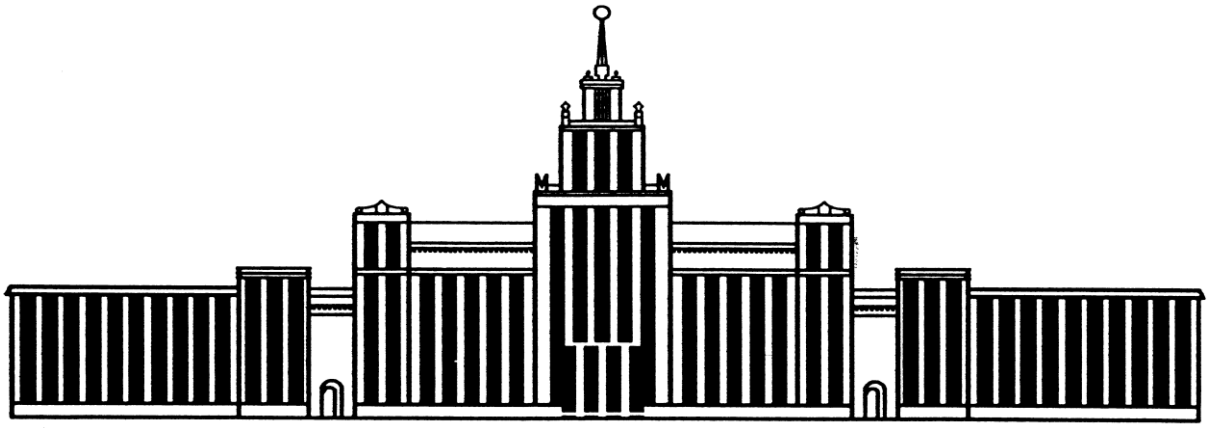
МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ



ЮЖНО-УРАЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

004.4(07)
К647

Е.А. Конова

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ С ПРИМЕРАМИ НА C++

Учебное пособие

Челябинск
2019

Министерство науки и высшего образования Российской Федерации
Южно-Уральский государственный университет
Кафедра информационных технологий в экономике

004.4(07)
К647

Е.А. Конова

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ С ПРИМЕРАМИ НА C++

Учебное пособие

Под редакцией Б.М. Суховилова

Челябинск
Издательский центр ЮУрГУ
2019

УДК 004.438 (075.8)
К647

Одобрено
учебно-методической комиссией
высшей школы экономики и управления.

Рецензенты:
И.В. Сафронова, А.М. Лоскутов

Конова, Е.А.

К647 Объектно-ориентированное программирование с примерами на C++: учебное пособие / Е.А. Конова; под ред. Б.М. Суховилова. − Челябинск: Издательский центр ЮУрГУ, 2019. − 161 с.

В учебном пособии излагаются основные положения теории объектно-ориентированного программирования, которые сопровождаются описанием инструментов объектно-ориентированного языка C ++, поддерживающих реализацию объектной модели. Синтаксис и семантика конструкций языка описаны подробно, иллюстрируются примерами. Приведены вопросы для самопроверки и упражнения для самостоятельной работы.

Учебное пособие подготовлено в соответствии с ФГОС ВПО 3-го поколения по направлениям 09.03.02 «Прикладная информатика» и 09.03.02 «Информационные системы и технологии».

УДК 004.438(075.8)

ВВЕДЕНИЕ

Основным содержанием пособия является изложение принципов объектно-ориентированного программирования и инструментов их реализации в языке программирования С++.

Подробный разговор об объектно-ориентированном анализе, проектировании и программировании невозможен в рамках небольшого пособия, поэтому для подготовки к изучению материала необходимо знакомство с соответствующей литературой, а именно, с содержанием книги Гради Буча «Объектно-ориентированный анализ и проектирование с примерами приложений на С++» [1].

Для изучения инструментов объектно-ориентированнного программирования требуется хорошее знание основ языка программирования С++, понимание концепции данных и механизма функций С++, а также умение использовать их при решении практических задач [2, 3, 4, 5].

В пособии подробно излагаются синтаксис и семантика инструментов разработки объектов на языке С++, рассматриваются возможности каждого из инструментальных средств и внутренние механизмы реализации. Изложение материала сопровождается примерами программного кода на классическом С++ на основе консольного приложения. Большинство примеров являются модельными, так как их простота помогает рассмотреть детали реализации конкретного инструментального средства. Примеры короткие, в сжатом виде излагают смысл отдельных положений теории. Текстовые вставки на языке С++ выделены шрифтом `Courier New`, как в интегрированной среде разработчика.

Материал излагается в порядке, соответствующем концепции объектного стиля, и усложняется по мере изложения. Чтобы научиться программировать в объектах, необходимо выполнять предложенные в пособии примеры. Рекомендации по разработке и примеры реализации кода можно найти в практикумах [8, 9].

Средства языка С++ описаны в соответствии со стандартом ISO/IEC 14882:2003. Все приведенные примеры отлажены и протестированы в интегрированной среде разработчика Visual Studio 2015. Описание среды разработчика можно найти во многих источниках [7, 11].

Следует отметить, что в тексте правила языка С++ превалируют над правилами русского языка, поэтому встречаются ситуации, когда предложение С++ завершается знаком «;», однако следующий абзац начинается с прописной буквы.

1. ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ (ООП)

В центре объектно-ориентированной модели стоит понятие объекта. Оно настолько общее, что дать ему строгое определение невозможно. Однако всегда понятие «объект» может заменить собой конкретные понятия реального или абстрактного мира, с которыми мы оперируем.

В реальном мире существуют простые понятия «собака», «дом» или обобщенные «животное», «жилище». Все они могут быть заменены термином «объект», абстрагирующим понятие, или порождающим класс одинаковых (однотипных) понятий. В абстрактном научном мире также существуют понятия, которые можно считать объектами, например «геометрическая фигура», «функция», «интеграл». Даже в мире сказок и фантазий можно найти объекты, такие как «волшебная палочка» или «золотое руно».

Проводя аналогию далее, можно сказать, что мир вокруг состоит из объектов, а реальная жизнь состоит из взаимодействия этих объектов. Например, объект «студент» может взять объект «интеграл», а объект «ребенок» завести объект «собака».

В основе идеи объектно-ориентированного программирования лежит природная способность человеческого мышления к абстрагированию и классификации. Понятие объекта в ООП приближено к определению этого понятия в реальном или абстрактном мире. Так, любой из объектов:

- находится в определенном состоянии;
- имеет определенное поведение;
- существует во взаимодействии с окружающим миром.

Например, объект «собака» имеет:

- неизменяемые атрибуты, такие как порода и имя, а также изменяемые атрибуты: возраст, состояние здоровья и другие;
- умеение выполнять команды, есть, спать, играть и другие;
- свойство понимать команды и интонации хозяина, а также умение поведением выразить свои требования к окружающим объектам.

Можно сформулировать формальное определение объекта.

Объект – это сущность, способная сохранять свое состояние (информацию) и обеспечивающая набор операций (поведение) для проверки и изменения состояния объекта.

В определении объекта отчетливо видны две составляющие части:

- **состояние** (синонимы «свойства», «данные», «атрибуты» объекта);
- **методы** (синонимы «набор операций» или «функции обработки данных» объекта).

Кроме того, объект должен иметь **имя**, что необходимо для идентификации конкретного объекта. Практически все физические или абстрактные

объекты уникально идентифицируются. Например, собака имеет кличку, ее хозяин – фамилию и имя. Оба они имеют и паспортные данные, но у каждого свой формат паспорта и различные атрибуты.

Объект не существует сам по себе, а только в рамках какой-то среды обитания. Из среды, где живет объект, к нему (объекту) может быть выполнено множество запросов. В ответ на запрос объект выполнит действие или изменит состояние, например, когда наступает ночь, собака спит. Объекты тоже взаимодействуют друг с другом, обмениваясь сообщениями на уровне запросов. Например, хозяин собаки дает команду (запрос), которую та выполняет, или собака просит у хозяина поесть (посылает запрос). Пока две собаки играют друг с другом (обмениваясь запросами), их хозяева беседуют, также обмениваясь запросами.

Можно утверждать, что объект, это абстрактная сущность, наделенная реальными характеристиками объектов окружающего мира. В программировании это методология написания программ, представляющая объектную модель предметной области.

1.1. Базовые понятия объектно-ориентированного программирования

В основе ООП находятся три базовых понятия, три кита объектного подхода:

- **инкапсуляция** (Encapsulation);
- **наследование** (Inheritance);
- **полиморфизм** (Polymorphism).

Кратко поясним эти базовые понятия. Далее по мере изложения материала смысл этих понятий будет разворачиваться.

Инкапсуляция. Под инкапсуляцией понимается слияние данных и действий, которые можно выполнить над этими данными. Действия реализуются с помощью функций и называются методами. Такое слияние порождает абстрактные типы данных, называемые классами.

Как пример рассмотрим упрощенную модель объекта «библиотека». В сложной объектной модели этого понятия можно выделить множество объектов, одним из которых, не самым важным, является объект «читатель». Читатель, это человек, который записан в библиотеку, имеет читательский билет, и идентифицирован по имени и номеру билета. Внутри объектной модели библиотеки характеристиками читателя как объекта, упрощенно являются:

- имя (фамилия, группа);
- список взятых книг.

Действия, которые может выполнить читатель:

- записаться и выписаться;
- взять книгу и сдать книгу.

Введя понятие объекта «читатель», увидим, что он имеет свойства:

- имя (неизменяемый атрибут);
- принадлежность (номер группы);
- список книг на руках (изменяемый атрибут).

Также читатель имеет методы:

- записаться в библиотеку;
- выписаться;
- сдать книгу;
- взять книгу.

Операции «сдать», «взять» изменяют список книг у данного читателя, значит, меняют его свойства. Операции «записаться» и «выписаться» порождают или уничтожают объект.

Аналогичным образом можно ввести множество объектов, характеризующих систему целиком: «читатель», «сотрудник», «единица хранения», «картотека» и другие, каждый со своими атрибутами и операциями.

Сложная система функционирования библиотеки (клиента) построена на взаимодействии объектов. Объекты взаимодействуют между собой, посылая сообщения (запросы). Например, библиотекарь напомнит читателю об истечении срока пользования книгой, в ответ читатель выполнит операцию «сдать книгу», следовательно, изменит свое состояние. Читатель при посещении библиотеки отвечает на запрос библиотекаря назвать свое имя. Читатель называет имя, значит, он должен иметь операцию идентификации. При регистрации нового читателя библиотекарь посылает запрос «зарегистрировать имя», и выполняется инициализация читателя как объекта.

В системе обычно существует множество одинаковых объектов (одного класса или однотипных). Например, читатели Иванов, Петров, Сидоров есть отдельные экземпляры класса «читатель». Все объекты одного и того же класса имеют один набор операций и реагируют одинаковым образом на одни и те же запросы.

Следовательно, класс – это тип объекта (абстрактный тип данных).

Наследование. Под наследованием понимается один из главных механизмов проектирования, который позволяет в высокой степени абстрагировать проект, разрабатывая базовые (общие) классы, порождающие новые (частные), наследующие свойства и методы классов предков и приобретающие новые качества.

Идея наследования объектов заимствована у живой природы, решает проблему модификации объектов и придает ООП гибкость. Классы наследуют свойства других классов и сами могут порождать новые классы. Так наследование позволяет легко расширять функциональность классов. Но самое главное, что дает наследование, это возможность выстраивать иерархию объектов, тем самым моделируя задачи предметной области. Объектная модель более приближена к реальной задаче, чем функциональная.

Например, объект «сотрудник», это и библиотекарь, и директор, и технический персонал. Объекты «сотрудник» и «читатель» имеют общие свойства, принадлежа классу «человек». Атрибуты «имя» и операция «назвать имя» – свойства человека, значит, они унаследованы читателем и библиотекарем от класса «человек».

С другой стороны, для класса «читатель» можно определить подклассы «студент» и «преподаватель». У них практически одинаковые атрибуты, но разные права, например, студент должен сдать книги в конце семестра, а преподаватель может взять большое число книг на неограниченное время. В объектной модели это дочерние классы «читатель-студент» и «читатель-преподаватель». Они имеют отличия в атрибутах и методах, но оба наследуют классу «читатель», значит, имеют общий набор операций.

Базовый класс содержит атрибуты, общие для дочерних объектов, а порожденные классы содержат атрибуты и методы, которые уточняют абстрактный смысл объекта. Так, объект «студент» идентифицируется номером группы, а «преподаватель» – названием кафедры и должностью.

Полиморфизм. Под полиморфизмом понимается способность объекта реагировать на запрос сообразно своему типу.

В узком смысле, это свойство, которое позволяет использовать одно и то же имя для решения одинаковых внешне, но технически разных задач. Например, мы знаем, как найти площадь геометрической фигуры. Обобщенный метод «площадь» реализуется различным образом для дочерних объектов: круг, прямоугольник, трапеция и другие, тем самым, алгоритм становится конкретным тогда, когда применяется к конкретному объекту.

Для класса «читатель» наследуемыми классами являются «студент» и «преподаватель». Оба они имеют одинаковые операции «взять книгу» и «сдать книгу», иными словами, имеют одинаковый интерфейс. Однако, операции, одинаковые внешне, могут быть определены в разных классах и реализованы различным образом, например, для студента есть ограничение на число взятых книг, а у преподавателя нет.

Объект полиморфичен, когда один и тот же метод применяется к объектам различных классов, а реакция объектов различная. Например, требуя сдать книгу, библиотекарь посылает один и тот же запрос «сдать книгу» как студенту, так и преподавателю. Запрос посылается одинаково объектам разных классов, но в ответ на запрос сама операция выполняется различным образом.

1.2. Отличие ООП от традиционного модульного подхода

Согласно определению Н. Вирта [5]:

Программа = Алгоритмы + Данные

Это означает, что программы представляют собой формулировки абстрактных алгоритмов, основанные на конкретных представлениях данных.

В традиционном (процедурно-ориентированном, или модульном) программировании программа строится как множество процедур обработки данных, организованных каким-либо способом. При таком подходе процедуры первичны, а данные вторичны, хотя реализация алгоритмов зависит от представления данных. Так, например, известны алгоритмы сортировки данных. Абстрактный смысл любой сортировки одинаков, но реализация различна для массивов, для строк, для табличных данных.

Управление сценарием работы приложения, то есть динамической моделью, осуществляется главным модулем программы.

В ООП программа строится как совокупность объектов, взаимодействующих друг с другом и окружающей средой. При таком подходе данные первичны, они сами определяют способы работы с собой. Управление динамической моделью осуществляет клиент, который порождает объекты и управляет ими.

Это соотношение приведено на рисунке 1.

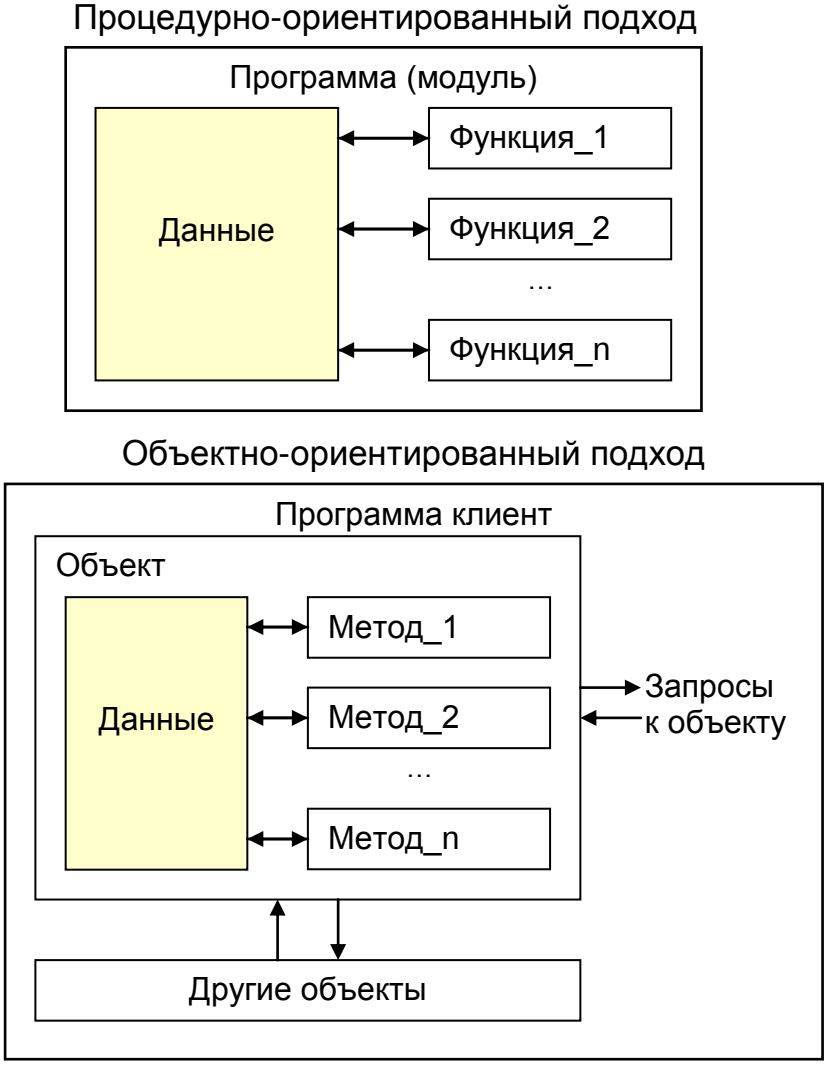


Рис. 1. Различия модульного и объектно-ориентированного подходов в технологии программирования

Программа, это модель некоторой задачи в некоторой предметной области. В соответствии с принципами моделирования, для представления задачи должны быть выделены главные составляющие и отсечены второстепенные. Основная цель, это упрощение исходной задачи в целом.

При использовании модульного стиля выполняется функциональная декомпозиция задачи, позволяющая поставить множество задач управления данными и реализовать алгоритмы их решения и интерфейсы к ним.

При использовании объектного стиля также выполнется разложение общей задачи на связанные между собой подзадачи, но иным образом. Каждая задача становится самостоятельным объектом, содержащим свои собственные данные и коды, относящиеся только к этому объекту. Абстрактные типы данных (классы) позволяют строить объектную модель прикладной задачи, выстраивать данные в их иерархии и определять необходимую функциональность.

Объектная модель имеет четыре главных элемента [1]:

- абстрагирование;
- инкапсуляция;
- модульность;
- иерархия.

Три дополнительных элемента:

- типизация;
- параллелизм;
- сохраняемость.

Каждый из этих принципов сам по себе не нов, но в объектной модели они впервые применены в совокупности.

Абстрагирование, это механизм, который выделяет существенные характеристики некоторого объекта, отличающие его от других видов объектов и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя.

Инкапсуляция, это выделение элементов объекта, определяющих его свойства и поведение. Объекты предоставляют интерфейсы, позволяющие использовать их функциональность, не раскрывая при этом детали внутреннего состояния и содержание алгоритмов.

Модульность, это свойство системы, которое позволяет разложить ее модули, внутренне связные, но слабо связанные между собой. Модульность в объектном программировании, это та же модульность, что и в структурном.

Таким образом, принципы абстрагирования, инкапсуляции и модульности являются взаимодополняющими. Объект логически определяет границы определенной абстракции, а инкапсуляция и модульность делают их физически незыблемыми.

Иерархия, это упорядочение абстракций, расположение их по уровням. Применительно к сложным системам, это структура классов (иерархия «is-a») и структура объектов (иерархия «part of»).

Типизация, это инструмент защиты от использования объектов одного класса вместо другого, или способ управления таким использованием.

Параллелизм, это свойство, отличающее активные объекты от пассивных в динамике объектной модели.

Сохраняемость, это способность объекта существовать во времени, переживая породивший его процесс, и в пространстве, перемещаясь из своего первоначального адресного пространства.

Помимо этих принципов объектной модели, можно назвать и другие полезные свойства объектов. Перечислим их.

Универсальность означет возможность повторного использования в других задачах. Однажды объявленный и отлаженный класс можно повторно использовать в разных сценариях различных приложений. Примерами являются известные библиотеки объектов, как встроенные в среду разработчика, так и независимые.

Расширяемость. Объект может расширять существующие объекты для обеспечения нового поведения.

Независимость. Объекты проектируются с минимальными зависимостями от других компонентов. Следовательно, они могут быть развернуты в любой подходящей среде без влияния на другие компоненты или системы.

Хорошими примерами реализации библиотек объектов являются библиотеки классов. Исторически первой является библиотека стандартных шаблонов STL (Standard Template Library), позднее включенная в стандарт языка C++[12]. STL представляет собой набор обобщённых алгоритмов и типов данных, средств доступа к их содержимому и различных вспомогательных функций.

Библиотека классов платформы .NET Framework [11] представляет собой библиотеку классов, интерфейсов и типов значений, которые обеспечивают доступ к функциональным возможностям системы. Она составляет основу для создания приложений, компонентов и элементов управления .NET Framework.

Библиотека Boost [10], это собрание библиотек классов, использующих функциональность языка C++ и предоставляющих удобный, кроссплатформенный, высокоуровневый интерфейс для кодирования различных задач программирования.

2. РЕАЛИЗАЦИЯ ОБЪЕКТНОЙ МОДЕЛИ В С ++. ИНКАПСУЛЯЦИЯ

Инкапсуляция, это инструмент, который объединяет данные и методы, управляющие этими данными, и позволяет защитить данные от внешнего воздействия или неправильного использования. Когда методы и данные объединяются таким способом, создается класс.

Важное различие между понятиями класса и объекта, Г. Буч [1] поясняет таким образом: «В то время как объект обозначает конкретную сущность, определенную во времени и пространстве, класс представляет собой лишь абстракцию существенных свойств объекта».

Определение 1. **Класс** – это абстрактный тип данных. Представляет собой производный структурированный тип на базе существующих типов, который задает набор данных и определяет набор операций над этими данными.

Пример. Класс «Вектор» объединяет данные, однозначно определяющие любой наперед заданный вектор, например, на плоскости, координатами начала и конца вектора, а также методы, которые обычно используются при работе с векторами: определение длины, направления, сложение векторов, и так далее.

Определение 2. **Объект** (экземпляр класса) – это переменная, тип которой определен как «класс».

Простейшим примером объекта является обычная переменная, для которой известно все, что полагается иметь объекту объявленного типа:

- диапазон значений;
- набор разрешенных операций;
- имя для идентификации объекта.

Пример. Так же точно, как можно объявить переменную любого базового типа, можно объявить переменную типа «Вектор» или массив векторов. Так порождаются реальные экземпляры векторов, над которыми можно выполнять все операции, обусловленные реализацией класса: присваивать значения, вычислять длину, выполнять сложение и другие операции.

Определение 3. **Клиент** – это программа, которая объявляет объекты класса и управляет ими посредством интерфейса объектов. Взаимодействие между клиентом и объектами, или между объектами внутри приложения, выполняется на уровне запросов. Получив запрос, объект изменяет состояние, или выполняет действие, или порождает запрос к другим объектам. Следовательно, клиент должен обеспечить общий интерфейс для манипулирования объектами в соответствии с динамическим сценарием.

Пример. Для управления созданными экземплярами векторов пишется программа, которая реализует сценарий работы приложения в целом: ини-

циализация, ввод и вывод данных, редактирование, поиск и другие. Как правило, это приложение, управляемое событиями.

Класс как абстрактный тип является конкретным представлением некоторого понятия задачи предметной области. Класс описывает множество объектов, обладающих общими атрибутами, поведение и семантикой.

2.1. Использование классов в C++

Класс является моделью некоторого объекта прикладной задачи. Для программиста классы, это типы данных, которые используются точно так же, как используются все базовые или структурированные типы.

В современных средах разработки существуют библиотеки классов, которые содержат описания абстрактных типов, и которыми программист может пользоваться для решения прикладных задач. Когда программисту необходимо ввести свой собственный абстрактный тип, то его необходимо определить. После определения классом можно пользоваться как типом.

Использование типа, определенного пользователем, не должно отличаться от использования встроенных типов.

Определение (спецификация) класса

Чтобы ввести в употребление абстрактный тип данных, необходимо определить класс. Синтаксические правила описания класса приведены ниже.

```
class Имя
{
private:
  // закрытые элементы;
protected:
  // защищенные элементы;
publuc:
  // открытые элементы;
};
```

Определение класса начинается с ключевого слова `class`, за которым следует имя класса (произвольный идентификатор), а далее в фигурных скобках определяется тело класса. Определение класса заканчивается знаком «точка с запятой», который указывает компилятору, что определение класса завершено.

Элементами класса могут быть как данные любых типов (определяют свойства, или состояние объекта класса), так и функции (определяют методы, или поведение объекта класса). В терминологии C++ они называются `data members` и `member functions`, где `member` означает «любой элемент, часть». Используются также общие для данных и методов синонимы «поля класса», «атрибуты класса», «члены класса», «элементы класса» и другие.

Метки прав доступа `private`, `protected` и `public` определяют режим доступа к элементам класса. Могут следовать в любом порядке и количестве. Действуют до следующей метки или до конца описания. По умолчанию способ доступа назначается `private`.

К закрытым (`private`) элементам имеют доступ только методы самого класса и друзья класса (об этом позже). К защищенным (`protected`) элементам имеют доступ методы самого класса и методы наследников класса (об этом позже). К открытым (`public`) элементам имеют доступ все: клиент, порождающий объекты и управляющий ими, и другие объекты.

Любые элементы класса могут быть как открытыми, так и закрытыми. Поскольку основным смыслом инкапсуляции является защита данных от нежелательного изменения, то элементы-данные закрывают от свободного доступа, а элементы-функции, как правило, открывают. Таким образом, открытые методы обеспечивают интерфейс класса. Доступ к данным организуется только средствами интерфейса класса.

Рассмотрим пример определения простого класса с именем `One`. Данными класса являются две переменные (`a` и `b`). Для работы с данными используются обычные для классов методы (функции), позволяющие присвоить значение данным `Set` и вернуть значения полей `get_a` и `get_b`.

Методы, позволяющие получить доступ к закрытым полям класса, называются аксессорами. Их имена, как правило, начинаются с префиксов `Get` или `Set`, к которым присоединено имя поля. Так, метод `get_a()` позволяет извлечь поле `a`, метод `set_a()` позволяет присвоить значение полю `a`. Аксессоры в обход защиты дают возможность работы с закрытыми данными класса, но по правилам интерфейса, определенным в классе.

```
class One {
private:
int    a;                  // Закрытые данные класса:
float  b;                  // числовые элементы a и b.
public:                    // Открытые методы класса:
void Set(int a1, float b1) {
  a = a1;                  // присвоить значения полям,
  b = b1;
}
int get_a(){               // вернуть значение данного a,
  return a;
  }
float get_b(){             // вернуть значение данного b.
  return b;
  }
}; //End of One
```

Требования к именам классов формулируются не строго. Как правило, имя отражает смысл объекта, например, `Shape` (фигура), `Window` (окно)

или с префиксом `My_class` (просто класс). Рекомендуется именовать абстрактные типы данных так, чтобы по внешнему виду они отличались от просто данных. Обычно имя класса начинается с большой буквы.

Имена методов, это интерфейс обекта, поэтому они должны быть значимыми, например, `Set` – присвоить, `Out` – вывести, `ToString` – сформировать строку.

Создание объектов класса

Класс, введенный таким образом, обладает правами типа, следовательно, объекты класса объявляются как обычные переменные с помощью известной конструкции:

```
Имя_типа Имя_объекта;
```

Например, запись

```
One    c1, c2;
```

объявляет две переменные с именами `c1` и `c2`, каждая из которых содержит по два данных (поля `a` и `b`).

В соответствии с таким объявлением, компилятор выделит память для хранения переменных `c1` и `c2`. Механизм хранения переменных типа «класс» имеет существенные особенности, а именно: в экземплярах класса хранятся только данные. Внутренние методы, предназначенные для работы с данными конкретных объектов класса, не тиражируются: нет смысла хранить коды, одинаковые для каждого экземпляра класса.

На уровне реализации место в памяти выделяется только для элементов данных каждого объекта класса. Так, для класса `One`, объявленного в примере, распределение памяти под объекты `c1`, `c2` приведено на рисунке 2.

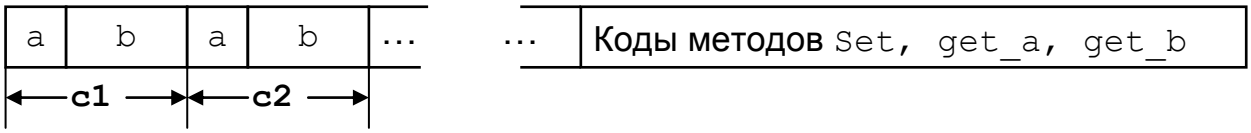


Рис. 2. Механизм выделения памяти при создании объектов классов

Метод класса отличается от обычной функции тем, что существует связь метода с экземпляром класса, включающим этот метод. Чтобы передавать данные в обычную функцию, существует механизм параметров. А чтобы метод класса получил данные объекта, параметры не нужны, так как адрес экземпляра объекта, для которого вызывается метод, неявно передается через указатель, имеющий имя `this`. Подробнее о нем далее.

Обращение к элементам класса

Обращение к элементам класса, объявленным в статической памяти, выполняется с использованием операции прямого доступа (разыменования) «.» (точка). Синтаксически обращение выглядит так:

```
Имя_объекта_класса.Имя_элемента_класса
```

Прямое обращение допускается только для открытых полей класса. Если обращение применяется к данному класса, то возвращается значение данного, если применяется к методу класса, то выполняется метод.

Пример клиентской программы для работы с объектами класса `One`.

```
void main(void){
// Объявление объектов класса.
One    c1, c2;
// Обращение к методам класса.
  c1.Set(1, 2.5);    // Присвоить значения данным c1.
  c2.Set(3, 4.0);    // Присвоить значения данным c2.
  int k1 = c1.get_a();   // Получить поле a объекта c1.
  float k2 = c2.get_b(); // Получить поле b объекта c2.
} // main
```

Независимо от типа элемента класса (данное или метод), обращение к ним выполняется одинаково, а именно:

```
Имя_класса.Имя_данного
```

Или

```
Имя_класса.Вызов_метода(параметры)
```

Поскольку данные, чаще всего, являются закрытыми (`private`) элементами, к ним нет прямого доступа извне, значит, прямое обращение невозможно.

```
int k = c1.a;         // Здесь ошибка: прямое обращение
                      // к закрытому (private) данному.
int k = c1.get_a();   // Обращение к закрытому данному с
                      // помощью открытого метода.
```

Указатель `this`

В соответствии с механизмами реализации объектов, каждый объект класса имеет собственную копию данных, тогда как коды методов сохраняются единожды (см. рис. 2). Когда вызывается метод класса, то он вызывается для одного конкретного экземпляра: `c1.Set(1,2.5);`.

При вызове метода ему автоматически (неявно) передается указатель на тот объект, для которого метод вызван, и это адрес конкретного экземпляра класса, с которым работает метод. Этот указатель называется `this`, и он неявно определен в каждом методе класса следующим образом:

```
Имя_класса * const this = адрес_объекта;
```

Имя `this` является ключевым словом. Явно его определить нельзя. Благодаря слову `const` изменить его нельзя, однако, в каждой функции класса он указывает именно на тот объект, для которого функция вызвана.

Иллюстрирует смысл указателя this рисунок 3.

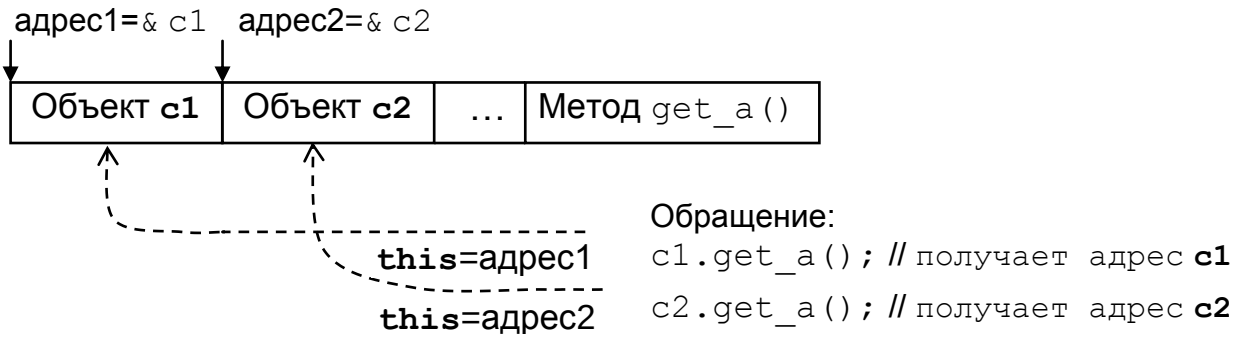


Рис 3. Роль указателя this при обращении к методу класса

Пусть объявлены объекты: One c1,c2;

Они размещены по адресам «адрес1» и «адрес2». При вызове метода c1.get_a() он получает адрес c1, при вызове метода c2.get_a() он получает адрес c2. Обобществленное имя адреса и есть this.

Можно сделать вывод, что указатель this, как адрес объекта, можно использовать для работы с данными класса внутри метода класса.

Метод Set класса One передает объекту внешние данные. В списке формальных параметров они названы именами a1 и b1: параметр a1 передает значение полю класса a, параметр b1 – полю класса b.

```
void Set(int a1, float b1) {
  a = a1;                  // Присвоить значения полям.
  b = b1;
}
```

Зная имена полей класса, можно конкретизировать имена внешних данных, называя их так же, как и поля класса, но в этом случае возникает коллизия имен, справиться с которой позволяет обобщенное имя this.

```
void Set(int a, float b) {
  this->a = a;             // Присвоить значения полям.
  this->b = b;
}
```

В этом примере показано, как, в общем случае, использовать this для разрешения коллизии имен, когда имена параметров совпадают с именами полей данных. Если обсуждать преимущества использования указателя this, то в этом примере их нет. Случай, когда использование этого указателя действительно необходимо, будет обсуждаться в разделе 2.5.

Массив объектов класса

Из объектов класса, как и из обычных переменных, можно строить более сложные структуры, например, массивы, структуры и прочие. Для класса One покажем, как ввести в употребление массив объектов класса.

Единственная особенность, которая здесь возникает, это необходимость двойного разыменования поля класса. Обращение к элементам массива выполняется через операцию разыменования [], а к полям класса через операцию «.»

```
void main(void) {
// Объявление массива из трех элементов типа One
One    c[3];
// Обращение к элементам массива выполняется через
// операцию разыменования [], а к полям класса через
// операцию «.»
for (int i=0; i<3; i++)  // Присвоить значения c[i]
  c[i].Set(i, i*2.);
for (int i=0; i<3; i++)  // Вывести значения
cout <<"a=" << c[i].get_a() << "b=" << c[i].get_b() << endl;
} // main
```

Обращение к одному элементу массива выглядит так:

```
c[i].Set(i, i*2.);
```

Сначала из области памяти, выделенной для массива, операцией [] извлекается один элемент, затем из области памяти, выделенной для хранения этого элемента, операцией «.» извлекаются поля объекта.

Динамические объекты

Как объект программного кода, объект класса можно создать динамически. Для этого объявляется указатель на объект класса, затем ему операцией `new` динамически выделяется память под размещение объекта.

Синтаксис операции `new`:

```
new Имя_типа;         //Любое имя типа.
```

Или с инизиализацией:

```
new Имя_типа инициализатор;
```

Здесь «`Имя_типа`» любое, в том числе имя объекта класса.

Семантика: `new` возвращает адрес, выделенный в динамической памяти, или `NULL`, если память не может быть выделена.

Механизм динамического выделения требует последовательного выполнения двух действий.

1. Определить указатель требуемого типа:

```
One *iP;
```

2. Выделить память и связать ее с указателем:

```
iP = new One;         // Динамический объект.
iP = new One[12];     // Динамический массив.
```

При необходимости память высвобождается операцией `delete`, синтаксис которой:

```
delete Имя_разрушаемого_объекта;
```

Например,

```
delete iP;
```

Для динамически созданных массивов нужно поставить скобки:

```
delete [] iP;
```

Доступ к элементам динамического объекта выполняется через операцию доступа (разыменования), которая называется косвенной, и записывается лексемой «->» (стрелка).

Покажем, как ввести в употребление динамическую переменную класса `One` и обратиться к ее данным.

```
void main (void) {
One    *c3;      // Объявить указатель на объект c3.
  c3 = new One; // Создать объект c3, выделить память.
  c3->Set(1, 2.0);   // Присвоить значения данным c3.
// Вывести значения объекта.
cout <<"a=" << c[i]->get_a() << "b=" << c[i]->get_b() << endl;
delete One;      // Освободить связанную память.
} // main
```

Память для динамических объектов выделяется в куче. Время жизни динамического объекта: от момента распределения памяти операцией `new` до момента высвобождения операцией `delete` или до окончания работы приложения.

2.2. Интерфейс классов. Сокрытие информации

При работе с абстрактными типами данных есть уникальная возможность сокрытия информации за барьером абстракции. Инкапсуляция объектов класса предполагает и легко реализует этот принцип. Механизмы инкапсуляции предлагают два простых правила реализации классов.

Для защиты своих данных от изменения класс использует метку прав доступа `private`. Метка означает, что класс запрещает доступ к закрытым данным из клиентской программы или из других объектов.

Для обмена информацией с клиентом или с другими объектами, класс имеет общедоступный интерфейс. Его обеспечивают открытые методы `public`, благодаря которым клиент или другие объекты могут манипулировать содержимым класса. Этот доступ является косвенным, так как определен по правилам интерфейса, отраженным в объявлении класса.

Из этих правил есть исключения. Во-первых, наследуемым классам разрешается доступ к полям базовых классов, имеющим тип `protected`. Во-вторых, дружественным функциям и классам разрешается доступ к полям своих друзей. Здесь есть определенное противоречие, но далее покажем, что в некоторых случаях такое нарушение прав доступа необходимо.

Область действия классов. Спецификация класса и реализация

Областью действия данных и методов класса является тело класса, то есть имена полей класса являются для него локальными.

Областью действия имени класса является файл, в котором он определен ключевым словом `class` и именем.

При работе с большим многофайловым проектом с использованием классов существуют требования:

- класс должен быть виден отовсюду, где он используется;
- абстрактная структура класса должна быть прозрачна;
- интерфейсная часть класса должна быть удобна для использования.

В нашем примере нет необходимости придерживаться этих требований, так как класс маленький. С ростом объема класса и усложнением алгоритмов, определение класса превращается в неуправляемую структуру, абстрактный смысл которой теряется. Поэтому принято разделять определение абстрактного типа на две части: спецификация класса и реализация класса.

Спецификация (определение класса) описывает поведение типа данных безотносительно его реализации. В спецификацию выносятся объявления данных класса и прототипы методов класса.

Объявления данных класса должны сопровождаться комментариями об их назначении. Прототипы методов класса должны сопровождаться комментариями, поясняющими смысл объявляемого метода и описание сигнатуры параметров.

Определение класса должно быть видно отовсюду, где предполагается использовать объекты данного класса, поэтому спецификация выносится в отдельный файл, и это файл заголовков (`.h` файл). В нем открыта интерфейсная часть класса.

Директивой `#include` файл заголовка включается в тот файл, которому нужно определение классов. Так определение становится доступным всем файлам проекта, которым оно необходимо.

Файлы реализации содержат коды конкретных методов. Здесь записываются описания (объявление и тело) всех методов класса. Файлы реализации имеют расширение `.cpp`. В составе проекта файлы реализации компилируются совместно с клиентской программой.

Пример организации многофайлового проекта для нашего простого класса приведен на рисунке 4.

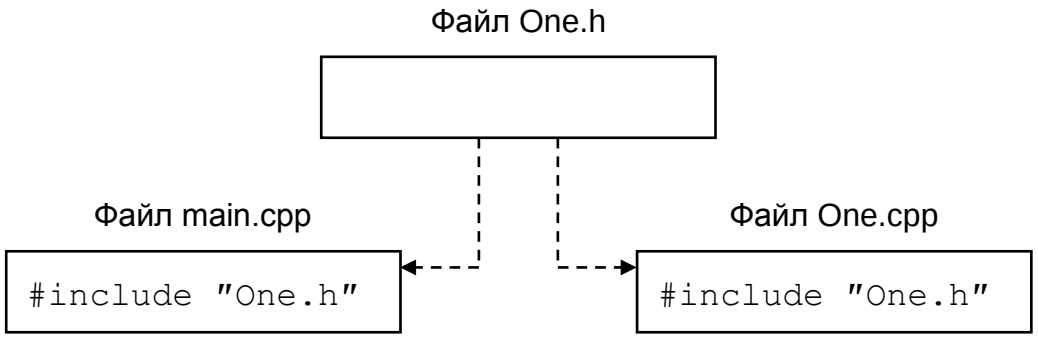


Рис. 4. Организация многофайлового проекта с использованием файла спецификации для определения класса и файла реализации

Для каждого объявляемого класса рекомендуется использовать свои собственные файлы спецификации и реализации, исключая случай, когда классы весьма близки друг другу.

Подобная организация проекта требует использования механизмов, позволяющих выполнить такое разделение, а именно, операции разрешения области видимости имен.

Операция разрешения области видимости имен (::)

Унарная операция разрешения области видимости имен C++ обозначается лексемой «::»:

```
:: Имя_объекта
```

Предназначена для разрешения коллизии имен, когда глобальная переменная скрыта локальной переменной с тем же именем. Операция «::» позволяет «присоединить» к имени переменной имя глобального объекта, тем самым выполняя обращение изнутри к внешней переменной.

Простой пример – доступ из тела функции к глобальной переменной.

```
// Глобальная переменная i.
int    i = 99;
void main(void) {
int    i = 10;        // Локальная переменная i.
cout <<"Локальная переменная= "<< i << endl;   //Равно 10.
cout <<"Глобальная переменная= "<< ::I << endl;//Равно 99.
} // main
```

В первом случае обращение выполняется к локальной переменной, во втором – к глобальной.

В объектном программировании операция «::» используется, чтобы присоединить имя метода или имя данного к тому классу, где они объявлены. Этот механизм используется, когда описания методов выносятся из тела класса в файл реализации. Функции, описываемые вне классов, должны быть привязаны к конкретному классу синтаксической конструкцией:

```
Имя_класса :: Имя_метода() {
// Тело функции
}
```

Этот же механизм используется и в других случаях, когда необходимо явно показать принадлежность имени конкретному классу, например, при реализации наследования, когда имена методов базового класса повторяются в производных.

Подставляемые функции

Если перед описанием функции записано ключевое слово `inline`, то функция будет подставляемой. Это означает, что компилятор не создает код функции, а копирует ее тело в код программы по месту вызова. Подставляемые функции используют, если тело функции короткое.

```
// Функция возвращает расстояние от точки с
// координатами(x1,y1) до точки с координатами (x2,y2)
```

```
inline float Line(float x1,float y1,float x2, float y2){
  return sqrt(pow(x1-x2,2)+pow(y1-y2,2));
}
```

В определении класса `One` описания всех методов записаны внутри класса, без разделения на объявление и определение. Такое описание упрощает запись класса, но все методы обладают свойством подстановки.

Иллюстрируем механизм подстановки на примере. Когда в тексте программы вызывается метод, определенный внутри класса, то компилятор подставляет его текст в точку вызова. Например, метод `get_a` класса `One`:

```
class One {
int a;
public:
int get_a(){    // Вернуть значение поля a.
return a;
} .// …
}; //End of One
```

При обычном вызове метода должно происходить обращение к функции:

```
void main(void) {
One    c1;                    int One::get_a() const {
int    k;                         return a;
k = c1.get_a();              }
…
} // main
```

Однако метод внутри, значит, используется подстановка, и программа выполняется так, как будто она была записана:

```
One    c1;
int    k = c1.a;
```

Это преобразование – результат оптимизации, выполняемой компилятором, и оно никак не видно программисту. Записывая определение метода в теле класса, мы даем компилятору указание применять подстановку, но совершенно не обязательно, что компилятор будет ее выполнять.

Если метод обязательно должен быть реализован методом подстановки, это требование можно указать явно ключевым словом `inline` перед определением метода. Оно носит рекомендательный характер и не гарантирует выполнение его компилятором.

```
inline int One :: get_a() {
  return a;
};
```

Итак, функции, описанные внутри класса, автоматически делаются подставляемыми (`inline`). Рекомендуется делать подставляемыми лишь небольшие простые функции из одного-двух операторов.

Явная рекомендация подстановки спецификатором `inline` не всегда будет выполнена компилятором. Не будут подставляемыми, несмотря на `inline`, следующие виды функций:

- функции, содержащие циклы и переключатели;
- рекурсивные функции;
- функции большого размера;
- функции, которые вызываются более одного раза в выражении.

Организация файлов спецификации и реализации

Теперь покажем, как должен выглядеть наш проект. Заголовочный файл «`Primer.h`» содержит абстрактное определение класса. Это объявление, оно содержит описания данных и объявления методов.

```
class One {
private:
int    a;        // Необходимы пояснения к содержанию.
float  b;
public:          // Объявления методов класса
void Set(int a1, float b1);
int    get_a();      // Как видим, в описании
float  get_b();      // присутствуют только прототипы
                     // методов класса.
}; //End of One
```

Файл реализации «`Primer.cpp`» содержит описание реализации методов класса.

```
#include "Primer.h"  // Обязательно везде, где
                     // используются объекты класса.
// Оператор :: показывает, чей метод.
void One :: Set(int a1, float b1) {
  a = a1;
  b = b1;
}
int One :: get_a() {
  return a;
};
float One :: get_b() {
  return b;
}
```

Простые методы `get_a()` и `get_b()` будут подставляемыми.

Файл клиентской программы «`Main.cpp`» объявляет и манипулирует объектами классов. Компилируется совместно с файлами реализации, для чего они объединяются в проект.

```
#include "Primer.h"
void main(void) {
One    C;
// Инициализировать объект C.
```

```
  C.Set(1, 9.9);
// Вывести данные объекта.
cout <<"\nОбъект: a= " << C.get_a() << "b= " << C.get_b();
} // main
```

2.3. Конструктор и деструктор

Инициализация переменных в C++, это присваивание значений объекту при его объявлении. Этот механизм разрешен для базовых и конструируемых типов, и позволяет избежать ошибок, вызванных неопределенным значением переменных, например:

```
int a = 10;      // Переменная a = 10.
double Arr[] = {1.5, 2.9, 6.7, 9.3};   // Массив Arr.
```

При объявлении объекта класса, значения его полей не определены. В приведенном примере полям класса значения присваиваются методом `Set` уже после того, как объект создан. Такое решение может вызвать ошибки.

В C++ существует инструмент, который позволяет инициализировать объекты класса при объявлении объекта, так же точно, как инициализируются переменные. Им является специальный метод класса, называемый конструктором. Название в точности отражает смысл метода, поскольку метод именно конструирует значение данного типа.

Определение 1. **Конструктор** – метод класса, который вызывается всегда при создании объекта класса (фактически при выделении памяти), в том числе операцией `new`.

Определение 2. **Деструктор** – метод класса, который вызывается при уничтожении объекта класса (при высвобождении памяти).

Конструктор предназначен для инициализации создаваемых объектов. Деструктор предназначен для высвобождения памяти при разрушении объекта. Конструктор и деструктор являются методами класса. Имя конструктора совпадает с именем класса, имя деструктора совпадает с именем класса, но предваряется знаком тильда ~.

Существенной особенностью конструктора и деструктора как функций является то, что они не возвращают никакого значения, даже `void`.

Данные класса могут быть определены внутри или переданы через параметры конструктора. Часто необходимо использовать несколько способов инициализации объекта, для этого следует описать несколько конструкторов. Когда класс имеет более одного конструктора, используются перегруженные функции.

Перегруженные функции

Цель перегрузки состоит в том, чтобы функция с одним и тем же именем могла выполняться различным образом в зависимости от способа обращения к ней (это пример реализации идеи полиморфизма). Необходимость перегрузки функции определяется логикой задачи. Разрешается перегрузка по типу аргументов и по числу аргументов.

Механизм реализации основан на обработке вызовов. Компилятор, обнаруживая в коде программы обращение к функции, сам выбирает нужную реализацию в зависимости от фактических параметров обращения.

Для каждого перегружаемого имени можно определить столько связанных с ним различных функций, сколько вариантов сигнатур допустимо при обращении. Отличаясь по сигнатуре параметров, все перегруженные функции возвращают значения одного типа. Перегрузка функций существенно улучшает читаемость кода.

Пример перегрузки по числу параметров

Пусть требуется многократно находить наибольшее значение из нескольких чисел (двух или трех). Следует написать две функции `max`. Обращение выполняется с двумя либо тремя аргументами.

```
int    max(int, int);          // Число аргументов 2
int    max(int, int, int);     // Число аргументов 3
void main(void) {
int    a = 1, b = 2, c = 3, d;
// Обращение к функциям выглядит одинаково.
  d = max(a, b);
  cout << "max=" << d << endl;
  d = max(a, b, c);
  cout << "max=" << d << endl;
}// main
// Реализация перегруженных функций различна.
int    max(int x, int y) {
  return x>y ? x : y;
}
int    max(int x, int y, int z) {
  return x>y ? (x>z ? x : z) : (y>z ? y : z);
}
```

При обращении компилятор анализирует количество фактических параметров и сам выбирает нужную реализацию функции.

Использование конструкторов, перегрузка конструкторов

Класс может не содержать конструкторов или иметь более одного конструктора. Число конструкторов определяется требованиями задачи.

Конструктор по умолчанию. Такой конструктор не имеет параметров, и если в классе не объявлено ни одного конструктора, то компилятор автоматически создаст конструктор, роль которого заключается в распределении памяти для экземпляра. Конструктор по умолчанию принято определять всегда.

Конструктор с параметрами. Позволяет инициализировать экземпляр при его создании, в том числе распределять память динамически, вызывать функции, выполнять ввод данных, отрисовку на экране и другие действия.

Конструктор копирования. Предназначен для создания экземпляра класса путем копирования данных из другого, уже существующего объек-

та. Конструктор копирования создается автоматически, и может быть использован только для данных, создаваемых в статической памяти. Для динамически создаваемых данных механизмы копирования иные.

Перепишем определение класса `One` с использованием конструкторов.

Вместо функции `Set()`, присваивающей значение полям класса, используются конструкторы. Это позволяет инициализировать значения объектов класса при их создании.

```
class One {
private:
int    a;
float  b;
public:
//Конструкторы с параметрами.
// 1. Инициализирует оба поля, имеет два параметра.
One(int a1, float b1) {
  a = a1;
  b = b1;
}
// 2. Инициализирует первое поле, имеет один параметр.
One(int a1) {
  a = a1;
  cout << "\nВведите b\n";
  cin >> b;
}
// 3. Инициализирует второе поле, имеет один параметр.
One(float b1) {
  b = b1;
  a = 0;
}
// 4. Конструктор без параметров (по умолчанию).
One() {
  a = 12;
  b = 9.99;
}
// Конструктор без параметров может быть и таким:
/*
One() {
  cout << "\nВведите a, b \n";
  cin >> a >> b;
} */
//Деструктор.
~Two() {
  cout << "\nУничтожение объекта One" << endl;
}
};// End of One
```

В этом примере показаны все варианты конструктора класса `One`.

Пример обращения к конструкторам и деструктору.

```
//Создание объектов с использованием конструкторов.
void main(void) {
// Вызывается конструктор с двумя параметрами.
  One  t1(1, 2.);
// Вызывается конструктор с параметром int.
  One  t2(3);
// Вызывается конструктор с параметром float.
  One  t3(4.);
// Вызывается конструктор без параметров.
  One  t4;
//Обращение к деструктору, здесь оно лишнее.
  ~ One();
} // main
```

Важный момент – при обращении к конструктору по умолчанию скобки не указываются:

```
One    t4;
```

Механизм умолчаний. Для конструкторов с параметрами можно использовать механизм умолчаний, если возможные значения параметров указать непосредственно в объявлении конструктора.

```
// Пусть объявлен конструктор:
One(int a1=1, float b1=1.5) { // Значения показаны
  a = a1;
  b = b1;
}
```

Если при объявлении объекта значения фактических параметров опущены, то поля будут проинициализированы умолчаниями, а если присутствуют, то поля будут проинициализированы фактическими параметрами.

```
// Пример обращения.
One A1;              // Поле a = 1, поле b = 1.5;
One A2(10, 12.5);    // Поле a = 10, поле b = 12.5;
```

Инициализация и присваивание. Конструктор копии

По умолчанию, когда конструктор не описан явно, то автоматически формируются конструктор без параметров и конструктор копирования.

Конструктор копирования имеет прототип вида:

```
Type :: Type(const Type &);
```

Здесь `Type` – абстрактное имя класса.

Этот конструктор вызывается, когда новый объект создается путем копирования существующего, а именно: когда описывается объект с инициализацией другим объектом, когда объект передается в функцию по значению, и когда объект возвращается из функции.

Благодаря этому возможно присваивание объектов классов, например:

```
One    t5(1,5.5);
One    t6 = t5; //Благодаря конструктору копирования
```

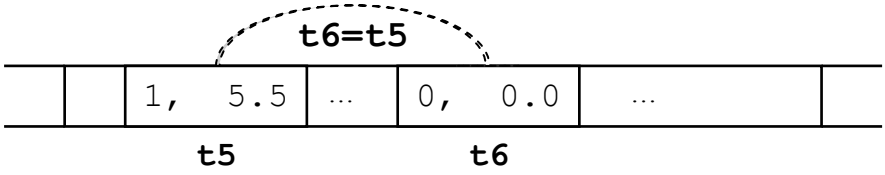
```
One    t7;
t7 = t6;
```

Операция копирования будет выполнена правильно только для объектов, объявленных статически: значения полей объекта-копии будут получены путем копирования значений полей исходного объекта.

Для динамических объектов будет выполнено поверхностное копирование, а именно, копирование адреса объекта, что приведет к утечке динамической памяти, и может вызвать ошибки.

Механизмы копирования иллюстрирует рисунок 5.

```
One    t5(1,5.5);
One    t6;
t6 = t5;
```




```
One *t5 = new One(1,5.5);
One *t6 = new One;
t6 = t5;
```

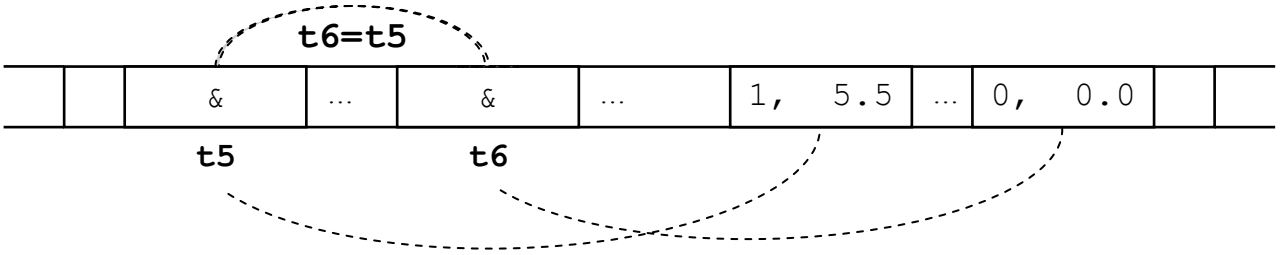


Рис. 5. Механизм копирования для статических и динамических объектов

Итак, что происходит при копировании: `t6 = t5;`

Для статических объектов происходит копирование содержимого области памяти, распределенной для переменной `t5`, в область памяти, распределенной для переменной `t6`.

Для динамических объектов указатели `t5` и `t6` содержат адреса динамической памяти (в области кучи), распределенной для хранения их значений. Если записано `t6 = t5` , то содержимое переменной `t5` копируется в `t6`, но это адрес. Таким образом, оба указателя будут хранить значение одного и того же адреса `t5`, или ссылаться на один и тот же объект, а адрес, на который ссылался `t6`, будет безвозвратно потерян.

Правильное присваивание значения одного динамического объекта значению другого динамического объекта имеет такой синтаксис:

```
*t6 = *t5;
```

Можно создать собственный конструктор копирования: он получает данные внешнего объекта, и присваивает их собственным полям.

```
One(One &T) {
  this->a = T.a;
  this->b = T.b;
}
```

В этом случае объект `this` получает значения полей объекта `T`.

Чтобы присваивание работало правильно для любых объектов, необходимо переопределить операцию копирования «=», перегружая операцию присваивания (см. далее в разделе 2.5).

Особенности конструктора

Конструктор, это метод класса, который вызывается при создании объекта, и позволяет инициализировать значения полей. Конструктор существует всегда, даже когда он не объявлен в описании класса.

Конструктор не возвращает значения, и нельзя получить указатель на конструктор.

По умолчанию, когда явного присутствия конструктора нет, формируются конструктор без параметров и конструктор копирования.

Если класс содержит константы и ссылки, их необходимо инициализировать, поэтому конструктор должен быть прописан обязательно.

Определять значения полей можно двумя способами.

1. Инициализация по умолчанию. Например:

```
One() {
  a = 12;b = 9.99;
}
```

При объявлении объекта используется неявный вызов конструктора:

```
One t4;              //Поля принимают значения 12 и 9.99
```

2. Инициализация через параметры конструктора. Например:

```
One(int a1, float b1) {
  a = a1; b = b1;
}
```

При объявлении объекта используется явный вызов конструктора:

```
One t1(1, 1.11);     //Поля принимают значения 1 и 1.11
```

В каждом классе конструкторов с параметрами может быть много, но только один с умалчиваемыми значениями параметров. Параметры конструктора определяются по механизму перегрузки функций. При объявлении объекта класса компилятор сам выбирает подходящий конструктор в соответствии с сигнатурой параметров.

Конструктор нельзя вызывать как обычный метод класса. Для вызова конструктора существует две формы. Синтаксис первой формы:

```
Имя_класса имя_объекта(фактические_параметры);
```

Например,

```
One    t5(1,1.11);
```

Синтаксис второй формы:

```
Имя_класса(фактические_параметры);
```

Например,

```
One t6 = One(5,6.88);
```

Здесь создается объект без имени, но с начальными значениями, который копируется в объект `t6`.

Например,

```
One t7 = One;   // t7 – экземпляр по умолчанию.
```

Ошибкой будет обращение вида:

```
One t8();
```

Для глобальных объектов конструктор вызывается с началом исполнения программы, для локальныхобъектов – при выполнении объявления, как только становится активной их область действия.

Деструктор для локальных объектов вызывается тогда, когда они выходят из области видимости, глобальные объекты разрушаются при завершении выполнения программы. Конструктор вызывается и при создании временного объекта, например, при передаче объекта из функции.

Конструкторы не наследуются.

Конструктор не может быть `const`, `static` и `virtual`.

Роль деструктора в уничтожении объектов класса

При работе с динамическими объектами класса операция `new` выделяет память для объекта и вызывает, если нужно, конструктор:

```
One *C = new One;
```

Операция `delete` высвобождает динамическую память и вызывает, если нужно, деструктор:

```
delete C;
```

Деструктор в классе может быть только один. Он не имеет параметров, и вызывается при разрушении экземпляра. Если класс не содержит объявления деструктора, компилятор автоматически создает пустой деструктор. Деструктор используется для явного высвобождения памяти, а также для того, чтобы уничтожить следы присутствия объекта в приложении.

Примером использования деструктора объекта в современных приложениях могут быть окна, например, диалога. Когда вызывается окно диалога, то динамически создается объект типа «диалог». Это может быть окно сообщения, окно открытия файла, окно настроек и другие. Интерфейсом диалога является графическое изображение окна на экране. По завершении работы с диалогом, пользователь закрывает окно. Тут-то и вызывается деструктор: его роль не только высвободить память, но и обновить графический экран.

Для объектов, создаваемых в статической памяти, использование деструктора не необходимо. Статические объекты имеют определенное время жизни и разрушаются при выходе из области видимости. Выполнив предыдущий пример в отладчике, можно увидеть момент вызова деструктора.

В приведенном примере для всех статических объектов `t1`, `t2`, `t4` , деструктор будет вызван при завершении программы.

Необходимо описывать деструктор явным образом в том случае, когда объект содержит указатели на память, выделяемую динамически. В этом случае при уничтожении объекта память, распределенная для полей указателей, не будет высвобождаться.

Деструктор не может быть объявлен как `const` или `static`.

Деструктор не наследуется.

Деструктор может быть виртуальным.

В следующем примере показан механизм вызова деструктора. Чтобы увидеть процесс разрушения динамических объектов, пример нужно выполнять в отладчике.

```
class One {
int    a;
float  b;
public:
One() {
  a = 5; b = 9.88;
}
One(int a1, float b1) {
  a = a1; b = b1;
}
// Деструктор: разрушаясь, объект прощается с нами.
~ One() {
  cout <<"\nПока!\n";
}
}; //End of One
void main (void) {
One *c1, *c2, *c3;
// Создаем объект, и тут же его разрушаем.
c1 = new One(1,2.);
  delete c1;
c2 = new One;
  delete c2;
c3 = new One;
  delete c3;
} // main
```

При каждом обращении к `new` вызывается конструктор, при каждом обращении к `delete` вызывается деструктор.

Инициализация массива объектов

При создании массива объектов класса может использоваться только конструктор по умолчанию, который будет вызван для каждого элемента массива.

Для присваивания произвольных значений элементам массива используются искусственные методы или допускается простая инициализация.

```
class One {
int    a;
float  b;
public: // Аксессоры Set обеспечивают доступ к полям.
void Set_a(int x) {
  a = x;
}
void Set_b(float y) {
  b = y;
}
void Out() {
  cout << "a= " << a << " b= " << b << endl;
}
One() {                    //Конструктор по умолчанию.
  a = 1; b = 9.99;
}
One(int a1, int b1) {      //Конструктор с параметрами.
  a = a1; b = b1;
}
~ One() {                  //Деструктор
  cout << "\nПока!\n";
}
}; //End of One
// Конструктор с параметрами не может быть вызван.
void main(void) {
int    i;
One C[4]; // Четырежды вызван конструктор по умолчанию.
for (i = 0; i < 4; i++)
  C[i].Out();         // Все элементы массива одинаковы.
// Можно присвоить значения элементам массива так:
for (i = 0; i < 4; i++) {
  C[i].Set_a(i*10);
  C[i].Set_b(i*0.1);
  }
for (i = 0; i < 4; i++)
  C[i].Out();         // Все элементы массива различны.
// Можно создать динамический массив:
One *D;
D = new One[4];       // Инициализированы умолчаниями.
for (int i = 0; i < 4; i++) {
  D[i]->Set_a(i*10);
  D[i]->Set_b(i*10+5);
  }
} // main
```

Настоящая инициализация данных массива выглядит как инициализация массивов размерности, большей, чем 1, например:

```
One Q[2] = {{1, 9.9},
            {2, 11.1}
            };
```

В этом случае наличие явно объявленных конструкторов может помешать, вызывая синтаксическую ошибку.

2.4. Статические элементы класса

В соответствии с механизмами реализации объектов, каждый экземпляр класса имеет собственную копию данных. Возможна ситуация, когда все экземпляры одного класса должны иметь общее поле, например, когда оно имеет одинаковое значение для всех объектов класса или является ссылкой на ресурс, разделяемый всеми объектами. Это поле класса объявляется как поле с атрибутом `static`, и существует в единственном экземпляре для всех объектов класса.

Синтаксис определения статического поля:

```
static Тип Имя;
```

Статическое поле создается в единственном экземпляре, и является общим для всех экземпляров класса. Память под статическое поле выделяется один раз при старте программы независимо от числа созданных объектов, и даже при их отсутствии. Память, занимаемая статическим полем, не учитывается при определении размера объекта операцией `sizeof`.

Иллюстрирует этот механизм рисунок 6.

```
class A {
int    a,b;
static int C;
...
}
// Пусть объявлены объекты:
A t1, t2;  // В каждом есть статическое данное.
```

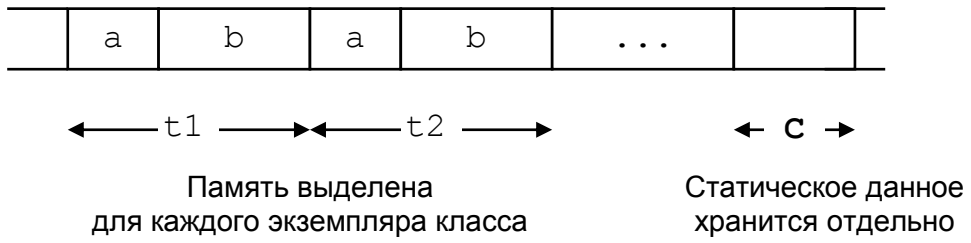


Рис. 6. Распределение памяти для статического элемента класса

Правила доступа распространяются и на статические элементы класса. Если они `public`, то к ним можно обращаться напрямую, а если они `private` или `protected`, что чаще всего случается, доступ можно получить только посредством методов класса.

Методы класса тоже могут быть статическими, тогда они могут обращаться только к статическим полям, поскольку не получают указателя this. С использованием статических методов инициализацию статических компонентов класса можно выполнить еще до создания экземпляра класса.

Вызов статического метода может быть как обычным:

```
Имя_объекта.Имя_метода(),
```

так и необычным, использующим квалифицированное имя вида:

```
Имя_класса :: Имя_статической_функции().
```

Пример использования статического данного.

Пусть есть класс Point, точка на плоскости. Необходимо знать количество одновременно существующих точек, что будет полезным при работе с динамическими объектами.

```
class Point {
int    x, y;
//Статическое данное номер точки (количество).
static int Cou;
public:
Point(int x1,int y1) {
  //Конструктор (точки нумеруются при создании).
  ++Cou;              // Первоначально должен быть 0.
  x = x1; y = y1;
}
// Статическая функция: количество точек.
static int counter() {
  return Cou;         // Возвращает значение Cou.
}
}; // End of Point
```

Программа клиент должна инициализировать нулем значение статического данного, когда еще нет ни одного объекта. Используется прямое обращение к имени статического данного через имя класса:

```
int Point :: Cou=0;
```

Это действие глобально.

Статическая функция counter возвращает значение статического данного. Обратиться к ней можно двумя способами: через имя объекта A.counter() или через имя класса Point :: counter(), что показано в примере.

```
// Инициализация статического данного.
int Point :: Cou=0;
void main(void) {
cout <<"Объектов нет, Cou=" << Point::counter() <<endl;
Point A(4,5);
cout <<"Есть точка с номером " << A.counter() << endl;
// Статическое поле хранится отдельно.
cout <<"Размер объекта: " << sizeof(Point) << endl;
// Два типа обращения к статическому данному.
```

```
Point B(9,10);
Point C(2,4);
// Обращение первого типа:
cout << "Теперь точек: ", Point::counter() <<endl;
// Обращение второго типа:
cout << "Теперь точек: ", A.counter() << endl;
}
```

Передача параметров по ссылке

В C++ определен способ передачи в функцию ссылки на объект. Синтаксис определения функции:

```
Тип_функции Имя_функции (тип &параметр, ...) {
// Далее в теле функции используется имя параметра.
}
```

Знак операции определения адреса **&** у имени параметра означает использование ссылки на переменную.

Пример передачи параметров по ссылке.

```
void Swap(int &x, int &y) {
int    buf = y;
  y = x;
  x = buf;           // Переменные x, y подлинники.
}
```

Механизм ссылок экономит память и время при передаче в функцию больших по объему данных. Кроме того, он позволяет избежать ошибок, связанных с вызовом конструкторов, когда объекты классов передаются в функцию по значению.

Ссылку можно вернуть из функции. Это полезно, например, при перегрузке операций определенных типов, когда необходима ссылка на созданный в теле функции объект. Синтаксис такого объявления:

```
int & f() {
  return Имя_объекта;
}
```

В этом случае функция не возвращает значение глобальной переменной `Имя_объекта`, а возвращает ее адрес в виде ссылки.

2.5. Перегрузка операций

Программирование в объектах, в основном, сводится к возможности определить функциональность абстрактного типа, то есть задать набор методов. Все обращения к объектам этого типа ограничиваются операциями из заданного набора. Для удобства использования абстрактных типов нужны еще некоторые расширения языка, одним из которых является перегрузка операций. Прежде, чем обсуждать способы перегрузки операций, необходимо ознакомиться с инструментом «дружественные функции».

Дружественные функции

Механизм управления доступом к полям класса выделяет три способа доступа: `public` (доступны отовсюду), `protected` (доступны внутри класса и в производных классах), `private` (доступны только в классе). Иногда необходимо разрешить доступ извне к каким-то полям класса в обход защиты. Таким расширением интерфейса класса являются дружественные функции.

Дружественная функция – средство для того, чтобы функция, не являющаяся членом класса, имела доступ к закрытым и защищенным полям этого класса.

Функция объявляется другом класса с его согласия при помощи ключевого слова `friend`, прописанного у прототипа функции внутри класса.

Пример. Друзья класса, это внешние функции.

```
class One {
int    a;
float   b;                  // Приватные члены класса.
// Функции увеличения приватной переменной.
friend void Inc_a(One &);
friend void Inc_b(One &);
public:
void Out_One() {
  cout << "a= " << a << " b=" << b << endl;
}
One(int a, int b) {       //Конструктор
  this->a=a;
  this->b=b;
}
}; // End of One
// Доступ к объектам – только через список параметров.
void Inc_a(One &A) {
  A.a++;
}
void Inc_b(One &A) {
  A.b+=1.;
}
void main(void) {
One O(12,9.99);
//До изменения
O.Print_One();
// Дружественная функция изменяет объект.
Inc_a(O);                  // Увеличит a.
Inc_b(O);                  // Увеличит b.
//После изменения
O.Print_One();
} // main
```

Дружественная функция, это инструмент, который позволяет обойти защиту класса. Чтобы не сломать защиту, действие выполняется под контролем класса, для чего требуется соблюдение двух условий.

Во-первых, сам класс должен объявить, что он имеет друга. Для этого прототип дружественной функции с ключевым словом `friend` размещается внутри определения класса.

Во-вторых, чтобы дружественная функция имела доступ к полям классов, объекты должны передаваться ей только через параметры. Имеет смысл передавать параметры по ссылке.

Особенности дружественных функций

1. Дружественная функция может быть глобальной.

```
class My_class {
friend int Method(My_class &T);
…
};// End My_class.
…
int Method(My_class &T) {
// тело_функции;
}
```

2. Дружественная функция может быть методом другого класса.

```
class My_class1 {
…
  int Method();
};
class My_class2 {
friend int My_class1::Method(My_class2 &T);
// Функция Method класса My_class1 является другом
// класса My_class2, значит, имеет доступ к его полям,
// но только через список параметров.
…
};
```

3. Дружественная функция может дружить с несколькими классами.

```
class My_class2;          //Опережающее объявление.
class My_class1 {
// Method – внешняя функция.
friend void Method(My_class1, My_class2);
…
};
class My_class2 {
// Method – внешняя функция.
friend void Method(My_class1, My_class2);
…
};
```

```
//Глобальное объявление функции
void Method(My_class1 x, My_class2 y) {
  // Тело_функции;
}
```

Аналогичным образом можно подружить и классы, об этом позже.

Операции над классами. Перегруженные операции

Стандартные операции C++ можно классифицировать по типу возвращаемого значения и по количеству операндов.

По типу возвращаемого значения операции могут быть, например, арифметические и логические.

По количеству операндов операции разделяются на унарные, бинарные и тернарную. Есть операции, которые могут быть как унарными, так и бинарными, в зависимости от числа операндов. Например, бинарная операция «*» означает умножение, а унарная – извлечение значения по адресу, бинарная операция «&» означает поразрядное умножение, а унарная – получение адреса.

Операция присваивания «=» и ее клоны изменяют значение данного.

Для каждой операции определен ранг и порядок выполнения.

Абстракция классов позволяет назначить операции над классами, аналогичные стандартным операциям над простыми типами данных – сложение, вычитание, умножение, деление и другие. Логический смысл этих операций для нестандартных типов данных очевиден либо неочевиден, но может быть использован.

Например, операции сложения, вычитания комплексных чисел, векторов и матриц выглядят при записи так же, как подобные операции для простых данных. Операция сложения для строк может присоединить вторую строку в конец первой. При работе с объектами, например, естественно вычитать объекты «Время», выраженные в часах, минутах, секундах, чтобы узнать разницу. При измерении углов можно складывать, вычитать значения углов, измеренных в градусах и минутах.

Полезно и необходимо иметь возможность сравнить два объекта класса, например, два вектора. Полезно присвоить значение одного объекта другому. Все эти возможности подчеркивают абстрактный смысл объекта как структуры, с которой можно обращаться как с единым целым.

Для перегрузки операций используются специальные функции, которые называются операции-функции.

Определение операции-функции

Новое поведение операции необходимо определить с использованием операции-функции следующим образом:

```
Тип_значения operator знак_операции (список_параметров)
{
// тело операции-функции;
}
```

Это синтаксис обычного определения функции с необычным именем.

Здесь `Тип_значения` – имя класса, или другой тип, знак_операции – почти любой знак операции из существующих в С++, список_параметров – объекты класса или других типов.

Имя операции-функции задают ключевое слово `operator` и следующий за ним знак операции, например, `operator +` расширяет действие стандартной операции сложения, `operator >` расширяет действие стандартной операции отношения.

В общем виде, если используется класс `Type`, то распространение произвольной операции `@` на объекты класса объявляется функцией

```
Type operator @ (Type &x, Type &y) {
// Описание алгоритма для объектов x и y класса Type.
}
```

В качестве параметров использована ссылка (`&`), что и рекомендуется делать всегда, так как в этом случае в функцию передается не весь объект (он может быть большим), а ссылка на него.

Тело операции-функции описывает алгоритм выполнения нестандартной операции, которая будет подменять собой стандартную, когда та будет вызвана для объектов класса. Для базовых типов семантика операции останется неизменной. Операция должна обеспечить явную связь с классом, для которого эта операция введена.

Количество параметров у операции-функции зависит от арности операции и способа определения операции-функции (их три):

- внешняя функция, у которой хотя бы один параметр имеет тип класс (или ссылка на класс);
- функция, дружественная классу;
- внутренняя функция класса.

Первый способ обладает большим числом недостатков, главным из которых является отсутствие доступа к закрытым полям класса. Если этот способ используется, то класс должен обладать открытыми методами, предоставляющими такой доступ.

В любом случае выбор способа перегрузки зависит скорее от предпочтения программиста, чем от требований задачи. С использованием внутренних методов класса особенности синтаксиса и семантики перегрузки существенны, и обусловлены тем, что первый операнд операции известен, и это `this`.

Перегрузка бинарных операций

Способ 1. Для перегрузки используется внешняя функция, то есть объявленная вне класса. Операндами бинарной операции являются объекты класса, передаваемые в функцию через список параметров, здесь другого способа доступа к объектам класса нет. Но поскольку они закрыты, для

доступа к полям класса используются специальные методы, аксессоры, позволяющие извлечь значения полей или присвоить.

```
class Two {
private:
int    a;
float  b;
public:
Two(int a, float b) {     //Конструктор с параметрами.
  this->a = a;
  this->b = b;
}
Two() {                   //Конструктор по умолчанию.
  a = b =0;
}
int Get_a() {             //Функции доступа к данным.
  return a;
}
float Get_b() {
  return b;
}
//Назначить значения полям данных.
Two Set(int a, float b) {
  this->a = a;
  this->b = b;
  return *this;
}
void Out() {
  cout <<"Объект класса: a=" << a <<" b=" << b << endl;
}
}; //End of Two
// Объявление операции сложения как внешней функции,
// ее аргументы – два объекта класса Two.
Two operator + (Two &T1, Two &T2) {
  Two tmp;           //Создается временный объект.
  // Его полям присваиваются значения, равные
  // суммам значений одноименных полей T1 и T2.
  tmp.Set(T1.Get_a()+T2.Get_a(),T1.Get_b()+T2.Get_b());
  return tmp;        //Возвращается полученное значение.
}
//Пример обращения к операции-функции.
void main (void) {
// Создаются два экземпляра класса Two и складываются.
  Two t1(1,2.0);
  t1.Out();
  Two t2(3,3.5);
  t2.Out();
```

```
  Two t3;
  t3 = t1 + t2;      //Внешний вид операции сложения.
  t3.Out();
// Второй вариант обращения – полная форма вызова,
// как обычное обращение к функции. Показывает, что
// операция-функция отличается от обычной функции
// именем и способом обращения, но не механизмом.
  Two t4;
  t4 = operator+(t1, t2);          // Обычное обращение.
  t4.Out();
  // Допускается цепочка вызовов.
  t4 = t1+t2+t3+t4;
  t4.Out();
}// main
```

Недостатком этого способа перегрузки является необходимость использования специальных интерфейсных методов, открывающих поля класса для доступа извне (методы `Get` и `Set`).

Способ 2. Для перегрузки операции используется дружественная функция. Она объявлена вне класса, но имеет доступ к его полям. Операндами бинарной операции являются объекты класса. В определении класса прототип операции-функции присутствует с ключевым словом `friend`.

```
class Two {
private:
int    a;
float  b;
//Дружественная функция. Внутри класса ее прототип
friend Two operator + (Two &T1, Two &T2);
public:              //Конструктор с параметрами.
Two(int a, float b) {
  this->a = a;
  this->b = b;
}
Two() {              //Конструктор по умолчанию.)
  a = b = 0;
}
void Out() {
  cout <<"Объект класса: a=" << a <<" b=" << b << endl;
}
}; //End of Two
// Объявление дружественной функции.
// Она не принадлежит классу, но использует его данные.
// Алгоритм сложения такой же, но нет необходимости
// использовать искусственные интерфейсные методы.
Two operator + (Two &T1, Two &T2) {
Two tmp;
  tmp.a =T1.a +T2.a;
```

```
  tmp.b =T1.b +T2.b;
  return tmp;
}
// Внешнее обращение такое же.
void main(void) {
  Two t1(1,2.0);
  t1.Out();
  Two t2(3,3.5);
  t2.Out();
  Two t3 = t1 + t2;
  t3.Out();
} // main
```

Способ 3. Перегрузка бинарной операции с помощью внутренней функции (метода) класса. В этом случае есть существенная особенность, а именно, первым операндом функции является тот объект класса, для которого вызывается метод (указатель `this`). Параметр у функции будет только один, соответствующий второму операнду бинарной операции. Заголовок будет выглядеть так:

```
Имя_типа operator Знак_операции (Имя_типа &параметр)
```

Здесь `Имя_типа` – имя класса, определенного пользователем.

Если объявлен класс `Type`, и существуют объекты `A`, `B` типа `Type`, то при обращении `A @ B` происходит вызов функции `A.@(B)`. Здесь операция выполняется с данными объекта `A`, для которого выполнен вызов, и данными объекта-параметра `B`.

```
class Two {
private:
int    a;
float  b;
public:                    //Конструктор с параметрами.
Two(int a, float b) {
  this->a = a;
  this->b = b;
}
Two() {                    //Конструктор без параметров.
  a = b = 0.0;
}
void Out() {
  cout <<"Объект класса: a=" << a <<" b=" << b << endl;
}
//Объявление метода внутри класса.
Two operator + (Two &T) {
  Two tmp;
  tmp.a = this->a +T.a;
  tmp.b = this->b +T.b;
  return tmp;
```

```
}
}; //End of Two
// Обращение к операции не изменяется.
void main(void) {
  Two t1(1,2.0);
  t1.Out();
  Two t2(3,3.5);
  t2.Out();
  Two t3 = t1 + t2;
  t3.Out();
// Другой вариант обращения. Операция-функция,
// принадлежащая классу, может быть вызвана явно
// с использованием операций «.» и «->».
Two *t4;
t4 = new Two(5, 1.1);
Two t5 = t4->operator + (t3);
t5.Out();
} // main
```

Примечание. Для динамических объектов вызов операции применяется к значению объекта, поэтому присутствие операции * (извлечение по адресу) обязательно в синтаксисе вызова операций, как в примере ниже.

```
Two *t4 = new Two(5, 1.);
Two *t5 = new Two(1, 2.);
Two *t5 = new Two;
*t5 = *t4 + *t5;
t5->Out();
```

Операция «+» вызывается для значений динамических объектов.

Перегрузка унарных операций

Перегрузка унарных операций может быть выполнена теми же тремя способами:

- если операция перегружается с помощью дружественной или внешней функции, то она имеет один параметр, он и является операндом;
- если операция перегружается с помощью внутреннего метода класса, то она не имеет параметра, потому что единственным доступным объектом будет указатель `this`.

Приведем пример перегрузки операции инкремента ++ как дружественной функции, а декремента -- как внутренней функции класса.

```
class Two {
private:
int    a;
float  b;
// Прототип дружественной функции инкремента.
friend Two operator ++ (Two &T);
public:              //Конструктор с параметрами.
Two(int a, float b) {
```

```
  this->a = a;
  this->b = b;
}
Two() {              //Конструктор по умолчанию.
  a = b = 0.0;
}

void Out() {
  cout <<"Объект класса: a=" << a <<" b=" << b << endl;
}
// Функция декремента как внутренний метод класса.
Two & operator -- () {
  a--;
  b-=1.;
  return *this;
}
}; //End of Two
// Объявление дружественной функции.
// Она не принадлежит классу.
Two operator ++ (Two &T) {
  T.a++;
  T.b+=1.;
  return T;
}
// Обращение к унарным операциям
// выглядит как обращение к простому типу.
void main(void) {
Two t1(1,2.5);
  t1.Out();
  cout << "Префиксные ++ и --\n";
  ++t1;
  t1.Out();
  --t1;
  T1.Out();
Two t2(1,2.5);
  t2.Out();
  cout << "Постфиксные ++ и --\n";
  t2++;
  t2.Out();
  t2--;
  t2.Out();
} // main
```

В примере показано обращение к перегруженным операциям ++ и -- в постфиксной и префиксной формах. Вспомним их механизм: если операция записана в префиксной форме, то она будет выполнена прежде других операций в выражении, а если в постфиксной, то после других операций.

На самом деле при вызове операции в постфиксной форме, среда разработчика выдает предупреждение, и не изменяет значения объекта. Почему так происходит, объясняется механизмом реализации перегруженной операции. Для префиксной формы операции `@` (`++t1`) выражение `@T` означает вызов метода `T.operator @()` или функции `operator @(T)`.

Для постфиксной формы `@` выражение `T@` (`t2--`) означает вызов метода `T.operator @()`, но это действие должно быть отложено.

Для правильного выполнения операций ++ и -- в постфиксной форме нужна иная реализация:

```
// Префикс
Two & operator -- () {
  a--;
  b-=1.;
return *this;
}
```

```
// Постфикс
Two &operator-- (const int){
Two Tmp = *this;
  a--;
  b-=1.;
return Tmp;
 }
```

Операция перегружается дважды. Для постфиксной формы – с фиктивным параметром `int`.

Примечание. Для динамических объектов унарные операции вызываются для значений динамических объектов, как в примере ниже.

```
Two *t1 = new Two(1,2);
++ *t1;
t1->Out();
```

Приведение типов

В перегрузке операций могут участвовать:

- два объекта одного класса;
- два объекта различных классов;
- объект класса и некласса.

Приведем пример реализации операции умножения для объекта класса и целого числа с использованием дружественной функции.

```
// Прототип операции *: второй параметр – число.
friend Two operator * (Two &t, int n);
// Описание операции:
Two operator * (Two &T, int n) {
  T.a = T.a*n;
  T.b = T.b*n;
  return T;
}
// Пример обращения:
t2 = t2*2;
t2.Out();
```

Можно перегрузить операцию по типу, чтобы в одном случае она выполнялась для объектов класса и некласса, а в другом для объектов класса. Внешнее обращение будет одинаковым.

В примере для той же функции параметрами являются объекты класса.

```
friend Two operator * (Two &t1, Two &t2);
// Описание операции:
Two operator * (Two &T, Two &t2) {
Two T;
  T.a = T1.a*T2.a;
  T.b = T1.b*T2.b;
  return T;
}
// Пример обращения:
t3 = t1*t2; // Здесь t1, t2 и t3 объекты класса Two.
t3.Out();
```

Перегрузка логических операций

Логические операции возвращают логическое значение. Перегружаются для объектов, когда объекты нужно сравнить. Алгоритмы сравнения могут быть различными: по всем полям, по одному полю, по какому-то значению, например, фигуры можно сравнить по площади – больше та фигура, у которой площадь больше, символы – в алфавитном порядке.

Тип возвращаемого значения – `int` или `bool`, более никаких особенностей нет.

Рассмотрим несколько примеров операций для типа данных «Точка», имеющего некий практический смысл.

Пример перегрузки операции «равно» для точек на плоскости выполним методом класса. Точки равны, если их координаты совпадают.

```
int operator ==(Point &t) {
  return x == t.x && y == t.y;
}
```

Пример перегрузки операции «больше» для точек на плоскости выполним с использованием дружественной функции по значению радиус-вектора – больше та точка, которая дальше от начала координат.

```
friend int operator >( Point &t1, Point &t2);
```

Реализация метода: возвращает `true`, когда первая точка больше.

```
int operator >( Point &t1, Point &t2) {
  return t1.R() > t2.R();
} // Метод R()вычисляет расстояние.
```

Перегрузка операций присваивания

При работе с динамическими объектами операция присваивания должна быть перегружена обязательно. Иначе выполняется поверхностное копирование, что уже обсуждалось в разделе 2.3 (конструктор копии). Операция присваивания перегружается только методом класса, и изменяет состояние объекта, который левый операнд. Пример реализации приведен для объекта «Точка».

```
Point & operator = (Point const & P) {
```

```
    x = P.x;
    y = P.y;
    return *this;
}
```

Сама операция должна быть применена к значениям объектов:

```
Point *Aa = new Point;
Point *Bb = new Point(1, 2);
*Aa = *Bb;
```

Перегрузка операций ввода и вывода в поток

При работе с консолью для ввода и вывода данных используется объектная библиотека <iostream>.

Для ввода в поток используется операция <<, для вывода – операция >>. Это перегруженные бинарные операции сдвига классического C++. Левым операндом является объект cin при вводе или cout при выводе.

```
int a;
float b;
cin << a << b;  // Ввод двух значений.
```

При вводе и выводе данных базовых типов потоки ввода и вывода знают правила преобразования данных согласно механизму перегрузки операций. При работе с типами, определенными пользователем, можно расширить действие этих операций применительно к новым типам введением новых перегруженных операций.

Операции перегрузки ввода и вывода для потоков << и >> бинарные, левым операндом является объект потока, правым объект произвольного (желаемого) типа.

Формат функции для перегрузки операции << должен быть:

```
ostream & operator <<(ostream & out, имя_типа имя_объекта) {
...
  out  <<...          //Вывод значений нового типа
  return out;         //Ссылка на объект класса ostream
}
```

Здесь ostream & – тип возвращаемого значения – объект потока, operator << – имя метода.

Формальные параметры: объект потока и собственный объект.

Аналогичным образом можно перегрузить операцию ввода >>.

```
istream & operator >>(istream & in, имя_типа& имя_объекта) {
...
  in >>...            //Ввод значений нового типа
  return in;          //Ссылка на объект класса istream
}
```

Здесь istream & – тип передаваемого значения – объект потока, operator >> – имя метода.

Пример. В поток должны передаваться объекты типа «Точка». Точка как объект имеет два атрибута. Перегрузка операций ввода и вывода точки в поток приведены ниже.

```
#include <iostream.h>
class Point {
float  x, y;
public:
  friend istream &operator >> (istream &in,  Point &A);
  friend ostream &operator << (ostream &out, Point A);
}; // End of Point
// Перегрузка операции >> для точек.
istream &operator >> (istream &in, Point &A) {
  cout << "\nВведи координаты точки: " << endl;
  in >> A.x;
  in >> A.y;
  return in;
}
// Перегрузка операций << для точек.
ostream &operator << (ostream &out, Point A) {
  out << "\n X= " << A.x <<  " Y=" << A.y;
  return out;
}
// Пример обращения к операциям.
void main (void) {
Point A;
  cin  >> A;
  cout << A;
} // main
```

Ограничения перегрузки операций

1. Собственные обозначения операций, отсутствующие в синтаксических определениях C++, вводить нельзя, например, знаки `@` и `$` не обозначают никаких операций в C++ и, значит, не могут перегружаться.

Также для обозначения операций нельзя ввести новые лексемы, такие как «`**`», «`-+`» или «`+-`».

2. Есть операции, не допускающие перегрузки, а именно: «`.`», «`.*`», тернарная «`?:`», «`::`», `sizeof` и `#`.

3. Нельзя изменить приоритеты операций и порядок их выполнения.

4. Нельзя переопределить синтаксис операции, т.е. чисто унарную операцию нельзя сделать бинарной, и наоборот.

5. Функции для перегрузки операций присваивания, разыменования и косвенного разыменования: `operator =`, `operator [ ]` и `operator ->` должны быть определены как методы класса. Это обеспечивает явное присутствие в качестве первого операнда указателя на объект класса.

6. Количество параметров у операции-функции определяется арностью операции и способом ее определения, а именно:

для бинарной операции:

- если функция – метод класса, то у нее один параметр, второй `this`;
- если функция дружественная, то у нее оба параметра;

для унарной операции:

- если функция – метод класса, то у нее нет параметров (`this`);
- если функция дружественная, то у нее один параметр.

7. Если параметры разного типа, и операция перегружается с использованием метода класса, то на первом месте должен быть объект класса. Это гарантирует обращение к указателю на объект класса.

2.6. Дружественные классы

Дружественные классы, так же как и функции, являются средством для доступа к закрытым полям класса. Используются, когда объекты класса должны быть тесно связаны друг с другом [4, 6].

Например, абстрактное математическое понятие «множество» может быть отображено в контейнер, могущий содержать данные разных типов. Интуитивно понятен смысл объектов «Множество значений», «Множество векторов» и им подобных. Пусть нужно построить множество точек на плоскости. Базовые объекты «Точка» находятся в контейнере, который должен предоставить методы для работы с набором точек. Классу «Множество» должны быть доступны поля класса «Точка», поэтому точка дружит с множеством, в котором она находится.

Если дружат классы, то один из них объявляет другого своим другом с. помощью ключевого слова `friend`, приписанного к имени класса внутри объявления другого класса.

Иллюстрация отношений дружбы приведена на рисунке 7. Здесь показаны определения классов и отношения между ними.

Классы `A` и `C` объявляют, что класс `B` является их другом. Достаточно объявить отношение в теле класса:

```
friend Имя_класса
```

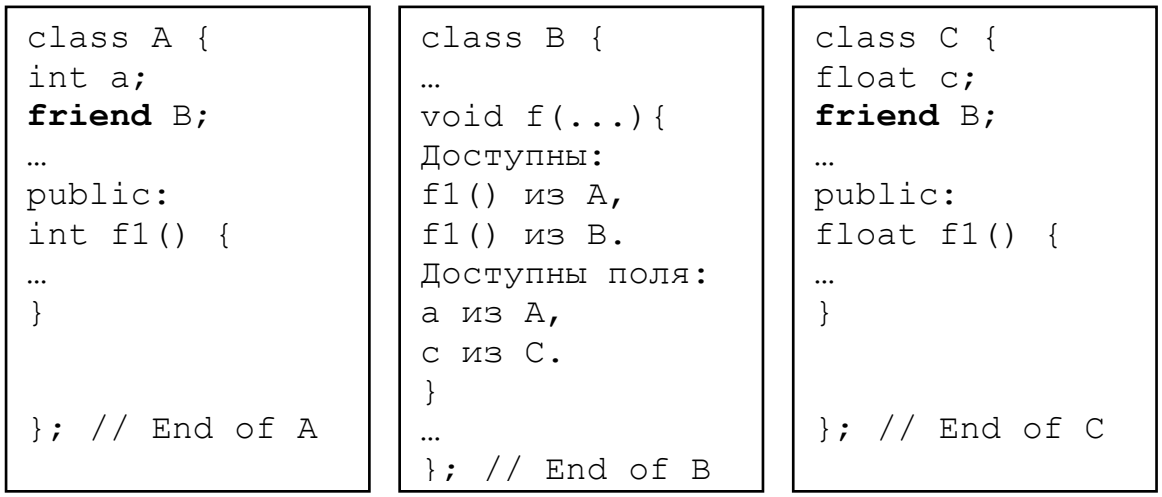


Рис. 7. Механизм дружественных классов

Классу `B` доступны все методы классов `A` и `C`, а также все их данные, в том числе закрытые (поле `a` из `A` и поле `c` из `C`), то есть он может ими пользоваться как своими. Наоборот, классы `A` и `C` не смогут воспользоваться закрытым методом `f()` и данными класса B.

Отношения дружбы имеют ограничения. Так же, как и для дружественных функций, необходимо выполнение двух условий.

Во-первых, класс должен объявить, что он имеет друга.

Во-вторых, объект дружественного класса может быть получен только через параметры метода. Параметром должен быть объект класса, чаще всего передаваемый по ссылке.

В примере покажем, как класс `My_Friend` получает доступ к закрытым полям класса `Three` при реализации механизма дружественности.

```
// Опережающее объявление дружественного класса.
class My_Friend;
class Three {
int    a;
float  b;
friend My_Friend; // Three объявляет о наличии друга.
public:
Three() {
  a = 1;
  b = 1.0;
}
void Out() {
  cout <<"Объект: a=" << a <<" b=" << b << endl;
}
}; //End for Three
class My_Friend {
// Другу Three доступны поля и методы Three, но объект
// класса может быть получен только через параметры.
```

```
int    x;
float  y;
public:
void Out() {
  cout <<"Друг объекта: x=" << x << " y="<< y <<endl;
}
// Метод Make имеет доступ к полям Three.
void Make(Three &);
}; // End of Friend
// Реализация метода Make: объект создаст точную копию.
void My_Friend::Make(Three &t) {
  x = t.a;      // Доступны поля Three и его методы.
  y = t.b;      // Поля класса равны полям t.
}
// Пример обращения.
void main(void) {
Three t1;
  t1.Out();          // Создаем объект t1 = (1,1).
My_Friend t2;        // Создаем объект t2.
// Используем Make, чтобы сделать их одинаковыми.
  t2.Make(t1);
  t2.Out();          // Убеждаемся, что получилось.
} // main
```

Итак, классу другу доступно многое, в том числе и конструктор, как метод класса. Можно определить конструктор так:

```
//Конструктор берет поля из Three.
My_Friend(Three & t) {
  x = t.a;
  y = t.b;
}
```

Обращение к конструктору выглядит так:

```
Three t1; t1.Out();
My_Friend t2(t1);    //Объект t2 получен конструктором.
t2.Out();
```

При объявлении объекта `t2` типа `My_Friend` вызывается конструктор, который получает ранее созданный объект типа `Three`, это `t1`, и заполняет поля `t2` значениями полей объекта `t1`.

Выводы. Дружба, это инструмент, позволяющий одному классу получить доступ к закрытым полям другого класса. Как и в жизни, дружба имеет ограничения. Во-первых, класс должен назвать друга, например:

```
friend Point;
```

Во-вторых, закрытые данные могут быть получены только через параметры методов:

```
void Set(Point P);
```

С помощью метода `Set` класса «Множество» можно изменить поля точки, которая указана в параметре.

2.7. Внутренний класс

Когда описание класса выполнено внутри другого класса, то класс называется внутренним или вложенным. Эта ситуация характерна для классов-контейнеров.

Пример: класс «Множество» содержит точки на плоскости.

```
class Set {
// В описание класса Set вложено описание класса Point.
class Point {
float x,y;
// Далее обычное описание класса.
Point(float xx, float yy) {
  x=xx;
  y=yy;
}
void Out_Point() {
  cout <<"Точка: x=" << x << " y="<< y <<endl;
}
}; // End of Point
// Далее описаны методы и данные множества:
Point A [100];       // Множество, это массив точек.
int Len;             // Реальное количество точек.
...
}; // End of Set
```

Существенным недостатком является факт, что внутренний класс доступен только в области видимости объемлющего класса, следовательно, нельзя записать:

```
Point T1(0.,0.); // Point не видна вне тела Set.
```

Вложенные классы вводятся в употребление, когда объемлющий класс и внутренний имеют весьма тесные связи, и внутренний класс, обладая своей четко выраженной структурой, должен предоставлять объемлющему классу прямой доступ к своим полям. Таким свойством должны обладать контейнерные классы, например, «Массив», «Таблица» и им подобные.

Вводя внутренние классы, можно легко строить модель таких абстрактных понятий как «Массив данных», «Множество точек», «Таблица функций» и им подобные. Однако, механизм вложенных классов рекомендуется использовать в простых случаях [2, 6]. Лучше описывать классы отдельно, а для реализации доступа к полям использовать инструменты дружбы, как в примере ниже.

```
class Point {
float x,y;
Point(float xx, float yy) {
  x=xx;
```

```
  y=yy;
}
}; // End of Point.
class Set {
friend class Point;
// Методы и данные таблицы:
Point A[100];        // Множество, это массив точек.
int Len;             // Реальное количество точек.
public:
int Add(Point &P);   // Добавление точки.
int Find(float Left, float Right);  // Поиск точки
…
};
```

Среди методов множества названы `Add`, получающий точку и добавляющий ее к множеству, и метод поиска, который должен найти точку, абсцисса которой принадлежит интервалу `[Left, Right]`.

Методу `Add` значения полей точки не нужны. А метод `Find` использует предикат поиска, извлекая абсциссу точки, поэтому ему просто необходим доступ к полю `x`. Но отношение дружбы позволяет получить доступ к полям объекта только в том случае, когда объект передается через параметры метода, значит, в этом случае `Point` обязана предоставить аксессор доступа `Get_x()`.

Полезным свойством вложенности является сокращение числа глобальных имен, а недостатком – отсутствие доступа к вложенным типам.

Итоги раздела

Объектно-ориентированное программирование использует природную способность человеческого сознания к абстрагированию и классификации. Если под объектом понимать сущность, которая имеет состояние и позволяет управлять им через интерфейс, то синтаксическая конструкция `class` отражает абстрактный объект в его программное представление.

Класс в C++, как и в других объектно-ориентированных языках, это тип данных, моделирующий объект некоторой предметной области. Инкапсулирует данные и методы, предоставляет интерфейс для управления объектом. Данные, определяющие состояние объекта, защищены от влияния извне. Методы могут быть закрытыми, решающими внутренние дела класса, и открытыми, предоставляющими интерфейс к объекту. Методы могут быть реализованы как операции, что расширяет абстракцию класса.

Объект, порожденный как экземпляр класса, ведет себя в соответствии с заложенными в него алгоритмами. Взаимодействует с другими объектами посредством запросов. Может находиться в определенных отношениях с другими объектами, например, отношение использования, это когда один объект использует возможности другого: менеджер выписывает счет заказчику или учитель проверяет работу ученика.

При решении сложных, больших задач используется декомпозиция – разложение общей задачи на связанные между собой подзадачи. Декомпозиция бывает модульная и объектная [1]. При использовании объектного стиля программирования каждая подзадача становится самостоятельным объектом, содержащим свои собственные данные и коды, относящиеся только к этому объекту.

Есть и определенные трудности. При объектном проектировании и программировании самое важное, это использовать объектное мышление − видеть сущности и декомпозировать задачу на взаимосвязанные объекты. Несмотря на то, что человек мыслит понятиями, научиться мыслить объектно доаольно сложно. Но это очень важно для того, чтобы найти нужные абстракции объектов для решения задач. Если абстракции выбраны и описаны хорошо, то модель задачи получает логику, оперируя объектами и взаимодействием между ними.

2.8. Вопросы для самопроверки

1. Что означает термин «инкапсуляция»?
2. Дайте определение класса.
3. Дайте определение объекта.
4. Дайте определение понятия «Клиент». Какова роль клиентской программы?
5. Что означает термин «запрос»? Как объект реагирует на запрос?
6. Что означает термин «динамический сценарий»?
7. Как объявить экземпляр класса? Каков механизм хранения переменных типа «класс»?
8. Опишите роль и механизм указателя `this`.
9. Какова роль динамических объектов?
10. Приведите синтаксис и опишите семантику объявления динамического объекта.
11. Приведите синтаксис и опишите семантику операции обращения к элементам класса для статических и динамических объектов.
12. Что означает «сокрытие информации за барьером абстракции»?
13. Как класс обеспечивает защиту от нежелательного изменения?
14. Что такое общедоступный интерфейс класса? Как обеспечить класс таким интерфейсом?
15. Какие поля класса должны быть `private`, а какие `public`?
16. Что такое статический элемент класса?
17. Что такое аксессор? Для чего нужны аксессоры?
18. Что означает термин «спецификация класса»? Что содержит спецификация класса?
19. Для каких типов данных в C++ разрешена инициализация?
20. Как инициализировать переменную объектного типа?
21. Дайте определение конструктора. Какова роль конструктора?

22. Дайте определение деструктора. Какова роль деструктора?

23. Каковы особенности именования конструктора и деструктора?

24. Какого типа значения возвращают конструктор и деструктор?

25. Какова цель перегрузки функции? На чем основан механизм реализации перегруженных функций?

26. Зачем объекту нужен конструктор по умолчанию?

27. Какова роль конструктора с параметрами?

28. Сколько конструкторов должен иметь класс?

29. Какова роль конструктора копирования? В каких случаях необходимо обязательно описать такой конструктор?

30. Что означает термин «поверхностное копирование»?

31. Зачем нужны `static` поля класса? Каков механизм действия статического поля класса?

32. Зачем нужны дружественные функции?

33. Назовите два условия, обеспечивающие отношение дружбы между классом и функцией.

34. Чем отличается синтаксис операции-функции от синтаксиса обычной функции?

35. В чем особенность перегрузки операций присваивания и разыменования: `operator =`, `operator [ ]` и `operator ->`?

36. Как перегружается операция, если параметры разного типа?

37. Что такое внутренний класс? В каких случаях имеет смысл использование внутренних классов?

2.9. Упражнения ко второму разделу

Упражнение 1. Инкапсуляция

1. Опишите класс «Треугольник». Данными класса являются длины отрезков, составляющих стороны треугольника, в общем случае типа `float`.

2. Определите методы объекта.

Три конструктора: конструктор по умолчанию, конструктор равностороннего треугольника, конструктор треугольника с произвольно заданными сторонами.

Внутренний метод логического типа для проверки условия существования треугольника. Этод метод должен быть применен везде, где определяется треугольник.

Интерфейсные методы: ввод треугольника, вывод на экран, определение периметра, определение площади треугольника.

3. Объявите равносторонний и произвольный треугольники как статические объекты созданного типа, найдите площадь и периметр каждого.

4. Объявите динамический объект по умолчанию, введите данные и выведите на экран, найдите периметр и площадь.

5. Объявите и инициализируйте массив из 2-3 треугольников. Выведите все треугольники на экран, найдите площадь каждого.

Указания. Проверка условия существования треугольника, это логический метод.

```
bool Is_Triangle () {
{
  return a<b+c && b< a+c && c<a+b;
}
```

Его следует сделать внутренним, потому что логика объекта должна не позволить в принципе создать неправильный объект. Вызывается везде, где объект создается или изменяются его стороны. Такими методами являются конструктор с параметрами и метод ввода. Именно они должны заботиться о создании правильного экземпляра объекта.

Упражнение 2. Перегрузка операций

1. Опишите класс «Рациональная дробь». Данными класса являются числитель и знаменатель дроби, натуральные числа.

2. Определите методы объекта.

Методы должны обеспечить минимальную функциональность объекта: конструкторы пустой дроби и с параметрами, ввод, вывод.

В реализации методов необходимо контролировать данные объекта:

- знаменатель всегда отличен от 0;
- необходимо иметь метод приведения дроби в случае, когда числитель больше знаменателя, с отбрасыванием целой части;
- необходимо иметь метод сокращения, когда числитель и знаменатель кратны, например, дробь 5/15 , это 1/3.

3. Перегрузите операции ввода и вывода в поток.

4. Перегрузите операцию сложения дробей с использованием метода класса, а операцию вычитания с использованием дружественной функции.

5. Перегрузите унарную операцию ++ для инкремента числителя дроби как метод класса, а −− как дружественную функцию.

6. Перегрузите операцию сравнения == для дробей.

7. Перегрузите операцию сравнения > для дробей.

8. Перегрузите операцию присваивания для дробей.

9. Проверьте работу всех операций на статических и динамических объектах.

Указания. Как и в предыдущем задании, методы проверки, приведения и сокращения нужно сделать внутренними. Это тем более важно потому, что значения дробей порождаются при выполнении операций, и могут быть такими, что потребуют внесения изменений в объект. Следовательно, объект сам контролирует свое состояние, исключая возможные ошибки.

Упражнение 3. Разработка класа

Этот пример часто используется в учебниках и статьях при описании основ объектного подхода, потому что его «объектность» очевидна, логична и не требует пояснений.

1. Опишите класс «Комплексное число».

Данными класса являются действительная и мнимая части комплексного числа, тип `float`.

2. Определите методы объекта:

конструктор по умолчанию, конструктор действительного числа, конструктор комплексного числа;

перегруженные операции >> ввода и << вывода в поток;

определение модуля комплексного числа.

3. Объявите действительное число и произвольное комплексное число как статические объекты созданного типа, найдите модуль каждого. Объявите динамический объект, введите данные, найдите модуль.

4. Перегрузите операции сложения и вычитания комплексных чисел.

5. Перегрузите операцию присваивания для комплексных чисел.

6. Перегрузите операцию сравнения = = для комплексных чисел.

7. Перегрузите унарные операции ++ и – – .

8. Проверьте работу всех операций на статических и динамических объектах.

Упражнение 4. Упражения на развитие объектного видения

В этом упражнении основная цель – научиться абстрагировать объект исследования, представить модель. В модели нужно определить атрибуты, однозначно описывающие объект: какие именно, какие их типы. Далее определить, что объект может сделать для себя (внутренние методы), и каким будет интерфейс объекта. Итогом может быть спецификация класса.

Пример. Расстояние (длина) измеряется в километрах и метрах, например, длина пути = 5 км., 300 м.

Представление данных очевидно: число километров и метров, целые числа. Методы: проведя измерение, получаем длину: может быть равна нулю, или измерена только в км. – (3 км.,0 м.), или полностью – (3 км.,100 м.). Если данные заданы некорректно, следует сделать приведение, например, длина = (3 км.,1057м.), это (4 км.,57 м.). Это должен контролировать класс. Можно реализовать преобразование типа к вещественному значению, например, (4 км.,57 м.), это 4,057 км. Можно вычислить длину в метрах: (4 км.,57 м.), это 4057 м.

```
class Dlina {
int km,m;        // Данные.
void Correct(); // Проверка и приведение.
public:
Dlina();
Dlina(1);        // Метров =0.
Dlina(1,500);
double KM();     // Преобразование к вещественному.
int M();         // Длина в метрах.
// И другие методы.
```

```
}// End of Dlina.
```

Задание 1. Опишите класс «Угол» для измерения углов.

Задание 2. Опишите класс «Вектор».

Задание 3. Опишите класс «Функция». Пусть функция может быть линейной $y=a\ x+b$ или квадратичной $y=a\ x^2+b\ x+c$.

Задание 4. Опишите класс «Вещество».

Пусть одной из характеристик будет проводимость (проводник, полупроводник, изолятор).

Задание 5. Опишите класс «Сигнал».

Пусть сигнал имеет синусоидальную форму $y=Asin(x+\varphi)$.

Задание 6. Опишите класс «Тара».

Важные характеристики тары: материал изготовления, вместимость.

Задание 7. Опишите класс «Контейнер для хранения».

Важные характеристики: размеры, материал изготовления, вместимость.

Для развития объектного видения рекомендуется тренироваться на любых предметах, явлениях, сущностях окружающего мира.

3. НАСЛЕДОВАНИЕ КЛАССОВ

В обыденном смысле наследование, это:

- когда один объект имеет черты другого (мы говорим «сын унаследовал черты матери»);
- когда какие-то вещи достаются дочернему объекту (дочь носит мамину шубку, берет папин автомобиль);
- когда один объект находится в подчинении другого (фирма имеет дочерние филиалы, а в отделе всегда есть начальник);
- когда один объект содержит в себе другие (семья, это все родственники близкие и дальние).

Человеку свойственно абстрактное мышление, в рамках которого легко анализировать прикладную задачу. Легко классифицировать расположение понятий в иерархии наследственных связей, где каждый дочерний вид является более специализированным, чем его прямые и косвенные родители, и более общим, чем все его потомки. Эта способность человеческого разума используется в объектно-ориентированном проектировании, чтобы выстроить модель задачи некоторой предметной области.

В отличие от функциональной модели, объектная модель предполагает, что задача строится как совокупность объектов (понятий), взаимосвязанных друг с другом. Инструменты объектно-ориентированного программирования основаны на классах. Класс описывает модель некоторого понятия предметной области. Но понятия существуют во взаимосвязи, например, понятия «окружность», «треугольник», «квадрат» являются геометрическими фигурами, а это более общее понятие, описывающее соотношение объектов. В данном случае это иерархия.

В ООП для представления отношений между объектами существует могучий инструмент наследования, механизмы реализации которого будут обсуждаться в этом разделе. Использовать эти механизмы можно двумя способами.

Во-первых, механизмы наследования могут использоваться утилитарно: можно определить новые классы добавлением возможностей к уже имеющемуся классу, не меняя существующую объектную модель.

Пример. Объект «Время» содержит атрибуты – «Часы», «Минуты», «Секунды», а также формат представления, например – чч.мм.сек.

Расширением объекта «Время» является объект «Мировое время», к которому добавлен атрибут – «Часовой пояс». Могут быть добавлены форматы представления – европейский, американский.

Во-вторых, механизмы наследования могут использоваться идеологически. Наследование позволяет построить объектную модель задачи, наиболее приближенную к предметной области, и описывающую отношения между объектами приложения.

Пример. Объекту «Фирма» **подчинены** объекты «Дочерние филиалы». Объект «Учебное заведение» **содержит** дочерние объекты – факультеты, кафедры, отделы.

Наконец, наследование позволяет обеспечить общий интерфейс для нескольких различных классов так, чтобы клиентская программа могла работать с объектами этих классов одинаковым образом. Так определяется абстрактная общность объектов классов, но это уже свойство полиморфизма, которое будет обсуждаться в следующем разделе.

Пример. Объект «Часы» относится к классу часов. Часы можно разделить на категории по принципу работы: атомные, электронные, механические, солнечные. Это иерархия.

Время, которое показывают часы, это атрибут, отражающий состояние объекта. Выводится посредством интерфейса объекта. Применительно к часам примером полиморфизма может быть конкретная реализация интерфейса: часы со стрелочным, цифровым циферблатом, песочные часы, гаджет на экране (программная реализация интерфейса часов). Любая реализация позволяет узнать время (атрибут), независимо от типа объекта.

3.1. Реализация наследования в C++

Если классы находятся в отношении наследования, то и объекты классов также будут находиться в иерархическом отношении. Процедура порождения производного класса называется наследованием классов. Новые классы определяются на основе уже имеющихся, базовых классов. Порождающие классы называют еще родительскими, а порожденные – дочерними, производными или классами потомками. Производный класс обладает всеми свойствами базового класса (данными и методами), он их наследует, а также своими уникальными свойствами.

Наследуемые компоненты не перемещаются в производный класс, а остаются в базовом. На рисунке 8 показано соотношение объектов базового и производного классов, и приведены пояснения к механизму наследования.

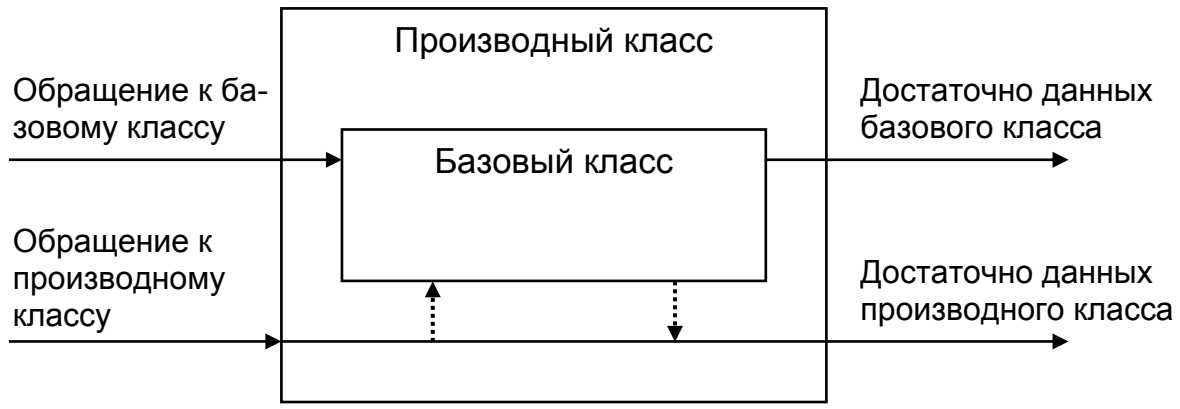


Рис. 8. Механизм наследования классов

При обращении к базовому классу используются его методы и данные. При обращении к производному классу используются его методы и данные, а если их недостаточно, автоматически происходит обращение к базовому классу.

Статус доступа к данным и методам класса при наследовании чрезвычайно важен. Напомним соглашение о доступе к компонентам класса.

Собственные (`private`) методы и данные доступны только внутри того класса, где они объявлены (локальные данные). Это механизм защиты данных. Исключением являются друзья класса.

Общедоступные (`public`) компоненты можно использовать вне класса. Это глобальные данные, так реализуется принцип открытого интерфейса.

Защищенные (`protected`) компоненты класса доступны внутри класса и во всех его производных классах, но вне класса недоступны.

Определение производного класса

В описании класса приводится через «:» список базовых классов, из которых он наследует данные и методы.

Пример. Определен базовый класс:

```
class Base {
private:
  // Закрытые поля;
protected:
  // Защищенные поля;
public:
  // Открытые поля;
}; // End of Base
```

Производные классы порождаются на основе этого определения. Должна быть объявлена схема наследования, которая определяет права доступа к объектам базового класса из производного класса. Существует четыре схемы наследования, для объявления которых используются метки прав доступа.

Способы доступа в наследуемых классах

1. Без метки доступа у имени класса.

```
class Derive: Base {

}; // End of Derive
```

Поля `protected` и `public` наследуются как `private`.

Поля `private` недоступны.

2. Защищенное наследование (наследует из `protected` класса).

```
class Derive: protected Base {

}; // End of Derive
```

Поля `protected` и `public` наследуются как `protected`.

Поля `private` недоступны.

3. Открытое наследование (наследует из public класса).

```
class Derive: public Base {

}; // End of Derive
```

Поля public наследуются как public.

Поля protected наследуются как protected.

Поля private недоступны.

4. Закрытое наследование (наследует из private класса).

```
class Derive: private Base {

}; // End of Derive
```

Поля protected и public наследуются как private.

Поля private недоступны.

Третий вариант, открытое наследование, наиболее естественный, но исходя из задачи, остальные варианты бывают необходимы.

В любом случае поля private базового класса недоступны в дочернем.

Замечание. Класс может быть объявлен не с помощью ключевого слова class, а struct. Тогда механизм наследования прав доступа в первом случае будет другой, а именно, поля public и protected будут наследоваться как открытые (public).

Замечание. При объявлении объекта класса вызывается конструктор, который инициализирует поля объекта при его создании. Порождение объекта производного класса вызывает порождение объекта базового класса, следовательно, из дочернего конструктора вызывается конструктор базового класса. Эта особенность накладывает требования к реализации конструкторов, которые будут обсуждаться позже в этом разделе.

Замечание. Имена методов и данных базового и порожденного классов могут перекрываться, например, часто используются такие имена, как Set, Input, Output. Согласно правилам локализации данных, в этом случае имена базового класса не видны из производного. Для разрешения проблемы используется оператор разрешения области видимости «::», который «присоединяет» имя к объекту, в котором тот определен. Синтаксис обращения из производного класса будет такой:

```
Имя_класса_предка :: Имя_метода_предка()
```

Прямое наследование

Когда в коде программы объявлен объект базового типа, то создается именно базовый объект, который не имеет потомков. Когда в коде программы объявлен дочерний объект, то объект его базового класса будет автоматически создан согласно механизмам наследования. При создании объектов вызываются констукторы. В случае наследования конструктор дочернего класса будет вызывать конструктор базового класса.

Пример. Рассмотрим открытое наследование.

```
class Base {
protected:
int    a;        // Поля защищены, значит,
float  b;        // доступны в производных классах.
public:
// Конструкторы базового класса.
Base() {
  a = 1;
  b = 1.;
}
Base(int a1, float b1) {
  a = a1;
  b = b1;
}
void Out() { // Обычное имя для функции вывода.
  cout << "Base: " << "a=" << a << " b=" << b << endl;
}
}; //End of Base
// Дочерний класс открыто наследует базовому.
class Derive: public Base {
int a;      // Имя поля перекрывается в дочернем классе.
float x,y;
public:
// Конструкторы дочернего класса.
// 1. Вызовет конструктор по умолчанию класса Base.
Derive() {
  a = 100;
  x = y = 0.;
}
// 2. Вызовет конструктор с параметрами класса Base.
// Параметры разделяются между объектами.
Derive(int aa, float x, float y, int a1, float b1):Base (a1, b1)
{
  a = aa;
  this->x = x;
  this->y = y;
}
// Метод вывода можно реализовать по-разному.
// 1. Здесь выводятся поля базового класса, потому что
// они protected, и доступны из производного.
void Out() { // Обычное имя для функции вывода.
  cout << "Base: a=" << Base::a << " b=" << b << endl;
  cout << "Derive: ";
  cout <<"a=" << a << " x=" << x << " y=" << y << endl;
}
// 2. Здесь используется метод Out базового класса.
```

```
// void Out() { // Обычное имя для функции вывода.
//   Base::Out();
//   cout << "Derive: ";
//   cout <<"a=" << a << " x=" << x << " y=" << y << endl;
//}
}; // End of Derive
void main(void) {
// Объекты базового класса существуют сами по себе.
Base A;
A.Out();
Base B(11,22.5);
B.Out();
// Объект дочернего класса создает базовый объект.
// Вызывается конструктор по умолчанию базового класса.
Derive C;
C.Out();
// Объект дочернего класса создает базовый объект.
// Вызывается конструктор с параметрами класса Base.
Derive D(1,11.5,2,0.,0.);
D.Out();
}
```

В этом примере иллюстрируются следующие особенности.

1. Данные `protected` базового класса доступны в производных классах, поэтому в методах `Derive` можно к ним обращаться напрямую.

2. Имена полей могут перекрываться (поле `a`). Чтобы определить, кому принадлежит поле, используется оператор «`::`». Так, `Base::a` в дочернем классе обращается к переменной `a`, принадлежащей базовому классу.

3. Используются конструкторы базового и производного классов. Конструктор по умолчанию дочернего объекта вызывает конструктор по умолчанию базового класса. Конструктор с параметрами дочернего объекта вызывает конструктор с параметрами базового класса.

Выполняя этот код в отладчике, можно увидеть механизмы создания объектов, и наблюдать последовательность вызовов конструкторов.

Особенности конструктора порожденного класса

При описании иерархии объектов используются конструкторы, заданные по умолчанию или явно. Когда объявлен дочерний объект, то вызывается его конструктор. Но конструктор дочернего класса будет вызывать конструктор базового класса, причем этот вызов осуществляется раньше, чем дочерний объект создается.

Базовый класс может иметь несколько перегруженных конструкторов.

В случае конструкторов с параметрами необходимо использовать схему распределения параметров между объектами иерархии.

Если объявление конструктора базового класса имеет вид:

```
Имя_базового_класса (формальные_параметры),
```

то синтаксис объявления конструктора производного класса имеет вид:

```
Имя_производного_класса (формальные_параметры):
  Имя_базового_класса (фактические_параметры)
```

Здесь для конструктора производного класса необходимо задать такое число параметров, чтобы их хватило для инициализации полей производного класса и полей базового класса, согласно сигнатуре параметров всех объявленных и используемых конструкторов.

В списке параметров производного класса перечисляются все параметры, необходимые для инициализации его самого и его предка, а в списке параметров базового класса перечисляются без указания типа (!) имена параметров, передаваемые конструктору базового класса. Объявление типов не нужно, потому что в этой схеме не объявляется конструктор базового класса, а происходит обращение к нему из конструктора производного класса.

Взаимодействие различных перегруженных конструкторов может быть разнообразным. Конструктор по умолчанию производного класса может использовать конструктор с параметрами базового класса, и наоборот.

Механизмы порождения объектов можно наблюдать в отладчике. На рисунке 9 показано содержимое созданных в примере объектов.

Имя	Значение
A	{a=1 b=1.0000000 }
a	1
b	1.0000000
B	{a=11 b=22.500000 }
a	11
b	22.500000
C	{a=100 x=0.00000000 y=0.00000000 }
Base	{a=1 b=1.0000000 }
a	1
b	1.0000000
a	100
x	0.00000000
y	0.00000000
D	{a=1 x=11.500000 y=2.0000000 }
Base	{a=0 b=0.00000000 }
a	0
b	0.00000000
a	1
x	11.500000
y	2.0000000

Рис. 9. Механизм порождения объектов при прямом наследовании

Объекты `C` и `D` содержат в себе объекты базового класса, в этом смысле базовый класс является частью производного класса.

Конструктор копии при наследовании

В разделе 2.3. обсуждались случаи, когда в классе обязательно должен быть конструктор копирования. Часто бывают ситуации, когда несколько дочерних объектов должны иметь одинаковое значение базового объекта.

Пусть есть базовый класс «Параллелепипед». Дочерним к нему является класс «Деталь», имеющий форму кирпичика, и собственные данные – материал изготовления, плотность.

```
class Parallelepiped {
double a,b,c;        // Размер.
// …
};// End of Parallelepiped.
class Detail {
std::string Material;
double Density;
// …
}; // End of Detail.
```

Требуется создать несколько объектов одинакового размера, но разного материала изготовления, например:

```
Detail A[0](1., 1., 10., "Сталь", 1.98);
Detail A[1](1., 1., 10., "Алюминий", 1.08);
Detail A[3](1., 1., 10., "Стекло", 1.22);
```

и так далее. Все объекты имеют одинаковые размеры, значит, можно породить один параллелепипед:

```
Parallelepiped P(1., 1., 10.);
```

а во всех деталях получать его копию:

```
Detail A[0](P, "Сталь", 1.98),
Detail A[1](P, "Алюминий", 1.08),
Detail A[2](P, "Стекло", 1.22).
```

Это тот случай, когда объект `P` передается в функцию, где должна быть создана его точная копия. Такой прием не только сокращает код, но и подчеркивает логику задачи: детали одинакового размера.

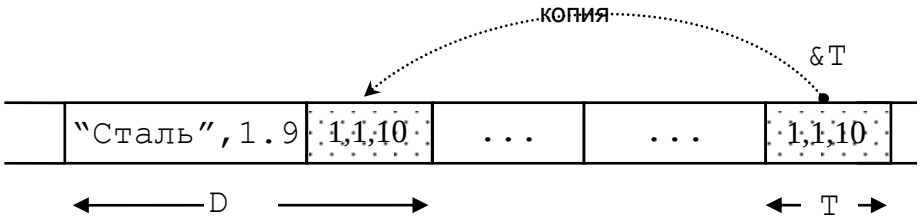
Конструктор дочернего объекта получает ссылку (`&`) на объект базового класса и передает в конструктор базового класса:

```
Detail(Parallelepiped &T,string mt,double Dd):Parallelepiped(T)
{
  Material = mt;
  Density = Dd;
}
```

&`T` передается базовому классу. Следовательно, тот должен иметь собственный конструктор копии: это специальный конструктор, который позволяет получить объект, идентичный к заданному, то есть получить копию уже существующего объекта.

```
Parallelepiped(Parallelepiped &T) {
  a = T.a;
  b = T.b;
  c = T.c;
}
```

В этом случае создается точная копия параллелепипеда `T` во всех дочерних объектах. Пример приведен на рисунке 10.



Дочерний объект:
```
Detail D (T, "Сталь",1.97)
```

Объект базового класса:
```
Parallelepiped T(1,1,10);
```

Рис. 10. Иллюстрация к конструктору копии

Конструктор копирования вызывается при объявлении объектов дочерних классов и получает копию T.

3.2. Классификация видов наследования

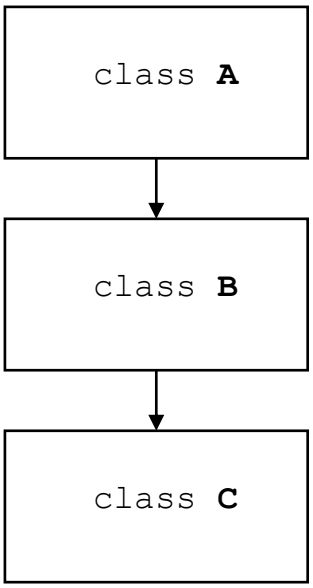
Любой производный класс может быть базовым для другого класса, при этом производные классы имеют доступ к своему родителю и родителю родителя. Так порождается цепочка наследования, в которой каждый класс может наследовать свойства и методы нескольких базовых классов и каждый класс может порождать несколько производных классов.

Наследование может быть прямым, косвенным, множественным и контейнерным. Рассмотрим реализацию различных схем наследования.

Косвенное наследование классов

Косвенное наследование – один из вариантов множественного наследования, предполагает наличие более сложной иерархии объектов. Иллюстрация схемы косвенного наследования приведена на рисунке 11.

Классу `A` наследует класс `B`, и классу `B` наследует класс `C`. Классу `C` доступны открытые и защищенные поля класса `B`, а классу `B` доступны открытые и защищенные поля класса `A`. Таким образом, дочернему классу `C` поля класса `A` доступны косвенно.




```
class B: public A
{
};
```

```
class C: public B
{
};
```

Рис. 11. Схема косвенного наследования классов

Приведем пример, иллюстрирующий правила реализации этой схемы. Класс `Base` содержит два поля данных и имеет наследника с именем `Son_of_Base`. Тот тоже имеет наследника с именем `Grandson_of_Base`, косвенно наследующего `public` и `protected` поля базового класса.

```
class Base {
static std::string Head; // Статический «заголовок».
protected:       // Доступны в дочерних классах.
int    a;
float  b;
public:
Base() {
  a = 1;
  b = 1.5;
}
void Out() {
  cout << "Я объект: ";  // Принадлежность.
  cout << Head;
  cout "a=" << a " b=" << b << endl;
}
// Метод вывода заголовка принадлежит базовому классу.
static void Out_head() {
  cout << Head;
}
}; // End of Base
// Сыновний класс открыто наследует Base.
class Son_of_Base: public Base {
protected:
int    a;            // Другой a.
int    b;            // Другой b, тип int.
```

```
public:
Son_of_Base() {
  a = 10;
  b = 50;
}
// Собственный метод с перекрывающим именем.
void Out() {
  cout << "Я сын: \n");  // Принадлежность.
  cout "a=" << a " b=" << b <<endl;
}
// Вывод информации о родительском классе.
void Out_my_ancestry() {
  cout << "Я сын. Мои предки:\n";
  cout << "Папа: ";
  Base::Out();  // Метод родительского класса.
}
}; // End of Son
// Son_of_Base имеет потомка, открыто наследующего.
class Grandson_of_Base: public Son_of_Base {
private:
int    a;            //Другой a.
char   c;
public:
Grandson_of_Base() {
  a = 100;
  c = '@';
}
// Собственный метод с перекрывающим именем.
void Out() {
  cout << "Я внук: \n");      // Принадлежность.
  cout "a=" << a " c=" << c << endl;
}
// Вывод информации о родительском классе.
void Out_my_ancestry() {
  cout << "Я внук. Мои предки:\n";
  cout << "Мой папа: ";
  // Обращение к методу прямого предка.
  Son_of_Base::Out();
  cout << "Мой дедушка: ";
  // Обращение к методу косвенного предка.
  Base::Out();
}
// Доступ к protected полям базовых классов напрямую.
void Out_a() {
  cout << "Поле Base: " << Base::a << endl;
  cout << "Поле Son_of_Base: << Son_of_Base::a << endl;
  cout << "Поле Grandson_of_Base: " << a << endl;
```

```
}
}; // End of Grandson
// В объектной модели показываем особенности.
void main(void) {
Base t1;                    // Объект.
Son_of_Base t2;             // Сын объекта.
Grandson_of_Base t3;        // Внук объекта.
// Инициализация статического элемента класса.
Base::Head = "Заголовок\0";   // Head принял значение.
// 1. Имена методов и данных повторяются в наследуемых
// классах. Обращение происходит именно к функции
// объекта того типа, для которого выполняется вызов.
cout <<"\nРис.1. Иллюстрация: названия одинаковы. \n";
  t1.Out();
  t2.Out();
  t3.Out();
// 2. Обращение к статическому методу базового класса.
// Наследники пользуются методом предка.
// Метод описан как public в базовом классе, значит,
// доступен объектам всей иерархии.
cout <<"\nРис.2. Обращение к статическому методу.\n";
  t1.Out_head();
  t2.Out_head();
  t3.Out_head();
// 3. Обращение к методам базовых классов.
cout << "\nРис.3. Обращение к методам предков.\n";
  t2.Out_my_ancestry();
  t3.Out_my_ancestry();
// 3. Иллюстрация принципа локализации имен.
// Имена полей повторены во всех производных классах:
// имя поля a перекрывается в каждом порожденном классе
// Наследование этого компонента не выполняется.
// Разрешим его увидеть из производных классов.
cout <<"\nРис.4. Обращение к полям предков.\n";
  t3.Out_a();
}// main
```

Особенности порождения объектов дочерних классов и механизмов вызова методов хорошо видны в отладчике, что показано на рисунке 12. Здесь отчетливо видно содержимое всех объектов иерархии.

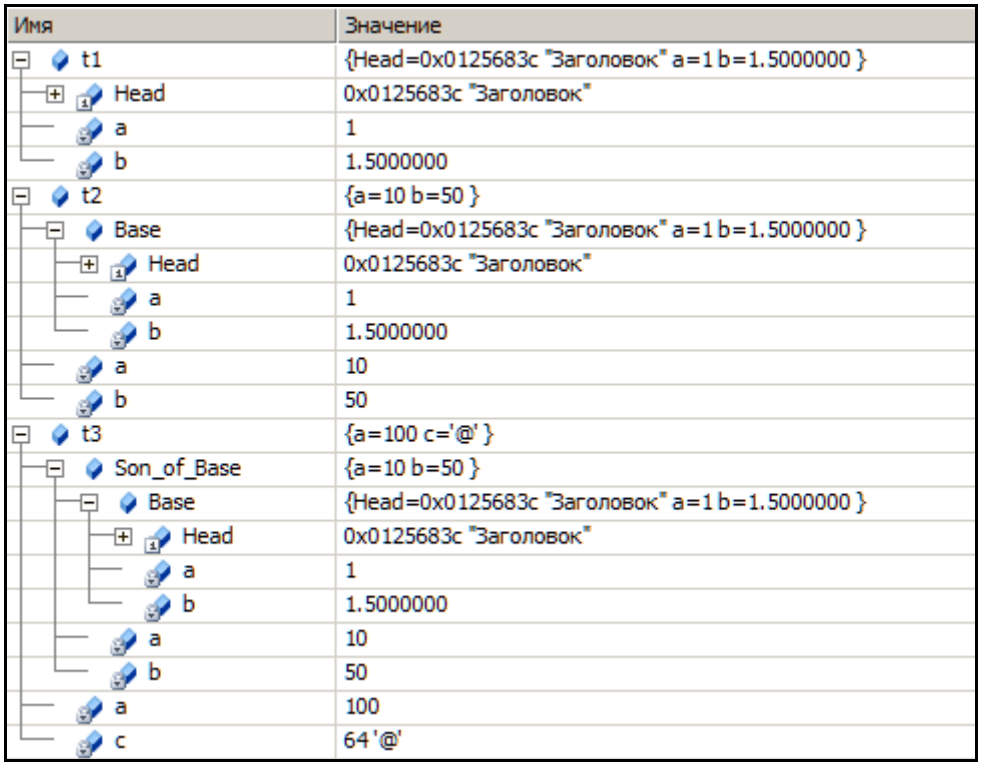


Рис. 12. Механизм порождения объектов при косвенном наследовании

В этом примере иллюстрируются следующие особенности:

- для создания объектов используются конструкторы по умолчанию, и исследуется механизм вызовов конструкторов в схеме наследования;
- поля данных базовых классов имеют метку `protected`, что позволяет напрямую использовать эти поля в дочерних классах;
- подчеркнуты особенности локализации имен: повторяются имена полей (`a`, `b`), хотя они имеют разные типы, и имена методов (`Out()`);
- статический элемент `Head` (заголовок) показывает, что механизм наследования не мешает механизму статических данных классов.

Замечание. Существенным недостатком примера является тот факт, что конструкторы объектов иерархии не имеют параметров. Для хорошей функциональности модели нужны конструкторы с параметрами. Конструктор каждого дочернего класса создает объект верхнего уровня иерархии, и передает параметры.

```
Base(int a1, float b1) {
  a = a1;
  b = b1;
}
```

Конструктор класса `Son_of_Base` передает параметры конструктору класса `Base`.

```
Son_of_Base(int aa, int bb, int a1, float b1): Base(a1, b1){
  a = aa;
  b = bb;
```

```
}
```

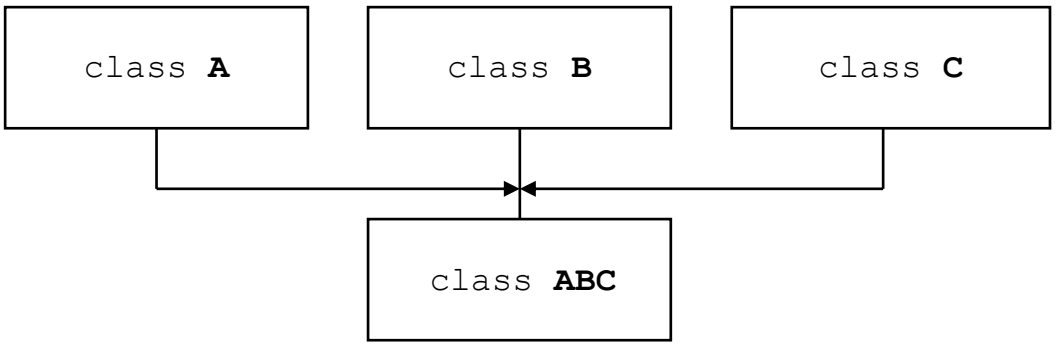
Конструктор класса `Grandson_of_Base` передает параметры конструктору класса `Son_of_Base`, а тот передает данные в базовый класс. Вызов конструктора `Base` неявный.

```
  Grandson_of_Base(int aa, int bb, int a1, float b1,
int A, char cc): Son_of_Base(aa, bb, a1, b1) {
    a = A;
    c = cc;
}
```

Множественное наследование классов

Множественное наследование предполагает наличие в иерархии объектов нескольких базовых классов у одного объекта – «многие к одному», или нескольких дочерних классов у одного объекта – «один ко многим». Могут также существовать сочетания разных видов наследования.

Рассмотрим случай, когда объект наследует нескольким базовым классам. Схема такого наследования приведена на рисунке 13.




```
class ABC: public A, public B, public C {
};
```

Рис. 13. Множественное наследование

Объект наследует трем базовым классам. При объявлении наследования перечисляются все базовые классы:

```
class ABC: public A, public B, public C {
  // Тело класса.
};
```

Особенность конструктора дочернего объекта заключается в том, что он должен создать все родительские объекты, сколько у него есть. Он должен иметь такое количество параметров, какое необходимо для создания всех своих предков и для себя. Будет иметь общий список параметров, из которого разделяет, кому какие, в том числе берет и себе.

Параметры конструктора производного класса объявляются с именами типов, так как это формальные параметры, а параметры конструкторов базовых классов − без имен типов, потому что это фактически вызов конструкторов базовых классов.

Примерный синтаксис объявления конструктора такой:

```
Имя_производного_класса (формальные_параметры) :
  Имя_базового_класса_A (фактические_параметры_A),
  Имя_базового_класса_B (фактические_параметры_B),
  Имя_базового_класса_C (фактические_параметры_C);
```

Сигнатура параметров конструктора производного класса определена сигнатурами параметров конструкторов базовых классов.

Пример. Два родительских класса, которым наследует один дочерний.

```
class Mother {
int    a;
float  b;
public:
// Конструктор первого родителя.
Mother(int a1, float b1) {
  a = a1;       // Параметры задают данные двух полей.
  b = b1;
}
void Out(void) {
  cout << "Мама: a=" << a << " b=" << b << endl;
}
}; // End of Mother
class Father {
int    a;
char   c;
public:
//Конструктор другого родителя.
Father(int a1, char c1) {
  a = a1;       // Параметры задают данные двух полей.
  c = c1;
}
void Out(void) {
  cout << "Папа: a=" << a << " c=" << c << endl;
}
}; // End of Father
// Сыновний объект имеет двух родителей,
// у каждого свой конструктор.
class Son: public Mother, public Father {
int    n;   // Собственное данное
public:
// Конструктор потомка получает четыре параметра, из
// которых первый отдает Mother и Father, второй отдает
// Mother, третий отдает Father, четвертый берет себе.
Son(int x, float y, char c, int nn):
  Mother(x,y), Father(x,c) {
  n = nn;
}
```

```
void Out(void) {
  cout << "Я наследник, вот мои предки:\n";
  Mother::Out();     // Использует методы предков.
  Father::Out();
  cout << " n= " << n << endl;
}
}; // End of Son
// Здесь покажем, как при инициализации объекта
// создаются его родительские объекты.
void main(void) {
Son M (1,2.3,'w',99);
// Родительские объекты показать напрямую нельзя.
  M.Out();
} // main
```

Содержимое дочернего объекта показано на рисунке 14.

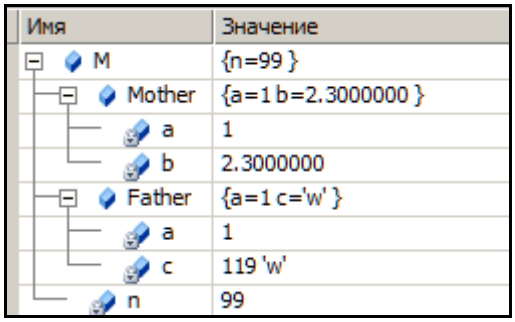
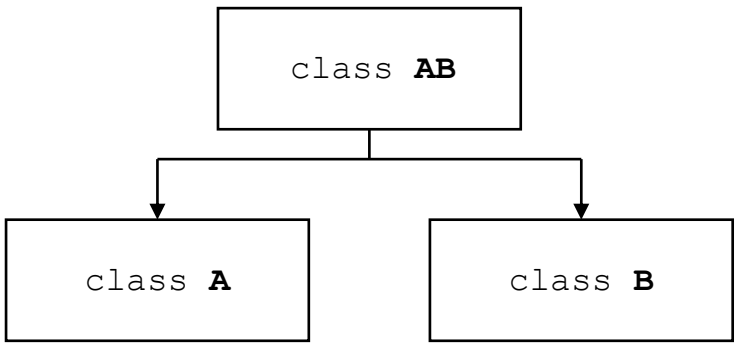


Рис. 14. Содержимое дочернего объекта

Оба родительских объекта входят в дочерний, и не видны напрямую.

Другой вариант множественного наследования предполагает, что одному базовому классу наследуют несколько дочерних. Эта ситуация приведена на рисунке 15.




```
class A: public AB   class B: public AB
```

Рис. 15. Множественное наследование классов

Объект порождает двоих наследников, и каждый из них наследует базовому классу. Синтаксис объявления обычный:

```
class A: public AB
class B: public AB
```

Пример. Объект родитель имеет двух потомков. Используются конструкторы по умолчанию и с параметрами.

```
class Father {
// Защищенные данные доступны наследникам класса.
protected:
int    a;
int    b;
public:
Father(void) {// Конструктор Father по умолчанию.
  a = 0;
  b = 0;
}
// Конструктор Father с параметрами.
Father(int a1, int b1) {
  a = a1;
  b = b1;
}
void Out(void) {
  cout <<"Класс Father: a=" << a << " b=" << b << endl;
}
}; // End of Father
// Первый наследник использует открытое наследование.
class Son1: public Father {
float c;        // Собственное данное.
public:
// Конструктор по умолчанию вызывает конструктор
// Father по умолчанию.
Son1(void): Father() {
  c = 0.0;
}
// Конструктор с параметрами вызывает конструктор
// Father с параметрами. Параметры разделяются.
Son1(int a, int b, float c1): Father(a,b) {
  c = c1;
}
// Можно изменить защищенные данные базового класса.
void Set(int a1, int b1) {
  a = a1; b = b1;
}
void Out(void) { // Используем метод базового класса.
  Father::Out();
  cout << "c=" << c << endl;
}
}; // End of Son1
```

```
// Второй наследник использует защищенное наследование.
class Son2: protected Father {
float x,y;      // Собственные данные класса.
public:
// Конструктор по умолчанию вызывает конструктор
// Father по умолчанию.
Son2(void): Father() {
  x = y = 0.;
}
// Конструктор с параметрами вызывает конструктор
// Father с параметрами. Параметры распределяются.
Son2(int a, int b, float xx, float yy): Father(a,b){
  x = xx;
  y = yy;
}
void Out(void) { // Используем метод базового класса.
  Father::Out();
  cout << "x=" << x << " y=" << y << endl;
}
}; // End of Son2
// Иллюстрируем использованные возможности.
void main(void) {
cout <<"1. Вызов конструкторов базового класса:\n");
Father a1, a2(10,10);
cout <<"Father по умолчанию:\n";
a1.Out();  //Метод Father.
cout << "Father с параметрами:\n";
a2.Out();  //Метод Father.
cout << "2. Конструкторы дочерних классов используют
конструкторы базового класса:\n";
Son1 b1, b2(20,30,0.4);       // Два конструктора Son1.
cout << "Son1 по умолчанию: ";
b1.Out();
cout << "Son1 с параметрами: ");
b2.Out();
Son2 c1, c2(50,60,1.5,1.6);
cout << "Son2 по умолчанию: ";
c1.Out();
cout << "Son2 с параметрами: ";
c2.Out();
// Можно изменить защищенное данное базового класса.
cout << "3. Son1 меняет значения полей предка: ";
b1.Set(111,222);
b1.Out();
}
```

Замечание. Каждый объект наследник при создании порождает для себя объект родительского класса. В рассматриваемой схеме наследования (см. рис. 15) показано, что оба сыновних объекта имеют общего предка. Это противоречит декларации о механизмах создания объектов и выполняется не в соответствии со схемой. А именно, у классов `Son1` и `Son2` будет не общий `Father`, а у каждого своя собственная копия родительского класса, свой собственный `Father`, что видно на рисунке 16.

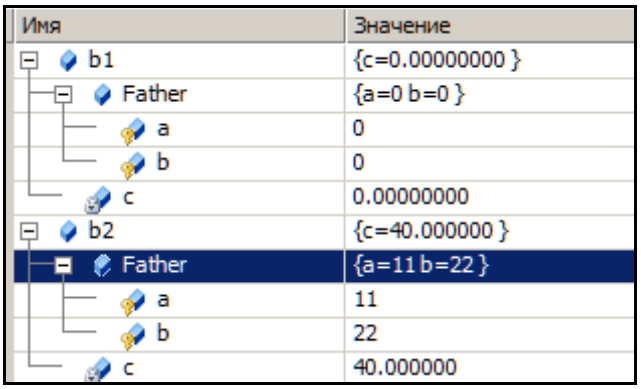


Рис. 16. Каждый дочерний объект имеет собственную копию базового

Чтобы избежать подобной ошибки (а это ошибка), в классическом C++ можно использовать виртуальное наследование:

```
class Son1: virtual public Father
class Son2: virtual private Father
```

Механизмы виртуального наследования кратко рассматриваются в следующем разделе.

В реализации управляемого C++/CLI множественное наследование запрещено, вместо него используются специальные классы − интерфейсы.

3.3. Правила наследования

Подведем итоги относительно механизмов наследованиия. Утверждение, что дочерний класс наследует все открытые и защищенные свойства и методы базового класса, верно не вполне. Не наследуются конструкторы, деструкторы и операция присваивания. Не наследуются также дружественные функции.

В классах наследниках абстрактное содержание и функциональность базовых классов расширяются путем определения новых полей и новых методов. В методах дочерних классов разрешается вызывать любые доступные методы базовых классов.

Если имя поля (данного или метода) в дочернем классе совпадает с именем поля базового класса, то действует принцип локализации имен. Прототипы методов могут не совпадать. Для обращения к полю родительского класса, его имя задается с префиксом класса, например:

```
Base::Out();
```

Статические поля, объявленные в базовом классе, наследуются обычным образом. Все объекты базового класса и всех его наследников разделяют единственную копию статических полей базового класса.

Поведение конструкторов в схеме наследования

1. Конструкторы классов не наследуются, поэтому каждый производный класс должен иметь собственные конструкторы.

2. Если конструктор базового класса имеет параметры, то он должен быть явно вызван в конструкторе производного класса. В списке параметров должны присутствовать параметры, инициализирующие поля дочернего и базового классов:

```
Derive(int aa, float x, float y): Base(x, y);
```

3. Если в конструкторе производного класса не прописан явный вызов конструктора базового класса, то автоматически вызывается конструктор базового класса по умолчанию.

4. Для иерархии, в которой более двух уровней, конструкторы базовых классов вызываются, начиная с самого верхнего уровня. После этого выполняются конструкторы элементов класса, которые являются объектами, в порядке их объявления в классе, а затем выполняется конструктор дочернего класса. Если базовых классов несколько, их конструкторы вызываются в порядке объявления.

5. Операцию присваивания при необходимости следует явно определить в дочерних классах.

6. В дочерних классах рекомендуется пользоваться вызовами методов базового класса. В случае совпадения имен, доступ к методу базового класса из производного выполняется через уточненное имя:

```
void Out() { // Обычное имя для функции вывода.
  Base::Out();
  cout << "Derive: ";
  cout << "a=" << a << " x=" << x << " y=" << y <<endl;
}
```

Поведение деструкторов в схеме наследования

Деструкторы не наследуются, и если в производном классе не описан деструктор, то он формируется по умолчанию и вызывает деструкторы всех базовых классов.

В коде деструктора производного класса не требуется явно вызывать деструкторы базовых классов, поскольку это будет сделано автоматически.

Для иерархии классов, состоящей из нескольких уровней, деструкторы вызываются в порядке, строго обратном вызову конструкторов: сначала вызывается деструктор дочернего класса, затем – деструкторы элементов класса, а потом деструктор базового класса.

Виртуальное наследование

Иерархия наследования классов в классическом C++ практически не имеет ограничений. Виды и формы наследования могут быть произвольными, что позволяет разрабатывать очень сложные структуры классов, выполняя декомпозицию от общего к частному или, наоборот, обобщать уже созданные классы, получая новые классы, отличные от исходных.

Возможно сочетание множественного и косвенного наследования. При этом класс может быть прямым базовым классом для нескольких классов, как ранее было рассмотрено в примерах, а также косвенным.

Рассмотрим возможное сочетание в примере схемы наследования, приведенной на рисунке 17. Объявление классов, соответствующее данной схеме, приведено в тексте рисунка.

Классы `A1` и `A2` являются прямыми потомками класса `A`. Они совместно порождают одного потомка, класс `C`. Класс `C` наследует классу `A` косвенно, через классы `A1` и `A2`.

Механизм наследования предполагает существование объекта базового класса в производном, отсюда существенные недостатки этой модели: внутри объекта класса `C` создаются объекты базовых классов: сначала `A1` и `A2`, далее в каждом из них создается объект прародитель `A`.

Вывод очевиден − имеет место дублирование объектов непрямого (косвенного) базового класса в производном.

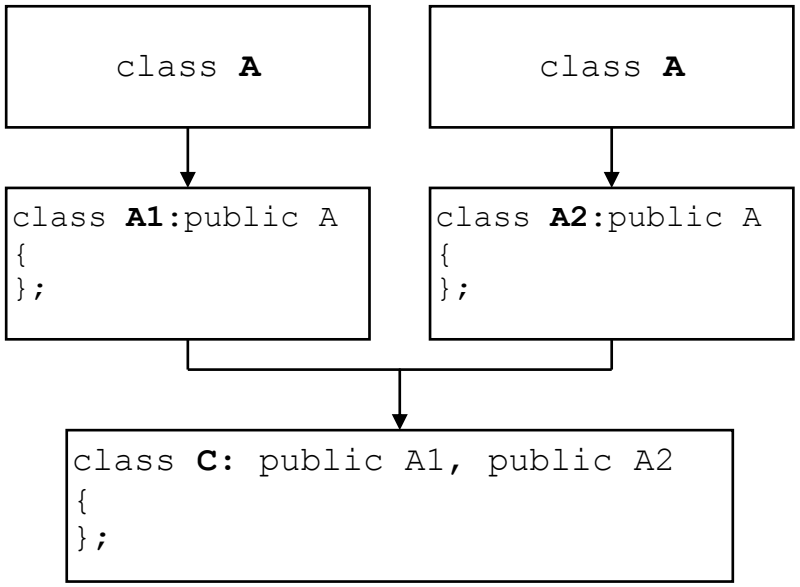


Рис. 17. Множественное и косвенное наследование

Механизмы реализации приведенной схемы наследования существуют, но почти всегда дублирование объектов непрямого базового класса является ошибочным, и нарушает логику. Для устранения коллизии используется виртуальный базовый класс. При виртуальном наследовании объект про-

изводного класса не содержит в себе объект базового класса, что показано на рисунке 18.

Ключевое слово `virtual` добавляется при объявлении наследования для объектов, которые являются прямыми наследниками общего базового класса (для `A1` и для `A2`). При этом базовый класс делается общим для своих прямых потомков, или виртуальным.

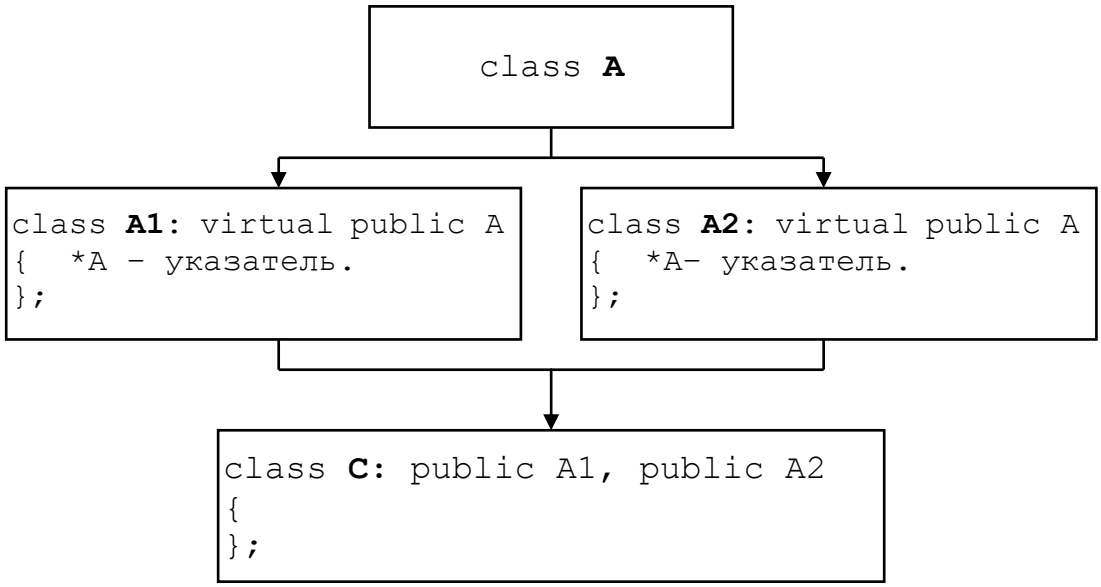


Рис. 18. Виртуальное наследование. Виртуальный базовый класс

Объект существует в единственном экземпляре для всех прямых наследников, объявляющих наследование виртуальным, а также для всех их наследников.

Механизм реализации виртуального объекта заключается в том, что объекты производных классов не содержат в себе базовые, но содержат указатель на объект базового класса. Следовательно, он действительно становится общим для всех своих потомков. Пример реализации данной модели приведен в [2, 5].

3.4. Композиция. Контейнерные классы

Отношение наследования между объектами называют иерархией. В иерархии одни объекты подчинены другим. Существует разновидность наследования, называемая композицией. Ее абстрактный смысл заключается в том, что один объект содержит в себе объекты других классов, или служит контейнером, вмещающим другие объекты.

Понятия, которые в объектной модели должны быть реализованы как контейнеры, включают в себя некоторое количество других понятий, возможно, разных типов. Так, «багаж», это набор вещей: сумок, чемоданов. Так, «записная книжка», это набор записей о друзьях и знакомых. Так, «ящик для инструментов», это контейнер для хранения разных полезных

вещей класса «инструмент». Объекты, заключенные в контейнер, в общем случае, могут быть разными по типу.

С точки зрения механизмов реализации иерархии и композиции, в C++ они одинаковы. Для иллюстрации похожести и отличия иерархии от композиции приведем рисунки.

Схема реализации чистого наследования приведена на рисунке 19. Подразумевается, что имеет место объявление:

```
class Derive: public Base
```

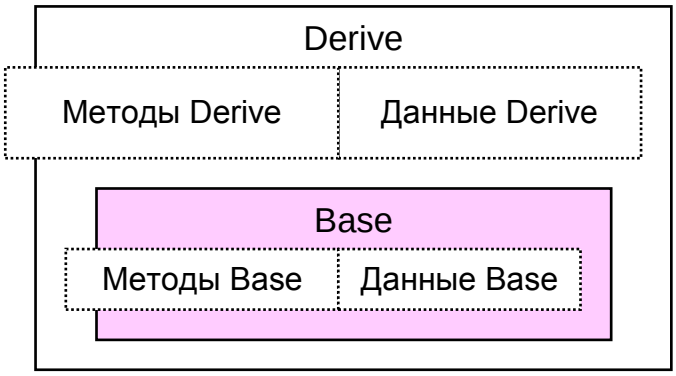


Рис. 19. Объекты в отношении наследования

Производный класс больше по объему, имеет доступ к данным и методам своего базового класса. В соответствии с механизмами реализации, дочерний объект порождает объект базового класса, который содержится внутри дочернего.

Схема реализации композиционного наследования приведена на рисунке 20. Подразумевается, что имеет место объявление:

```
class Container{
Inner Some;     // Inner – тип внутреннего объекта.
...
};
```

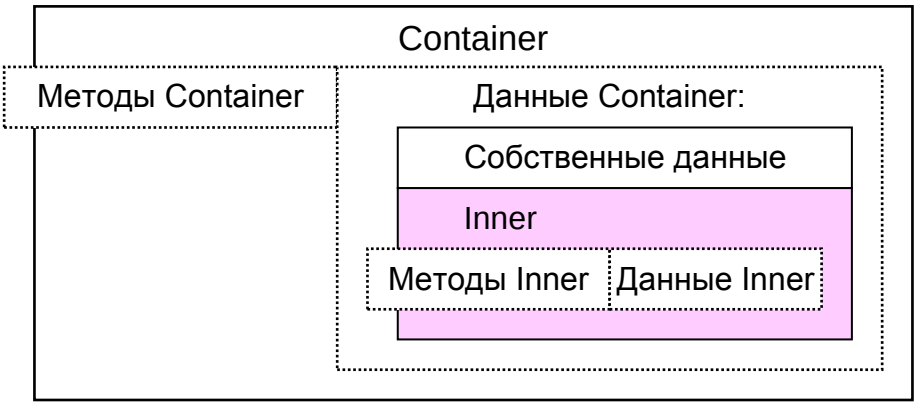


Рис 20. Объекты в отношении композиции

Класс контейнер больше по объему, и содержит внутри себя хотя бы один экземпляр другого класса (`Inner Some`), который по механизму реализации можно считать базовым. Все соглашения относительно механизмов наследования в этом случае остаются в силе. Класс контейнер абстрактно может считаться наследником своего внутреннего класса.

Существуют абстрактные контейнеры, которые служат очевидными примерами хороших классов, – это массивы, таблицы, множества, списки, словари и другие. Каждый такой класс содержит в себе некоторое количество абстрактных данных одного вида. Каждый такой класс имеет операции добавления в контейнер и извлечения из контейнера. Обычно имеется операция проверки: есть ли указанный элемент в контейнере. Могут быть определены операции просмотра элементов, поиска, упорядочивания, удаления и другие, их набор определен абстрактным смыслом контейнера.

При контейнерном наследовании существуют ограничения на доступ из контейнера к полям внутреннего класса. Класс контейнер имеет доступ только к открытым полям объекта, содержащегося внутри, но не имеет доступа к его закрытым данным.

Конструкторы контейнерного класса имеют отличие: внутренний объект создается как конкретный экземпляр.

Приведем пример реализации контейнерного класса.

```
class Inner {   // Класс внутри контейнера.
int    a, b;
public:
// Конструктор по умолчанию.
Inner(void) {
  a = 0;
  b = 0;
}
// Конструктор с параметрами.
Inner(int a1, int b1) {
  a = a1;
  b = b1;
}
// Для доступа к закрытым полям класса Inner.
void Set(int a1, int b1) {
  a = a1;
  b = b1;
}
void Out(void) {
  cout << "Класс Inner: a=" <<a " b=" << b << endl;
}
}; // End of Inner
// Класс контейнер содержит экземпляр класса Inner.
class Container {
float  c;       // Собственное данное Container.
```

```
Inner  AB;       // Экземпляр класса Inner.
public:
// Конструктор контейнера по умолчанию вызывает Set.
Container(void) {    // Методы контейнерного класса
  AB.Set(99,99);     // могут управлять объектом Inner
  c = 0.5;           // только с помощью его методов.
}
// Конструктор с параметрами:
// данные передаются экземпляру объекта.
Container(int a1, int b1, float c1): AB(a1,b1) {
  c = c1;
}
void Out(void) {
  AB.Out();          // Обращение к методу экземпляра.
  cout << "c=" << c <<endl;
}
}; // End of Container
// Пример работы с объектами Inner и Container.
void main(void) {
  cout << "Объекты класса Inner:\n";
Inner I1;
  I1.Out();
Inner I2(99,99);
  I2.Out();
  cout <<"Контейнерный класс:\n";
Container C1;                  // Экземпляр AB внутри.
  cout << "C1 по умолчанию:";
  C1.Out();
Container C2(11,12,99.99);     // Экземпляр AB внутри.
  cout << "C2 с параметрами:";
  C2.Out();
}
```

Вывод. Сложность реализации контейнерного класса в том, что методы контейнера обращаются к экземпляру объекта внутреннего класса, данные которого, как правило, защищены от доступа извне. Решением проблемы должно быть использование способов, разрешающих такой доступ: дружественные функции и классы или внутренние классы.

Пример контейнерного класса

Обычным примером контейнера является класс «Массив». Массив, это множество значений указанного типа в определенном количестве. Массив реализуется как последовательная структура данных с прямым доступом. Доступ к элементам массива обеспечивает операция разименования [].

Как абстрактная структура данных, массив предоставляет пользователю множество методов обработки данных произвольного типа, скрывая детали реализации.

Пример. Рассмотрим простой вариант – массив целочисленных значений. Хранит данные целого типа, имеет длину. Длина статического массива определяется при его объявлении и не может быть изменена. Длина динамического массива определяется при распределении памяти операцией `new` и, при необходимости, может быть изменена. Если длина динамического массива изменяется, то нужно выполнять действия, связанные с перераспределением памяти: выделить новый блок памяти в куче, копировать в него данные блока, связанного с массивом, освободить ранее занятую память.

Обычными операциями для массива являются: присвоить значения элементам массива, визуализировать содержимое, сортировать, выполнить поиск в массиве, добавить элемент, удалить указанный элемент, найти сумму элементов, и другие.

Введем понятия «емкость» и «длина» массива. Емкость, это количество данных, которое может поместиться в отведенной для массива памяти, – в коде, это константа `LEN_MAX`. Емкость не изменяется. Длина, это реальное количество данных, имеющееся в массиве, – в коде, это переменная `len`.

Спецификация класса описывает набор данных и функциональность.

```
#define LEN_MAX 10   // Емкость массива.
class My_Array {
  int *a;            // Динамический массив.
  int len;           // Фактическая длина массива.
public:              // Прототипы методов:
  My_Array(void);    // Конструктор.
  My_Array(int L);   // Конструктор.
  ~My_Array(void);   // Деструктор.
  void Input();      // Метод ввода.
  void Output();     // Метод вывода.
  int Sum();         // Метод суммирования.
  void Sort();       // Метод сортировки Bubble sort.
  // Функциональность может быть расширена.
};
```

В спецификации класса его абстрактный смысл очевиден: поля определяют набор данных класса, прототипы методов определяют функциональные возможности класса. Краткие комментарии поясняют суть методов. Функционал легко расширяется добавлением новых методов.

В реализации методов массива использованы обычные алгоритмы.

```
#include "My_Array.h"
My_Array::My_Array(void) {
  len = LEN_MAX;          // Длина = емкость = LEN_MAX.
  a = new int[len];
  for (int i=0; i<len; i++)
     a[i]=0;
}
```

```
My_Array::My_Array(int L) { // Длина = L или LEN_MAX.
len = (L < LEN_MAX) ? L : LEN_MAX;
  a = new int[len];
  for (int i=0; i<len; i++)
     a[i] = 0;            // Все элементы = 0.
}
My_Array::~My_Array(void) {
// Освободить динамическую память.
  delete [] a;
  }
void My_Array::Input() {
  cout << "Input " << len << " elements\n";
  for (int i=0; i<len; i++)
     cin >> a[i];
}
void My_Array::Output() {
  cout << "Is elements\n";
  for (int i=0; i<len; i++)
     cout << a[i] << " ";
  cout << endl;
}
int My_Array::Sum() {
int S = 0;
  for (int i=0; i<len; i++)
     S += a[i];
  return S;
}
void My_Array::Sort() {
int flag;       // Сигнал, что упорядочен.
int tmp;
  do
  {
     flag = 1;
     for (int i=0; i<len-1; i++)
        if (a[i] > a[i+1]) {
           tmp = a[i];
           a[i] = a[i+1];
           a[i+1] = tmp;
           flag = 0;
           break;
        }
  }while (flag!=1);
}
```

Обращение к массиву как к объекту класса внешне выглядит вполне абстрактно, и это главное достоинство класса в сравнении с обычной реализацией массива.

```
#include "My_Array.h"
void main(void) {
// В массиве A по умолчанию LEN_MAX элементов.
My_Array A;
  A.Input();
  A.Output();
  int S = A.Sum();
  cout << "Sum = " << S << endl;
// В массиве с именем B 5 элементов.
My_Array B(5);
  B.Input();
  B.Output();
  B.Sort();
  B.Output();
}
```

Функциональность класса легко расширяется путем добавления методов. Так, можно определить методы добавления элемента в массив, поиска элемента, удаления элемента и другие.

Для добавления элемента в массив напишем метод `Add`. Его реализация зависит от требований, предъявляемых к массиву. Если массив упорядочен, то новый элемент вставляется внутрь так, чтобы порядок не был нарушен, иначе элемент просто добавляется в конец, длина массива увеличивается, как в нижеследующем примере.

```
int My_Array::Add(int T) {
  if (len<LEN_MAX) {
     a[len++] = T;
     return 1;
     }
  else
     return 0;
}
```

Метод имеет логическое значение. Это, с одной стороны, обеспечивает контроль выхода за границу массива, с другой – сигнализирует об успехе операции.

Для добавления числа в массив нужно обратиться к методу `Add`, параметром которого является добавляемое значение:

```
A.Add(100)      // Элемент добавлен в конец массива.
```

Для класса «массив» можно перегружать операции, например, операции присваивания, сравнения, сложения массивов и другие.

Пример. Перегрузка операции присваивания для массивов

Операция присвавания реализуется методом класса. Порождает новый массив, использует метод `Add`.

```
My_Array & My_Array::operator = (My_Array &Right) {
  len = 0;           // Новый массив пуст.
  for (int i = 0; i < Right.len; i++)
```

```
      Add (Right.a[i]);
  return *this;
}
```

Эта реализация позволяет выполнить присваивание для массивов:

```
My_Array Arr(3);     // В массиве три элемента.
  Arr.Input();
  Arr.Output();
My_Array Tmp;        // По умолчанию длина = 0.
  Tmp = Arr;
  Tmp.Output();
  // Получена копия массива Arr в массиве Tmp.
```

Пример. Перегрузка операции сложения для массивов.

Операция сложения может иметь разный абстрактный смысл: можно складывать элементы массивов, имеющие одинаковые индексы, а можно выполнить конкатенацию – присоединение второго массива в конец первого. В примере показана реализация операции первым способом. Если длины исходных массивов не совпадают, порождается массив, длина которого равна наименьшей длине двух исходных. Метод `Add` позволяет получать значения элементов нового массива.

```
My_Array & My_Array::operator + (My_Array &Right) {
int Count;
  Count = len < Right.len ? len : Right.len;
My_Array *New_Arr = new My_Array(0);   // Новый массив.
  for (int i = 0; i < Count; i++)
     New_Arr->Add(A[i] + Right.A[i]);  // Сложение
  return *New_Arr;
}
```

Эта реализация позволяет выполнить сложение для массивов:

```
My_Array One(3);
  One.Input();
My_Array Two(5);
  Two.Input();
My_Array Tmp;
// Сложить массивы.
  Tmp = One + Two;   // Сложить поэлементно.
  Tmp.Output();
```

Пример. Перегрузка операции [] для массивов.

Перегрузка операции индексации [ ] позволяет получить непосредственный доступ к элементам массива – извлечь элемент по его номеру. Ее использование нарушает абстрактный смысл объекта как единого целого, тем не менее, покажем, что это возможно.

Код операции извлечения элемента по индексу должен быть внутренним методом класса.

```
int & My_Array::operator [] (int n) {  // Будет A[i].
  if (n>=0 && n <len)
```

```
      return a[n];
    else
      exit(1);
  }
```

Если операция [ ] перегружена, то возможно непосредственное обращение к элементам массива.

```
My_Array D(5);
for (int i=0; i<5; i++)
  D[i] = i+1;
D.Output();
// Вместо обращения к Output можно использовать цикл:
for (int i=0; i<5; i++)
  cout << " " << D[i];
cout << endl;
```

Еще раз подчеркнем, что перегрузка операции [ ] нарушает абстрактный смысл контейнера.

Пример контейнерного класса

Класс «Массив» может содержать данные произвольных типов, в том числе объектных. Общий принцип построения массива одинаков. Когда содержимым массива являются объекты, единственная сложность в том, что контейнеру недоступны напрямую поля класса, который в нем содержится.

На основе предыдущего примера создадим массив объектов класса `Inner`, для которого в начале раздела 3.4. приведен пример реализации.

Пусть спецификация `Inner` находится в файле `"Classes.h"`, а класс контейнер имеет имя `I_Array`.

```
#define LEN_MAX 10
#include "Classes.h"
class I_Array {
  Inner *a;     // Данные контейнера – объекты.
  int len;      // Фактическая длина.
public:
  I_Array(void);          // Прототипы методов.
  I_Array(int L);
  ~I_Array(void);
  void Input();
  void Output();
  Inner Sum();            // Сумма элементов: тип Inner.
  void Sort();
  Inner & operator [] (int n);     // Извлекает a[i].
  int Add(Inner T);
};
```

В массиве хранятся данные типа `Inner`. Если функциональность класса `I_Array` должна быть такой же, как и у класса `My_Array`, то специфика-

цию и описание класса массива можно получить копированием кода предыдущего примера с последующей заменой имен.

Далее начинается самое интересное. Компилятор C++ покажет, какие изменения нужно внести в код, чтобы получить работающее приложение. Покажем требования в том порядке, в котором их определит компилятор.

Сообщение об ошибке:

```
error C2679: бинарный "=": не найден оператор, принимающий правый операнд типа "int"
```

относится к конструктору массива, в котором все элементы созданного массива обнуляются. В самом деле, для экземпляра класса `Inner` невозможно присваивание нулю, но возможна инициализация методом `Set`, тогда код конструктора будет таким:

```
I_Array::I_Array(void) {
// Определить длину массива, выделить память.
  len = LEN_MAX;
  a = new Inner[len];
  for (int i=0; i<len;i++)
     a[i].Set(0,0);           // Вместо a[i]=0;
}
```

Следующая ошибка в реализации метода суммирования:

```
error C2677: бинарный "+=": не найден глобальный оператор, принимающий тип "Inner"
```

относится к абстрактному смыслу суммирования для объектов `Inner`. Сумма элементов массива, это объект типа `Inner`. Операцию += нужно перегрузить следующим образом.

```
Inner operator += (Inner &T) {
  this->a = a + T.a;
  this->b = b + T.b;
  return *this;
}
```

Возможно, правильнее будет перегрузить отдельно сложение и присваивание.

Следующая ошибка относится к строке `Inner S = 0;` метода суммирования:

```
error C2440: инициализация: невозможно преобразовать "int" в "Inner"
```

Действительно, объекту `S` нельзя присвоить нуль, поэтому его нужно инициализировать при вызове конструктора:

```
Inner S(0,0);
```

Следующая ошибка относится к методу сортировки:

```
error C2676: бинарный ">": "Inner" не определяет этот оператор или преобразование к типу приемлемо к встроенному оператору
```

и означает, что компилятору нужно научиться сравнивать объекты Inner по значению. Алгоритм сравнения должен быть определен в класе Inner перегрузкой операции отношения >.

```
int operator > (Inner &T) { // Больше по модулю.
  return a*a + b*b > T.a*T.a + T.b*T.b;
}
```

Таким образом, компилятор сам показывает, что необходимо изменить в спецификации объекта Inner.

В методах массива необходимо внести изменения в методы ввода и вывода, а именно, следует использовать методы класса Inner. При вводе i-тый элемент массива получает значение методом In класса Inner.

```
cout << "Input " << len << " elements\n";
for(int i=0; i<len; i++)
  a[i].In();
```

При выводе i-тый элемент выводится методом Out класса Inner.

```
cout << "Is elements\n";
for(int i=0; i<len; i++)
  a[i].Out();
```

Напомним, что при создании массива объектов для каждого его элемента вызывается конструктор по умолчанию, следовательно, поля всех элементов обнулены.

Обращение к контейнеру показывает, что смысл объекта контейнера такой же, и обращение такое же.

```
I_Array T(3);
  T.Input();
  T.Output();
Inner S = T.Sum();   // Сумма имеет тип Inner.
  S.Out();
Inner I(11,6);
  T.Add(I);
  T.Sort();          // Сортировка по правилам Inner.
  T.Output();
```

Еще раз приведем спецификацию класса Inner.

```
class Inner {
int a;
float b;
public:
  Inner();
  Inner(int aa, float bb);
  void Set(int aa, float bb);
  void In();
  void Out();
  Inner operator +=(Inner &T);
  int operator > (Inner &T);
  ~Inner();
```

```
};
```

Абстрактный смысл объекта очевиден. Новые методы `operator +=` и `operator >` необходимы, чтобы обеспечить классу контейнеру доступ к полям `Inner`'а.

Реализация класса массива показывает, что операция [] не нужна при работе с контейнером. Исключая ее, мы выводим объект на более высокий уровень абстракции.

На этом же примере можно построить другие реализации абстрактного контейнера, например, «Множество» Отличается от массива как функциональным набором методов, так и алгоритмами их реализации. Например, при добавлении элемента во множество (метод `Add`) следует учесть, что множество не должно содержать повторяющихся элементов. Метод `Sort` не имеет смысла, если множество должно быть упорядоченным. Для упорядоченного множества реализация `Add` должна быть такова, чтобы элементы сохранялись в порядке возрастания значений.

Индексация и итерация. Класс итератор

Для контейнерного класса, где содержанием объекта является последовательность данных, в общем случае, произвольного типа, можно интерпретировать индексацию перегрузкой операции [ ]. Параметром операции-функции `operator[]` является индекс, как правило, целочисленный.

Перегрузка операции [ ] позволяет получить прямой доступ к элементам данных массива, что противоречит правилам создания абстрактных типов – сокрытие информации за барьером абстракции.

Однако задача поэлементного перебора элементов массива есть всегда, даже когда внутри контейнерный класс организован не как массив, а, например, множество или список, со своими правилами работы с данными.

Для перебора элементов коллекции принято писать классы-итераторы. **Итератор**, это объект, который позволяет получить доступ к элементам коллекции, не раскрывая ее внутренней реализации. Итератор предоставляет контейнеру абстрактный интерфейс для доступа к его элементам.

Итераторы бывают пассивные или активные, в зависимости от того, как они соотносятся с классом, для которого используются. Об итераторах подобности можно найти в учебниках многих авторов [3, 4, 5, 6, 8].

Итак, чтобы иметь возможность пробежаться по всем элементам контейнера и выполнить какие-то действия, используется итерация. Итерация является альтернативой индексации, и требует создания класса итератора. Можно использовать два способа создания итератора.

1. Итератор, использующий адресные операции

Напомним синтаксис операции [] для массива. Пусть объявлен массив:

```
float Arr[MAX_SIZE];
```

Прямая адресация выполняется операцией [ ]. Извлекает значение по номеру.

```
// len – фактическая длина массива.
for (i = 0; i<len; i++)
  // Что-то делать с Arr[i];
```

Косвенная адресация в массивах использует семантику указателей. Для того же массива значение извлекается по адресу:

```
float *ip;      // Указатель на float.
for (ip = Arr; ip < Arr+len; ip++)
  // Что-то делать с *ip;
```

Здесь *`ip` – указатель на `float`: переменная, адресующая 4-х байтовый объект. Присваивание `ip = Arr` устанавливает указатель на начало массива. Операция `ip++` сдвигает адрес на `sizeof(float)` байт, получая значение очередного элемента. Условие завершения итерации по массиву: достижение переменной `ip` адреса конечного элемента: `ip<Arr+len`.

Заметим, что операция ++ для указателей выполняется корректно для любого типа данных, в том числе для объектов классов.

Напомним семантику адресных операций. Унарная операция * применяется к указателю и возвращает значение, хранимое по этому адресу. Значение имеет тип, который определен для указателя при его объявлении. Унарная операция & применима к переменной любого типа, и возвращает адрес, по которому переменная хранится.

Управление пробегом по массиву следует вынести из описания класса массива в описание класса итератора, настроенного на массив `Inner`'ов.

```
class Iterator {
const I_Array * p_A;   // Указатель на массив Inner'ов.
int i;                 // Текущий индекс.
public:
Iterator(const I_Array &T) {  // Конструктор.
  p_A = &T;            // p_A принимает адрес массива.
  i = 0;               // Индекс =0.
}
Inner * Begin() {      // Начало.
  return &p_A->a[0];   // Адрес первого элемента.
}
Inner * Current() {    // Адрес текущего значения.
  return &(p_A->a[i]);
}
Inner * Last() {       // Адрес значения за последним.
  return &p_A->a[p_A->len];
}
Inner * Next() {       // Итерация к следующему.
  i++;
  return &(p_A->a[i]);
}
}; // End of Iterator.
```

В конструкторе итератор настраивается на указанный массив. После этого методы `Begin`, `Last`, и `Next` позволяют пробежать по массиву независимо от его типа. Метод `Current` извлекает текущее значение.

Очевидное достоинство итератора – в его абстрактности, обособленности от того объекта, которым он управляет.

Для перехода к следующему элементу вместо метода `Next` можно перегрузить операцию ++, тода получим итератор-указатель [3, 6].

```
Inner * operator ++(int) {
  i++;
  return &(p_A->a[i]);
}
```

Для итерации по элементам массива следует объявить объект-итератор:

```
Iterator It(T); // Итератор It указывает на массив T.
```

Для управления циклом объявить рабочую переменную:

```
Inner *p;
```

И пробежать по массиву от начала до конца:

```
for (p = It.Begin(); p != It.Last(); p = It.Next()) {
  It.Current()->Out();   // Любая обработка.
  // Можно что-то делать с p: это очередной элемент.
  }
```

2. Итератор, использующий перегрузку операции ()

Сначала приведем некоторые сведения об операции ().

Операция обращения к функции:

```
Имя_функции (Фактические_параметры)
```

трактуется как бинарная операция, в которой `Имя_функции` является левым операндом, а `Фактические_параметры` – правым.

Эту операцию можно перегружать. Функция будет иметь имя `operator()`, а список фактических параметров вычисляется и проверяется по типам в соответствии с механизмом передачи параметров. Функция `operator()` должна быть внутренним методом класса.

Перегрузка операции вызова функции имеет смысл для типов, с которыми возможна только одна операция, или для типов, одна из операций над которыми настолько важна, что всеми остальными можно пренебречь.

Чтобы ввести в употребление итератор для массива объектов, определим специальный класс `Iterator`, задачей которого является извлечение элементов из массива в некотором порядке. Чтобы итератор имел доступ к данным, хранимым в массиве, он объявляется дружественным классу `I_Array`, для чего в теле класса добавлено объявление:

```
friend class Iterator;
```

Класс итератор можно определить произвольным образом. Если цель итератора – пробежать по всем элементам, то можно объявить его так:

```
class Iterator {
const I_Array * p_A; // Указатель на массив Inner'ов.
int i;               // Текущий индекс.
public:
Iterator(const I_Array &S) {
  p_A = &S;          // p_A принимает адрес массива,
  i = 0;             // Индекс =0.
}
// Перегружена операция ().
Inner * operator()() {
  if (i <p_A->len)   // Возвращает очередной элемент.
     return &p_A->a[i++];
  else
     return NULL;    // Итерация закончена.
}
};
```

Конструктор итератора инициализирует объект типа `Iterator`, настраивая его на массив: `p_A = &S`.

При обращении к итератору с помощью операторной функции (), возвращается указатель на следующий по порядку элемент этого массива. При достижении конца массива возвращается `NULL`.

Вот пример перебора элементов через итератор.

```
Iterator It(T);      // It настраивается на массив T.
Inner * p;
while (p = It())     // Итерация.
  // Что-то делать с p: это очередной элемент.
  p->Out();
```

Итерация имеет определенные преимущества:

- итератор может иметь собственные данные, в которых хранится информация о ходе итерации;
- можно одновременно запустить несколько итераторов одного типа;
- виды итераторов могут быть различными: прямой использует операции `Begin()`, `Next()` и `Last()`, и разрешает движение от начала к концу. Можно описать обратный итератор для движения в обратном порядке, или двунаправленный.

Следует отметить, что итераторы как объекты встроены практически во все современные языки программирования – C++, C#, Python, Java и другие. Предоставляют абстрактный интерфейс, позвляя программисту решать задачу извлечения элементов из последовательной структуры. Существенно отличаясь в синтаксисе, имеют одинаковую реализацию.

Итоги раздела

Наследование, один из основополагающих принципов ООП, это могучий механизм, который позволяет представить логику прикладной задачи в виде совокупности объектов, связанных друг с другом отношениями.

Наследование позволяет обеспечить общий интерфейс для нескольких различных классов так, чтобы клиентская программа могла работать с объектами этих классов одинаковым образом.

Первый из видов наследования моделирует иерархию, или тип отношения «является» или «is a» между двумя объектами. Иерархии классифицируют объекты по признакам «от общего к частному», или представляют прогрессию с течением времени. Утверждение «язык программирования C++ унаследовал возможности языка C, дополнил и расширил их новыми возможностями» можно рассматривать в двух проекциях. Иерархия подразумевает создание дочерних объектов путём сохранения свойств и поведения базовых объектов, и расширения их в дочерних. Позволяет структурировать и повторно использовать код.

Второй вид наследования моделирует композицию, или тип отношения «имеет» или «part of» между двумя объектами. Представляет отношение целого к части. Так, смартфон имеет встроенную камеру, дети в детском саду объединены в группы. Композиция позволяет создавать сложные объекты, объединяя простые и управляемые части.

Конструкторы классов позволяют правильно инициализировать объекты иерархической модели в соответствии с основным принципом наследования. При порождении объекта класса конструкторы вызываются один за другим иерархически, начиная с базового класса и заканчивая последним производным классом. Деструкторы вызываются в обратном порядке.

Контейнерное наследование дает возможность конструировать объект как совокупность частей, связанных между собой: цветок, это стебель, лепестки, листья. Мощность контейнера как типа в том, что он позволяет объединить данные произвольных типов в группы, имеющие функциональность контейнера, и предоставляющие операции доступа к данным.

Множественное наследование имеет место, когда класс имеет более одного наследника, или класс наследует двум и более объектам. Существуют проблемы наследования типа «из одного многие», вызванные неоднозначностью в механизмах реализации. Именно поэтому наследование должно быть тщательно спроектировано.

3.5. Вопросы для самопроверки

1. Дайте определение наследованию, приведите примеры.
2. Что означает термин «утилитарное использование наследования»?
3. Каково общее назначение механизма наследования? Какова его роль в построении объектной модели задачи?

4. Каким образом наследование позволяет обеспечить общий интерфейс для нескольких различных классов?

5. Каков механизм наследования изнутри: соотношение базового и дочернего объектов?

6. Приведите синтаксис объявления дочернего класса.

7. Какова роль меток прав доступа при объявлении наследования?

8. Назовите схемы наследования по признаку доступа к компонентам базового класса из дочерних.

9. Каков механизм вызова конструкторов при порождении объекта производного класса?

10. Каковы особенности реализации конструкторов дочерних классов?

11. Каковы правила локализации имен в схеме наследования?

12. Какова роль `protected` данных базового класса?

13. Каковы особенности `private` данных базового класса?

14. Какова роль статических данных в базовом классе?

15. Зачем базовому классу несколько перегруженных конструкторов?

16. Назовить виды наследования.

17. Что такое косвенное наследование? Приведите примеры.

18. Что такое множественное наследование? Приведите примеры.

19. Что такое наследование «многие к одному», и что такое наследование «один ко многим»? Приведите примеры.

20. Каковы особенности конструктора дочернего объекта?

21. Что означает термин «виртуальное наследование»?

22. Какие компоненты базового класса не наследуются?

23. Как наследуются статические поля, объявленные в базовом классе?

24. Как вызываются конструкторы базовых классов в иерархии, в которой более двух уровней?

25. В каких случаях используется уточненное имя?

26. В каком порядке вызываются деструкторы при разрушении объектов в случае множественного наследования?

27. Каков абстрактный смысл композиции? Приведите примеры объектов-контейнеров.

28. Приведите примеры абстрактных контейнеров.

29. Назовите основные операции контейнера.

30. Каковы ограничения на доступ из контейнера к полям внутреннего класса, и как обойти их?

31. Как перегрузить операцию присваивания для контейнеров?

32. Каков смысл перегрузки операции индексации [ ] для контейнеров?

32. Что такое итератор? Зачем нужны итераторы?

33. В чем смысл перегрузки операции вызова функции?

34. В чем преимущества итерации перед индексацией?

3.6. Упражнения к третьему разделу

Упражнение 1. Прямое наследование

Задание 1

1. Опишите класс «**Конус**».

Данные: координаты центра основания, радиус основания, высота.

Конструкторы: конструктор по умолчанию, конструктор конуса с центром в начале координат, конструктор произвольного конуса.

2. Определите методы: вычисление площади поверхности, вычисление объема. Для ввода и вывода на экран перегрузите операции >> и <<.

3. Объявите несколько конусов с использованием различных конструкторов, найдите площадь поверхности и объем каждого.

Задание 2

1. Опишите класс «Усеченный конус», производный от конуса.

2. Определите конструкторы по умолчанию и с параметрами.

3. Переопределите методы вычисления площади поверхности и объема для усеченного конуса с использованием методов базового класса.

4. Для ввода и вывода на экран перегрузите операции >> и <<.

5. Объявите несколько усеченных конусов, выведите на экран их площади поверхности и объемы.

6. Определите методы сравнения конусов.

7. Определите несколько усеченных конусов, наследующих одному базовому конусу.

Указания. В этой объектной модели никакой сложности нет, это пример реализации утилитарного наследования.

Упражнение 2. Прямое наследование

Задание 1.

1. Опишите класс «**Шар**».

Данные класса: радиус.

Конструкторы класса: конструктор по умолчанию и конструктор с параметрами.

2. Определите метод вычисления объема шара.

3. Для ввода и вывода на экран перегрузите операции >> и <<.

4. Объявите несколько объектов с использованием различных конструкторов, найдите их объемы.

Задание 2.

1. Опишите класс «Деталь», производный от шара, имеющий собственные данные – материал изготовления и плотность, например, в шарикоподшипниках детали изготовлены из стали.

2. Определите конструкторы детали по умолчанию и с параметрами.

3. Найдите массу детали.

4. Для ввода и вывода на экран перегрузите операции >> и <<.

5. Определите методы сравнения деталей по размеру `operator >`, по материалу изготовления `operator ==`.

6. Определите несколько деталей, наследующих одному базовому классу – шару одного и того же размера.

Указания. В этой объектной модели пример реализации утилитарного наследования. Материал детали описывается как данное типа `std::string`. Задания пункта 5 расширяют логику объекта. В п. 6 использовать конструктор копии базового объекта.

Упражнение 3. Косвенное наследование

1. Создайте иерархию типов: «Океан», «Море», «Залив».

В качестве информации об этих объектах обычно называют название, местоположение, размер, глубину, площадь поверхности и другие значимые признаки. При описании полей класса следует выбрать их разумное количество. Обратить внимание, что уникальное название должно быть присуще каждому объекту иерархии.

2. Каждый из классов должен иметь конструкторы по умолчанию и с параметрами, перегруженные операции ввода из потока и вывода в поток.

3. Создайте несколько объектов каждого типа, покажите, как производный объект использует данные и методы базового класса.

Указания. Здесь роль конструкторов исключительно важна по следующим причинам. Море может принадлежать океану, например, море Лаптевых находится в Северном Ледовитом океане, или быть внутренним, например, Каспийское море. Залив, по определению − часть океана, моря или другого водоема. Может принадлежать морю, например, Ботнический залив находится в северной части Балтийского моря, или океану, например, Авачинский залив находится в Тихом океане у полуострова Камчатка.

Иерархия показана на рисунке 21.

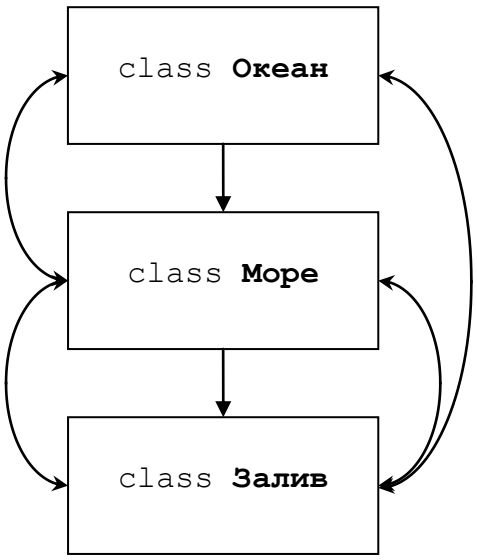


Рис. 21. Иллюстрация к задаче

Сделаем пояснения. Оленёкский залив находится в море Лаптевых, которое принадлежит Северному Ледовитому океану, значит, это чистое косвенное наследование. Ботнический залив принадлежит Балтийскому морю, которое являяется внутренним, значит, базовый класс океан – пустой. Авачинский залив принадлежит Тихому океану, значит, класс моря в этой схеме − пустой. Роль конструкторов – правильное построение схемы наследования для реальных зкземпляров объектов.

Упражнение 4. Множественное наследование

1. Создайте иерархию типов: «Сотрудник фирмы», «Студент», а также «Студент, подрабатывающий в фирме».

2. Каждый из классов должен иметь конструкторы по умолчанию и с параметрами, перегруженные операции ввода из потока и вывода в поток.

3. Создайте несколько объектов каждого типа и покажите, как производный объект использует данные и методы базового класса.

Указания. Объектная модель очевидна. Для объекта «Студент» важными атрибутами, относящими его к классу учащихся, являются название учебного заведения, специальность. Для объекта «Сотрудник фирмы» важными атрибутами являются должность, возможно, заработная плата. Тот и другой объекты имеют одинаковые атрибуты: имя, данные паспорта, ИНН и другие атрибуты. Если студент работает на фирме, то помимо собственных атрибутов он получает должность, например, курьера, и заработную плату.

Следует хорошо продумать механизмы конструкторов, чтобы понять, что происходит при порождении объекта «Студент, подрабатывающий в фирме»: какие атрибуты объекта находятся в дочернем классе, а какие передаются в базовые классы.

Упражнение 5. Контейнерное наследование

В реализации этой задачи используйте абстрактное понятие множество, которое наилучшим образом отражает смысл задачи.

1. Опишите класс «Слово».

Данные класса: строка типа `std::string`.

Конструкторы класса: конструктор по умолчанию генерирует пустое слово, конструктор произвольного слова.

Методы класса:

проверка существования слова (не пусто), ввод слова, вывод на экран.

2. Опишите контейнерный класс «Словарь» как множество слов.

Методы класса: добавление слова, удаление слова, поиск слова по значению, редактирование.

Необходим метод просмотра всех элементов словаря.

Указания. Поскольку словарь создается не как массив, а на основе множества, то алгоритмы реализации методов имеют особенности.

1. Слово не добавляется, если такое уже есть.

2. Имеет смысл создание упорядоченного словаря. Добавление слова происходит в лексикографическом порядке (тип `string` позволяет выполнить такое сравнение). Алгоритм поиска реализуется как бинарный поиск.

Упражнение 6. Контейнерное наследование

В реализации этой задачи используйте абстрактное понятие массив, для которого в разделе 3.4 приведен пример.

1. Опишите класс «Мой_знакомый».

Данные класса: имя (ник), номер телефона, важность знакомства.

Методы класса должны обеспечить минимальную функциональность объекта.

2. Опишите класс «Записная книжка». Это массив, содержащий определенное количество знакомых.

Методы класса: добавление, удаление, редактирование знакомого, поиск, просмотр списка.

Указания. Важным методом, используемым внутри других методов, является поиск. Варианты поиска различны. В этой задаче логично реализовать два варианта поиска: по имени и степени важности. Поскольку имя уникально в пределах списка знакомых, метод будет возвращать одно значение или нулевой указатель. Поиск по степени важности – чисто интерфейсный, может просто вывести на экран все найденные элементы списка.

Упражнение 7. Контейнерное наследование

Разработайте и иллюстрируйте применение класса «Нечеткое множество». Нечеткое множество, это множество, содержащее пары $\{A|\mu_A\}$, где A обозначает элемент множества, а μ_A − значение функции принадлежности этого элемента нечеткому множеству [0,1].

Реализуйте операции над множествами: конструктор, просмотр значений элементов множества, добавление элемента во множество, поиск элемента, построение функции принадлежности для указанного множества.

Указания. Для представления элементов множества напишите класс «Пара»: `{string, float}`. Пара должна сама контролировать значение $0 <= \mu_A <= 1$. Выберите предметную область, например, постройте нечеткое множество для утверждения «Красивый» для множества сказочных персонажей, например, {(Василиса Премудрая|0,9), (Иван царевич|0,7), (Баба Яга|0,1)}. Множество не может содержать одинаковые элементы.

4. ПОЛИМОРФИЗМ

Полиморфизм (дословно, многоформенность) – способность объектов реагировать на запрос сообразно своему типу.

Запросы являются средством и инструментом взаимодействия объектов в среде обитания. Запрос можно считать вызовом метода объекта.

Внешне многие запросы выглядят одинаковым образом, но могут быть посланы объектам разного типа. Например, шахматная фигура делает ход. В объектной модели метод «сделать ход» для каждой фигуры, то есть для каждого типа, выполняется различным образом. Понятно, что тип объекта определяет вид реакции объекта на запрос, и для каждого типа объекта реакция своя. Свойство полиморфизма позволяет использовать одно и то же имя для решения одинаковых внешне, но технически разных задач.

В языке C++ реализация идеи полиморфизма поддержана с помощью различных инструментов. Нами уже приведены примеры реализации этой идеи без введения названия «полиморфизм». Так, полиморфичными являются:

- перегруженные функции;
- перегруженные операции.

В любом из этих случаев для каждого типа объекта существует своя собственная реализация алгоритма. Нужную реализацию выбирает компилятор, когда обрабатывает вызов: зависит от типа или числа аргументов в первом случае, и от типа операндов во втором.

Вершиной реализации идеи полиморфизма являются виртуальные функции. Главная идея их использования заключается в том, что тип объектов, создаваемых динамически, может быть неизвестен при компиляции, а определен в момент создания объекта. Тогда для объекта распределяется память и генерируется код нужных объекту методов.

Рассматривается также параметрический полиморфизм, реализованный с помощью шаблонов (`template`).

4.1. Виртуальные функции

Очень часто в объектной модели встречается ситуация, когда у базового класса существует несколько производных классов, и в каждом из них есть некоторая функция, которая должна различным образом выполняться для каждого производного класса.

Например, шахматная фигура есть абстрактный объект. Фигуры отличаются друг от друга внешним видом и системой ходов. Фигура имеет метод «сделать ход». Суть метода различна для конкретной фигуры – пешки, ферзя, коня и других. На запрос «сделать ход» каждая фигура выполнит свое собственное действие.

Например, геометрическая фигура есть абстрактный объект. Каждая фигура имеет метод «вычислить площадь». Суть метода различна для конкретной фигуры – квадрата, трапеции, круга и других. Нельзя вычислить площадь просто фигуры, ее тип должен быть конкретизирован.

В базовом классе «фигура» может быть объявлена виртуальная функция, вместо которой будет вызвана реальная функция одного из производных классов. Реализация виртуальной функции различна для всех производных классов, так как определяется типом, но в базовом классе объявляется общность метода.

С помощью виртуальных функций реализована идея полиморфизма. Классы с виртуальными функциями являются полиморфными.

Объявление виртуальной функции

Виртуальная функция объявляется в базовом классе с ключевым словом `virtual` перед именем. Имя функции одинаково во всех производных классах, где она должна использоваться, а также одинаковыми должны быть тип функции и сигнатура параметров.

Служебное слово `virtual` (виртуальная) показывает, что функция в разных производных классах имеет собственные реализации, а выбор нужной версии при ее вызове, это задача компилятора. Определение виртуальной функции дается для того класса, в котором она была впервые описана, если только она не является чистой виртуальной функцией.

Пример. Базовый класс `Base` имеет два производных класса.

```
// Базовый класс.
class Base {
...
virtual тип function (параметры);
...
}; // End of Base
// Дочерние классы.
class B: public Base {
...
тип function (параметры);
...
}; // End of B
class C: public Base {
...
тип function (параметры);
...
}; // End of C
```

Имя **`function`** в базовом классе объявлено с ключевым словом `virtual`, а в каждом дочернем классе описана конкретная реализация метода **`function`**. Тип функции и сигнатура параметров одинаковы.

Механизм виртуальных функций реализует полиморфизм времени выполнения: какой именно виртуальный метод вызовется, известно только во время выполнения программы.

Механизм работает только для динамических объектов. Дочерний объект создается как указатель на объект базового класса.

Прежде чем рассматривать реализацию механизма виртуальных функций, несколько слов о преобразовании указателей.

Преобразование указателей

Пусть есть открытый базовый класс `Base`:

```
class Base {
...
};
```

Пусть у него есть производный класс `Derive`:

```
class Derive: public Base {
...
};
```

Объект базового класса является частью объекта производного класса, в соответствии с механизмами наследования (см. рисунок 8).

Согласно правилам преобразования типов, объект «меньшего» типа может быть преобразован в объект «большего типа» без потерь, а обратно только с потерей данных. Это правило относится и к указателям, а операция присваивания для указателей требует явного преобразования типов.

Пример.

```
int a = 1;
int *p_a = &a;            // Адресует 4 байта.
long b = 99.0;
long *p_b = &b;           // Адресует 8 байт.
// Требуется явное преобразование типа.
p_a = (int *) p_b;        // Адресует 4 байта из восьми.
//p_b = (long *) p_a;     // Адресует что?
```

Механизмы, связанные с таким преобразованием, очень тонкие. Подробное их обсуждение на примерах можно найти в соответствующих разделах учебников [2, 3, 5].

Для классов указатель на производный класс `Derive` можно присваивать переменной типа указатель на `Base`, не используя явное преобразование типа.

```
Derive D;                 // Больший тип.
Base *p_b = &D;           // Неявное преобразование.
// p_b = new Derive;      // Или чаще всего так.
```

Обратное преобразование указателя на `Base` в указатель на `Derive` должно быть явным:

```
Derive *p_d = p_b;              // Ошибка: Base* меньше.
Derive *p_d = (Derive*) p_b;    // Явное преобразование.
```

Вывод. Объект производного класса при работе с ним через указатель можно рассматривать как объект его базового класса, обратное неверно.

Пример.

```
Base   *B;
Derive p_d;
// Производный объект как объект базового класса.
p_d = &B;
// Или чаще всего
Base   *B;
B = new Derive;
```

Выводы

Пусть задан базовый класс `Base` и производный от него класс `Derive`.

```
Base  *p_b;   // Указатель на объект базового класса.
Derive *p;    // Указатель на объект производного класса.
Base  B;      // Объект базового класса.
Derive  D;    // Объект производного класса.
```

Переменная, объявленная как указатель на объект базового класса, может применяться как указатель на объект производного класса.

```
p = &B;     // Указатель на объект базового класса.
p = &D;     // Указатель на объект производного класса.
```

Путем использования указателя `p` можно получить доступ ко всем элементам объекта `D`, унаследованным из `B`, но нельзя получить доступ к собственным компонентам `Derive`.

Если задан указатель на объект производного класса, то через него нельзя получить доступ к компонентам базового класса.

Механизм обращения к виртуальным функциям

Пусть в базовом и производном классах объявлена функция с одним и тем же именем, типом и параметрами.

Сначала рассмотрим пример, в котором функции не являются виртуальными. Указатель на объект базового класса (`A *a`) может указывать на объект производного класса, при этом выполняется преобразование указателей в соответствии с правилами преобразования типов (не наоборот).

В этом случае для объекта производного класса происходит обращение к функции базового класса. Почему? Потому что выбор функции зависит от типа указателя, а им является тип A.

```
class Base {
public:
void f(void) {
  cout << "Невиртуальная функция f базового класса\n";
}
};// End of Base
class B: public Base {
int d;
public:
```

```
void f(void) {
  cout << "Функция f класса B\n";
}
}; // End of B
class C: public Base {
int e;
public:
void f(void) {
  cout << "Функция f класса C\n";
}
}; // End of C
// Преобразование указателей.
void main(void) {
Base *a;
a = new Base;        //Указатель на объект класса Base.
// Выполняется метод f класса Base.
  a->f();
  a = new B;         //Показывает на производный класс B.
// Хочется, чтобы был выполнен метод f класса B, но
// этого не произойдет.
  a->f();
  a = new C;         //Показывает на производный класс C.
// Хочется, чтобы был выполнен метод f класса C.
  a->f();
// Чтобы получить доступ к методам классов B и C,
// нужно явно указывать область действия имен, то есть
// создавать динамически объекты типов B и C.
B *b;
  b->f();
C *c;
  c->f();
}
```

Вывод. Если в базовом классе функция не виртуальная, то происходит обычная перегрузка: каждый класс имеет собственный экземпляр функции. Это называется механизмом статического или раннего связывания, и выполняется на стадии компиляции, когда генерируются коды функций всех используемых объектов.

Преобразование указателей не может обеспечить вызов метода, соответствующего реальному типу указываемого объекта. Проблема распознавания класса, к которому принадлежит объект, не может быть разрешена.

Теперь рассмотрим предыдущий пример с использованием виртуальных функций. Для этого ключевое слово `virtual` достаточно поместить в базовом классе перед определением функции.

```
class Base {
public:
virtual void f(void) {
```

```
  cout << "Виртуальная функция f базового класса\n";
}
}; // End of Base
class B: public Base {
int    d;
public:
void f(void) {
  cout << "Функция f класса B\n";
}
}; // End of B
class C : public Base {
int    e;
public:
void f(void) {
  cout << "Функция f класса C\n";
}
}; // End of C
// Пример виртуальности.
void main (void) {
Base *a;
Base = new A;
//Выполняется метод класса A.
  a ->f();
a = new B;
//Выполняется метод класса B.
  a ->f();
a = new C;
//Выполняется метод класса C.
  a ->f();
} // main
```

Доступ к методам производных классов выполняется через указатель на базовый класс, только в этом случае проявляется свойство виртуальности. Если он имеет значение объекта базового класса, то вызывается метод базового класса, если значение объекта производного класса, то вызывается метод производного класса. Это называется динамическим или поздним связыванием.

Поясним механизм. Внутри класса, имеющего виртуальную функцию, компилятор добавляет скрытый элемент – указатель на таблицу виртуальных функций. Во время выполнения программы происходит замена функции базового класса функцией производного класса по следующей схеме:

- создается объект;
- генерируется код функции, необходимый для объекта данного типа;
- вызывается нужная функция.

Объявление `virtual` достаточно в базовом классе. В производных классах функции с таким же именем, типом и параметрами автоматически делаются виртуальными.

Механизм виртуальных функций обеспечивает динамическое связывание только при использовании указателей.

Некоторые особенности виртуальных функций

Прототипы виртуальной функции в базовом классе и во всех производных классах должны быть одинаковы. Иначе функция будет рассматриваться как перегруженная.

Виртуальной может быть только функция, не являющаяся статической.

Конструктор не может быть виртуальным.

Деструктор может быть виртуальной функцией.

Более того, деструктор полиморфного базового класса должен объявляться виртуальным. Этим обеспечивается корректное разрушение объекта производного класса через указатель на базовый класс.

Рассмотрим модельный пример. Все классы имеют минимальный набор методов, только для того, чтобы иллюстрировать семантику объектов.

```
// Класс, который будет внутренним для Derive
class Inner {
public:
Inner() {
  cout << "Inner: конструктор" << endl;
}
~Inner() {
  cout << "Inner: деструктор" << endl;
}
}; //End of Inner.
// Базовый класс имеет виртуальный деструктор.
class Base {
public:
Base() {
  cout << "Base: конструктор" << endl;
}
~Base() {
  cout << "Base: деструктор" << endl;
}
virtual void print() = 0;
}; // End of Base.
// Производный класс. Внутри порождается объект One.
class Derive: public Base {
Inner One;      // Внутренний объект.
public:
Derive() {
  cout << "Derive: конструктор" << endl;
}
```

```
~Derive() {
  cout << "Derive: деструктор" << endl;
}
void print() {
}
}; // End of Derive.
```

В функции `main` указателю на базовый класс присваивается адрес динамически создаваемого объекта производного класса `Derive`. Затем этот объект разрушается.

```
Base * T;
  T = new Derive;
  delete T;
```

Наличие виртуального деструктора базового класса обеспечивает вызовы деструкторов всех классов в ожидаемом порядке, а именно, в порядке, обратном вызовам конструкторов соответствующих классов.

Вывод программы с использованием виртуального деструктора в базовом классе будет следующим:

```
Base: конструктор
Inner: конструктор
Derive: конструктор       // Три объекта порождены.
Derive: деструктор
Inner: деструктор
Base: деструктор          // Три объекта разрушены.
```

Если деструктор в базовом классе невиртуальный, то уничтожение объекта производного класса даст неопределенный результат: практически будет разрушена только часть объекта, соответствующая базовому классу.

Если в этом коде убрать ключевое слово `virtual` перед деструктором базового класса, то вывод программы будет таким:

```
Base: конструктор
Inner: конструктор        // Три объекта порождены.
Derive: конструктор
Base: деструктор          // Один объект разрушен.
```

Данное внутри класса `Derive` также не разрушается

Существует правило: если базовый класс полиморфичен, то деструктор его должен быть виртуальным.

4.2. Абстрактные классы

Абстрактные функции

Абстрактной функцией в базовом классе называется пустая (чистая) виртуальная функция.

Она имеет определение:

```
virtual тип_функции имя_функции (параметры) = NULL;
virtual тип_функции имя_функции (параметры) = 0;
```

Механизмы чистых виртуальных функций:

- такая функция недоступна для вызовов;
- не имеет реализации;
- служит только основой для подменяющих ее функций производных классов.

Абстрактные классы

Абстрактным классом является такой базовый класс, в котором есть хотя бы одна чистая виртуальная функция.

Класс не только называется, а фактически является абстрактным, потому что объект такого класса не может быть создан и не создается никогда. На его основе создаются только объекты производных классов. Назначение абстрактных классов – описание общих свойств других, реально существующих объектов. Примеры таких абстрактных объектов: «шахматная фигура», «геометрическая фигура», «единица хранения» и прочие.

Абстрактный класс может иметь данные, может и должен иметь методы. Факт виртуальности метода говорит о том, что метод реализуется не в этом классе, а только в его производных.

Особенности использования абстрактных классов перечислены ниже.

1. Главное назначение абстрактного класса – это определение интерфейса для всей иерархии наследуемых классов.

2. Объект абстрактного класса не существует, а служит основой для создания объектов производных классов.

3. Абстрактный класс может иметь свои атрибуты и методы, которые будут унаследованы производыми классами.

4. Чистый виртуальный метод должен быть переопределен во всех классах, производных от него.

5. Абстрактный класс может иметь конструктор.

6. Абстрактный класс может иметь деструктор, который следует сделать виртуальным и определить реализацию.

Приведем пример реализации абстрактной функции и абстрактного класса.

```
class Base {
public:
virtual void f(void) = NULL;       // Без реализации.
}; // End of Base
class B: public Base {
int    d;
public:
void f(void) {
  cout << "Функция f класса B\n";
}
}; // End of B
class C: public Base {
int    e;
```

```
public:
void f(void) {
  cout << "Функция f класса C\n ";
}
}; // End of C
// Пример порождения дочерних объектов через указатель.
void main(void) {
Base *a;
// Невозможно создать объект абстрактного класса.
// a = new Base;
// a->f();
a = new B;
  a->f();            // Вызывается функция f() класса B.
a = new C;
  a->f();            // Вызывается функция f() класса C.
} // main
```

Пример коллекции графических объектов

Существует абстрактный объект «геометрическая фигура», частными случаями которого являются точка, окружность, прямоугольник и другие. Будем рисовать их на плоскости экрана. Сначала выделим общность фактических объектов: каждый из них, независимо от типа, привязан к точке координат экрана и может быть создан, изображен, разрушен.

Абстрактный класс должен определить глобальные атрибуты объекта, то есть его координаты. Он может имет конструктор для инициализации глобальных атрибутов и деструктор, роль которого – разрушение объекта.

Определим абстрактный класс `Shape` (фигура) и производные от него `Point` (точка), `Circle` (окружность) и `Rect` (прямоугольник). Атрибуты точки совпадают с атрибутами фигуры, у окружности добавляется атрибут радиус, у прямоугольника длина и ширина.

```
class Shape {              // Абстрактный класс «фигура».
protected:
int    x,y;                // Координаты любой фигуры.
public:
Shape ();                  // Конструктор по умолчанию.
// Виртуальное рисование фигуры.
virtual void Draw() = 0;
// Виртуальный деструктор с реализацией.
virtual ~Shape() = 0;
}; // End of Shape
class Point : public Shape {
// Собственные данные точке не нужны.
public:
Point();
void Draw();    // Настоящее рисование точки.
};// End of Point
```

```
class Circle : public Shape {
// Собственное данное радиус.
int    r;
public:
Circle();
void Draw();    // Настоящее рисование окружности.
}; // End of Circle
class Rect : public Shape {
// Собственные данные ширина и высота.
int    h,w;
public:
Rect();
void Draw();    // Настоящее рисование прямоугольника.
}; // End of Rect
// Реализация методов:
Shape::Shape(){
// Конструктор объекта: задает x, y.
  x = random (getmaxx()-50);
  y = random (getmaxy()-50);
}
Shape::~Shape() {
  delete this;
}
Point::Point(): Shape() {
// Конструктор точки пустой.
}
void Point::Draw() {          // Метод точки.
  setcolor (BLUE);
  circle (x, y, 2);
}
Circle::Circle(): Shape() {
//Конструктор окружности добавляет r.
  r = 10 + random(40);
}
void Circle:: Draw() {   // Метод окружности.
  setcolor(RED);
  circle(x, y, r);
}
Rect::Rect(): Shape() {
//Конструктор прямоугольника добавляет h,w.
  h = 10 + random(90);
  w = 10 + random(90);
}
void Rect:: Draw() {     // Метод прямоугольника.
  setcolor(GREEN);
  rectangle(x, y, x+w, y+h);
}
```

```
// Создаются объекты произвольного размера
// в произвольных местах экрана.
void main(void) {
//
int    i=0, key;
// Объекты абстрактного класса не создаются.
// Shape Fg;    // Нельзя создать просто фигуру.
// Fg.Draw();   // Нельзя нарисовать просто фигуру.

do {
  i++;
  switch (i%3) {
  case   0: {
  // Создается виртуальная точка.
  Shape *p = new Point;
       p -> Draw();
       delay (500);
       delete p;              // Объект разрушается.
       break;
  }
  case   1: {
  // Создается виртуальная окружность.
       Shape *c = new Circle;
       c -> Draw();
       delete c;
       delay (500);
       break;
  }
  case  2: {
  // Создается виртуальный прямоугольник.
       Shape *d = new Rect;
       d -> Draw();
       delete d;
       delay (500);
       break;
  }
  };   // Конец switch.
  } while ((key=bioskey(1))==0);
} // End of main
```

Реализация виртуального деструктора

Чистый виртуальный деструктор может выполнять задачи разрушения объектов независимо от их типа. Помимо высвобождения памяти, занятой объектом, он может выполнять все задачи, связанные с прекращеним жизни объекта. Например, в данном примере, можно было бы «стереть» изображение с экрана или вызвать эффект вспышки или еще что-нибудь.

Пример синтаксиса объявления такого деструктора:

```
clacc A {
...
public:
virtual ~A() = 0;
};
...
A::~A {
// Реализация деструктора;
  delete this;
}
```

Хранение объектов разного типа в одной структуре

Механизм абстрактных классов, помимо прочих преимуществ, дает возможность сохранять данные различных типов в одной структуре, статической или динамической. Покажем, как можно хранить разные графические объекты в одной структуре.

Пусть объявлен массив указателей на объекты базового класса.

```
Shape *array [3];
// Значения элементов массива определены так:
array [0] = new Point;
array [1] = new Circle;
array [0] = new Rect;
```

При этом каждый элемент массива `array` способен ссылаться на данное любого из типов `Point`, `Circle`, `Rect`. Механизм размещения в памяти графических объектов приведен на рисунке 22.

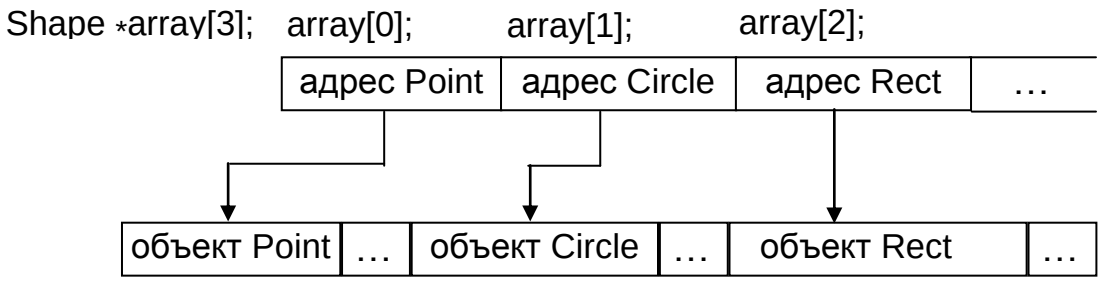


Рис. 22. Объединение данных разных типов в массиве

Такой способ позволяет намного упростить работу с данными различных типов, обращаясь к ним одинаковым образом для реализации одинаковых задач. Выполняя запрос, мы не задумываемся над тем, к какому объекту он адресуется, так как уверены в том, что каждый объект сам знает, что он должен делать.

Итоги раздела

Полиморфизм, один из трех основных принципов ООП, это способность обьекта иметь различное поведение, зависящее от типа объекта.

Кратко смысл полиморфизма выражен фразой: «Один интерфейс, множество реализаций». В C++ полиморфными являются перегруженные функции и перегруженные операции.

Вершиной идеи реализации полиморфизма являются объекты, порождаемые через механизм виртуальных функций и абстрактных классов. В этом случае классы с одинаковой спецификацией имеют различную реализацию, используя методы производного класса, который не существует на момент создания базового. Абстрактный объект создается «на лету», при этом имеет присущую только ему специфику.

Полиморфизм позволяет еще больше абстрагировать объектную модель задачи. Общие свойства групп объектов объединяются единым интерфейсом, следовательно, внешняя общность проявляется как одинаковый набор методов с одинаковыми именами и сигнатурами.

Полиморфизм абстрактных классов является динамическим, и проявляется во время выполнения программы. В C++ способом смены интерпретации объекта является преобразование указателя [2, 4]. Преобразование указателя базового класса на объект производного класса приводит к динамическому порождению объектов производных классов, каждого со своей реаллизацией, и, следовательно, со своим интерфейсом.

Механизм реализации виртуальных функций отражает единство данных и алгоритма. Дочерний объект создается как базовый, но со своим интерфейсом, присущим именно этому типу объекта.

Абстрактные классы позволяют работать с группами однородных объектов универсальным способом.

4.3. Вопросы для самопроверки

1. Дайте определение полиморфизма. Приведите примеры объектов, которые имеют одинаковый интерфейс.
2. Что такое запрос? Как объект реагирует на запрос?
3. Как зависит реакция объекта на запрос от типа объекта? Приведите примеры.
4. Назовите инструменты C++, поддерживающие реализацию идеи полиморфизма.
5. Что такое перегруженная функция?
6. Что такое перегруженная операция?
7. Что такое виртуальная функция?
8. Как объявить виртуальную функцию в базовом и в дочерних классах?
9. Каковы правила преобразования типов для указателей?
10. Что такое динамическое связывание?
11. Что изменится при порождении дочерних объектов, если прототипы виртуальной функции в базовом классе и в производных классах будут различаться?

12. Почему `static` функция не может быть виртуальной?

13. Почему деструктор в базовом классе может быть виртуальным?

14. Почему конструктор в базовом классе не может быть виртуальным?

15. Дайте определение абстрактной функции.

16. Каков синтаксис определения абстрактной функции?

17. Дайте определение абстрактного класса.

18. Какое значение имеют данные и методы, объявленные в базовом классе?

19. Какова роль невиртуальных методов, объявленных в абстрактном классе?

20. Каково главное назначение абстрактных классов?

21. Как абстрактные классы обеспечивают общий интерфейс группы объектов?

4.4. Упражнения к четвертому разделу

Упражнение 1. Полиморфизм

1. Создайте абстрактный базовый класс «Функция», для которого можно виртуально найти значение в произвольной точке и виртуально построить таблицу значений, зная диапазон аргумента.

2. Создайте производные классы «Прямая», «Парабола», «Гипербола». Определите методы вычисления значения и построения таблицы для каждого производного объекта.

3. Выполните проверку для каждого вида производного объекта.

Указания. Уравнение прямой: $y=a \cdot x + b$, уравнение параболы: $y=a \cdot x^2 + b \cdot x + c$, уравнение гиперболы: $y=a / x + b$.

В базовый класс выносятся все данные, общие для всех дочерних объектов, а также прототипы общих методов.

Общими данными являются коэффициенты уравнений *a*, *b*, класс параболы расширяется коэффициентом *c*.

В базовом абстрактном классе «Функция» определите виртуальные методы вычисления значения в указанной призвольной точке:

```
virtual float F(float x) = 0;
```

и метод вывода таблицы значений:

```
virtual void Table(float x1, float xn, int Count)=NULL;
```

Во всех дочерних классах напишите конструкторы пустой и с параметрами, перегрузите операции ввода и вывода в поток.

Переопределите виртуальные методы в каждом дочернем классе для конкретного типа функции.

Выполняя тестирование, порождайте дочерние объекты на основе указателя на базовый класс так, как это делается в примере этого раздела.

Упражнение 2. Полиморфизм

1. Создайте абстрактный базовый класс «Работник», один из методов которого, это получение заработной платы согласно условиям трудового

договора. Поскольку условия договора могут быть разными, метод начисления зарплаты виртуальный. Данными объекта являются имя, должность.

2. Создайте производные классы: «Служащий на окладе», «Служащий с почасовой оплатой» и «Служащий с процентной ставкой». Переопределите метод начисления зарплаты согласно условиям трудового договора.

3. Для проверки определите массив ссылок на абстрактный класс, которым присваиваются адреса объектов разного типа.

4. Напишите метод, который сформирует ведомость зарплаты сотрудников с выводом на печать.

Указания. Базовый класс «Работник» не пустой, он хранит данные о работнике – имя, должность. Имеет виртуальный метод вычисления зарплаты:

```
virtual float Sum() = 0;
```

Нужно написать конструктор и метод вывода данных о сотруднике. Вопрос в том, может ли метод вывода быть виртуальным методом базового класса? Переопределите конструкторы дочерних объектов.

В дочерних классах определены данные, и известен конкретный алгоритм начисления зарплаты: для работника на окладе это оклад, для работника с почасовой оплатой, это число часов и стоимость часа, для работника с процентной ставкой, это оклад и процентная ставка. Метод вычисляет и возвращает сумму заработной платы для каждого дочернего объекта в соответствии с его типом.

Метод вывода ведомости зарплаты не может принадлежать базовому классу, и тем более, классам дочерним. Как реализовать этот алгоритм?

5. ПАРАМЕТРИЧЕСКИЙ ПОЛИМОРФИЗМ

Обобщенное программирование – парадигма программирования, заключающаяся в таком описании алгоритмов и данных, которое можно применять к различным типам данных без изменения самого описания. Поддерживается разными языками программирования в том или ином виде: C++, Java, Object Pascal, Eiffel, языках .NET и других.

Параметрический полиморфизм в C++ реализуется на основе шаблонов классов и функций. Разделяя методологии программирования, можно утверждать, что виртуальные функции и абстрактные классы, это объектно-ориентированное программирование, а шаблоны – это обобщенное программирование. Тем не менее, будем изучать этот инструмент, относя его к ООП.

Шаблоны (`template`) дают возможность создавать функции и классы «в общем виде», тем самым, поддерживают идею полиморфизма. Этот инструмент служит основой для создания компилятором функций и классов, конкретизирующих некоторое общее описание, заданное программистом.

Шаблоны функций расширяют и совершенствуют механизм перегруженных функций. Например, для классов контейнеров такие алгоритмы как сортировка, поиск, суммирование выполняются одинаковым образом независимо от типа данных. Поэтому, описывая абстрактные алгоритмы, можно перегрузить функции (методы) для каждого типа данных, но можно описать одну функцию, которая получает тип данных, с которыми ей придется работать.

Шаблоны классов используются для создания абстрактных типов данных. Бывают полезными при определении контейнерных классов, где обработка данных выполняется одинаковым образом, независимо от типа. Для того, кто пишет класс-контейнер, тип содержащихся в нем объектов не имеет значения, но для пользователя конкретного контейнера этот тип является существенным.

Примером контейнера, который может содержать данные любого типа, является абстрактный класс «Множество».

Множество относится к неопределимым объектам математики, и определяется как совокупность объектов, связанных по какому-либо признаку. Множество отличается от массива.

Существуют правила построения множеств:

- объект либо входит в множество (принадлежит), либо нет;
- объект не может входить во множество дважды;
- элементы множества однотипны;
- множество может быть упорядоченным или нет;

В классической теории множеств для них определены операции:

- добавление одного элемента во множество;

- удаление одного элемента;
- определение принадлежности элемента ко множеству;
- определение включения множества во множество;
- объединение множеств;
- пересечение множеств;
- разность множеств;
- проверка на равенство / неравенство множеств;
- проверка на пустоту.

Как видно из определения, элементы множества должы быть одного типа, но этот тип может быть абсолютно любым: точки на плоскости, геометрические фигуры, все, что угодно. Приблизительная спецификация класса «Множество» приведена в разделе 5.4.

5.1. Шаблоны функций

Шаблоны функций позволяют описывать алгоритмы, которые могут одинаковым образом обрабатывать данные разных типов. Примеров алгоритмов, которые выполняются одинаковым образом для данных различных типов, множество, например, поиск, сортировка, сжатие и им подобные.

Функция обработки данных может быть определена однократно и это объявление параметризовано. Можно отнести к параметрам:

- тип возвращаемого значения;
- типы параметров функции (при фиксированной сигнатуре).

Определение шаблона функции

Шаблон функции является почти обычным определением функции. Статус шаблона функции придают ключевое слово `template` и список параметров шаблона, где параметром является имя типа данных. Заголовок шаблона функции имеет синтаксис:

```
template <список_параметров_шаблона> тип_функции
имя_функции (формальные_параметры)
{
// Тело шаблона, обычное описание функции;
}
```

Список формальных параметров шаблона заключается в угловые скобки < >, каждый формальный параметр шаблона обозначается ключевым словом `class`, за которым следует имя параметра шаблона. Слово `class` здесь означает «любой тип». В теле шаблона определяется алгоритм функции, в которой тип возвращаемого значения и типы параметров обозначаются именами параметров шаблона, введенными в заголовке. Они же в теле функции могут обозначать типы локальных объектов.

Так порождается не одна, а целое семейство функций, отличающихся друг от друга типом данных, с которыми функция работает.

Пример, кочующий из одного руководства в другое, это шаблон функций Swap для обмена двух значений произвольного (любого) типа.

```
template <class Type> void Swap (Type &x, Type &y)
{
Type   z;
  z = x;
  x = y;
  y = z;
}
```

Здесь **Type** – параметр шаблона функции: обобщенное имя типа. Используется:

- в заголовке для спецификации формальных параметров;
- в теле шаблона для объявления локальной переменной.

Механизм шаблона функции

Шаблон функции определяет общее, абстрактное описание функции, которое не может быть реально выполнено, потому что тип объектов, участвующих в алгоритме, не определен в шаблоне. Реальный код функции формируется в процессе компиляции, когда компилятор обрабатывает вызовы функций, которые он обнаружил в коде программы. В зависимости от реальных, указанных при обращении, типов параметров и возвращаемого значения, генерируется столько обработчиков, сколько необходимо.

Например, если в программе объявлены объекты:

```
float a,b;
  a = 2.3;
  b = 7.9;
```

То для обращения вида Swap(a,b); будет генерирован код:

```
void Swap (float &x, float &y) {
float  z;
  z = x;
  x = y;
  y = z;
}
```

Далее будет выполнено обращение именно к этой функции.

Для переменных другого типа, в том числе отличном от базового, генерируется другой код функции, с другим именем типа. Пусть есть некоторый тип My_class, объявленный программистом, и есть объекты:

```
My_class A,B;
```

Для обращения Swap(A,B); будет генерирован код:

```
void Swap (My_class & x, My_class & y) {
My_class z = x;
  x = y;
  y = z;
}
```

И будет выполнено обращение именно к этой функции.

Приведем полный пример работы с шаблоном этой функции. В данном примере параметром шаблона может быть любой тип данного, в том числе произвольного класса. Для правильного выполнения функции класс должен иметь конструктор копирования. Шаблонной также является функция `OutXY`, которая выводит два значения любого типа.

```
#include <iostream.h>
// Шаблон функции: здесь T , это обобщенное имя типа.
template <class T> void Swap (T &x,T &y) {
  T    z = x;
  x = y;
  y = z;
}
template <class T> void OutXY (T &x, T &y) {
  //Вывод двух значений любого типа.
  cout << x <<" "<< y<< "\n" ;
}
// Шаблоны функций для классов.
class My_class {     // Объявим класс.
int    a;
float  b;
public:
My_class (int a, float b) {
  this->a = a;
  this->b = b;
}
//Перегрузка вывода для классов.
void Out() {
  cout << a << " " << b << "\n";
}
My_class operator = (My_class &T) {
  a = T.a;
  b = T.b;
  return *this;
}
}; // End of My_class
// Обращение к Swap одинаково для данных любого типа.
void main (void) {
int    x = 1;
int    y = 2;
  cout << "Перед обменом (целые)\n";
  OutXY(x, y);
  Swap(x, y);
  cout << "После обмена\n";
  OutXY(x, y);
float  x1 = 1.5;
float  y1 = 5.9;
```

```
  cout << "Перед обменом (float)\n";
  OutXY(x1, y1);
  Swap(x1, y1);
  cout << "После обмена\n";
  OutXY(x1, y1);
My_class t1(1, 1.5), t2(2, 9.9);
  cout << "Перед обменом (классы)\n";
  t1.Out();
  t2.Out();
  Swap(t1, t2);
  cout << "После обмена\n";
  t1.Out();
  t2.Out();
}
```

Шаблоны функций используются для работы с данными произвольных типов. Приведем пример шаблона функции поиска наибольшего значения в массиве `Arr`, содержащего данные обобщенного типа `T`. Функция возвращает адрес элемента массива. В `main` это значение обнуляется. Шаблонной также является функция вывода массива: выводит массив, независимо от типа элементов.

```
#include <iostream.h>
template <class T> T & Find_max(T Arr[], int len) {
int    i_max=0;
  for (int i=0; i<len; i++)
     i_max = Arr[i_max] > Arr[i] ? i_max : i;
  return Arr[i_max];
}
template <class T> void Print_arr(T Arr[], int len) {
  cout << "Массив:\n";
  for(int i=0; i<len; i++)
     cout << Arr[i] << ' ';
  cout << '\n';
}
// Обращение к шаблонным функциям.
void main (void) {
// Обобщенное имя T будет int.
int a[] = {12, 22, 44, 19};
int    len_a = 4;
  Find_max(a, len_a) = 0;     // Параметр при обращении
  Print_arr(a, len_a);        // целочисленный массив.
// Обобщенное имя T будет float.
float b[] = {2.2, 3.3, 1.1};
int len_b = 3;
  Find_max(b, len_b) = 0;     // Параметр при обращении
  Print_arr(b, len_b);        // вещественный массив.
// Обобщенное имя T будет char.
```

```
char c[] = {'f', 'x', 's'};
int    len_c = 3;
  Find_max(c, len_c) = 0;     // Параметр при обращении
  Print_arr(c, len_c);        // символьный массив.
}
```

В примере мы показали, что шаблонная функция, написанная единожды, работает для данных любого из базовых типов. Если применить к данным объектных типов, описанных пользователем, то функция будет работать при условии, что класс предоставит необходимый для этого функциональный набор. Так, объекты класса должны уметь сравнить себя, присвоить и вывести. Пусть есть описание класса `My_class`, тогда обращение к шаблонной функции будет ровно таким же:

```
// Обобщенное имя T будет My_class.
My_class D[3] = {{1,0},{111,0},{11,0}};
  Find_max(D, 3) = 0;     // Параметр при обращении
  Print_arr(D, 3);        // массив My_class'ов.
```

Класс `My_class` должен иметь функциональность, соответствующую тем задачам, в которых он принимает участие. Опишем функциональность класса.

```
class My_class {
int a, b;
public:
My_class (int a, int b) {
  this->a = a;
  this->b = b;
}
My_class() {
  a = b = 0;
}
// Для поиска наибольшего уметь сравнить два объекта.
int operator > (My_class &T) {
  if (a > T.a) return 1;
     else return 0;
}
// Перегрузить присваивание.
My_class operator = (My_class &T) {
  a = T.a;
  b = T.b;
  return *this;
}
// Перегрузить присваивание нулю.
My_class operator = (int n) {
  a = n;
  b = n;
  return *this;
}
```

```
// Вывод в поток не зависит от типа.
friend ostream &operator << (ostream &out, My_class A);
// Этот метод для использования в шаблоне не подходит.
void Out() {
cout << "a=" << a << " b=" << b << endl;
}
};
// Перегружена операция вывода в поток.
ostream &operator << (ostream &out, My_class A) {
  out << "a= " << A.a << " b=" << A.b << endl;
  return out;
}
```

Шаблонная функция одинаковым образом отрабатывает свою функциональность для данных любого типа.

Еще раз напомним, как реализован механизм шаблона функций. Компилятор, обрабатывая вызовы функций, узнает тип данных при обращении к функции. Соответственно, генерирует код функции нужного типа, заменяя абстрактное (параметризованное) имя реальным именем типа.

5.2. Шаблоны классов

Шаблоны классов определяются аналогично шаблонам функций, позволяют работать с данными разных типов, порождают конкретные типы в зависимости от типа, переданного при обращении.

Замечание. В управляемом C++/CLI, реализованном в Microsoft Visual Studio существуют ограничения на использование шаблонов, вызванные особенностью реализации объектной модели.

Определение шаблона класса

Синтаксис определения шаблона класса, в отличие от определения просто класса, предваряется ключевым словом `template` и списком параметров класса.

Заголовок шаблона класса имеет синтаксис:

```
template <список_параметров_шаблона>
определение_класса
```

В таком определении объявляется не просто класс, а семейство классов. Имя класса в этом случае – параметризованное имя семейства, где параметрами являются имена типов.

Замечательный пример шаблона класса – массив. Тип элементов массива может быть любым, в том числе отличным от базовых типов.

Механизм шаблона классов

Если в коде есть описание шаблона класса, то компилятор, обнаруживая объявление переменной такого типа, должен породить объект класса. Для этого ему нужно знать, какой именно тип нужен здесь и сейчас. Чтобы

передать компилятору эти знания, нужны фактические параметры шаблона, которые подставляются на место формальных.

Синтаксис объявления переменной, создаваемой на основе шаблона, требует указания типа, к которым порождается объект:

```
Имя_класса <фактические_параметры>
Имя_объекта (могут_быть_параметры_конструктора)
```

Угловые скобки в объявлении обязательны, как признак имени типа для компилятора при обработке кода.

Пример реализации шаблона класса рассмотрим на примере шаблона массива элементов любого типа. Для массивов обычно реализуются алгоритмы: ввод данных, вывод, сортировка, поиск, редактирование и им подобные. С использованием шаблонов такие распространенные алгоритмы пишутся единожды. Тип элементов массива определен обобщенным именем `Atype`, имя класса – `array`.

```
template <class Atype> class array {
Atype  *a;           // Тип элементов и длина массива.
int    len;
public:
array(int size);          // Конструктор массива.
~array(){                 // Деструктор массива.
  delete [] a;
}
// Перегрузка операции разименования.
Atype &operator [](int n);
void Sort(void);          // Сортировка массива.
}; // End of array.
// Конструктор массива выделяет память для размещения
// элементов и заполняет их нулями.
template <class Atype> array <Atype>::array(int size) {
int    i;
  len = size;
  a = new Atype[size];
  for (i = 0; i<size; i++)
       a[i] = 0;
}
// Операция [] перегружена. Это бинарная операция,
// у которой первый операнд – имя массива,
// второй – номер элемента внутри массива.
template <class Atype>
  Atype &array <Atype> :: operator [](int n) {
  if (n<0 || n>len-1) {
       cout << "Выход за границу массива";
       exit(1);
  }
  return a[n];
```

```
}
// Сортировка элементов массива методом пузырька.
template <class Atype> void array <Atype>::Sort(void) {
int    i, j;
Atype  b;
  for (i = 0; i<len-1; i++) {
       for (j = 0; j<len-i-1; j++) {
            if (a[j]>a[j+1]) {
                 b = a[j];
                 a[j] = a[j+1];
                 a[j+1] = b;
            }
       }
  }
}
// Пример обращения.
void main (void) {
// Создание и работа с целочисленным массивом.
array <int> My_Int(20);
int    i;
for (i = 0; i<20; i++)
  My_Int[i] = 20-i;
My_Int.Sort();
cout << "\nОтсортированный массив\n";
for (i = 0; i < 20; i++)
  cout << My_Int[i] << " ";
cout << endl;
// Создание и работа с символьным массивом.
array <char> My_Char(20);
for (i = 0; i<20; i++)
  My_Char[i] = 116-i;
My_Char.Sort();
cout << "\nОтсортированный массив\n";
for (i = 0; i<20; i++)
  cout << My_Char[i] << " ";
cout << endl;
} // main
```

Рассмотрим пример использования шаблона для типов данных, отличных от базовых. Пусть тот же шаблон используется для данных типа `My_class`, описанных в примере выше по тексту (раздел 5.1). Можно записать обращение «по образцу», и это будет работать.

```
array < My_class > Arr(20);
for (i = 0; i<20; i++)
  Arr[i] = 116-i;
Arr.Sort();
cout << "Отсортированный массив\n";
```

```
for (i = 0; i<20; i++)
  cout << Arr[i] << " " ;
```

5.3. Особенности использования шаблонов функций и классов

1. Формальные параметры шаблона.

Каждый параметр шаблона фактически заменяет произвольный тип. При определении класса параметр шаблона используется как имя типа (формальное описание). При обращении к шаблону имя заменяется на фактическое, и шаблон автоматически расширяется компилятором до полного описания класса.

2. Шаблонные функции класса.

Функции, описанные внутри класса, являются `inline`. При внешнем определении, вне тела шаблона, используется полное объявление функции, включая параметр шаблона «`Имя_класса <Type>`», так:

```
template <class Type>
Type имя_класса <Type>::имя_функции (...) {
  // тело функции;
}
```

3. Перегрузка шаблонов функций.

Шаблон функции порождает общую функцию обработки данных. Эта функция может быть перегруженной, если требуется. Например, `Swap` элементарно реализуется для базовых типов, но для строк реализация должна быть другой, а для объектов классов реализация должна быть третьей. В любом случае для данных нестандартного типа используется перегрузка функций или операций. Допускается как перегрузка по типу параметров, так и по типу функции. Например, `Swap` для обмена значений двух строк типа `char` будет выглядеть так:

```
void Swap(char *s1,char *s2) {
int len;
len = (strlen(s1)>=strlen(s2))? strlen(s1): strlen(s2);
  char *tmp = new char[len+1];
  strcpy(tmp,s1);
  strcpy(s1,s2);
  strcpy(s2,tmp);
  delete tmp;
}
```

4. Перегрузка операций для шаблонов.

Во многих случаях хочется и можно воспользоваться готовыми шаблонами, применяя их к новым типам данных, определенных пользователем. Например, наша `Swap` могла бы успешно переменить местами не только данные базовых типов, но и данные производных типов, благодаря тому, что конструктор копирования существует по умолчанию. Однако если бы мы попытались каким-то образом упорядочить массив точек на плоскости, наш метод `Sort` для массивов не справился бы с этой задачей. И совсем не

потому, что задача абстрактно не имеет смысла, а потому, что пришлось бы сравнить с помощью операции "<" или ">" две точки. Логическое решение, это сравнить радиус-векторы точек, программное решение, это перегрузить операцию "<" или ">" таким образом, чтобы она сравнила радиус-векторы точек и вернула логическое значение. Во многих случаях требуется также перегружать конструктор копирования.

5. Дружественные функции.

Шаблоны классов могут содержать друзей. Функция `friend`, не использующая спецификацию шаблона, будет дружественной для всех экземпляров шаблона класса. Функция `friend` с параметрами шаблона будет дружественной только для того класса, экземпляр которого фактически создается. Например,

```
template <class Type> class Matrix {
// Шаблон матрицы типа Type.
…
// Универсальная функция.
friend void Norma();
…
// Экземпляр для класса.
friend Vector <Type> Sum (Vector <Type> V);
…
} // End of Matrix
```

Прочие особенности дружественных функций сохраняются.

6. Статические элементы.

Статические элементы не универсальны, а являются собственными данными для каждой реализации объекта.

```
template <class Type> class My_Static {
public:
static int Count;
…
} // End of My_Static.
// Инициализация статического данного
template <class Type> int My_Static::Count = 0;
// Объявление
Type <int> A;
Type <char> B;
```

Определены две статические переменные `My_Static <int>::Count` и `My_Static <char>::Count`.

7. Число параметров шаблона.

Число параметров шаблона (функции или класса) не ограничено, типы могут быть различны. Например, можно параметризовать не только тип элементов массива, но и число данных в нем.

```
template <int n, class Type> class Array {
public:
Type   a[n];
…
};
```

Нетипизированные параметры шаблона должны быть заменены фактическими константными выражениями. Так, при обращении указаны два параметра: количество данных и их тип.

```
Array <50,double> One,Two;
```

8. Наследование.

Параметризованные классы могут использоваться в схеме наследования. Могут комбинироваться в различных формах. Класс шаблон может быть порожден из обычного, как и обычный класс может быть порожден от шаблона.

5.4. Пример спецификации шаблона класса «Множество»

Приведем пример описания шаблона класса «Множество» элементов любого типа.

```
// Класс - множество элементов любого типа.
// Имя Atype - тип элементов множества.
template <class Atype> class Set {
  Atype *a;          // Тип элементов и количество.
  int  len;
public:
Set (int size);      // Конструктор.
~ Set() {            // Деструктор.
  delete [] a;
}
bool Is_Empty();     // Проверка на пустоту.
bool Is_Full();      // Проверка на заполнение.
bool Add(Atype T);   // Включение элемента.
bool In_Set(Atype T);     // Принадлежность элемента.
Atype Get(Atype T);       // Извлечение элемента.
// Другие методы.
}; // End of Set.
// Пример реализации одного из шаблонных методов.
// Реализация вынесена из тела класса.
// Метод – шаблонная функция.
template <class Atype> Set <Atype> :: Set (int size) {
  len = size;
  a = new Atype[size];
  for (int i = 0; i < size; i++)
       a[i] = 0;     // Для базовых типов.
                     // Для нестандартных определить.
}
```

Реализация методов множества отличается от реализации схожих по именам методов массива. Так, в методе `Add` включения элемента во множество, должен выполняться контроль: элемент, который уже есть во множестве, не включается. Метод `In_Set` позволяет проверить вхождение элемента во множество, и используется другими методами, например, методом `Get` извлечения элемента из множества.

Для реализации методов можно использовать перегрузку операций. Так, вполне логично операции включения и исключения реализовать именно операциями:

```
Set operator +(Atype T);
Set operator -(Atype T);
```

Тогда обращение к множествам выглядит абстрактно.

```
Set <Point> SetP(10);     // Множество точек.
Set <Ball> SetB(30);      // Множество шаров.
Point A(0,0);
//Сравните абстракцию операции:
SetP.Add(A);              SetP = SetP + A;
Ball B(1.5);
//Сравните абстракцию операции:
SetB.Add(B);              SetB = SetB +B;
```

К обычным операциям над множествами относятся операции пересечения, объединения, вычитания (дополнения), включения. Они также могут быть реализованы шаблонами операторных функций.

```
Set operator +(Set T);
Set operator *(Set T);
```

И последний комментарий: шаблон может содержать данные любых типов. Данные должны быть готовы для работы в шаблонных функциях, то есть содержать весь необходимый функциональный набор.

5.5. Библиотеки шаблонов

Помимо стандартных библиотек, встроенных в реализацию языка, сторонними разработчиками создано множество дополнительных библиотек для решения различных прикладных задач, из которых многие являются объектно-ориентированными.

Использование сторонних библиотек сокращает время разработки и объем кода. Решения, использованные разработчиками библиотек, имеют хорошие показатели качества, поэтому их использование оптимизирует работу программы.

Знание и использование полезных библиотек позволяет разработчику быстро и эффективно создавать качественные приложения.

Категории задач, для которых можно использовать библиотеки, самые разнообразные: веб-разработка, работа с графикой, решение математических задач и многое другое.

По внутреннему содержанию библиотеки можно категоризировать по разным параметрам, например, есть объектно и процедурно ориентированные библиотеки, есть кроссплатформенные и ориентированные на конкретную операционную систему.

Чаще всего, это бесплатно распространяемые библиотеки, некоторые − с открытым кодом, для которых разработчики предоставляют техническую документацию.

Дадим краткий обзор некоторых библиотек шаблонов, позволяющих эффективно программировать прикладные задачи.

Библиотека стандартных шаблонов STL

Стандартная библиотека шаблонов **STL** (Standard Template Library), это набор согласованных обобщённых алгоритмов, контейнеров, средств доступа к их содержимому и различных вспомогательных функций в C++. Встроена в стандарт C++.

Подробное описание библиотеки можно найти по адресу http://www.solarix.ru/for_developers/cpp/stl/stl.shtml.

Библиотека предоставляет набор абстрактных структур данных, моделирующих задачи представления данных, и набор алгоритмов для работы с ними. Все шаблонные алгоритмы работают как со структурами данных, предоставляемыми библиотекой, так и с встроенными структурами данных C++. Например, все алгоритмы работают с обычными указателями.

Ортогональный проект библиотеки позволяет программистам использовать библиотечные структуры данных со своими собственными алгоритмами, а библиотечные алгоритмы – с собственными структурами данных. Хорошо определённые семантические требования и требования сложности гарантируют, что компонент пользователя будет работать с библиотекой, и что он будет работать эффективно.

Подробнее об этой библиотеке далее в разделе 5.6.

Boost

Проект Boost – это одновременно сообщество разработчиков и набор свободно распространяемых библиотек на C++. Web-сайт находится по адресу http://boost.org.

Boost предоставляет набор библиотек классов, использующих функциональность языка C++ и предоставляющих кроссплатформенный высокоуровневый интерфейс для профессиональной разработки приложений: работа с данными, алгоритмами, файлами, потоками и другими.

Библиотеки Boost охватывают следующее:

- алгоритмы;
- многопоточное программирование;
- контейнеры;
- структуры данных;
- функциональные объекты;

- обобщённое программирование;
- графы;
- работа с геометрическими данными;
- ввод-вывод;
- итераторы;
- математические и числовые алгоритмы;
- синтаксический и лексический разбор;
- обработка строк и текста;
- и другие.

Пример. Для решения задач линейной алгебры, Boost включает библиотеку uBLAS с операциями для векторов и матриц.

Пример, показывающий умножение матрицы на вектор.

```
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/io.hpp>
using namespace boost::numeric::ublas;
// Вектор y равен произведению матрицы A на вектор x.
int main() {
vector<double> x(2);      // Создан и заполнен вектор.
x(0) = 1; x(1) = 2;
matrix<double> A(2,2);    // Создана и заполнена матри-
ца.
A(0,0) = 0; A(0,1) = 1;
A(1,0) = 2; A(1,1) = 3;
  vector<double> y = prod(A, x); // Умножение.
  std::cout << y << std::endl;
return 0;
}
```

Лаконичность этого кода поражает.

Описание библиотек Boost представлено разработчиками на официальном сайте.

Отметим, что Boost предоставляет эффективную реализацию графов. Можно выбрать способ представления графа и тип данных. Одним из типов может быть `std::vector из STL`. Среди предоставляемых реализаций алгоритмов:

- поиск в ширину;
- поиск в глубину;
- алгоритм Беллмана-Форда;
- алгоритм Дейкстры;
- алгоритм Прима;
- алгоритм Краскала;
- нахождение компонент связности графа;
- и другие.

Библиотека Net.Framework

.NET Framework, поддерживая разработку и исполнение приложений, ставит своей целью решение следующих задач:

- объектно-ориентированная среда программирования;
- среда исполнения, решающая проблемы конфликта версий;
- среда безопасного исполнения кода (безопасность типов и другие особенности управляемого кода);
- среда исполнения, улучшающая производительность за счет управления памятью и кэширования результатов компиляции;
- интеграция и переносимость приложений достигается использованием общего промежуточного языка и библиотеки типов.

Библиотека типов − .NET Framework Class library (FCL), это больше, чем библиотека шаблонов, это идеология …

Библиотека FCL включает в себя Common Language Specification (CLS – общая языковая спецификация), которая устанавливает основные правила языковой интеграции. Спецификация CLS определяет минимальные требования, предъявляемые к языку платформы .NET. Компиляторы, удовлетворяющие этой спецификации, создают объекты, способные взаимодействовать друг с другом. Поэтому любой язык, соответствующий требованиям CLS, может использовать все возможности библиотеки FCL.

В .NET включены сборки библиотеки классов .NET Framework FCL, содержащие определения нескольких тысяч типов, каждый из которых предоставляет определенную функциональность. Наборы схожих типов собраны в отдельные пространства имен. Например, пространство имен `System` содержит базовый класс `Object`, из которого порождаются все остальные типы. Таким образом, всякая сущность в .NET является объектом со своими полями и методами.

Кроме того, `System` содержит типы для целых чисел, символов, строк, обработки исключений, консольного ввода/вывода, группу типов для безопасного преобразования одних типов в другие, форматирования данных, генерации случайных чисел и выполнения математических операций. Типами из пространства имен `System` пользуются все приложения.

Можно измененить существующий FCL-тип созданием собственных производных типов. Можно создавать свои собственные типы, в том числе, основанные на типах FCL, можно создавать пространства имен. Все это строго соответствует общим принципам объектной модели данных, предлагаемым платформой .NET.

Приведем распространенные пространства имен FCL с кратким описанием содержащихся там типов.

`System` содержит фундаментальные типы данных и классы.

`System::Collections` содержит абстрактные типы данных − хэш-таблицы, списки, массивы и другие контейнеры.

`System::Drawing` содержит классы для вывода графики (GDI+).

`System::IO` содержит классы файлового и потокового ввода/вывода.

`System::Reflection` содержит классы для чтения и записи метаданных.

`System::Threading` содержит классы для создания и управления потоками.

`System::Wndows::Forms` содержит классы для приложений с графическим интерфейсом пользователя.

Часть FCL описывает базовые типы. Определение фундаментальных данных облегчает совместное использование языков программирования в .NET Framework. Все вместе это называется Common Type System (единая система типов).

Единство каркаса

Каркас одинаков для всех языков среды. Поэтому разработка на любом языке программирования использует классы одной и той же библиотеки.

Многие классы библиотеки, составляющие общее ядро, используются всеми языками. Отсюда единство интерфейса приложения, на каком бы языке оно не разрабатывалось, единство работы с коллекциями и другими контейнерами данных, единство связывания с различными хранилищами данных и прочая универсальность.

Встроенные примитивные типы

Классы, описывающие базовые типы, считаются встроенными в язык программирования. В библиотеке FCL они называются примитивными, причем типы каркаса покрывают все множество встроенных типов, встречающихся в языках программирования. Типы конкретного языка программирования проецируются на соответствующие типы каркаса.

Например, тип `Integer` языка Visual Basic и тип `int` языка C++ проецируются в один и тот же тип каркаса `System::Int32`. В каждом языке программирования, наряду с родными для языка названиями типов, разрешается пользоваться именами типов, принятыми в каркасе.

Структурные типы

Частью библиотеки являются структурные типы, задающие организацию данных: строки, массивы, перечисления, структуры (записи).

Модульность

Число классов библиотеки FCL – несколько тысяч. Классы с близкой функциональностью объединяются в группы, называемые пространством имен (`Namespace`). Для динамического компонента CLR физической единицей, объединяющей классы и другие ресурсы, является сборка (`assembly`).

Основным пространством имен библиотеки FCL является пространство `System`, содержащее как классы, так и другие вложенные пространства имен. Так, уже упоминавшийся примитивный тип `Int32` непосредственно

вложен в пространство имен `System` и его полное имя, включающее имя пространства – `System::Int32`.

В пространство `System` вложены другие пространства имен. Например, в пространстве `System::Collections` находятся классы и интерфейсы, поддерживающие работу с коллекциями объектов – списками, очередями, словарями. В пространство `System::Collections` также вложено пространство имен `Specialized`, содержащее специальные коллекции, элементами которых являются только строки. Пространство имен `System::Windows::Forms` содержит классы, используемые при создании Windows-приложений. Класс `Form` из этого пространства задает форму – окно, обеспечивающее интерактивное взаимодействие с пользователем.

Библиотека OpenGl

Для разработки графического пользовательского интерфейса (Graphical User Interface, GUI) существуют библиотеки, использующие различные технологии для создания графического интерфейса Их использование позволяет эффективно разрабатывать графическую часть приложения.

Назовем некоторые из них:

SFML (Simple and Fast Multimedia Library) − кроссплатформенная графическая библиотека для 2D-графики.

Qt − набор инструментов для быстрого и удобного проектирования GUI.

Cairo − кроссплатформенная библиотека для отрисовки векторных изображений.

wxWidgets − одна из старейших графических библиотек для отрисовки GUI.

Подробнее назначение и примерный набор инструментов покажем на примере OpenGL (Open Graphics Library). Это не объектно-ориентированная и не шаблонная библиотека, это открытый и мобильный графический стандарт в области компьютерной графики, разработанный и утвержденный в 1992 году. Разработчики OpenGL – ведущие фирмы разработчики оборудования и программного обеспечения.

Все программы, написанные с помощью OpenGL, кроссплатформенные. Если устройство поддерживает какую-то функцию, то эта функция выполняется аппаратно, если нет, то библиотека выполняет её программно.

С точки зрения программиста, OpenGL, это программный интерфейс для графических устройств. Он включает в себя около 150 различных команд, с помощью которых программист может определять различные объекты и воспроизведить (Rendering) их.

Для объектов изображения программист описывает объекты, задает их положение в трёхмерном пространстве, определяет параметры (поворот, масштаб и другие), задает свойства объектов (цвет, текстура, материал и

другие), определяет положение наблюдателя. Далее библиотека OpenGL воспроизводит это на экране. OpenGL имеет процедурный интерфейс.

Основные возможности OpenGL.

1. Геометрические и растровые примитивы: точки, линии, полигоны; растровые: битовый массив(bitmap) и образ(image).

2. B-сплайны используются для рисования кривых по опорным точкам.

3. Видовые и модельные преобразования позволяют располагать обьекты в пространстве, вращать их, изменять форму, или изменять положение наблюдателя.

4. Работа с цветом в режиме RGBA (красный-зелёный-синий-альфа) или в индексном режиме, где цвет выбирается из палитры.

5. Удаление невидимых линий и поверхностей.

6. Одинарная и двойная буферизация. Двойная буферизация используется для того, чтобы устранить мерцание при мультипликации.

7. Наложение текстуры позволяет придавать объектам реалистичность.

8. Сглаживание позволяет скрыть ступенчатость, свойственную растровым дисплеям.

9. Освещение позволяет задавать источники света, их расположение, интенсивность.

10. Атмосферные эффекты, такие как туман, позволяют придать объектам или сцене реалистичность.

Основные возможности реализованы через функции библиотеки, которые можно разделить на пять категорий.

1. Функции описания примитивов определяют объекты нижнего уровня иерархии (примитивы), которые может отображать графическая подсистема: точки, линии, многоугольники и т.д.

2. Функции описания источников света служат для описания положения и параметров источников света, расположенных в трехмерной сцене.

3. Функции задания атрибутов позволяют задать атрибуты объектов, которые определяют, как будет выглядеть на экране отображаемые объекты: цвет, характеристики материала, текстуры, параметры освещения.

4. Функции визуализации позволяют задать положение наблюдателя в виртуальном пространстве и параметры объектива камеры. Зная эти параметры, система сможет не только правильно построить изображение, но и отсечь объекты, оказавшиеся вне поля зрения.

5. Функции геометрических преобразований позволяют выполнять различные преобразования объектов – поворот, перенос, масштабирование.

Дополнительные операции, это использование сплайнов для построения линий и поверхностей, удаление невидимых фрагментов изображений, работа с изображениями на уровне пикселей и другие.

OpenGL состоит из набора библиотек. Базовые функции, предоставляющие практически все возможности для моделирования и воспроизве-

дения трёхмерных сцен, хранятся в основной библиотеке GL. Помимо нее, OpenGL включает в себя несколько дополнительных библиотек.

Первая из них называется библиотекой утилит GLU (GLU – GL Utility), поставляется вместе с главной библиотекой OpenGL. В библиотеку утилит входит реализация более сложных функций, таких как набор популярных геометрических примитивов (куб, шар, цилиндр, диск), функции построения сплайнов, реализация дополнительных операций над матрицами.

Библиотека GLUT (GL Utility Toolkit) предоставляет минимально необходимый набор функций для создания OpenGL-приложения. Предоставляет функции для работы с окнами, клавиатурой и мышью. В библиотеке GLUT есть команды для определения окна, в котором будет работать OpenGL, определение прерываний от клавиатуры или мышки. GLUT обеспечивает кроссплатформенность приложения, поддерживает довольно много операционных систем.

Библиотека GLAUX разработана фирмой Microsoft для операционной системы Windows. Во многом схожа с библиотекой GLUT, предназначена только для Windows.

Пример. В этом примере показана структура GLUT-приложения и иллюстрируются некоторые основы OpenGL. Программа в графическом окне выводит случайный набор цветных прямоугольников, который меняется при нажатии левой кнопки мыши. Выбор правой кнопки мыши меняет режим заливки прямоугольников.

```
#include <stdlib.h>
#include <gl/glut.h>
#define random(m) (float)rand()*m/RAND_MAX
#define Esc 27
// Ширина и высота окна.
GLint Width = 512, Height = 512;
int   Times = 100;   // Число прямоугольников в окне.
int   FillFlag = 1;  // С заполнением или нет.
long  Seed = 0;
// Функция рисует прямоугольник.
void DrawRect(float x1,float y1,float x2,float y2,int FillFlag)
 {
   glBegin(FillFlag ? GL_QUADS : GL_LINE_LOOP);
   glVertex2f(x1, y1);
   glVertex2f(x2, y1);
   glVertex2f(x2, y2);
   glVertex2f(x1, y2);
   glEnd();
 }
// Функция управляет выводом на экран.
void Display(void) {
int i;
float x1, y1, x2, y2;
```

```
float r, g, b;
srand(Seed);
glClearColor(0, 0, 0, 1);
glClear(GL_COLOR_BUFFER_BIT);
for( i = 0; i < Times; i++ ) {
  r = g = b = random(1);
  glColor3f( r, g, b );
  x1 = random(1) * Width;
  y1 = random(1) * Height;
  x2 = random(1) * Width;
  y2 = random(1) * Height;
  DrawRect(x1, y1, x2, y2, FillFlag);
  }
glFinish();
}
// Функция вызывается при изменении размеров окна.
void Reshape(GLint w, GLint h) {
  Width = w;
  Height = h;
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(0, w, 0, h, -1.0, 1.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}
// Функция обрабатывает сообщения от мыши.
void Mouse(int button, int state, int x, int y) {
  if( state == GLUT_DOWN ) {
  switch( button ) {
     case GLUT_LEFT_BUTTON: {
        Seed = random(RAND_MAX);
        break; }
     case GLUT_RIGHT_BUTTON: {
        FillFlag = !FillFlag;
        break; }
  }
  glutPostRedisplay();
  }
}
// Функция обрабатывает сообщения от клавиатуры.
void Keyboard(unsigned char key, int x, int y ) {
  if( key == Esc )
     exit(0);
}
// Пример main.
main(int argc, char *argv[])
```

```
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGB);
  glutInitWindowSize(Width, Height);
  glutCreateWindow("Rect draw example (RGB)");
  glutDisplayFunc(Display);
  glutReshapeFunc(Reshape);
  glutKeyboardFunc(Keyboard);
  glutMouseFunc(Mouse);
  glutMainLoop();
} // main
```

Не претендуя на полноту, пример показывает, как происходит обращение к функциям OpenGL. Краткость кода скрывает детали реализации методов.

5.6. Библиотека стандартных шаблонов STL

Библиотека стандартных шаблонов STL (Standard Template Library) была разработана Александром Степановым и Менг Ли, и сначала была сторонней разработкой. Включена в стандарт C++, является неотъемлемой частью языка.

Структура библиотеки

Библиотека содержит пять основных видов компонентов.

1. контейнер (`container`) управляет набором объектов в памяти.

2. итератор (`iterator`) обеспечивает для алгоритма средство доступа к содержимому контейнера.

3. алгоритм (`algorithm`) определяет вычислительную процедуру.

4. функциональный объект (`function object`) инкапсулирует функцию в объекте для использования другими компонентами.

5. адаптер (`adaptor`) адаптирует компонент для обеспечения различного интерфейса.

Структура библиотеки ортогональная: все алгоритмы работают со всеми данными: с базовыми, с контейнерами STD, с типами, определенными программистом. Например, одна функция поиска работает с каждым из видов контейнеров.

Структуру библиотеки можно представить как куб: одно измерение представляет различные типы данных (например, `int`, `double`), второе измерение представляет различные контейнеры (например, вектор, связный список, файл), а третье измерение представляет различные алгоритмы с контейнерами (например, поиск, сортировка).

Если размеры измерений i, j и k, тогда должно быть разработано $i^* j^* k$ различных версий кода. При использовании шаблонных функций, которые параметризуют типы данных, число версий $= j * k$.

Если же и алгоритмы будут работать с разными контейнерами, то число версий = $j+k$.

Это не только упрощает разработку программ, но позволяет гибким способом использовать компоненты в библиотеке вместе с компонентами, определяемыми пользователем. Пользователь может определить специализированный контейнерный класс и использовать для него библиотечную функцию сортировки. Для сортировки пользователь может выбрать собственную функцию сравнения объектов через обычный указатель на функцию, либо через функциональный объект. Для функционального объекта переопределен `operator()`, который сравнивает. Передвижение по контейнерам выполняется через итераторы. Но если существующих возможностей недостаточно, используется адаптер.

Операции (Operators)

Для сравнения объектов переопределены операции отношения:

```
bool operator!=(const T1& x, const T2& y)
bool operator>(const T1& x, const T2& y)
bool operator<=(const T1& x, const T2& y)
bool operator>=(const T1& x, const T2& y)
```

Пара (Pair)

Библиотека включает шаблоны для разнородных пар значений.

```
template <class T1, class T2> struct pair {
T1 first;
T2 second;
  pair() {
  }
  pair(const T1& x, const T2& y): first(x), second(y){
  }
};
```

Для пар перегружены операции `operator ==` и `operator<`.

Шаблонная функция `make_pair`, упрощает инициализацию.

```
pair<int, double>(5, 3.1415926);   // Явное объявление.
make_pair(5, 3.1415926);           // Типы определяются.
```

Контейнеры

Контейнерные классы STL (коллекции) наиболее востребованы, предоставляют широкий функциональный набор. Контейнеры STL делятся на три категории:

- последовательные;
- ассоциативные;
- адаптеры.

Последовательные контейнеры, это контейнерные классы, элементы которых расположены в последовательности. Как правило, имеют прямой доступ к элементу по его номеру.

Перечислим последовательные контейнеры.

vector;
deque;
list;
forward_list;
basic_string.

Класс **vector** (вектор), это динамический массив, коллекция элементов произвольного типа. Автоматически увеличивается по мере необходимости. Обеспечивает прямой доступ к элементам операцией [], поддерживает вставку и удаление элементов. #include <vector>

Класс **deque** (дек), это двусторонняя очередь. Реализована как динамический массив, поддерживающий вставку и удаление элементов с двух сторон.

Класс **list** (список), это двусвязный список, организованный по всем правилам реализации списков: каждый элемент ссылается на предыдущий элемент списка и на следующий элемент. Это контейнер с последовательным доступом: доступ предоставляется к началу и к концу списка. Преимуществом двусвязного списка является высокая скорость операций добавления и удаления элементов. #include <list>

Классы **string** и **wstring** (строки) обычно не относят к последовательным контейнерам, но их можно рассматривать как векторы с элементами типа char или wchar. #include <string>

Ассоциативные контейнеры, это контейнерные классы, которые сохраняют элементы в упорядоченном виде. Основаны на хеш-таблицах.

Класс **set** (множество), это коллекция уникальных значений Key, каждое из которых является также и ключом. В set хранятся только уникальные элементы, которые сортируются по значениям. К ключу предъявляются те же требования, что и для map. Использовать set имеет смысл, если определен класс, хранящий пару ключ-значение и определяющий операцию сравнения по ключу. #include <set>

Класс **multiset** (множество), это множество set, в котором допускаются повторяющиеся элементы.

Класс **map** (ассоциативный массив), это множество, в котором каждый элемент является парой «ключ-значение» (pair<const Key, Value>). Ключ используется для сортировки и индексации данных и должен быть уникальным. Значение, это произвольные данные. Предназначена для быстрого поиска значения по уникальному ключу Key. В качестве ключа используется строка или int , дубликаты ключа не допускаются. Поиск значения по ключу осуществляется быстро, потому что пары хранятся в сортированном виде. Скорость вставки новой пары обратно пропорциональна количеству элементов в коллекции. #include <map>.

Класс **multimap** (словарь), это map, который допускает дублирование ключей. Ключи отсортированы в порядке возрастания, и значение извлекается по ключу. Так, поиск по ключу вернет набор значений, сохраненных с данным ключом. `#include <map>`

Адаптеры, это специальные контейнерные классы, назначенные (адаптированные) для выполнения типовой задачи. Можно выбрать, какой последовательный контейнер должен использовать адаптер.

Класс **stack** (стек), это контейнерный класс, элементы которого работают по принципу LIFO («**L**ast **I**n, **F**irst **O**ut» – последним пришёл, первым вышел). Элементы добавляются в конец контейнера и удаляются из конца контейнера. Обычно в стеках используется deque в качестве последовательного контейнера по умолчанию.

Класс **queue** (очередь), это контейнерный класс, элементы которого работают по принципу FIFO («**F**irst **I**n, **F**irst **O**ut» – первым пришёл, первым вышел). Элементы вставляются в конец контейнера, а удаляются из начала контейнера. По умолчанию в очереди используется deque в качестве последовательного контейнера.

Класс **priority_queue** (очередь с приоритетом), это тип очереди, в которой элементы отсортированы. Элемент с наивысшим приоритетом находится в начале такой очереди, а удаление элементов выполняется с начала.

STL Строки

STL строки поддерживают форматы ASCII и Unicode.

`string` – коллекция символов типа `char` в формате ASCII. `#include <string>`

`wstring` – коллекция двухбайтных символов `wchar_t`, которые используются для представления набора символов в формате Unicode. `#include <xstring>`

Методы коллекций

Назовем методы, которые присутствуют почти во всех STL коллекциях.

`capacity` – емкость коллекции, определяет реальный размер буфера коллекции. Не равен длине. При создании коллекции память для хранения элементов распределяется неявно. В процессе работы с коллекцией элементы добавляются, и когда размер буфера делается недостаточным, память распределяется в новом буфере, куда копируются все элементы старого. Размер нового буфера в два раза больше, – такая стратегия позволяет уменьшить количество операций перераспределения памяти.

В реализации STL от Microsoft распределение памяти происходит не в конструкторе, а при добавлении первого элемента коллекции. Фрагмент программы ниже демонстрирует, что размер и емкость коллекции – две разные сущности:

```
vector <int> Vc;
cout << "Real size of vector: " << Vc.capacity() << endl;
```

```
for (int j = 0; j < 10; j++) {
  vec.push_back (10);
}
cout << "Real size of vector: " << Vc.capacity() << endl;
return 0;
```

Выдача этого кода:

```
Real size of vector: 0
Real size of vector: 10
```

Другие методы

`empty` – определяет, является ли коллекция пустой.

`size` – определяет реальный размер коллекции.

`clear` – удаляет все элементы коллекции. Если элементами коллекции являются объекты, содержащие динамические данные, они должны имеет явную реализацию деструктора.

`erase` – удаляет элемент или несколько элементов из коллекции.

`reserve(n)` – распределяет память для контейнера под `n` элементов.

`resize(n)` – изменяет размер контейнера (только для векторов, списков и очередей с двумя концами).

`swap(x)` – меняет местами два контейнера.

`operator =(x)` – контейнеру присваиваются элементы контейнера `x`.

`assign(n, x)` – присваивание контейнеру `n` копий элементов `x` (не для ассоциативных контейнеров).

`assign(first, last)` – присваивание элементов из диапазона `[first:last]`.

`find(k)` – находит элемент с ключом `k`.

Итераторы

Итераторы в STL, это объекты для доступа и управления коллекциями. Являются основным способом доступа к элементам коллекций. Итераторы, это абстракция, которая ведет себя, как указатель, и чаще всего, является объектной оберткой указателя.

Существует пять типов итераторов.

1. Итераторы ввода (`input iterator`) поддерживают операции равенства, разыменования и инкремента: `==`, `!=`, `*i`, `++i`, `i++`, `*i++`. Специальным случаем итератора ввода является `istream_iterator`.

2. Итераторы вывода (`output iterator`) поддерживают операции разыменования, допустимые только с левой стороны присваивания, и инкремента: `++i`, `i++`, `*i = t`, `*i++ = t`. Специальным случаем итератора вывода является `ostream_iterator`.

3. Однонаправленные итераторы (`forward iterator`) поддерживают все операции итераторов ввода/вывода и, кроме того, позволяют без ограничения применять присваивание: `==`, `!=`, `=`, `*i`, `++i`, `i++`, `*i`.

4. Двунаправленные итераторы (`bidirectional iterator`) обладают всеми свойствами `forward` итераторов, а также имеют дополнительную операцию декремента (`--i`, `i--`, `*i--`), что позволяет им проходить контейнер в обоих направлениях.

5. Итераторы произвольного доступа (`random access iterator`) обладают всеми свойствами двунаправленных итераторов и поддерживают операции сравнения и адресной арифметики. Дают непосредственный доступ к элементу по индексу: `i+=n`, `i+n`, `i-=n`, `i-n`, `i1-i2`, `i[n]`, `i1<i2`, `i1<=i2`, `i1>i2`, `i1>=i2`.

Поддерживаются обратные итераторы (`reverse iterators`): это двунаправленные итераторы или итераторы произвольного доступа, проходящие последовательность в обратном направлении.

Операции для итераторов перегружены, имеют семантику указателей.

Операция `*` возвращает элемент, на который в данный момент указывает итератор.

Операция ++ перемещает итератор к следующему элементу контейнера. Большинство итераторов также предоставляют оператор ––– для перехода к предыдущему элементу.

Операции == и `!=` используются, чтобы узнать, указывают ли два итератора на один и тот же элемент. Для сравнения значений, на которые указывают два итератора, нужно эти итераторы разыменовать, и применить операции == или `!=` к значению.

Операция = присваивает итератору новое значение (обычно начало или конец элементов контейнера). Чтобы присвоить значение, на которое указывает итератор, другому объекту, нужно итератор разыменовать, и применить оператор = к значению.

Методы контейнерных классов для работы с операцией =

`begin()` возвращает прямой итератор, указывающий на начало коллекции.

`end()` возвращает прямой итератор, указывающий на конец коллекции, и представляющий элемент, который находится после последнего элемента в контейнере. Сигнализирует о достижении конца последовательности.

`rbegin()` возвращает обратный итератор, указывающий на начало коллекции.

`rend()` возвращает обратный итератор, указывающий на конец коллекции, указывает на элемент, следующий за последним.

`cbegin()` возвращает константный (только для чтения) итератор, представляющий начало элементов контейнера.

`cend()` возвращает константный (только для чтения) итератор, представляющий элемент, который находится после последнего элемента в контейнере.

Все контейнеры предоставляют (как минимум) два типа итераторов:

`container::iterator` – итератор для чтения/записи;

`container::const_iterator` – итератор только для чтения.

Замечание. Изменение структуры коллекции может привести к непригодности итератора.

Рассмотрим несколько примеров использования итераторов.

Пример итерации по вектору

```
// 1. Объявление итератора.
vector < int >::iterator i;
// 2. Итерация по вектору вперед.
for (i = MyArray.begin(); i != MyArray.end(); ++i)
   cout << *i << " ";
cout << endl;
// 1. Объявление итератора.
vector <My_Class>::iterator J;
// 2. Итерация по вектору вперед.
for (J = My_C.begin(); J != My_C.end(); J++)
   J->Out();
// 1. Объявление итератора.
vector <My_Class>::reverse_iterator I;
// 2. Итерация по вектору назад.
for (I = My_C.rbegin(); I < My_C.rend(); I++)
   I->Out();
```

Функторы

Функторы (функции-объекты), это специальный инструмент, который дает возможность использовать объект как функцию. Как правило, это класс, перегружающий операцию () обращения к функции. В STL используются ассоциативными контейнерами для упорядочения элементов, управляют работой алгоритмов с суффиксами `if` типа `find_if`, используются в алгоритме `for_each`.

Пример описания функтора:

```
void printInt (int number) {
   cout << number << endl;
}
```

Оператор получает числовое значение и выводит его.

Или так:

```
class Less {
bool operator() (const int &t1, const int &t2) {
  return t1 < t2;
  }
};
```

Оператор получает два значения, возвращает `true`, если первое меньше второго.

Функтор может быть шаблоном.

```
template <typename T> class Mult {
public:
T operator()(T &t1, T &t2) {
  return t1 * t2;
  }
};
```

Пример использования функтора

```
// Описание функтора.
void printInt (int number) {
   cout << number << endl;
}
// Использование функтора для vector <int> MyArray;
for_each(MyArray.begin (), MyArray.end (), printInt);
```

Что происходит: создается объект **printInt.** Он передается в функцию `for_each`, которая есть алгоритм, перебирающий коллекцию. Каждый элемент коллекции передается функции `operator()` через указатель функтор `printInt.`

Функторы STL созданы по образцу указателей на функции C++, поэтому действует правило, согласно которому объекты функций передаются по значению (копия объекта в теле функции). Это правило демонстрирует объявление алгоритма `for_each`, который получает и передает по значению объекты функций:

```
template <class Iterator, class Function> Function;
// Передача по значению
for_each(Iterator first, Iterator last, Function f);
```

В библиотеке определен набор функциональных объектов для арифметических операций:

`plus`, `minus`, `multiplies`, `divides`, `modulus`, `negate`.

Используются, чаще всего, вместе с алгоритмами.

Предикаты

Для многих алгоритмов STL необходимо задать условие, посредством которого алгоритм определяет, что ему необходимо делать с тем или иным элементом коллекции. Эта возможность передоставляется предикатами.

В общем случае, предикат, это функция, принимающая один или более параметров и возвращающая логическое значение.

Предикат может быть функцией или функтором.

В втором случае, предикаты, это подмножество функторов, в которых тип возвращаемого значения `operator() bool.` Предикаты используются в алгоритмах сортировок, поиска, а также во всех, имеющих суффикс `_if`, например, `find_if.`

Объект-функция, которая является предикатом, возвращает логическое значение в зависимости от выполнения какого-то условия: либо объект

удовлетворяет требованиям, либо это результат сравнения двух объектов по определенному признаку.

Существует набор стандартных предикатов:

`equal_to`, `not_equal_to`, `greater`, `less`, `greater_equal`, `less_equal`, `logical_and`, `logical_or`, `logical_not`.

Они выполняют операции срванения и логические операции, возвращают `bool`.

Пример использования предиката:

```
class DividedByTwo {      // Число четное.
public:
bool operator()(const int x) {
   return x%2 == 0;
}
};
// Обращение покажем для массива целых чисел.
const std::size_t N = 6;
int A[N] = {3, 2, 5, 6, 8,12};
// Найдет число элементов, удовлетворяющих предикату.
cout << count_if(A, A + N, DividedByTwo());
// Для сортировки нужен стандартный предикат less.
std::sort(A, A + N, std::less<int>());
```

Алгоритмы

Алгоритмы STL, это объекты, предоставляющие реализацию множества алгоритмов обработки данных различных типов. С использованием алгоритмов код приложения получается компактным и эффективным.

Каждый алгоритм представлен шаблоном функции или набором шаблонов функций. Таким образом, один алгоритм может работать с разными контейнерами, содержащими значения различных типов. Алгоритмы, которые возвращают итератор, для сообщения о неудаче используют конец входной последовательности. Алгоритмы, возвращающие итератор, возвращают итератор того же типа, что и на входе.

Алгоритмы в STL реализуют большинство распространенных универсальных операций с контейнерами, такие как просмотр, сортировка, поиск, вставка и удаление элементов.

Алгоритмы определены в заголовочном файле `<algorithm>`.

Можно разделить на три основных группы.

1. Функции для перебора всех членов коллекции и выполнения определенных действий над каждым из них:

`count`, `count_if`, `find`, `find_if`, `adjacent_find`, `for_each`, `mismatch`, `equal`, `search copy`, `copy_backward`, `swap`, `iter_swap`, `swap_ranges`, `fill`, `fill_n`, `generate`, `generate_n`, `replace`, `replace_if`, `transform`, `remove`, `remove_if`, `remove_copy`, `remove_copy_if`, `unique`, `unique_copy`, `reverse`, `reverse_copy`, ro-

tate, rotate_copy, random_shuffle, partition, stable_partition.

2. Функции для сортировки коллекций:

sort, stable_sort, partial_sort, partial_sort_copy, nth_element, binary_search, lower_bound, upper_bound, equal_range, merge, inplace_merge, includes, set_union, set_intersection, set_difference, set_symmetric_difference, make_heap, push_heap, pop_heap, sort_heap, min, max, min_element, max_element, lexographical_compare, next_permutation, prev_permutation.

3. Функции для выполнения арифметических действий над элементами коллекции:

accumulate, inner_product, partial_sum, adjacent_difference.

Смысл алгоритма, чаще всего, понятен из названия. Если нет, то Microsoft предлагает достаточно подробную документацию как часть MSDN для своей реализации STL [10].

Опишем некоторые алгоритмы.

Немодифицирующие операции

for_each() выполняет операции для каждого элемента последовательности.

find() находит первое вхождение значения в последовательность.

find_if() находит первое соответствие предикату в последовательности.

count() подсчитывает количество вхождений значения в последовательность.

count_if() подсчитывает количество выполнений предиката в последовательности.

search() находит первое вхождение последовательности как подпоследовательности.

search_n() находит n-ое вхождение значения в последовательность.

Модифицирующие операции

copy() копирует последовательность.

swap() меняет местами два элемента.

replace() заменяет элементы с указанным значением.

replace_if() заменяет элементы при выполнении предиката.

replace_copy() копирует последовательность, заменяя элементы с указанным значением.

replace_copy_if() копирует последовательность, заменяя элементы при выполнении предиката.

fill() заменяет все элементы заданным значением.

remove() удаляет элементы с заданным значением.

`remove_if()` удаляет элементы при выполнении предиката.

`remove_copy()` копирует последовательность, удаляя элементы с указанным значением.

`remove_copy_if()` копирует последовательность, удаляя элементы при выполнении предиката.

`reverse()` меняет порядок следования элементов на обратный.

`random_shuffle()` перемещает элементы согласно случайному равномерному распределению («тасует» последовательность).

`transform()` выполняет заданную операцию над каждым элементом последовательности.

`unique()` удаляет равные соседние элементы.

`unique_copy()` копирует последовательность, удаляя равные соседние элементы.

Сортировка

`sort()` сортирует последовательность.

`partial_sort()` сортирует часть последовательности.

`stable_sort()` сортирует последовательность, сохраняя порядок следования равных элементов.

`lower_bound()` находит первое вхождение значения в отсортированной последовательности.

`upper_bound()` находит первый элемент, больший чем заданное значение.

`binary_search()` определяет, есть ли данный элемент в отсортированной последовательности.

`merge()` сливает две отсортированные последовательности.

Работа с множествами

`includes()` проверка на вхождение.

`set_union()` объединение множеств.

`set_intersection()` пересечение множеств.

`set_difference()` разность множеств.

Минимумы и максимумы

`min()` меньшее из двух значений.

`max()` большее из двух значений.

`min_element()` наименьшее значение в последовательности.

`max_element()` наибольшее значение в последовательности.

Перестановки

`next_permutation()` следующая перестановка в лексикографическом порядке.

`prev_permutation()` предыдущая перестановка в лексикографическом порядке.

Пример. Приведем пример кода, в котором на основе вектора иллюстрируются некоторые приемы работы с итераторами, функторами, с алгоритмами `for_each`, `find_if` и `sort`.

Задача: есть набор данных типа `Person`. Для каждого объекта заданы поля: имя, профессия и возраст. Данные хранятся в векторе. Прикладная задача: выбрать тех, кто старше 30 лет, выбрать тех, кто старше указанного возраста, сортировать последовательность по убыванию возраста.

Первая задача решается с использованием предиката-функции: условие «старше 30 лет». Вторая – с использованием предиката-функтора: условие «старше n лет». Для поиска используется алгоритм `find_if`. Для сортировки используется алгоритм `sort`, он позволяет сортировать последовательность по убыванию возраста, для чего необходимо написать предикат `PLess`: принимает два объекта, возвращает `true`, если возраст первого меньше, чем возраст второго.

```
//Для многих алгоритмов STL необходимо задать предикат,
//посредством которого алгоритм определяет, что ему
//делать с тем или иным членом коллекции.
//Предикат может быть функцией или функтором.
//Существует набор стандартных предикатов.
//Приведены способы использования предикатов
//на примере алгоритмов find_if и sort.
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;
class Person;
// Перегружена операция вывода в поток.
ostream& operator << (ostream& os, Person& P);
class Person {
  string Name;
  string Profession;
  int Age;
public:
Person() {
Name = Profession = "";
Age = 0;
}
// Конструктор.
Person(string name, string Profession, int age) {
  Name = name;
  this ->Profession = Profession;
  Age = age;
}
```

```
// Аксессоры дают доступ к полям Person.
int GetAge() {
  return Age;
}
void SetAge(int age) {
  Age = age;
}
string GetName() {
  return Name;
}
void SetName(string name) {
  Name = name;
}
string GetProfession() {
  return this ->Profession;
}
void SetProfession(string profession) {
  this ->Profession = profession;
}
void PrintInfo() {
  cout << (*this);
}
}; // End of Person.
ostream& operator << (ostream& out, Person& P) {
  out << " Name: " << P.GetName() << " ";
  out << " Age: " << P.GetAge() << " ";
  out << " Profession: " << P.GetProfession() << " ";
  out << " \n------------------------------" << endl;
  return out;
};
// Предикат-функтор: сравнение объектов по возрасту
// нужно задать для сортировки.
class PLess {
public:
bool operator ()(Person& P1, Person& P2) {
  return P1.GetAge() < P2.GetAge();
}; // End of PLess.
// Предикат-функция: условие "старше 30 лет".
bool OlderThan30(Person &P) {
  return P.GetAge() > 30;
}
// Предикат-функтор: условие "старше n лет".
class OlderThan {
  int m_age;
public:
OlderThan (int age) {
  m_age = age;
```

```
}
bool operator ()(Person &P) {
  return P.GetAge() > m_age;
}
}; // End of OlderThan.
// В основном коде создан вектор объектов Person.
int main(void)
{
// Порождены четыре объекта Person.
Person P1("Ann", "Prorgammer", 23);
Person P2("Filipp", "Manager", 30);
Person P3("Ivan", "Prorgammer", 32);
Person P4("Sophia", "Manager", 29);
vector <Person> Persons;
// И добавлены в вектор.
  Persons.push_back(P1);
  Persons.push_back(P2);
  Persons.push_back(P3);
  Persons.push_back(P4);
// Найти и вывести всех, кто старше 30 лет.
cout << " 1. Найти всех, кто старше 30." << endl;
// Алгоритм find_if находит всех OlderThan30.
vector <Person>::iterator p =
find_if(Persons.begin(), Persons.end(), OlderThan30);
// Вывести операцией cout.
  while (p != Persons.end()) {
       cout << (*p);
       p++;
  }
// То же самое с использованием функтора:
cout << " 2. Найти всех, старше 25." << endl;
p = find_if(Persons.begin(), Persons.end(), OlderThan(25));
for_each(p, Persons.end(),
                    mem_fun_ref(&Person::PrintInfo));
// Сортировка по возрасту использует предикат PLess.
cout << " 3. По убыванию возраста: " << endl;
sort(Persons.begin(), Persons.end(), PLess());
for_each(Persons.begin(), Persons.end(),
                    mem_fun_ref(&Person::PrintInfo));
  return 0;
}
```

Замечание. Для передачи указателя на метод класса `PrintInfo`, которую необходимо вызвать для каждого элемента контейнера, использован адаптер метода `mem_fun_ref`. Чтобы им воспользоваться, необходимо включить `<functional>`.

Адаптеры методов используются, когда есть задача применить ко всем элементам контейнера один и тот же метод класса, содержащегося в контейнере – нужен указатель на функцию.

Пример. Пусть есть функция:

```
void Out(int a) {
{ cout << a << endl;
}
```

Применить к контейнеру `int` легко:

```
int A[3] = {1, 2, 3};
for_each (A, A+3, Out);
```

Но если это не простой массив, а контейнер, то `mem_fun_ref` вызывает метод через ссылку: `mem_fun_ref(&Person::PrintInfo))`.

Выдача этого примера приведена на рисунке 23.

```
1. Find all person older than 25
Name: Ivan  Age: 32  Profession: Prorgammer
-----------------------------------
Name: Sophia  Age: 29  Profession: Manager
-----------------------------------
2. Find all person older than 25
Name: Filipp  Age: 30  Profession: Manager
-----------------------------------
Name: Ivan  Age: 32  Profession: Prorgammer
-----------------------------------
Name: Sophia  Age: 29  Profession: Manager
-----------------------------------
3. Sorted list of person:
Name: Ivan  Age: 32  Profession: Prorgammer
-----------------------------------
Name: Filipp  Age: 30  Profession: Manager
-----------------------------------
Name: Sophia  Age: 29  Profession: Manager
-----------------------------------
Name: Ann  Age: 23  Profession: Prorgammer
-----------------------------------
```

Рис. 23. Выдача примера: результат выполения алгоритмов `for_each`, `find_if` и `sort`

Замечание. Некоторые контейнеры содержат собственные методы, имена которых совпадают с именами алгоритмов STL. Например, в ассоциативных контейнерах существуют функции `count`, `find`, в контейнере `list` существуют функции `remove`, `remove_if`, `sort`, `merge`. Если контейнер имеет такой метод, то его использование предпочтительнее, чем обращение к алгоритму. Методы классов работают эффективнее, потому что интегрированы с обычным поведением контейнеров.

Тип vector

Класс вектор, это реализация динамического массива с изменяемой длиной. Может содержать данные базовых и нестандартных типов. Предоставляет операции прямого доступа к элементам массива, добавления,

удаления элементов и многие другие. Для подключения необходимо инклудировать заголовочный файл <vector>.

В классе vector определены конструкторы пустого вектора, вектора с заданным количеством элементов, вектора с заданным количеством элементов и их значениями, конструктор копирования. Приведем примеры объявления векторов, содержащих данные разных типов.

```
vector <int> A;
vector <double> X(5);
vector <char> C(5, '*');
vector <int> B(A);        // Вектор B - копия A.
```

Общие принципы работы с вектором позволяют хранить в нем данные произвольных типов при условии, что для любого объекта, который будет храниться в векторе, должен быть определены:

конструктор по умолчанию;

операции < и ==.

Так, в примере создан вектор тип элементов которого – My_Class:

```
vector <My_Class> My_C;
My_Class A(1, 2);
My_Class B(11, 22);
My_C.push_back(A);   // Добавление.
My_C.push_back(B);   // Добавление
B = My_C.at(0);      // Извлечение по номеру.
```

Операции, определенные для класса вектор:

операции отношения ==, <, <=, !=, >, >=;

операция разименования [ ].

Некоторые методы вектора.

1. Добавление новых элементов: insert(), push_back(), resize(), assign().

2. Удаление элементов: erase(), pop_back(), resize(), clear().

3. Доступ к элементам через итератор: begin(), end(), rbegin(), rend().

4. Типичные алгоритмы работы с контейнером, такие как сортировка, поиск и другие реализованы через алгоритмы. Для их подключения необходимо включить заголовочный файл <algorithm>.

Работа с вектором через итератор приведена в разделе выше.

Приведем примеры обращения к некоторым методам вектора.

Чтобы показать, что vector готов работать с данными любого типа, создан класс «Пара».

```
class My_Pair {
  string T;
  int Key;
public:
My_Pair() {
```

```
  T = "";
  Key = 0;
}
My_Pair (string tt, int kk) {
  T = tt;
  Key = kk;
}
My_Pair(const My_Pair &M) {
  T = M.T;
  Key = M.Key;
}
// Аксессоры для извлечения полей.
string get_Text() {
  return T;
}
int get_Key() {
  return Key;
}
~My_Pair() {
}
void Out() {
  cout << "Key=" << Key << " Text=" << T << endl;
}
}; // End of My_Pair.
```

Функция `printPair` визуализирует объект.

```
void printPair(My_Pair P) {
  cout <<"Text="<< P.get_Text() <<" Key="<< P.get_Key();
}
```

Вектор создан и наполнен значениями:

```
// 1. Объявление вектора.
vector <My_Pair> My_C;
My_Pair A("One", 2);
My_Pair B("Two", 22);
My_Pair C("Three", 222);
My_Pair D("Four", 2222);
// 2. Заполнение вектора.
My_C.push_back(A);
My_C.push_back(B);
My_C.push_back(C);
My_C.push_back(D);
```

1. Доступ к элементам: `at(i)` получает доступ по индексу с проверкой. При выходе за границу генерирует исключение.

```
// 3. Извлечение значения.
B = My_C.at(0);
// 3. Извлечение значения.
B = My_C.at(5);      // Выбросит исключение.
```

```
// 4. Итерация по вектору
vector<My_Pair>::iterator J;
for (J = My_C.begin(); J != My_C.end(); J++)
  J->Out();
// 4. Использование функтора printPair.
for_each(My_C.begin(), My_C.end(), printPair);
```

2. Добавление новых элементов: `push_back()` добавляет элемент в конец вектора. `insert()` – через итератор, определяющий место включения нового элемента.

```
// 5. Добавление новых значений.
My_Pair F("New", 1);
My_C.push_back(F);        // Добавление в конец.
// Нужны итераторы.
vector<My_Pair>::iterator first, last;
first = My_C.begin();     // Положение начала.
My_C.insert(first, F);    // F добавить в начало.
first = My_C.begin();     // Дать значение итератору.
My_C.insert(first,3, A); // В начало 3 штуки.
first = My_C.begin();     // Дать значение итератору.
My_C.insert(first+5, A); // В позицию 5 3 штуки.
for_each(My_C.begin(), My_C.end(), printPair);
```

3. Удаление элементов: `pop_back()` удаляет элемент из конца вектора. `erase()` – через итератор, определяющий место исключения элемента.

```
// Удаление.
My_C.pop_back();          // Один элемент с конца.
My_C._Pop_back_n(3);      // Три элемента с конца.
first = My_C.begin();
last = My_C.begin() + 1; // Итератор определяет место.
if (last < My_C.end())
  My_C.erase(first, last);//Второй.
for_each(My_C.begin(), My_C.end(), printPair);
```

Итоги раздела

Шаблоны представляют собой мощное и эффективное средство обращения с различными типами данных, которое называют параметрическим полиморфизмом. Шаблоны функций и классов обеспечивают безопасное использование типов и являются средством реализации идей обобщенного программирования и метапрограммирования.

Предназначены для грамотного использования и требуют знания многих тонкостей. Так, программа, порождающая объекты на основе шаблона, содержит код каждого порожденного типа. С некоторыми типами данных шаблоны могут работать не эффективно.

Встроенная в C++ библиотека STL предоставляет набор шаблонов для различных способов организации хранения и обработки данных. Реализации абстрактных типов данных представлены контейнерными классами,

моделирующими массив, список, множество, ассоциативный массив. Алгоритмы STL работают с данными базовых типов, контейнерами STL и типами, разработанными пользователем. Библиотека STL может и должна использоваться для решения прикладных задач.

Сторонние библиотеки на основе шаблонов, такие как Boost, широко используются для решения многих классов прикладных задач. Являются кроссплатформенными.

5.7. Вопросы для самопроверки

1. Дайте определение понятию «обобщенное программирование».
2. Как в C++ реализуется параметрический полиморфизм?
3. Что такое шаблон функции?
4. Приведите примеры алгоритмов, которые выполняются одинаковым образом для данных различных типов.
5. Что такое параметр шаблонной функции? Что следует отнести к параметрам?
6. Приведите синтаксис описания шаблона функции.
7. Поясните семантику списка формальных параметров шаблона.
8. Как реализован механизм шаблона функций?
9. Что такое шаблон класса?
10. Приведите синтаксис определения шаблона класса.
11. Как реализован механизм шаблона классов?
12. Приведите синтаксис объявления объекта шаблонного типа. Каков механизм фактических параметров шаблона?
13. Являются ли шаблонными методы шаблонного класса?
14. Как перегрузить операцию для шаблона класса?
15. Какова семантика статических элементов шаблона?
17. Что такое библиотека шаблонов?
18. Назовите основные достоинства библиотеки STL.
19. Опишите структуру библиотеки STL.
20. Назовите основные типы данных, реализованных в STL.
21. Назовите основные алгоритмы, реализованные в STL.
22. Дайте определение итератора применительно к STL.
23. Дайте определение функтора. Как можно задать функтор?
24. Дайте определение предиката. Какова роль предикатов в STL?
25. Опишите вектор как инструмент реализации контейнерного класса.

5.8. Упражнения к пятому разделу

Упражнение 1. Шаблон функции

Пусть есть три массива: массив точек в декартовых координатах, массив данных типа `float`, массив комплексных чисел. Объявите и инициализируйте массивы. Классы точек и комплексных чисел должны быть реализованы в предыдущих упражнениях.

Напишите шаблонные функции, реализующие типовые алгоритмы: суммирование элементов массива, поиск наибольшего, сортировка.

Указания. Цель задачи – показать, что шаблон функции, это универсальный алгоритм, решающий поставленную задачу для любого типа данных. Алгоритмы – самые простые, общеизвестные. Начните с простого – напишите и протестируйте код всех шаблонных функций для массива вещественных чисел. Далее возьмите реализацию класса комплексных чисел. Задание приведено в упражнении 3 ко второму разделу, и обратитесь к шаблонным функциям, передавая как параметр тип `<Complex>`. Здесь может возникнуть ситуация, которая обсуждалась в третьем разделе: методов класса недостаточно, чтобы реализовать алгоритм функции. Тогда следует расширить функционал класса добавлением требуемых операций. Далее такую же процедуру примените к классу точек, пример реализации которого есть во втором разделе.

Следует сделать замечание, что массивы данных могут быть просто массивами, а могут быть классами. В этом упражнении исследуется тема шаблонных функций, поэтому реализация должна быть для массивов.

Упражнение 2. Шаблон класса «Множество».

Прототип для класса «Множество» приведен в разделе 5.4. На его основе напишите реализацию шаблона класса «Множество». Тестируйте шаблон на данных типа `<int>` – множество целых чисел. Далее объявите множество точек, множество комплексных чисел, множество слов (тип `<string>`). Напишите прямой итератор для просмотра элементов множества.

Указания. Здесь должен быть универсальным класс-контейнер. В него можно положить данные любого типа. Данные должны быть готовы к тому, что контейнер с ними будет манипулировать, следовательно, должны предоставить контейнеру доступ к своему набору данных и методов.

Для того, чтобы понять смысл и принципы работы с шаблонами классов, не обязательно реализовать все методы, которые могут быть функциональным содержанием множества. Обязательно следует реализовать методы:

- проверки на пустоту и на заполнение;
- добавления элемента во множество, исключая повторения;
- проверки принадлежности элемента ко множеству;
- извлечения элемента из множества.

Как-то нужно реализовать инициализацию и вывод содержимого множества. Здесь целесообразно использовать итератор.

Полезно также перегрузить операции для работы со множествами, например, операции объединения, пересечения, включения. Идеальное множество должно позволить работать с данными любых типов.

Упражнение 3. Работа с библиотекой STL

Цель этой работы – изучение структур данных и алгоритмов библиотеки STL, освоение приемов работы с предикатами, итераторами, алгоритмами библиотеки на прикладной задаче.

Прикладная задача сформулирована так. Коротышки провели психологическое тестирование «Узнай себя». Запись о результатах сохранена в файле в виде таблицы, где приведена оценка для трех качеств в виде итогового балла.

	Умный	Смелый	Добрый	Всего
Знайка	10	5	8	
Незнайка	2	10	5	
Пончик	5	3	10	
Шурупчик	8	8	7	
Пилюлькин	8	5	10	
И так далее	…	…	…	

Для обработки данных требуется:

1. Найти самого умного, самого смелого, самого доброго из коротышек.
2. Найти общий балл «Всего» для каждого коротышки, найти самого-самого по общему баллу.
3. Найти среднее значение общего балла.
4. Задать предикат для сравнения по общему баллу, и выбрать всех коротышек, для которых общий балл выше среднего.
5. При желании можно сформулировать еще задачи: сортировать по убыванию общего балла, найти самого среднего, найти самого глупого, трусливого, самого плохого из коротышек, хотя это жестоко, и так далее.

Указания.

Для сохранения данных нужно разработать сущность «Коротышка», в которой реализовать необходимые методы, в том числе метод вывода. Вывод лучше всего реализовать через функтор.

Таблицу записей организовать на основе типа vector библиотеки STL.

Визуализировать содержимое вектора, используя функтор вывода и прямой итератор. Этим приемом придется воспользоваться несколько раз.

Для поиска наилучших коротышек использовать алгоритмы `min`, `max`. Для них потребуется написать предикаты: выбор по каждой колонке таблицы записей. Для сортировки использовать метод `sort` вектора. Для сравнения объектов использовать предикат `less`, или написать новый.

Таблицу читать из файла в вектор при старте приложения, отсортированную таблицу выгрузить в файл при завершении работы.

Реализовать интерфейс для добавления и отображения записей.

Реализовать интерфейс для вывода лучших коротышек по отдельным категориям и лучшего по общему баллу.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Буч, Гради: Объектно-ориентированный анализ и проектирование с примерами приложений на C++ / Г. Буч; пер. с англ. под ред. И. Романовского, Ф. Андреева, М., СПб.: Бином: Невский диалект, 2001.

2. Подбельский, В.В.: Язык Си++: учебное пособие для вузов по направлениям «Прикладная математика» и др. / В.В. Подбельский, М.: Финансы и статистика, 2008.

3. Страуструп, Бьярн: Язык программирования C++: Специальное издание / Б. Страуструп; пер. с англ. С. Анисимова, М. Кононова, под ред. Ф. Андреева, А. Ушакова, М.: Бином-Пресс, 2008.

4. Дейтел, Х.М.: Как программировать на C / Х.М. Дейтел, П.Д. Дейтел, пер. с англ. под ред. В.В. Тимофеева, М.: Бином, 2006.

5. Шилдт, Г.: C++: базовый курс / Г. Шилдт, пер. с англ. Н.М. Ручко, М. и др.: Вильямс, 2007.

6. Павловская, Т.А.: C/C++: Программирование на языке высокого уровня: учебник для вузов по направлению «Информатика и вычислительная техника» / Т.А. Павловская, СПб. и др.: Питер, 2013.

7. Пахомов, Б.И.: C/C++ и MS Visual C++ 2008 для начинающих / Б.И. Пахомов, СПб.: БХВ-Петербург, 2009.

8. Подбельский, В.В.: Практикум по программированию на языке Си: Учебное пособие для вузов по направлениям «Прикладная математика и информатика», «Информатика и вычислительная техника» / В.В. Подбельский, М.: Финансы и статистика, 2004.

9. Павловская, Т.А.: C++: Объектно-ориентированное программирование. Практикум: учебное пособие для вузов по направлению «Информатика и вычислительная техника» / Т.А. Павловская, Ю.А. Щупак, и др, СПб.: Питер, 2008.

10. Сик, Джереми: C++ Boost Graph Library / Д. Сик, Л.К. Ли, Э. Ламсдэйн, СПб: Питер, 2006.

11. Библиотека MSDN: http://msdn.microsoft.com/ru-ru/library.

12. Техническая документация Misrosoft Visual Studio:

13. Плаугер, П.Д. : STL − стандартная библиотека шаблонов C++. / П.Д. Плаугер, А. Степанов, Менг Ли, Д. Массер. Пер. с англ. А. Никифоров, СПб: БХВ-Петербург, 2004.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 3
1. ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ (ООП) 4
1.1. Базовые понятия объектно-ориентированного программирования 5
1.2. Отличие ООП от традиционного модульного подхода 7
2. РЕАЛИЗАЦИЯ ОБЪЕКТНОЙ МОДЕЛИ В С ++. ИНКАПСУЛЯЦИЯ .. 11
2.1. Использование классов в С++ 12
Определение (спецификация) класса 12
Создание объектов класса 14
Обращение к элементам класса 14
Указатель `this` 15
Массив объектов класса 16
Динамические объекты 17
2.2. Интерфейс классов. Сокрытие информации 18
Область действия классов. Спецификация класса и реализация 18
Операция разрешения области видимости имен (::) 20
Подставляемые функции 20
Организация файлов спецификации и реализации. 22
2.3. Конструктор и деструктор 23
Перегруженные функции 23
Использование конструкторов, перегрузка конструкторов 24
Инициализация и присваивание. Конструктор копии 26
Особенности конструктора 28
Роль деструктора в уничтожении объектов класса 29
Инициализация массива объектов 30
2.4. Статические элементы класса 32
Передача параметров по ссылке 34
2.5. Перегрузка операций 34
Дружественные функции 35
Особенности дружественных функций 36
Операции над классами. Перегруженные операции 37
Определение операции-функции 37
Перегрузка бинарных операций 38
Перегрузка унарных операций. 42
Приведение типов 44
Перегрузка логических операций 45
Перегрузка операций присваивания 45
Перегрузка операций ввода и вывода в поток 46
Ограничения перегрузки операций 47
2.6. Дружественные классы 48

2.7. Внутренний класс ... 51
Итоги раздела ... 52
2.8. Вопросы для самопроверки ... 53
2.9. Упражнения ко второму разделу ... 54
3. НАСЛЕДОВАНИЕ КЛАССОВ ... 58
3.1. Реализация наследования в C++ ... 59
Определение производного класса ... 60
Способы доступа в наследуемых классах ... 60
Прямое наследование ... 61
Особенности конструктора порожденного класса ... 63
Конструктор копии при наследовании ... 65
3.2. Классификация видов наследования ... 66
Косвенное наследование классов ... 66
Множественное наследование классов ... 71
3.3. Правила наследования ... 76
Поведение конструкторов в схеме наследования ... 77
Поведение деструкторов в схеме наследования ... 77
Виртуальное наследование ... 78
3.4. Композиция. Контейнерные классы ... 79
Пример контейнерного класса ... 82
Пример контейнерного класса ... 87
Индексация и итерация. Класс итератор ... 90
Итоги раздела ... 94
3.5. Вопросы для самопроверки ... 94
3.6. Упражнения к третьему разделу ... 96
4. ПОЛИМОРФИЗМ ... 100
4.1. Виртуальные функции ... 100
Объявление виртуальной функции ... 101
Преобразование указателей ... 102
Механизм обращения к виртуальным функциям ... 103
Некоторые особенности виртуальных функций ... 106
4.2. Абстрактные классы
Абстрактные функции ... 107
Абстрактные классы ... 108
Пример коллекции графических объектов ... 109
Реализация виртуального деструктора ... 111
Хранение объектов разного типа в одной структуре ... 112
Итоги раздела ... 112
4.3. Вопросы для самопроверки ... 113
4.4. Упражнения к четвертому разделу ... 114
5. ПАРАМЕТРИЧЕСКИЙ ПОЛИМОРФИЗМ ... 116
5.1. Шаблоны функций ... 117
Определение шаблона функции ... 117

Механизм шаблона функции ... 118
5.2. Шаблоны классов ... 122
Определение шаблона класса ... 122
Механизм шаблона классов ... 122
5.3. Особенности использования шаблонов функций и классов ... 125
5.4. Пример спецификации шаблона класса «Множество» ... 127
5.5. Библиотеки шаблонов ... 128
Библиотека стандартных шаблонов STL ... 129
Boost ... 129
Библиотека Net.Framework ... 131
Библиотека OpenGl ... 133
5.6. Библиотека стандартных шаблонов STL ... 137
Структура библиотеки ... 137
Операции (Operators) ... 138
Пара (Pair) ... 138
Контейнеры ... 138
STL Строки ... 140
Методы коллекций ... 140
Итераторы ... 141
Функторы ... 143
Предикаты ... 144
Алгоритмы ... 145
Тип vector ... 151
Итоги раздела ... 154
5.7. Вопросы для самопроверки ... 155
5.8. Упражнения к пятому разделу ... 155
БИБЛИОГРАФИЧЕСКИЙ СПИСОК ... 158

Учебное издание

Конова Елена Александровна

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ
ПРОГРАММИРОВАНИЕ НА С++

Учебное пособие

Под редакцией Б.М. Суховилова

Техн. редактор *А.В. Миних*

Издательский центр Южно-Уральского государственного университета

Подписано в печать 04.07.2019. Формат 60×84 1/16. Печать цифровая.
Усл. печ. л. 9,53. Тираж 30 экз. Заказ 292/8.

Отпечатано в типографии Издательского центра ЮУрГУ.
454080, г. Челябинск, проспект Ленина, 76.