типу */
        }
        B& pr = dynamic_cast<B&> (r);
        /* использование ссылки pr */
}

void g(){
    try {
        cout << "f (new B, * new B) - correct using" << '\n';
        f (new B, *new B); // правильный вызов
        cout << "f (new C, * new C) - incorrect using";
        cout << '\n';
        f (new C, *new C); // выход в перехватчик (C – из
                           // другой иерархии, основанной
                           // на том же базовом классе)
    }
    catch (bad_cast){
        cout << "Bad_cast" << '\n';
        // обработка исключительной ситуации
    }
}

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

Однако корректность преобразования, зависит не только от типов указателей и ссылок, но и от типов соответствующих объектов. Пусть наследование классов представлено следующей схемой:



Тогда, если объект создан на основе класса *E*, то указатель на него, имеющий тип указателя на базовый класс *B*, может быть преобразован в указатель на базовый класс *C*. Кроме того, если объект создан на основе класса *E*, то указатель на него, имеющий тип указателя на базовый класс *D*, может быть преобразован в указатель на базовый класс *B*, несмотря на то, что классы *B* и *D* не имеют общего базового класса. Это связано с тем, что объект, созданный на основе класса, находящегося на нижней ступени иерархии, содержит в своей структуре структуру всех выше расположенных классов.

Пример:

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

class A{
   public:
      virtual void fx () {
           cout << "A::fx" << '\n';
      }
};

class B: public A{
   public:
      void fx () {
           cout << "B::fx" << '\n';
      }
};

class C: public A{
   public:
      void fx () {
           cout << "C::fx" << '\n';
      }
};

class D {
   public:
      virtual void fd () {
           cout << "D::fd" << '\n';
      }
};
```

```
class E: public B, public C, public D {};

void f (C* p, C& r){
   if (B* pp = dynamic_cast<B*> (p)){
      cout << "using of pp in f" << '\n';
   }
   else {
      cout << "NULL in f" << '\n';
   }
   B& pr = dynamic_cast<B&> (r);
   /* использование ссылки pr */
}

void f2 (D* p, D& r){
   if (B* pp = dynamic_cast<B*> (p)){
      cout << "using of pp in f2" << '\n';
   }
   else {
      cout << "NULL in f2" << '\n';
   }
   B& pr = dynamic_cast <B&> (r);
   /* использование ссылки pr */
}
void g(){
   try {
      cout << "f(new E, *new E)" << '\n';
      f (new E, *new E);
   }
   catch (bad_cast){
      cout << "Bad_cast in f" << '\n';
      // обработка исключительной ситуации
   };
   try {
      cout << "  f2 (new E, *new E)" << '\n';
      f2 (new E, *new E);
   }
   catch (bad_cast){
      cout << "Bad_cast in f2" << '\n';
      // обработка исключительной ситуации
   };
}

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

В приведенном примере не будет возбуждена ни одна из двух исключительных ситуаций, а указатели будут иметь ненулевые значения.

Статическое приведение типов (операция ***static_cast***) возможно для объектов родственных классов (полиморфность типов, участвующих в операции, при этом может отсутствовать), относящихся к одной иерархии классов. Статическое приведение возможно также для свободных указателей (***void****), которые могут преобразовываться в значения любых типов указателей, и *для преобразований между арифметическими типами*.

Однако при этом невозможно проверить корректность преобразований с использованием исключительной ситуации *bad_cast*. Кроме того, невозможно преобразование указателя к указателю из другой ветви иерархии наследования, основанной на том же базовом классе, даже если объект создан на основе класса, являющегося наследником классов рассмотренных указателей. Так, если в приведенном выше примере динамическое преобразование позволяет преобразовать указатель типа *C** к указателю *B**, если объект является объектом типа класс *E*, то статическое преобразование не дает возможности преобразовать указатель типа *C** к указателю *B**.

Примечание. Кроме динамического и статического преобразований имеются и другие преобразования. Например, преобразование ***reinterpret_cast*** (управляет преобразованиями между произвольными несвязанными типами. Результатом является значение нового типа, состоящее из той же последовательности битов, что и аргумент преобразования). Константное (***const_cast***) преобразование аннулирует действие модификатора ***const*** (ни статическое, ни динамическое преобразования действие модификатора **const** аннулировать не могут). Подробнее о преобразованиях см. в литературе по С++, например, в [12].

15. Параметрический полиморфизм

Параметрический полиморфизм позволяет применить один и тот же алгоритм к разным типам данных. При этом тип является параметром тела алгоритма. Механизм шаблонов, реализующий параметрический полиморфизм, позволяет легче разрабатывать стандартные библиотеки.

Шаблон представляет собой предварительное описание функции или типа, конкретное представление которых зависит от параметров шаблона. Так, если необходимо написать функции нахождения суммы элементов числовых массивов разных типов (например, ***int***, ***float*** или ***double***), то вместо трех различных функций можно написать один шаблон.

15.1. Параметры шаблона

Для описания шаблонов используется ключевое слово ***template***, вслед за которым указываются аргументы (формальные параметры шаблона), заключенные в угловые скобки. Формальные параметры шаблона перечисляются через запятую, и могут быть как именами объектов, так и параметрическими именами типов (встроенных или пользовательских). Параметр-тип описывается с помощью служебного слова ***class*** или служебного слова ***typename***.

В соответствии со Стандартом ISO 1998 C++ (см. [2], раздел 14.1.4) параметром шаблона может быть:

- параметрическое имя типа
- параметр-шаблон
- параметр одного из следующих типов:
  - ✓ интегральный
  - ✓ перечислимый,
  - ✓ указатель на объект любого типа (встроенного или пользовательского) или на функцию
  - ✓ ссылка на объект любого типа (встроенного или пользовательского) или на функцию
  - ✓ указатель на член класса, в частности, указатель на метод класса

Интегральные типы (раздел 3.9.1 Стандарта):

- знаковые и беззнаковые целые типы,
- ***bool***, ***char***, ***wchar_t***

Примечание. Перечислимые типы не относятся к интегральным, но их значения приводятся к ним в результате целочисленного повышения.

При использовании типа в качестве параметра перед параметром, являющимся параметрическим именем типа, необходимо использовать одно из ключевых слов: либо ***class*****,** либо ***typename***.

Примечание. Не все компиляторы обрабатывают вложенные шаблоны (например, Microsoft* Visual C++ 6*: Visual Studio 98*). Тем не менее, шаблонный класс может использоваться в качестве значения параметра по умолчанию. Далее будет рассмотрен шаблонный класс со следующим прототипом:

```
template <class T , class A=allocator <T> > class vector
{ . . .};
```

В разделе 14.1.7 Стандарта явно сказано, что нетиповый параметр не может иметь тип ***void***, пользовательский тип, или плавающий тип, например:

```
// template <double  d> class X{ . . . };  // ошибка
   template <double* d> class X{ . . . };  // OK
   template <double& d> class X{ . . . };  // OK
```

Таким образом, параметром шаблона не может быть формальный параметр нецелочисленного типа, например, ***float*** или ***double***. Это можно объяснить тем, что основным назначением формальных параметров в качестве параметров шаблона является определение числовых характеристик параметрических типов (например, размер вектора, стека и так далее).

Вместе с тем параметром шаблона может быть указатель, в том числе и на объект нецелочисленного типа. При этом, естественно, указатель – фактический параметр может содержать как адрес одного объекта, так и массива однородных объектов.

Использование шаблонов позволяет создавать универсальные алгоритмы, без привязки к конкретным типам.

15.2. Шаблоны функций.

$$\mathbf{template}\ \left\langle \begin{matrix} \textit{Список_параметров} \\ \textit{шаблона} \end{matrix} \right\rangle \ \begin{matrix} \textit{Тип_возвращаемого} \\ \textit{значения} \end{matrix} \quad \begin{matrix} \textit{Имя} \\ \textit{функции} \end{matrix} \left(\begin{matrix} \textit{формальные} \\ \textit{параметры} \end{matrix} \right) \{..\};$$

Пример:

```
template <class T> T sum(T array[], int size){
   T res = 0;
   int i;
   for (i = 0; i < size; i++ )
      res += array[i];
   return res;
}
```

При обращении к функции-шаблону после имени функции в угловых скобках указываются фактические параметры шаблона - имена реальных типов или значения объектов:

$$\begin{matrix} \textit{Имя} \\ \textit{функции} \end{matrix} \left\langle \begin{matrix} \textit{Список_фактич.} \\ \textit{параметров_шаблона} \end{matrix} \right\rangle \left(\begin{matrix} \textit{фактические} \\ \textit{параметры} \end{matrix} \right);$$

Для конкретного использования приведенного выше шаблона для массивов целых чисел размерности 10 следует написать:

```
int iarray[10];
int i_sum;
//...
i_sum = sum <int> (iarray, 10);
```

Встретив такую конструкцию, компилятор сгенерирует конкретную реализацию функции sum с учётом специфицированных параметров (такая функция называется **порожденной функцией**).

Так как допускается параметризовать шаблоны не только именами типов, но и объектами, то аргумент *size* можно указать в виде параметра шаблона:

```
template <class T, int size> T sum (T array[]) { /* ... */ }
```

Тогда вызов sum будет выглядеть соответственно:

```
i_sum = sum <int, 10> (iarray);
```

Замечание. Следует отметить, что использование шаблонов сокращает текст программы, но не сокращает программный код. В программе реально будет сгенерировано столько порожденных функций, сколько имеется вызовов функций с разными наборами фактических параметров шаблона.

15.3. Специализация шаблонной функции

По сути, при использовании функции-шаблона используется механизм, позволяющий автоматически перегружать функции компилятором. Однако функцию-шаблон можно перегрузить и явно.

Если шаблонный алгоритм является неудовлетворительным для конкретного типа аргументов или неприменим к ним, то можно описать обычную функцию, список типов аргументов и возвращаемого значения которой соответствуют объявлению шаблона. Такая, перегружающая шаблон, функция, называется **специализацией шаблонной функции**.

Следует отметить, что при обращении к функции-шаблону после имени функции можно и не указывать в угловых скобках фактические параметрические типы шаблона. В этом случае компилятор автоматически определяет фактические параметрические типы шаблона по типам фактических параметров вызова функции. Это называется **выведением типов аргументов шаблона**. Выведение типов аргументов возможно при условии, что список фактических параметров вызова функции однозначно идентифицирует список параметров шаблона.

Так, описанную выше шаблонную функцию можно вызвать следующим образом:

```
int iarray[10];
int i_sum;
//...
i_sum = sum (iarray, 10);
```

Так как тип первого фактического параметра – ***int***, то компилятор автоматически трактует данный вызов, как

```
i_sum = sum <int> (iarray, 10);
```

Если параметр шаблона можно вывести из более, чем одного фактического параметра функции, результатом каждого выведения должен быть один и тот же тип. Иначе вызов будет ошибочным.

Пример:

```
template <class T> void f(T i, T* p){ /* . . . */ }
//  . . .
void g(int i){
   //  . . .
   f(i, &i); // правильный вызов
   //  . . .
   // f(i,”hello world”); - ошибка, т.к. по первому
   // параметру T – int, по второму – const char
}
```

Версия шаблона для конкретного набора фактических параметров шаблона также называется **специализацией**.

При **выведении** типов аргументов шаблона по типам фактических параметров функции нельзя применять никаких описанных выше преобразований параметров, кроме преобразований **Точного** отождествления, то есть:

- **Точное** совпадение
- Совпадение с точностью до ***typedef***
- Тривиальные преобразования:

  T[] <-> T*
  T <-> T&
  T -> ***const*** T

Пример:

```
template <class T> T max(T t1, T t2){. . .}
int main(){
   max(1,2);          // max<int>(1,2);
   max('a','b');      // max<char>('a','b');
   max(2.7, 4.9);     // max<double>(2.7, 4.9);
   //max('a',1);      // ошибка - неоднозначность,
                      // стандартные преобразования не
                      // допускаются
   //max(2.5,4);      // ошибка - неоднозначность,
                      // стандартные преобразования не
                      // допускаются
}
```

Неоднозначности не возникают при использовании явного квалификатора:

```
   max <int>('a', 1);
   max <double>(2.5, 4);
```

15.4. Алгоритм поиска оптимально отождествляемой функции (с учетом шаблонов)

С одним и тем же именем функции можно написать несколько шаблонов и перегруженных обычных функций. При этом одна специализация считается **более специализированной**, чем другая, если каждый список аргументов, соответствующий образцу первой специализации, также соответствует и второй специализации.

Алгоритм выбора перегруженной функции с учетом шаблонов является обобщением правил выбора перегруженной функции.

1. Для каждого шаблона, подходящего по набору формальных параметров, осуществляется формирование специализации, соответствующей списку фактических параметров.
2. Если могут быть два шаблона функции и один из них более специализирован, то на следующих этапах рассматривается только он (порядок специализаций описан далее).
3. Осуществляется поиск оптимально отождествляемой функции из полученного набора функций, включая определения обычных функций, подходящие по количеству параметров. При этом если параметры некоторого шаблона функции были определены путем выведения по типам фактических параметров вызова функции, то при дальнейшем поиске

оптимально отождествляемой функции к параметрам данной специализации шаблона нельзя применять никаких описанных выше преобразований, кроме преобразований **Точного** отождествления
4. Если обычная функция и специализация подходят одинаково хорошо, то выбирается обычная функция.
5. Так же, как и при поиске оптимально отождествляемой функции для обычных функций, если полученное множество подходящих вариантов состоит из одной функции, то вызов разрешим. Если множество пусто или содержит более одной функции, то генерируется сообщение об ошибке.

15.5. Шаблонные классы

Так же, как и для функций, можно описать шаблоны для классов. Механизм шаблонов при описании класса позволяет, например, обобщенно описывать множество классов, единственное отличие которых заключается в используемых типах данных.

Объявление шаблона класса:

$$\textbf{\textit{template}} \left\langle \begin{matrix} \textit{Список_параметров} \\ \textit{шаблона} \end{matrix} \right\rangle \textbf{\textit{class}} \begin{matrix} \textit{Имя} \\ \textit{класса} \end{matrix} \{\ldots\};$$

Процесс генерации объявления класса по шаблону класса и фактическим аргументам шаблона называется **инстанцированием шаблона**. Обычно он совмещается с объявлением объекта соответствующего конкретного типа. Синтаксис такого объявления:

$$\begin{matrix} \textit{Имя} \\ \textit{класса} \end{matrix} \left\langle \begin{matrix} \textit{Список_параметров} \\ \textit{шаблона} \end{matrix} \right\rangle \begin{matrix} \textit{Идентификатор} \\ \textit{объекта} \end{matrix};$$

Функции-члены класса-шаблона автоматически становятся функциями-шаблонами. Для них не обязательно явно задавать ключевое слово ***template***.

Пример: описание стека для хранения величин разных типов данных:

```
template <class T> class stack{
      T* body;
      int size;
      int top;
   public:
      stack(int sz = 10){
         size = sz;
         top = 0;
         body = new T[size];
      }
      ~stack(){delete[] body;}
      T pop(){
         top--;
         return body[top];
```

```
        }
        void push(T x){
           body[top] = x;
           top++;
        }
};
int main(){
   stack<int>     S1( 20);
   stack<char>    S2(256);
   stack<double> S3( 16);
   //  . . .
}
```

Так же, как и в шаблонах функций, использование шаблонов классов сокращает текст алгоритма, но не сокращает размер кода программы. Реально генерируется столько описаний классов, сколько было объявлений объектов с разными параметрами шаблонов.

Шаблонные классы могут иметь дружественные функции и классы. Дружественная функция, которая не использует параметры шаблона, имеется в единственном экземпляре, то есть она является дружественной для всех инстанцирований класса.

Дружественная функция, которая использует параметры шаблона, сама является шаблоном. Конкретная реализация такой функции с учётом специфицированных параметров (**порожденная функция**) является дружественной для такого инстанцирования класса, которое совпадает по типам с фактическими типами аргументов порожденной функции.

Пример:

```
template <class T> class Y{ . . .};

template <class T> class X
{
   //  . . .
   public:
     friend void f1();          // функция f1() является
                                // дружественной ко всем
                                // инстанцированиям
                                // класса
     friend Y<T> f2(Y<T> par1); // порождение функции f2()
                               // для фактического типа T
                               // является дружественной
                               // только к тому
                               // инстанцированию класса
                               // X, которое подходит по
                               // фактическому
                               // типу-параметру.
      //  . . .
};
```

```
template <class T> Y<T> f2(Y<T> par1)
{ /* . . . */ }
```

Статические члены создаются для каждого инстанцирования класса.

Пример:

```
template <class T> class X {
   static T x1;
   //  . . .
};

int X <int> :: x1 = 0;

double X <double> :: x1 = 1.5;

int main() {
   X <int> xx1;
   X <double> xx2;
   . . .
   return 0;
}
```

15.6. Эквивалентность типов

При объявлении объектов с использованием шаблона задаются фактические параметры типа. При задании одного и того же набора аргументов шаблона получается один и тот же тип объекта.

Использование ключевого слова ***typedef*** задает новое имя (синоним) для типа, никакой новый тип при этом не создается. При задании типа с использованием шаблонов эквивалентность типов проверяется с учетом синонимов (***typedef***).

Пример:

```
typedef unsigned int uint;

template <class T, int s> class vect
{ /* . . . */ };

int main(){
   vect <uint,16> v1;
   vect <unsigned int,16> v2;
   . . .
}
```

Объекты *v1* и *v2* имеют один и тот же тип.

Кроме того, эквивалентность типов определяется с точностью до вычисления константных выражений на этапе компиляции. Поэтому объекты *v1* и *v3*

```
vect <uint,16> v1;

vect <unsigned int,10+6> v3;
```

имеют одинаковый тип.

16. Стандартная Библиотека шаблонов STL

STL (Standard Template Library) является частью стандарта C++. Основные компоненты этой библиотеки – иерархии шаблонов классов и функций. Библиотека STL является важной составной частью стандартной библиотеки.

Ядро STL состоит из четырех основных компонентов:

- **контейнеры**
- **итераторы**
- **алгоритмы**
- **распределители памяти** (аллокаторы)

16.1. Контейнеры

Контейнер – тип данных (класс), предназначенный для хранения объектов какого-либо типа (возможна реализация контейнера, который хранит объекты разных типов: в этом случае в нем хранятся указатели на базовый тип для всех желаемых типов, то есть формально хранятся объекты одного типа, а фактически указатели ссылаются на элементы разных типов из одной иерархии классов).

Примерами контейнеров являются массив, дерево, список.

Стандартные контейнеры библиотеки STL

```
• Vector < T >           - динамический массив
• List < T >             - линейный список
• Stack < T >            - стек
• Queue < T >            - очередь
• Deque < T >            - двусторонняя очередь
• Priority_queue < T >   - очередь с приоритетами
• Set < T >              - множество
• Bitset < N >      - множество битов (массив из N бит)
• Multiset < T >    - набор элементов, возможно, одинаковых
• Map < key, val >    - ассоциативный массив
• Multimap < key, val > - ассоциативный массив для хранения
//                       пар «ключ-значение», где с каждым
//                       ключом может быть связано более
//                       одного значения.
```

Примечание. Строго говоря, стек, очередь, очередь с приоритетами не считаются стандартными контейнерами. Они построены с ограничениями функциональности на базе других контейнеров. Тем не менее, они включены в библиотеку STL наряду с другими стандартными контейнерами.

В каждом классе-контейнере определен набор функций для работы с этим контейнером, причем все контейнеры поддерживают стандартный набор базовых операций (функции, одинаково называющиеся, имеющие одинаковый профиль и семантику, их примерно 15-20). Например, функция `push_back()` помещает элемент в конец контейнера, функция `size()` выдает текущий размер контейнера. Основные операции включаются в следующие группы:

- Доступ к элементу
- Вставка элемента
- Удаление элемента
- Итераторы

Операции, которые не могут быть эффективно реализованы для всех контейнеров, не включаются в набор общих операций. Например, обращение по индексу введено для контейнера `vector`, но не для `list`.

Каждый контейнер в своей открытой области содержит набор определений стандартных имен типов. Среди них есть следующие имена:

```
value_type                    - тип элемента,
allocator_type                - тип распределителя памяти,
size_type            - тип, используемый для индексации,
iterator, const_iterator              - итератор,
reverse_iterator, const_reverse_iterator
                                      - обратный итератор,
pointer, const_pointer              - указатель на элемент,
reference, const_reference          - ссылка на элемент.
```

Эти имена определяются внутри каждого контейнера так, как это необходимо для соответствующего контейнера. При этом реальные типы инкапсулированы. Это позволяет писать программы с использованием контейнеров, не зависящие от типов данных, реально используемых в контейнерах.

16.2. Распределители памяти

Каждый контейнер имеет аргумент, называемый **распределителем памяти** (*allocator*), который используется при выделении памяти под элементы контейнера и предназначен для того, чтобы освободить разработчиков контейнеров, а также алгоритмов, от подробностей физической организации памяти.

Распределитель памяти обеспечивает стандартные способы выделения и перераспределения памяти, а также стандартные имена типов для указателей и ссылок. Стандартная библиотека обеспечивает стандартный распределитель памяти. Кроме того, можно задать свои распределители памяти, предоставляющие

альтернативный доступ к памяти (можно использовать разделяемую память, память со сборкой мусора, память из заранее выделенного пула и прочее).

Стандартные контейнеры и алгоритмы получают память и обращаются к ней через средства, обеспечиваемые распределителем памяти.

Стандартный распределитель памяти, задаваемый стандартным шаблонным классом *allocator* из заголовочного файла <*memory*>, выделяет память при помощи операции ***new*** и по умолчанию используется всеми стандартными контейнерами.

```
template <class T> class allocator {
   public:
      typedef T* pointer;
      typedef T& reference;
      //  . . .
      allocator() throw();
      //  . . .
      pointer allocate (size_type n);  // выделение памяти для
                                       // n объектов типа T
      void deallocate (pointer p, size_type n);
      // освобождает память для n объектов типа T,
      //деструкторы не вызываются

      void construct (pointer p, const T& val);
      // инициализация памяти, на которую указывает p,
      // значением val

      void destroy (pointer p);
      // вызывает деструктор для *p, не освобождая память
      //  ...
};
```

16.3. Итераторы

Каждый контейнер содержит итераторы, поддерживающие стандартный набор итерационных операций со стандартными именами и смыслом.

Итератор – это класс, объекты, которого по отношению к контейнерам играют роль указателей. Итераторы поддерживают абстрактную модель совокупности данных как последовательности объектов (что и представляет собой любой контейнер). Обычно, основное действие с последовательностью элементов – перебор. Он организуется с помощью итераторов. Итератор – это класс, чьи объекты выполняют ту же роль по отношению к контейнеру, которую выполняют указатели по отношению к массиву. Указатель может использоваться в качестве средства доступа к элементам массива, а итератор – в качестве средства доступа к элементам контейнера. Но, понятия "нулевой итератор" не существует. При организации цикла для последовательного обращения к элементам контейнера окончание цикла фиксируется на основе применения специальной функции для сравнения с концом последовательности элементов контейнера.

Классы итераторов и функции, предназначенные для работы с ними, находятся в библиотечном файле <*iterator*>.

Каждый контейнер содержит ряд ключевых методов, позволяющих найти концы последовательности элементов в виде соответствующих значений итераторов. Это:

`iterator begin()` – возвращает итератор, который указывает на первый элемент последовательности.

`const_iterator begin()` ***const***

`iterator end()` – возвращает итератор, который указывает на элемент, следующий за последним элементом последовательности (используется при оформлении циклов).

`const_iterator end ()` ***const***

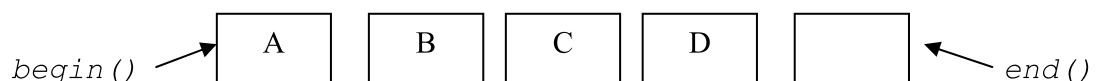
`reverse_iterator rbegin()` - возвращает итератор, указывающий на первый элемент в обратной последовательности (используется для работы с элементами последовательности в обратном порядке).

`const_reverse_iterator rbegin()` ***const***

`reverse_iterator rend()` - возвращает итератор, указывающий на элемент, следующий за последним в обратной последовательности.

`const_reverse_iterator rend ()` ***const***

«Прямые» итераторы:



«Обратные» итераторы:

Пусть *p* – объект-итератор. К каждому итератору можно применить, как минимум, три ключевые операции:

- `*p` – элемент, на который указывает итератор («разыменование» итератора),
- `p++` - переход к следующему элементу последовательности,
- `==` - операция сравнения.

Пример:

`iterator p = v.begin()`

Такое присваивание верно независимо от типа контейнера *v* . Теперь **p* – содержимое первого элемента контейнера *v*.

Замечание: при проходе последовательности как прямым, так и обратным итератором переход к следующему элементу будет *p++* (а не *p--* !).

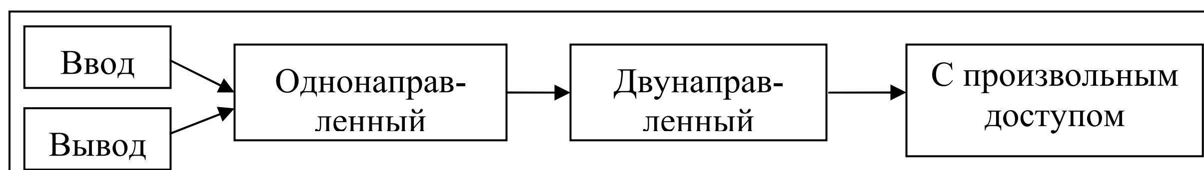
Не все виды итераторов поддерживают один и тот же набор операций.

В библиотеке STL введено **5 категорий итераторов:**

1. **Ввода** (input)
2. **Вывода** (output)
3. **Однонаправленный** (forward)
4. **Двунаправленный** (bidirectional), контейнеры *list*, *map*, *set*
5. **С произвольным доступом** (random_access), контейнеры *vector* и *deque*

Итераторы	Чтение	Доступ	За-пись	Изменение	Сравнение
Вывода			*p=e	p++ ++p	
Ввода	x=*p	p->f		p++ ++p	p==q p!=q
Однонаправ-ленные	x=*p	p->f	*p=e	p++ ++p	P==q p!=q
Двунаправ-ленные	x=*p	p->f	*p=e	p++ ++p p-- --p	P==q p!=q
Произволь-ный доступ	x=*p	p->f p[n]	*p=e	p++ ++p p-- --p p+n n+p p-n p-q p+=n p-=n	p==q p!=q p<q p>q p>=q p<=q

Каждая последующая категория, начиная с третьей, является более мощной, чем предыдущие категории.



Принято такое соглашение, что при описании алгоритмов, входящих в STL используют стандартные имена формальных параметров. В зависимости от названия итератора в профиле алгоритма, должен использоваться итератор уровня «не ниже чем». То есть по названию параметров шаблона можно понять, какого рода итератор требуется, а, следовательно, и к какому контейнеру применим алгоритм.

Пример: шаблонная функция *find()*.

Для этой функции нужны: итератор, указывающий на элемент контейнера, с которого начинается поиск, итератор, содержащий элемент, на котором поиск заканчивается, и элемент, поиск значения которого осуществляется. Для целей функции достаточно итератора ввода (из контейнера).

```
template < class InputInterator, class T >
    InputIterator find (InputIterator first,
                        InputIterator last,
                        const T& value)
   {
     while ( first != last && *first != value )
         first ++;
     return first;
   }
```

16.4. Алгоритмы

Алгоритмы STL (их около 60) реализуют некоторые распространенные операции с контейнерами, которые не реализуются методами каждого из контейнеров (например, просмотр, сортировка, поиск, удаление элементов и прочие). Такие операции являются универсальными для любого из контейнеров и поэтому находятся вне этих контейнеров. Зная, как устроены алгоритмы, можно писать необходимые дополнительные алгоритмы обработки, которые не будут зависеть от контейнера.

Каждый алгоритм представлен шаблоном функции или набором шаблонов функций. Все стандартные алгоритмы находятся в пространстве имен *std*, а их объявления – в библиотечном файле *<algorithm>*.

Можно выделить три основные **группы алгоритмов**:

1. **Немодифицирующие алгоритмы**, те, которые извлекают информацию из контейнера (о его устройстве, об элементах, которые там есть и т. д.), но никак не модифицируют сам контейнер (ни элементы, ни порядок их расположения).

Примеры:

find() – поиск первого вхождения элемента с заданным значением

count() – количество вхождений элемента с заданным значением

for_each() – для применения некоторой операции к каждому элементу, не изменяющей элементы контейнера

2. **Модифицирующие алгоритмы**, которые каким-либо образом изменяют содержимое контейнера. Либо сами элементы меняются, либо их порядок, либо их количество.

Примеры:

transform()	– для применения некоторой операции к каждому элементу, изменяющей элементы контейнера в отличие от алгоритма *for_each*
reverse()	– переставляет элементы в последовательности
copy()	– создает новый контейнер

3. **Сортировка.**

Примеры:

sort()	– простая сортировка
stable_sort()	– сохраняет порядок следования одинаковых элементов
merge()	– объединяет две отсортированные последовательности

16.5. Достоинства и недостатки STL-подхода

\+ Каждый контейнер обеспечивает стандартный интерфейс в виде набора операций, так что один контейнер может использоваться вместо другого, причем это не влечет существенного изменения кода

\+ Дополнительная общность использования обеспечивается через стандартные итераторы.

\+ Каждый контейнер связан с распределителем памяти (аллокатором), который можно переопределить с тем, чтобы реализовать собственный механизм распределения памяти.

\+ Для каждого контейнера можно определить дополнительные итераторы и интерфейсы, что позволит оптимальным образом настроить его для решения конкретной задачи.

\+ Контейнеры по определению однородны, т.е. должны содержать элементы одного типа, но возможно создание разнородных контейнеров как контейнеров, содержащих указатели на общий базовый класс.

\+ Алгоритмы, входящие в состав STL, предназначены для работы с содержимым контейнеров. Все алгоритмы представляют собой шаблонные функции, следовательно, их можно использовать для работы с любым контейнером.

\- Контейнеры не имеют фиксированного стандартного представления. Они не являются производными от некоторого базового класса. Это же верно и для итераторов. Использование стандартных контейнеров и итераторов не подразумевает никакой явной или неявной проверки типов во время выполнения.

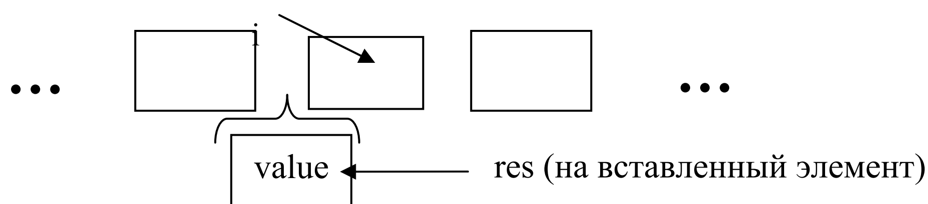
\- Каждый доступ к итератору приводит к вызову виртуальной функции. Эти затраты по сравнению с вызовом обычной функции могут быть значительными.

\- Предотвращение выхода за пределы контейнера по-прежнему возлагается на программиста, при этом каких-то специальных средств для такого контроля не предлагается.

16.6. Контейнер вектор

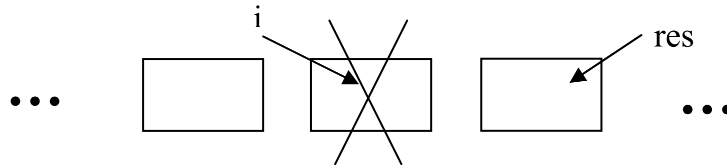
```
template <class T , class A = allocator <T> > class vector {

// vector – имя контейнера,
// T – тип элементов контейнера (value_type),
// A – распределитель памяти (allocator_type) –
       необязательный параметр.
   .......
   public:
      // Типы - typedef
      // . . .
      // Итераторы
      // . . .
      //
      // Доступ к элементам:
      //
      reference operator[] (size_type n);  // доступ без
                                           // проверки
                                           // диапазона

      const_reference operator[] (size_type n) const;

      reference at(size_type n);
      // доступ с проверкой диапазона (если индекс выходит за
      // пределы диапазона, то возбуждается исключение
      // out_of_range)

      const_reference at(size_type n) const;

      reference front();            // первый элемент вектора

      const_reference front() const;

      reference back();           // последний элемент вектора

      const_reference back() const;

      // Конструкторы:
      //
      // Конструкторы, которые могут вызываться с одним
      // параметром, для предотвращения случайного
      // преобразования объявлены explicit.
      // Это означает, что конструктор может вызываться
      // только явно:
      // vector<int>v=10 - ошибка, попытка неявного
      // преобразования числа 10 в vector<int>

      explicit vector(const A&=A());
      // конструктор умолчания – создается вектор нулевой
      // длины
```

```
explicit vector(size_type n, const T& value = T(),
                                        const A& = A());
// создается вектор из n элементов со значением value
// (или со значениями типа T, создаваемыми по умолчанию,
// если второй параметр отсутствует. В этом случае
// конструктор умолчания в классе T – обязателен)

template <class I> vector (I first, I last,
                                        const A& = A());

// I - итератор. Инициализация вектора копированием
// элемента, на который указывает итератор first, во все
// элементы из диапазона [first, last) (уже было
// отмечено, что функция end() возвращает итератор,
// который указывает на элемент, следующий за последним
// элементом).

vector(const vector < T, A > & obj);
// конструктор копирования

~vector(); // деструктор
// . . .
// Некоторые методы класса vector
//
vector& operator = (const vector < T, A > & obj);
bool empty () const {...}       //истина, если контейнер пуст

size_type size () const {...} //выдача текущего размера

iterator insert (iterator i, const T& value) {...}
      // вставка перед элементом
```

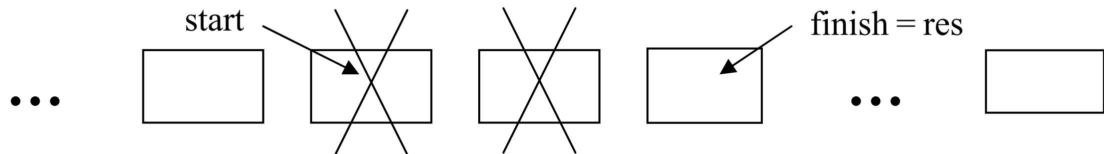



```
iterator insert (iterator i, size_type number,
                              const T & value){...}
// вставка нескольких одинаковых элементов перед элементом

void push_back (const T&value) {insert(end(),value);}
     //вставка в конец контейнера
```

```
void clear () {erase (begin(), end());}
     // уничтожение всех элементов, при этом память не
     // освобождается, так как деструктор самого вектора не
     // вызывается

iterator erase (iterator i) { ... return (res); }
     // уничтожение заданного элемента и выдача итератора
     // элемента, следующего за удалённым
```




```
iterator erase (iterator start, iterator finish)
     // уничтожение диапазона [start,finish) и выдача
     // итератора элемента, следующего за последним удалённым

{ ... return (finish);}
```




```
void pop_back () {erase (end() - 1); }
// уничтожение последнего элемента

reference front () {return * begin (); }
// содержимое первого элемента

reference  back () {return *(end () – 1); }
// содержимое последнего элемента

reference operator [](size_type i) {
     return * (begin () + i); // индексация вектора
}
reference at (size_t i) { ... } // (аналог индексации) выдает
                                // содержимое элемента i.
                                // Метод at() может возбудить
                                // исключение out_of_range.
}
```

Для организации поиска в контейнере в обратном порядке (от конца к началу) обычно пишутся такие циклы:

```
     template <class C> typename C::const_iterator find_last
     (
                       const C& c, typename C::value_type v){
          typename C::const_iterator p = c.end ();
          while (p != c.begin ()) if (* -- p == v) return p;
          return c.end ();
     }
```

Применив обратный итератор, можно воспользоваться библиотечной функцией поиска со всеми её преимуществами и без потери эффективности:

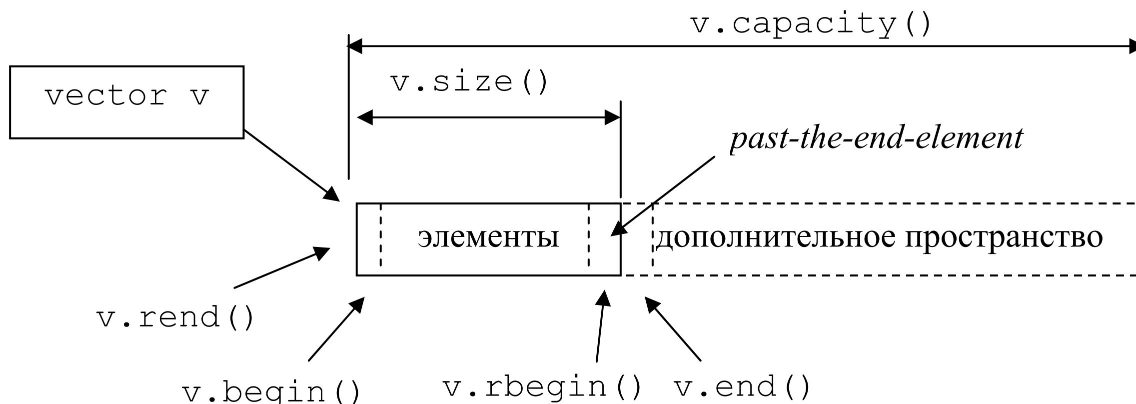
```
template <class C> typename C :: const_iterator find_last
(
                 const C& c, typename C::value_type v)
{
     typename C::const_reverse_iterator ri = find (
                            c.rbegin (), c.rend (), v);
     if (ri == c.rend ())
              return c.end ();
     typename C::const_iterator  i = ri.base ();

     return -- i;
}
```

Для обратного итератора выполнение операции *ri.base()* выдаёт значение типа *iterator*, указывающее на один элемент вперёд позиции, на которую указывает сам обратный итератор *ri*. Между итератором *i* и соответствующим обратным итератором существует фундаментальное отношение, выражающееся равенством

```
&*(reverse_iterator(i)) == &*(i - 1)
```

Операции *insert()* и *erase()* определены только для обычных итераторов, поэтому организуя циклы по обратным итераторам, эти обратные итераторы надо сначала преобразовывать к обычным, а лишь затем выполнять вставку или уничтожение элементов:

```
vector <int> :: reverse_iterator ri = v.rbegin ();

while (ri != v.rend ())
   if (* ri ++ == Element){
         vector<int>::iterator i = ri.base ();
         v.insert (i, - Element); // перед заданным
                                  // элементом
                                  // вставить ещё один,
                                  // с обратным знаком
         break;
   }
```

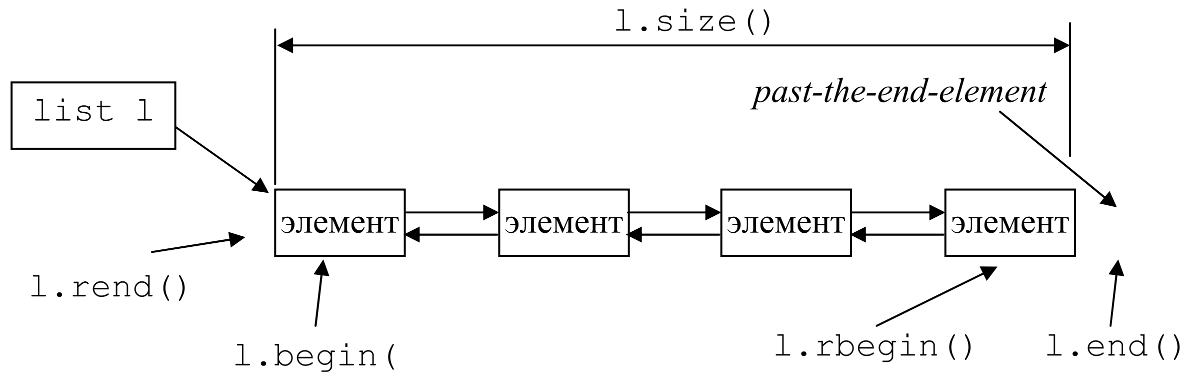


16.7. Контейнер список

Контейнер Список имеет аналогичный с контейнером список набор основных методов.

```
template <class T, class A = allocator <T> > class list {

     // list - имя контейнера,
     // T - тип элементов, которые будут храниться в списке,
     // A - распределитель памяти.
     //  ..........
   public:
      // Типы - typedef
      //  . . .
      // Итераторы
      //  . . .
      //
      // Доступ к элементам
      //
      reference front();  // первый элемент списка

      const_reference front() const;

      reference back();   // последний элемент списка

      const_reference back() const;
      //
      // Конструкторы:
      //
      explicit list(const A&=A()); // создается список
                                   // нулевой длины
      explicit list(size_type n, const T& value = T(),
                                        const A& = A());
      // создается список из n элементов со значением value
      //(или со значениями типа T, создаваемыми по умолчанию,
      // если второй параметр отсутствует. В этом случае
      // конструктор умолчания в классе T - обязателен)
```

```
    template <class I> list(I first, I last, const A& = A());
    // I - итератор. Инициализация списка копированием
    // элемента, на который указывает итератор first, во все
    // элементы из диапазона [first, last) (уже было
    // отмечено, что функция end() возвращает итератор,
    // который указывает на элемент, следующий за последним
    // элементом).

    list(const list <T,A> & obj);// конструктор копирования

    ~list(); // деструктор
    // . . .
    // Некоторые методы класса list
    // . . .
    list& operator=(const list<T,A> & obj);

    iterator erase(iterator i);
    // удаляет элемент, на который указывает данный
    // итератор. Возвращает итератор элемента, следующего
    // за удаленным элементом.

    iterator erase (iterator st, iterator fin);
    // Удаляются все элементы между st и fin, но fin не
    // удаляется. Возвращается fin.

    iterator insert(iterator i, const T& value = T());
    // вставка значения value перед i. Возвращает итератор
    // вставленного элемента.

    void insert (iterator i, size_type n, const T& value);
    // вставка n копий элементов со значением value перед
    // i.

    void push_back(const T& value); // добавляет элемент в
                                    // конец списка

    void push_front(const T& value);  // добавляет элемент
                                      // в начало списка
    void pop_back();       // удаляет последний элемент.
                           // Значение не возвращается.
    void pop_front();      // удаляет первый элемент списка

    size_type size() const;  // выдает количество элементов
                             // списка
    bool empty() const;   // возвращает значение true, если
                          // вызывающий список пуст
    void clear(); //удаляет все элементы списка
    . . .
}
```



16.8. Пример решения задачи с использованием контейнеров STL

Задача:

Написать функцию ***g()*** с тремя параметрами:

- непустой и неизменяемый контейнер-вектор типа *vector<**float**>*,
- непустой контейнер-список типа *list<**float**>*,
- целое число – шаг по первому контейнеру.

Функция должна исследовать

- элементы **списка**, выбираемые от его **конца** с **шагом 1**, и
- элементы **вектора**, выбираемые от его **начала** с **шагом**, равным **третьему параметру**.

Если обнаруживаются пары элементов **разных знаков**, то у текущего элемента **списка** должен меняться знак.

После окончания сравнения контейнеров функция должна вывести на устройство вывода изменённый список.

Функция **возвращает** общее **количество неотрицательных элементов списка**.

Решение:

```
#include <iostream>
#include <Vector>
#include <List>

using namespace std; // средства стандартной библиотеки,
                     // и, в частности, STL, определены в
                     // пространстве имен std.

typedef vector<float> V;

typedef   list<float> L;
```

```
int g(const V& vect, L& lst, int step)
{
   V:: const_iterator vp = vect.begin();
   L:: reverse_iterator lp = lst.rbegin();
   int t = 0;

   do {
     if (*lp * *vp < 0)
          *lp = -*lp;
     if (vect.end() - vp <= step)
          break;
     ++ lp;
     vp += step;
   } while (lp != lst.rend());

   L::iterator rp = lst.begin();
   while (rp != lst.end()){
      cout << *rp << ' ';
      if (*rp >= 0)
          t++;
      ++rp;
   };
   cout << endl;
   return t;
}

 int main()
 {
    V vect1(15,0.0); // описание тестовых объектов
                     // определенных типов
    L list1(15,0.0);  // вектор чисел типа float и список
                     // чисел типа float размером по 15
                     // элементов с начальным
                     // заполнением нулями.
   int i;
   . . .
   i = g(vect1,list1,1);
   cout<< '\n' << "positive elements:  " << i << '\n';
   return 0;
}
```

17. Литература

1. Standard for the C Programming Language ISO/IEC 9899, 1990.
2. Standard for the C++ Programming Language ISO/IEC 14882, 1998.
3. Архангельский А.Я. Компоненты общего назначения библиотеки C++ Builder 5. - М.: ЗАО "Издательство БИНОМ" 2001 - 416 с.
4. Архангельский А.Я. Язык Pascal и основы программирования в Delphi - М.: ООО "Бином-Пресс", 2004.
5. Бокс Д. Сущность технологии COM. Библиотека программиста. 4-е изд. - СПБ.: Питер, 2001 - 400 с.
6. Глушаков С.В. Программирование на Visual C++ - М.: ООО "Издательство АСТ"; Харьков "Феликс", 2003 - 726 с.
7. Гослинг Д., Арнольд К. Язык программирования Java. /Пер. с англ. - СПБ.: Питер, 1997 - 304 с.
8. Лясин Д.Н., Саньков С.Г. Объектно-ориентированное программирование на языке C++: Учебное пособие./ Волгоград.гос.техн.ун-т,- Волгоград, 2003 – 83 с.
9. Пол Айра Объектно - ориентированное программирование на C++. 2-е изд./Пер. с англ. - СПБ.; М.: "Невский Диалект" - "Издательство БИНОМ", 1999 - 462 с.
10. Пратт Т., Зелковиц М. Языки программирования: разработка и реализация. 4-е изд. - СПБ.: Питер, 2002 - 688 с.
11. Столяров А.В. Методы и средства визуального программирования. Раздел «введение в язык C++». М.: МГТУ ГА, 2008 – 112 с.
12. Страуструп Б. Язык программирования C++. Специальное изд./Пер. с англ. - М.: "Бином", 2005.
13. Шилдт Г. C# учебный курс. - СПБ.: Питер; К.: Издательская группа BHV, 2003 - 512 с.
14. Шилдт Г. Самоучитель C++. 3-е изд., - СПБ.: БХВ-Петербург, 2002.