

Министерство образования и науки Российской Федерации

Федеральное агентство по образованию

Государственное образовательное учреждение высшего профессионального

образования

«Московский государственный технический университет имени Н.Э.Баумана»

Факультет «Робототехника и комплексная автоматизация»

Кафедра «Системы автоматизированного проектирования»



ЛЕКЦИИ

по курсу «Объектно-ориентированное программирование»

|  |  |
| --- | --- |
| Разработчики курса: | Волосатова Т.М., Родионов С.В. |
|  |  |

[Аннотация 5](#_Toc533367412)

[Глава 1. Введение в ООП 6](#_Toc533367413)

[1.1. Основные принципы и этапы объектно-ориентированного программирования 6](#_Toc533367414)

[1.2. Языки ООП 8](#_Toc533367415)

[1.3. О языке программирования C++ 10](#_Toc533367416)

[Глава 2. Средства ООП в C++ 11](#_Toc533367417)

[2.1. Прототипы функций 11](#_Toc533367418)

[2.2. Определение класса 12](#_Toc533367419)

[Глава 3. Классы 14](#_Toc533367420)

[3.1. Описание объектов 14](#_Toc533367421)

[3.2. Инициализация полей объектов 15](#_Toc533367422)

[3.3. Конструкторы 17](#_Toc533367423)

[3.4. Деструкторы 22](#_Toc533367424)

[3.5. Указатели на элементы класса 23](#_Toc533367425)

[3.6. Правила использования указателей на методы класса 24](#_Toc533367426)

[3.7. Неявный параметр this 25](#_Toc533367427)

[3.8. Статические элементы классов 27](#_Toc533367428)

[3.9. Статические методы 29](#_Toc533367429)

[3.10. Рекомендации по составу класса 30](#_Toc533367430)

[Глава 4. Дружественные классы и функции 32](#_Toc533367431)

[4.1. Дружественные функции 32](#_Toc533367432)

[4.2. Дружественный класс 34](#_Toc533367433)

[Глава 5. Локальные и вложенные классы 36](#_Toc533367434)

[5.1. Локальный класс 36](#_Toc533367435)

[5.2. Вложенный класс. 37](#_Toc533367436)

[Глава 6. Наследование 39](#_Toc533367437)

[6.1. Наследование 39](#_Toc533367438)

[6.2. Правила наследования различных методов 43](#_Toc533367439)

[6.3. Множественное наследование 44](#_Toc533367440)

[6.4. Виртуальное наследование 48](#_Toc533367441)

[Глава 7. Полиморфизм 50](#_Toc533367442)

[7.1. Полиморфизм 50](#_Toc533367443)

[7.2. Виртуальные методы 52](#_Toc533367444)

[7.3. Абстрактные классы 56](#_Toc533367445)

[Глава 8. Параметризованные функции и классы 58](#_Toc533367446)

[8.1. Шаблоны функций. 58](#_Toc533367447)

[8.2. Шаблоны классов 60](#_Toc533367448)

[8.3. Контейнерные классы 63](#_Toc533367449)

[Глава 9. Динамическое выделение памяти 70](#_Toc533367450)

[9.1. Динамическое выделение памяти 70](#_Toc533367451)

[Глава 10. Перегрузка операций 72](#_Toc533367452)

[10.1. Перегрузка операций 72](#_Toc533367453)

[10.2. Перегрузка унарных операций 74](#_Toc533367454)

[10.3. Перегрузка бинарных операций 76](#_Toc533367455)

[10.4. Перегрузка операции присваивания 77](#_Toc533367456)

[10.5. Перегрузка операций new и delete 78](#_Toc533367457)

[10.6. Перегрузка операций приведения типа 81](#_Toc533367458)

[10.7. Правила выбора реализации перегружаемых функций в C++ 82](#_Toc533367459)

[10.8. Переопределение операций ввода-вывода 83](#_Toc533367460)

[10.9. Перегрузка операций вызова функций 85](#_Toc533367461)

[10.10. Перегрузка операций индексирования 86](#_Toc533367462)

[Глава 11. Структуры данных 88](#_Toc533367463)

[11.1. Стек 88](#_Toc533367464)

[11.2. Очереди 90](#_Toc533367465)

[11.3. Деревья 92](#_Toc533367466)

[11.4. Определение линейного списка. Программирование связных списков в C++ 102](#_Toc533367467)

[11.5. Линейные связные списки. Двусвязные списки 107](#_Toc533367468)

[11.6. Хранение целых списков 110](#_Toc533367469)

[Глава 12. Классы ввода-вывода 111](#_Toc533367470)

[12.1. Классы стандартной библиотеки 111](#_Toc533367471)

[12.2. Потоковые классы. 112](#_Toc533367472)

[12.3. Форматирование данных 119](#_Toc533367473)

[12.4. Манипуляторы 122](#_Toc533367474)

[12.5. Методы обмена с потоками 124](#_Toc533367475)

[12.6. Классы с самоадресацией 128](#_Toc533367476)

[Глава 13. Взаимосвязь и различия C и C++ 129](#_Toc533367477)

[13.1. Области видимости и время жизни переменных 129](#_Toc533367478)

[13.2. Оператор разрешения области видимости (::). Пространство имён 130](#_Toc533367479)

[13.3. Булевы типы данных 132](#_Toc533367480)

[13.4. Модификатор const 134](#_Toc533367481)

[13.5. Константы char 135](#_Toc533367482)

[13.6. Комплексные типы данных 136](#_Toc533367483)

[13.7. Указатель на void 137](#_Toc533367484)

[13.8. Альтернативное правописание 137](#_Toc533367485)

[13.9. Поддержка расширенных символов 137](#_Toc533367486)

[Глава 14. Исключения и их обработка 138](#_Toc533367487)

[Приложение 1. 141](#_Toc533367488)

[1.1. Построение магического квадрата 141](#_Toc533367489)

[1.2. Поиск числовых палиндромов 146](#_Toc533367490)

[Приложение 2. Лабораторные работы 147](#_Toc533367491)

[Разработка программы-калькулятора дробей 147](#_Toc533367492)

[Разработка программы выполнения теоретико-множественных операций 153](#_Toc533367493)

[Разработка программы игры 157](#_Toc533367494)

[Пример программной реализации игры 160](#_Toc533367495)

[Варианты лабораторной работы по теме «Наследование» 162](#_Toc533367496)

[Приложение 3. Контрольные работы 170](#_Toc533367497)

[Виртуальные методы 170](#_Toc533367498)

[Перегрузка операторов 185](#_Toc533367499)

[Варианты контрольной работы 195](#_Toc533367500)

[Приложение 4. Вопросы к экзамену 202](#_Toc533367501)

# Аннотация

В учебном курсе рассматриваются основные конструкции языка С++ и принципы объектно-ориентированного программирования (ООП). Рассмотрен основной принцип ООП - абстрагирование и способ его реализации с помощью классов. Определен механизм наследования и его реализация. Рассмотрены механизмы полиморфизма - статического и динамического. Определены основные средства языка и стандартной библиотеки С++.

Учебный курс разработан для студентов вузов, изучающих ООП на языке С++ и выполняющих практические работы по программированию в различных операционных средах.

Разработчики курса**: к.т.н. доцент Волосатова Т.М., старший преподаватель Родионов С.В.**

# Глава 1. Введение в ООП

## Основные принципы и этапы объектно-ориентированного программирования

В теории программирования объектно-ориентированного программирования (ООП) определяется как технология создания сложного программного обеспечения, которая основана на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного типа (класса), а классы образуют иерархию с наследованием свойств. Взаимодействие объектов в такой системе осуществляется путем передачи сообщений между объектами.

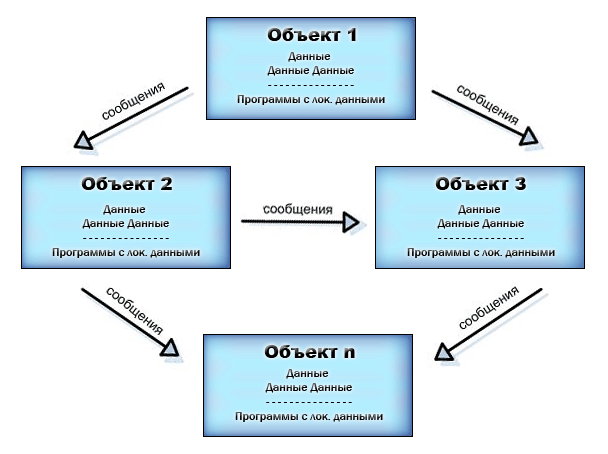


Рис. 1.

Такое представление программы было представлено в языке Simula. Принцип ООП был использован в языках Паскаль, C++.

Основное достоинство ООП – сокращение количества межмодульных вызовов и уменьшение объемов информации, передаваемой между модулями, по сравнению с модульными программами. Это достигается за счет более полной локализации данных и интегрирования их с подпрограммами обработки, что позволяет вести практически независимую разработку отдельных частей программы (объектов). Объектный подход предполагает новые средства разработки (наследование, полиморфизм, композиция, наполнение), позволяющие конструировать более сложные программные продукты.

Основной недостаток – некоторое снижение быстродействия за счет сложной организации программных систем.

В основу ООП заложены следующие принципы:

* абстрагирование
* ограничение доступа
* модульность
* иерархичность
* типизация
* параллелизм
* устойчивость

Абстрагирование четко определяет особенности данного объекта с точки зрения дальнейшего рассмотрения и анализа. Современный уровень абстракции предполагает объединение всех свойств объекта (состояние объекта, поведение) в единую программную единицу – некий абстрактный тип (в C++ – класс).

Необходимое ограничение доступа предполагает разграничение 2-х частей в описании абстракции: интерфейса и реализации.

В результате ограничения доступа в ООП позволяют разработчику:

* выполнять конструирование системы поэтапно, не отвлекаясь на особенности реализации используемых абстракций.
* легко модифицировать реализацию отдельных объектов. Если система разработана правильно, то модификация не потребует изменения всех объектов.

Принцип модульности унаследован от модульного программирования. Следование этому принципу упрощает проектирование и отладку программы.

В ООП используется два вида иерархии - общая, дробь, частная. Эта иерархия используется при разработке структуры классов, когда классы строятся на базе более простых, путем добавления к ним новых характеристик и, возможно, уточнения имеющихся.

Один из важных механизмов ООП – наследование свойств в иерархии общее/частное. Наследования – такое соотношение между абстракциями, когда одна из них использует структурную либо функциональную часть другой или нескольких других абстракций (простое и множественное наследование).

Использование принципа типизации обеспечивает раннее обнаружение ошибок, которые связаны с недопустимыми операциями над программными объектами.

* упрощение документации
* возможность генерации более эффективного кода программы. Тип может связываться с программным объектом
* статически (раннее связывание, тип объекта определен при компиляции)
* динамически (позднее связывание. Тип определяется во время выполнения программы.)

Реализация позднего связывания в языке программирования позволяет создавать переменные – указатели на объекты, принадлежащие разным классам (так называемые полиморфные объекты).

## Языки ООП

Объектно-ориентированное программирование включает в себя три основных понятия: абстрактные типы данных, наследование, полиморфизм. Языки ООП (объектно-ориентированные языки) поддерживаются этой парадигмой с помощью классов, методов, объектов, передаваемых сообщений.

Основные вопросы при обсуждении объектно-ориентированных языков:

* Исключительность объектов.
* Подклассы и подтипы.
* Реализация интерфейсного наследования.
* Проверка типов.
* Одиночное и множественное наследование.
* Полиморфизм (статическое и динамическое связывание).
* Удаление объектов из памяти.

Smalltalk – чисто объектно-ориентированный язык. Все сущности в этом языке являются объектами, все вычисления производятся через передачу сообщений. Методы конструируются из выражений, выражение описывает объект, который представляет собой значение выражения. Управляющие структуры конструируются с использованием объектов и сообщения. В этом языке все подклассы являются подтипам. Все проверки типов и связывание сообщений с методами выполняется динамически, и любое наследование является одиночным. Язык не имеет явного оператора удаления объектов из памяти.

C++ обеспечивает поддержку абстракции данных, наследование, возможное динамическое связывание сообщений с объектами наряду с остальными свойствами языка C.

C++ имеет две различных системы типов, также обеспечивается множественное наследование, явное удаление объектов из памяти, разнообразные средства управления доступа к сущностям в классе, некоторые из которых позволяют подклассам быть подтипами. Класс может содержать неявные методы конструктора и деструктора.

Язык Java, в отличие от C++, является полностью объектно-ориентированным. Все объекты размещаются в динамической памяти и доступны через ссылки на них. В языке нет явной операции удаления объектов из памяти. Методами являются только подпрограммы, и вызывать их можно только через объекты или классы. Непосредственно поддерживается одиночное наследование, возможна реализация множественного наследования.

Все связывания сообщений с методами динамические, за исключением методов, которые не могут быть замещаемыми. Кроме классов в качестве второй конструкции инкапсуляции Java содержит пакеты.

ADA 95 обеспечивает поддержку ООП с помощью меченых типов, которые могут использовать наследование, возможное динамическое связывание с использование ассоциированных типов. Производные типы являются расширением родительских типов. В случае, если они не определены в библиотеке дочерних пакетов, сущности родительского типа могут быть исключениями из производного типа. В небиблиотечных дочерних пакетах все подклассы являются подтипами.

Eiffel – объектно-ориентированный язык, который не основан ни на одном из существующих языков. Содержит средства управления доступом к сущностям в классах, средства ограничения доступа к классам клиентов, подклассов, их обоих. В нем нет явной операции удаления из памяти. Пользователь может определять конструкторы и деструкторы в своих классах. Все связывания сообщений с методами являются динамическими.

Golang (также известен как Go) – новый объектно-ориентированный язык программирования без поддержки наследования реализации (то есть, наследуется не сам объект, а только его интерфейсы). Создан в 2009 году. Интересен тем, что в нём реализована многопоточность, которая предусматривается ещё на этапе компиляции (то есть, программа разделяется на несколько потоков ещё в начале запуска, а не по ходу её выполнения). Позиционируется как замена C/C++ для разработки высоконагруженных решений, исполняющихся на современных многопроцессорных и распределённых системах.

Python – объектно-ориентированный язык программирования высокого уровня, полностью реализовывающий парадигму ООП, а также позволяющий писать, используя функциональное программирование. Разрабатывающийся ещё с 1980-х годов, Python стал очень популярным в последнее время из-за активного развития, а также из-за простоты разработки и отладки приложений на нём. В настоящее время Python активно используется в таких областях, как разработка нейронных сетей, разработка систем компьютерного зрения и др.

## О языке программирования C++

Разработчиком языка C++ является Бьёрн Страуструп (Bjarne Stroustrup). В своей работе он опирался на опыт создателей языков Simula, Modula-2, абстрактных типов данных. Основные работы велись в исследовательском центре компании Bell Labs. Непосредственный предшественник С++ – язык C с классами – появился в 1979 году. Если говорить коротко, то С++ является объектно-ориентированной версией языка С. С++ построен на основе версии С89, включающей все усовершенствования 1995 года, и теперь версию С89 называют С-подмножеством языка С++. В 1997 году был принят международный ANSI/ISO-стандарт С++, а затем в 2011 году был принят обновлённый стандарт C++ 2011, добавивший в язык частичную поддержку парадигмы функционального программирования. Актуальная на сегодняшний день версия – C++ 2017.

Принятие стандарта обеспечило единообразие всех реализаций языка С++. Не менее важным результатом стандартизации стало то, что в процессе выработки и утверждения стандарта язык был уточнен и дополнен рядом существенных возможностей. На сегодня стандарт утвержден Международной организацией по стандартизации ISO. Его номер ISO/IEC 14882 (актуальная версия имеет номер ISO/IEC 14882:2017). ISO бесплатно стандарты не распространяет. Его можно получить на сайте Международного комитета по стандартам информационных технологий ([INCITS](http://www.incits.org/)). В России следует обращаться в [ВНИИ Сертификации](https://vniis.ru/). Проекты стандарта [имеются в свободном доступе](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4778.pdf).

Несмотря на то, что язык С++ был задуман как набор объектно-ориентированных расширений языка С, вскоре он развился в самостоятельный язык программирования. В настоящее время его новые средства почти удвоили объем исходного языка. На данный момент С++ - один из самых мощных компьютерных языков. Язык С++ является универсальным языком программирования, в дополнение к которому разработан набор разнообразных библиотек. Поэтому, строго говоря, он позволяет решить практически любую задачу программирования. Тем не менее, в силу разных причин (не всегда технических) для каких-то типов задач он употребляется чаще, а для каких-то – реже. С++ как преемник языка С широко используется в системном программировании. На нем можно писать высокоэффективные программы, в том числе операционные системы, драйверы и т.п. Язык С++ – один из основных языков разработки трансляторов. Поскольку системное программное обеспечение часто бывает написано на языке С или С++, то и программные интерфейсы к подсистемам ОС тоже часто пишут на С++. Соответственно, те программы, даже и прикладные, которые взаимодействуют с операционными системами, написаны на языке С++. Распределенные системы, функционирующие на разных компьютерах, также разрабатываются на языке С++. Обработка сложных структур данных – текста, бизнес-информации, Internet-страниц и т.п. – одна из наиболее распространенных возможностей применения языка. В прикладном программировании, наверное, проще назвать те области, где язык С++ применяется мало. Разработка графического пользовательского интерфейса на языке С++ выполняется, в основном, тогда, когда необходимо разрабатывать сложные, нестандартные интерфейсы. Однако для разработки интерфейсов лучше пользоваться средствами графических библиотек операционной системы, таких как WPF, DirectX, OpenGL, X11, GTK+ 3 и другими). Простые программы чаще пишутся на таких языках, как Python. Веб-программирование в основном производится на языках Java (редко, но встречается), JavaScript, PHP.

В целом необходимо отметить, что язык С++ в настоящее время является одним из наиболее распространенных языков программирования в мире.

# Глава 2. Средства ООП в C++

## 2.1. Прототипы функций

В C++ все функции должны иметь прототипы, а в языке C прототипы формально необязательны, но весьма желательны. Общая форма прототипа такова:

<тип\_результата> <имя\_функции>(<список\_параметров>);

По сути, прототип – это просто объявление типа возвращаемого значения, имени и списка параметров функции, завершающееся точкой с запятой.

*Пример 1*

**// Пример прототипа функции fn()**

**float fn(float x); // прототип**

**{**

**// Определение функции**

**}**

Чтобы задать прототип функции, которая принимает переменное количество аргументов, следует поставить три точки в том месте, с которого начинается список, состоящий из переменного числа параметров

*Пример 2*

**int printf(const char \*format, ...);**

Компилятор C++ позволяет объявлять более одной функции под одним и тем же именем, если у них различные списки аргументов.

При задании прототипа для перегруженной функции каждая версия этой функции должна иметь собственный прототип. При объявлении функции-члена внутри своего класса используется прототип этой функции.

## 2.2. Определение класса

В C++ класс - это абстрактный тип данных, в котором объединяются все свойства объектов. Так же, как и в других языках программирования, формат описания класса имеет вид:

*Пример 1*

**сlass имя\_класса**

**{**

**[private:]**

**//описание скрытых компонентов класса**

**[protected:]**

**//описание защищенных компонентов класса**

**public:**

**//описание доступных компонентов класса**

**};**

В качестве компонентов класса фигурируют поля. Внутренние компоненты класса (private) доступны только компонентам функций того же класса и функциям, объявленным «дружественными» описываемому классу.

Компоненты класса, объявленные как защищенные (protected), называются защищенными. Они доступны функциям не только данного класса, но и его потомкам. При отсутствии наследования интерпретируются как внутренние.

Компоненты, объявленные в public называются общими. Они доступны за пределами данного класса в любом месте программы. Именно в этой секции осуществляются объявления полей и методов интерфейсной части класса. Если при описании секции класса тип доступа к компонент не указан, то по умолчанию присваивается тип private. Поля класса всегда описываются внутри класса. Методы могут быть описаны как внутри определения класса, так и вне него. В последнем случае определение класса должно содержать прототип этих функций. А заголовки функций должны содержать описатель видимости “::” (::private). Таким образом компилятору сообщается, что определенные функции доступны внутри поля класса.

*Формат описания функций:*

<тип функции> <имя класса>::<имя функции>(параметры)

{

<тело функции>

}

По правилам C++, если тело компонентной функции размещено в описании класса, то она по умолчанию считается встраиваемой функцией (inline). Для таких функций компилятор помещает машинные коды, реализующие эту функцию, непосредственно вместо вызова, что значительно ускоряет работу. Встраиваемые компилятором функции можно описать и вне определения класса, добавив к заголовку функции служебное слово (спецификатор функции) inline. При определении компонентов функции следует иметь в виду, что внутри класса запрещено определять следующие типы функций:

* рекурсивные функции;
* виртуальные функции.

Тела таких функций размещаются вне пределов класса.

*Пример 2*

*Пример класса, моделирующего персонажа компьютерной игры:*

**class monstr**

**{**

**int health, ammo;**

**public:**

**monstr(int he = 100, int am = 10)**

**{**

**health = he;**

**ammo = am;**

**}**

**void draw(int x, int y, int sale, int position);**

**int get\_health() {return health;}**

**int get\_ammo() {return ammo;}**

**};**

Определение класса может размещаться перед текстом программы, а может помещаться в отдельный файл, который подключается к программе при описании объектов этого класса с помощью директивы компилятора **#include**. Если файл находится в каталогах автоматического поиска, то **#include<имя файла>.**

# Глава 3. Классы

## 3.1. Описание объектов

В программе, использующей определенные ранее классы, по мере необходимости объявляются объекты классов. Такое объявления выглядит следующим образом:

**<имя класса> <список объектов или указателей>**

Как только объект класса определен, обращение к полям и функциям объекта может осуществляться с помощью полных имен, каждое из которых имеет следующий вид:

**<имя объекта>.<имя класса>::<имя поля или функции>**

Чаще доступ к компонентам объекта обеспечивается с помощью укороченного имени, в котором имя класса и двоеточие опускаются. В этом случае доступ к полям и методам осуществляется по аналогии с обращением к полям структур:

**<имя объекта>.<имя поля или функции>**

**<имя указателя на объект> -> <имя поля или функции>**

**<имя объекта>[индекс].<имя поля или функции>**

*Пример 1*

*Примеры объявления объектов класса:*

**monstr Vasia; /\* Объект класса monstr с параметрами по умолчанию \*/**

**monstr Super(200, 300); /\* Объект с явной инициализацией \*/**

**monstr stado[100]; /\* Массив объектов класса monstr с параметрами по умолчанию \*/**

**monstr \*beavis = new monstr(10); /\* Динамический объект, второй параметр по умолчанию \*/**

**monstr &butthead = Vasia; /\* Ссылка на объект \*/**

Так как объекты некоторого класса являются обычными переменными, то на них распространяется общие правила длительности существования (времени жизни) и области действия переменных. При создании и удалении объектов определенного класса соблюдаются следующие правила:

* Глобальные и локальные статические объекты создаются до вызова функции main и уничтожаются по завершении работы программы.
* Автоматические объекты создаются каждый раз при их объявлении и уничтожаются при выходе из функции, в которой они появились.
* Объекты, память под которые выделяется динамически создаются функцией new и уничтожаются функцией delete.

## 3.2. Инициализация полей объектов

При объявлении полей класса не допускается их инициализация, поскольку в момент описания поля память для его размещения еще не выделена. Выделение памяти осуществляется не для класса, а для объектов этого класса. Поэтому возможность инициализации полей появляется после объявления данного класса.

Значение может заноситься в поле объекта во время выполнения программы несколькими способами:

* присваиванием значения полю объекта
* внутри любой компонентной функции используемого класса
* согласно общим правилам C++ с использованием оператора инициализации.

Эти правила применимы только для инициализации общедоступных полей, описанных в секции public. Инициализация полей, описанных в секциях private и protected возможно только с помощью компонентной функции.

*Пример 1*

*Пример метода, определенного в другом месте программы с помощью операции доступа к области видимости (::)*

**void monstr::draw(int x, int y, int scale, int position)**

**{**

**<тело метода>**

**}**

*Пример 2*

*Пример описания метода, определенного как встроенный вне класса с пометкой директивы inline.*

**inline int monstr::get\_ammo()**

**{**

**return ammo;**

**}**

*Пример 3*

*Примеры доступа к элементам объекта аналогично доступу к полям структуры.*

**. – для обращение через имя объекта**

**-> – через указатель**

**int n=Vasia.get\_ammo();**

**stdio[5].draw(10,10,1,1);**

**cout<<beavis-> get\_health();**

В случае, когда обращаются к const объекту, к нему применяется только const методы.

Объект типа const (константный объект)- объект, значение полей которого изменять запрещено.

*Пример 4*

**class monstr**

**{…**

**int get\_health() const {return health;}**

**};**

**const monstr Deaf(0,0); /\* константный объект \*/**

**cout<<Dead.get\_health();**

## 3.3. Конструкторы

В C++ предназначены специальные методы для инициализации объектов - конструкторы. Они выполняются автоматически при создании объекта. Конструктор имеет то же самое имя, что и класс и обладает следующими свойствами:

* конструктор не возвращает значения, даже типа void; нельзя получить указатель на конструктор
* класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации (при этом используется перегрузка)
* конструктор может не иметь параметров, тогда это конструктор по умолчанию
* параметры конструктора могут иметь любой тип, кроме типа этого же класса; можно задавать значения по умолчанию, но их может содержать только один конструктор
* если не указано ни одного конструктора, будет вызван конструктор по умолчанию. Такой конструктор вызывает конструкторы по умолчанию для полей класса и конструкторы по умолчанию базовых классов. В случае, когда класс содержит константы или цифры, при попытке создания объекта класса будет выдана ошибка, так как их необходимо инициализировать конкретными значениями, а конструктор по умолчанию этого делать не умеет
* конструкторы не наследуются
* конструкторы нельзя записать с модификаторами: const, virtual, static
* конструкторы глобальных объектов вызываются до вызова функции main

В иерархии классов конструкторы выполняются в порядке их "классового происхождения".

Локальные объекты создаются, как только становятся активными области памяти конструктор записывается при создании временного объекта, например при передаче объекта из функции.

Как и любая другая компилируемая функция, конструктор может быть переопределенным, указыв. конкретные данные для различных списков параметров. Этот способ, как правило, применяется для инициализации полей объектов. В последнем случае будет вызван конструктор, соответствующий указанным в скобках параметрам. Конструктор не может сообщать значение, чтобы сообщить об ошибке во время инициализации.

**Конструктор копирования**

Конструктор копирования - специальный вид конструктора, получающий в качестве единственного параметра указатель на объект того же класса.

**T::T(const T&)**

**{**

**/\* Тело конструктора \*/**

**}**

Конструктор вызывается, когда новый объект создается путём копирования существующего:

* при передаче нового объекта с инициализацией другим объектом
* при передаче объекта в функцию по значению
* при возврате объекта из функции.

Конструктор копирования выполняет поэлементное копирование полей. В случае, когда в классе есть указатели или ссылки, использование конструктора копирования будет неправильным.

*Пример 1*

**monst::monstr( const monstr&M )**

**{**

**if( M.name )**

**{**

**name = new char[strlen(M.name)+1];**

**strcpy(name, M.name);**

**}**

**else name = 0;**

**health = M.health;**

**ammo = M.ammo;**

**skin = M.skin;**

**}**

В конструкторах может использоваться список инициализации, который отделяется от заголовка конструктора символом «:» и состоит из записей вида: <имя>(<список выражений>), где в качестве имени может быть использовано:

имя поля данного класса

имя объекта другого класса, включенного в данный класс

имя базового класса

Список выражений определяет значения, использованные для инициализации объекта. Чаще всего конструктор со списком инициализации применяется для задания назначенных значений объекта, в котором используются фиксированные и ссылочные поля, являющиеся объектами других, ранее определенных классов.

Иногда возникает необходимость создать объект, не инициализируя его поля. Такой объект называется неинициализированным и под него только резервируется память. Для создания такого объекта используется неинициализирующий конструктор. У такого конструктора нет списка параметров и отсутствует тело. В этом случае значения полей объекта не определяются.

Пример 2

**/\* Создается объект без имени и копируется \*/**

**monst Super(200, 300), Vasia(50), z;**

**monst x=monstr(1000);**

**monst y=500;**

**lnum color{red, green, blur} /\* возможные значения цветов \*/**

**class monstr**

**{**

**int health, ammo;**

**color skin;**

**char \*name;**

**public:**

**monstr(in the = 100, int am = 10);**

**monstr( color 5k);**

**monstr( char \* name);**

**int get\_health() {return health;}**

**int get\_ammo() {return ammo;}**

**};**

**monstr::monst(int he, int am)**

**{**

**health = he;**

**ammo = am;**

**skin = red;**

**name = ?;**

**}**

**monstr::monst(color sk)**

**{**

**switch(sk)**

**{**

**case red:**

**health = 100;**

**ammo = 10;**

**skin = red;**

**name = 0;**

**break;**

**case green:**

**health = 100;**

**ammo = 20;**

**skin = green;**

**name = 0;**

**break;**

**case blue:**

**health = 100;**

**ammo = 40;**

**skin = blue;**

**name = 0;**

**break;**

**}**

**}**

**monstr::monstr(char \* name)**

**{**

**nam = new char[strlen(name) + 1];**

**strcpy(name, nam);**

**health = 100;**

**ammo = 10;**

**skin = red;**

**}**

**monstr \*m = new monstr (“Ork”);**

**monstr Green(green);**

Существует еще один способ инициализации полей в конструкторе - инициализация с помощью списка инициализации, расположенного после «:» между заголовком тела и конструктором.

**monstr::monst(in the, int am):health(he), ammo(am), skin(red), name(0);**

Поля перечисляются через запятую. Для каждого поля в скобках указывается инициализирующее значение, которое может быть выражением.

**Неконвертируемый конструктор**

Неконвертируемый конструктор создается при помощи спецификатора функций explicit, который применяется только к конструкторам. Обычно, если имеется конструктор, принимающий только один аргумент, то для инициализации объекта ob можно использовать варианты: ob(x), либо ob = x (т.к. при создании такого конструктора мы неявно указываем преобразование из типа аргумента в тип данного класса). Если же конструктор определен с помощью explicit, то он будет задействован только в случае, когда для инициализации используется обычный синтаксис конструктора, т.е. ob(x). При этом автоматических преобразований выполняться не будет, и вариант ob = x не будет разрешен.

## 3.4. Деструкторы

Деструктор - особый вид метода, применяемого для освобождения памяти, занимаемой объектом. Имя деструктора совпадает с именем класса, но перед ним ставится символ «~» (тильда). Деструктор определяет операции, которые необходимо выполнить при уничтожении объекта. Деструктор вызывается автоматически, когда объект выходит из области видимости для локальных объектов - при выходе из блока, в котором они были объявлены; для глобальных – как часть процедуры выхода из main. Для объектов, заданных через указатели, деструктор выполняется неявно при использовании операции delete.

Деструктор:

* не имеет аргументов и возвращающего значения
* не может быть объявлен как const или static (но может быть virtual)
* не наследуется

В иерархии классов деструкторы, в отличие от конструкторов (которые выполняются в порядке своего "классового происхождения"), выполняются в обратном порядке.

Если деструктор не определен явным образом, то компилятор создает автоматически пустой деструктор. Явным образом деструктор требуется описывать в случае, когда объект создает указатели на память, выделяемую динамически. Иначе при уничтожении объекта память, на которую ссылались его указатели, не будет помечена как свободная. Указатель на деструктор определить нельзя.

Деструктор можно вызывать явным образом путем указания полностью уточненного имени.

Деструкторы, вызываемые неявно. Если программа завершается с использованием exit, то вызывается деструктор только глобальных объектов. При завершении программы, в которой использовались объекты некоторого класса функции abort, деструктор не вызывается.

*Пример 1*

**class String {**

**public:**

**// набор конструкторов**

**String();**

**String( const char\* );**

**String( const String& );**

**// деструктор**

**~String();**

**private:**

**...**

**}**

## 3.5. Указатели на элементы класса

К элементам класса можно обращаться с помощью указателей. Для этого определены следующие операции: .\* и –> . Указатели на поля и методы класса определены по-разному.

Формат указателя на метод класса: **<возвращаемый\_тип> (<имя\_класса>::\*<имя\_указателя>)(<параметр>)**

*Пример 1*

**int get\_health() {return health;}**

**int get\_ammo() {return ammo;}**

**int (monstr::\*pget)();**

Если указатель задавать в качестве параметра функции, это дает возможность передавать в функцию имя метода.

*Пример 2*

**void fun(int (monstr::\*pget)())**

**{**

**(\*this.\*pget)(); /\* \*/**

**(this -> pget)(); /\* \*/**

**}**

Можно создать указатель на конкретный метод с помощью операции взятия адреса:

*Пример 3*

**/\* присваивание значения указателю\*/**

**pget = &monstr::get\_health;**

**monstr Vasia.\*p;**

**p = new monstr;**

**/\* вызов функции через операцию .\* \*/**

**int Vasia\_health=(Vasia.\*pget)();**

**/\* вызов функции через операцию ->\* \*/**

**int p\_health=(p->\*pget)();**

## 3.6. Правила использования указателей на методы класса

* Указателю на метод можно присваивать только адреса методов, имеющих соответствующий заголовок.
* Нельзя определить указатель на статический метод класса.
* Нельзя преобразовать указатель на метод в указатель на обычную функцию, не являющуюся элементом класса.

Как и указатели на обычные функции, указатели на методы класса используются в том случае, когда возникает необходимость вызывать метод, имя которого неизвестно. Однако существуют различия между указателем на метод и на функцию (имя переменной). Указатель на метод не ссылается на определенный адрес памяти. Указатель на метод имеет сходство с индексом в массиве, так как задает смещение. Конкретный адрес в памяти получается путем сочетания указателя на метод класса с указателем на определенный объект.

Формат указателя на поле класса: **<тип\_данных>(<имя\_класса>::\*<имя\_указателя>);**

В определение указателя можно включить его инициализацию в такой форме: **&имя\_класса::имя\_поля;** При это поле должно быть в области **public**.

## 3.7. Неявный параметр this

Когда компонентная функция вызывается для конкретного объекта, то этой функции автоматически и неявно передается в качестве параметра указатель на тот объект, для которого она вызывалась. Этот указатель имеет специальное имя и определен как константный в каждой функции класса.

**<имя класса>\*const this = <адрес объекта>**

В соответствии с описанием этот указатель менять нельзя, однако в каждой принадлежащей классу функции он указывает на тот объект, для которого данная функция вызывается. То есть указатель **this** является дополнительным (скрытым) параметром каждой нестатической компонентной функции. Этот параметр используется для доступа к полям конкретного объекта. Для иллюстрации использования указателя **this** добавим в приведенный класс **monstr** новый метод, возвращающий ссылку на более здорового из двух монстров, один из которых вызывает метод, а другой передается ему в качестве параметра. Метод необходимо поместить в секцию **public**

*Пример 1*

**monstr &the\_best(monstr &M)**

**{**

**if(health > M.get\_health())**

**return \*this;**

**return M;**

**}**

**monstr Vasia(50), Super(200);**

**/\* Новый объект Best инициализируется значениями полей Super \*/**

**void cure(int health, int ammo)**

**{**

**this-> health+=health; /\* использование this \*/**

**monstr::ammo+=ammo; /\* использование операции :: \*/**

**}**

Использование параметра **this** не дает ощутимого преимущества при программировании, так как данные конкретного объекта уже доступны в принадлежащих классу функциях по именам. Удобным и необходимым он становится, когда в теле принадлежащей классу функции требуется явно задать адрес, для которого она была вызвана.

Ниже приведён пример кода, в котором использование this делает код более читаемым и удобным для написания.

*Пример 2*

**class MyClass{**

**void foo(){**

**MyClass \*ptr = new MyClass(\*this);**

**//Тут некоторый код**

**delete ptr;**

**}**

**};**

**int main(){**

**MyClass obj;**

**}**

Без указателя this этот код выглядел бы примерно так:

*Пример 3*

**class MyClass{**

**void foo(MyClass\* obj){**

**MyClass \*ptr = new MyClass(\*obj);**

**//Тут некоторый код**

**delete ptr;**

**}**

**public:**

**void init(MyClass\* obj){**

**foo(obj);**

**}**

**};**

**int main(){**

**MyClass obj;**

**obj.init(&obj);**

**}**

## 3.8. Статические элементы классов

Статическими компонентами называются компоненты класса, которые объявлены с модификатором static. Статические компоненты класса являются частью класса, но не включаются в объекты этого класса. Имеется ровно одна копия статических полей класса. Она является общей для всех объектов данного класса. Статические поля никогда не дублируются.

Особенности:

память под статические поля выделяется один раз при их инициализации. Независимо от числа созданных объектов (и даже при их отсутствии) инициализируются статические поля с помощью операции доступа к области действий, а не с помощью оператора выбора (т.е. определение статического поля должно быть описано вне функции).

*Пример 1*

**class A**

**{**

**public:**

**static int count; /\* Объявление в классе \*/**

**};**

**int A::count; /\* Объявление в глобальной области. По умолчанию инициализируется нулем \*/**

**int A::count = 10; /\* Пример инициализации произвольным значением \*/**

Статические поля доступны как через имя класса, так и через имя объекта.

*Пример 2*

**A\*a,b;**

**cout<<A::count<<a->count<<b.count;**

На статические поля распространяется действие спецификаторов доступа, следовательно описанные как private статические поля нельзя изменять с помощью операций доступа к области действия. Это можно сделать только с помощью статических методов. Память, занимаемая статическим полем, не учитывается при определении размера объекта с помощью операции sizeof. Принципиально любой метод класса может обратиться к статическому полю и изменить его значение. Но существует возможность обращения к статическому полю при отсутствии объектов данного класса. Такой доступ осуществляется с помощью статических компонентных функций, которые объявляются со спецификатором static. Статическая функция не ассоциируется с каким-либо объектом и не получает параметра this. Она не может без указателя объекта обращаться к нестатическим полям класса. При необходимости ссылка на конкретный объект может быть передана в списка параметров и тогда статическая функция может обратиться к нестатическим полям следующим образом:

**<имя объекта>.<имя нестатического поля класса>**

При обращении к статическому компоненту класса, являющемуся принадлежностью всех объектов данного класса, можно вместо имени объекта указать имя класса.

**<класс>::<компонент>**

## 3.9. Статические методы

Предназначены для обращения к статическим полям класса. Обращение к статическим методам производится так же, как и к статическим полям, а именно:

а) через имя класса,

б) через имя объекта, если он уже создан.

*Пример 1*

**class A**

**{**

**static int count; /\* поле count - скрытое \*/**

**public:**

**static void int\_count() {count++}**

**};**

**A::int count; /\* определение в глобальной области\*/**

**void f()**

**{**

**A.a;**

**/\* a.count++ - нельзя, так как поле count скрытое.**

**Изменение поля count выглядит следующим образом: \*/**

**a.inc\_count();**

**/\*или :\*/**

**A::inc\_count();**

**}**

*Пример 2*

**class point**

**{**

**int x, y, color; /\* нестатическое поле \*/**

**static int obj\_count; /\* Статическое поле - счетчик обращений \*/**

**public:**

**point();**

**static void draw\_point(point &p); /\* статическая функция \*/**

**};**

**int point::obj\_count = 0; /\* инициализация статического поля \*/**

**void point::draw\_point(point &p) /\* имя объекта передано в списке параметров \*/**

**{**

**putpixel(p.x, p.y, p.color);**

**obj\_count++; /\* обращение к статическому и нестатическому полю \*/**

**}**

## 3.10. Рекомендации по составу класса

Класс, как тип, определяемый пользователем, должен содержать скрытые поля (private) и следующие функции:

* конструкторы, определенные как инициализирующиеся объекты класса;
* методы, реализующие свойства класса (методы, возвращающие значения скрытых полей класса описываются с модификатором const, который указывает, что они не должны изменять значения этих полей);
* операции, позволяющие копировать, присваивать, сравнивать объекты и производить над ними другие действия, требуемые по сути класса;
* необходим еще функциональный класс исключений, используемый для сообщений об ошибках, с помощью генерации исключительных ситуаций.

Могут также использоваться функции, которые работают с классом или несколькими классами через интерфейс (т.е. доступ к скрытым полям им не требуется). Эти функции можно описать вне классов, чтобы не перегружать интерфейс. Чтобы обеспечить логическую связь, можно поместить их в общее с этим классом пространство имен.

*Пример 1*

**namespace staff**

**{**

**class monstr{...};**

**class hero{...};**

**void interact(hero, monstr);**

**}**

*Пример 2*

**class String {**

**public:**

**// набор конструкторов**

**// для автоматической инициализации**

**// String strl; // String()**

**// String str2( "literal" ); // String( const char\* );**

**// String str3( str2 ); // String( const String& );**

**String();**

**String( const char\* );**

**String( const String& );**

**// деструктор**

**~String();**

**// операторы присваивания**

**// strl = str2**

**// str3 = "a string literal"**

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

**String& operator=( const char\* );**

**// операторы проверки на равенство**

**// strl == str2;**

**// str3 == "a string literal";**

**bool operator==( const String& );**

**bool operator==( const char\* );**

**// перегрузка оператора доступа по индексу**

**// strl[ 0 ] = str2[ 0 ];**

**char& operator[]( int );**

**// доступ к членам класса**

**int size() { return \_size; }**

**char\* c\_str() { return \_string; }**

**private:**

**int \_size;**

**char \*\_string;**

**}**

# Глава 4. Дружественные классы и функции

## 4.1. Дружественные функции

Применяются для доступа к скрытым полям класса и представляют собой альтернативу метода, т.к. метод как правило используется для реализации свойств объекта. Дружественные функции используются, когда оформляется действие, не представляющее собой свойство класса, но входящее в интерфейс класса и нуждающееся в доступе к его скрытым полям.

Свойства дружественных функций:

* объявляются внутри класса, к элементам которого им необходим доступ. При этом используется служебное слово friend. В качестве параметра дружественной функции передается объект или ссылка на объект в классе, т.к. this ей не передается (поскольку она не является членом класса).
* она может быть обычной функцией, либо методом другого, ранее определенного класса. На дружественные функции не распространяется действие спецификаторов доступа. Место ее размещения и объявления в классе безразлично.
* функция, объявленная дружественной, может являться таковой сразу в нескольких классах.

*Пример 1*

**class monstr; /\* предварительное объявление класса \*/**

**class hero**

**{**

**public:**

**void kill(monstr &);**

**};**

**class monstr**

**{**

**friend int steal\_ammo(monstr &);**

**friend void hero::kill(monstr &);**

**/\* класс hero должен быть определен ранее \*/**

**};**

**int steal\_ammo(monstr &M)**

**{**

**return --M.ammo;**

**}**

**void hero::kill(monstr&M)**

**{**

**M.health = 0;**

**M.ammo = 0;**

**}**

Использование дружественных функций по возможности нужно избегать, так как они нарушают принцип инкапсуляции.

## 4.2. Дружественный класс

Если все методы какого-либо класса должны иметь доступ к скрытым полям другого класса, то он объявляется дружественным классом с помощью ключевого слова friend.

*Пример 1*

**class hero**

**{**

**friend class mistress;**

**};**

**class mistress**

**{**

**void f1(...);**

**void f2(...);**

**};**

Класс сам определяет, какие функции являются дружественными, а какие нет. Функции f1 и f2 – дружественные по отношению к классу hero и имеют прямой доступ ко всем полям класса.

*Пример 2*

**#include <stream.h>**

**struct employee {**

**friend class manager;**

**employee\* next;**

**char\* name;**

**short department;**

**virtual void print();**

**};**

**struct manager : employee {**

**employee\* group;**

**short level;**

**void print();**

**};**

**void employee::print()**

**{**

**cout << name << "\t" << department << "\n";**

**}**

**void manager::print()**

**{**

**employee::print();**

**cout << "\tlevel " << level << "\n";**

**}**

**void f(employee\* ll)**

**{**

**for ( ; ll; ll=ll->next) ll->print();**

**}**

**main ()**

**{**

**employee e;**

**e.name = "J. Brown";**

**e.department = 1234;**

**e.next = 0;**

**manager m;**

**m.name = "J. Smith";**

**m.department = 1234;**

**m.level = 2;**

**m.next = &e;**

**f(&m);**

**}**

# Глава 5. Локальные и вложенные классы

## 5.1. Локальный класс

Если класс объявляется внутри некоторой функции, то он называется локальным. Функция, в которой объявлен локальный класс, не имеет непосредственного доступа к компонентам локального класса. Доступ осуществляется с помощью указателя имени объектов встроенного класса.

Локальный класс не может иметь статических полей. Объекты локального класса могут быть созданы только внутри функции в области действия объявления класса, причем все компонентные функции локального класса должны быть встраиваемыми:

*Пример 1*

**int x;**

**void f()**

**{**

**static int s;**

**int x;**

**extern int g();**

**struct local {**

**int g() { return x; } // ошибка, auto x**

**int h() { return s; } // OK**

**int k() { return ::x; } // OK**

**int l() { return g(); } // OK**

**};**

**// ...**

**}**

**local\* p = 0; // ошибка: нет local в текущем контексте**

## 5.2. Вложенный класс.

Класс, объявленный внутри другого класса, называют вложенным. Вложенный класс находится в области действия того класса, внутри которого он объявлен. Соответственно, объекты вложенного класса могут использовать компоненты внешнего класса, а компонентные функции и статические компоненты вложенного класса могут быть описаны вне глобального класса.

*Пример 1*

**class Figura {**

**class Point // вложенный класс Point**

**{**

**int x, y, color;**

**public:**

**int getx() { return x; }**

**int gety() { return y; }**

**int getcolor() { return color; }**

**.......**

**};**

**class Line // вложенный класс Line**

**{**

**Point Tn, Tk; // начало и конец линии**

**public:**

**void draw(void) // метод рисования линии**

**{**

**....**

**line(Tn.getx(), Tn.gety(), Tk.getx(), Tk.gety());**

**}**

**void setline(int ncolor) {....}**

**};**

**};**

*Пример 2*

**int x;**

**int y;**

**class enclose {**

**public:**

**int x;**

**static int s;**

**class inner {**

**void f(int i)**

**{**

**x = i; // ошибка: присваивание enclose::x**

**s = i; // нормально: присваивание enclose ::s**

**::x = i; // нормально: присваивание глобальному x**

**y = i; // нормально: присваивание глобальному y**

**}**

**void g(enclose\* p, int i)**

**{**

**p->x = i; // нормально: присваивание enclose ::x**

**}**

**};**

**};**

**inner\* p = 0; // ошибка: `inner' вне области видимости**

# Глава 6. Наследование

## 6.1. Наследование

Наследование позволяет строить иерархии, в которых производные классы получают элементы родительского (родительских) или базового классов и могут дополнять или изменять его свойства. Классы, находящиеся ближе к началу иерархии, объединяют в себе наиболее общие черты для всех нижележащих классов. По мере продвижения вниз иерархии, классы приобретают все более индивидуальные черты. Множественное наследование позволяет одному классу обладать свойствами многих родительских. Причем любой класс может быть объявлен потомком ранее описанного класса.

Доступ к полям, описанным в классе родителя, осуществляется так же, как в собственном. Поиск метода в иерархии класса выполняется следующим образом:

* в первую очередь компилятор устанавливает тип объекта
* ищет метод в классе объекта, если находит, то подключает его
* если метод в классе объекта не найден, то идет поиск в классе родителя. В случае успеха вызывается метод класса родителя. Если метод в классе родителя не найден, то поиск продолжается в классе предков до тех пор, пока метод не будет найден
* если метод не найден, выдается ошибка

Механизм наследования обеспечивает создание дерева родственных классов. Дерево может иметь несколько уровней, причем на каждом из них могут добавляться новые поля и методы.

Ключи доступа при наследовании

При описании класса в его заголовке перечисляются все классы, являющиеся для него базовыми. Возможность обращения к элементам этих классов регулируется с помощью ключей доступа: private, protected, public.

*Пример 1*

**class name:[private|protected|public] parent\_class;**

**class A{...};**

**class B{...};**

**class C{...};**

**class D:A, protected B, public C {...};**

|  |  |  |
| --- | --- | --- |
| **Ключ доступа** | **Спецификатор в базовом классе** | **Спецификатор в производном классе** |
| private | private | no |
|  | protected | private |
|  | public | private |
| protected | private | no |
|  | protected | protected |
|  | public | protected |
| public | private | no |
|  | protected | protected |
|  | public | public |

Private элементы базового класса в производном классе недоступны в производном классе независимо от ключа доступа. Обращение к ним может осуществляться только через элементы базового класса.

Protected элементы с ключом private становятся private, в остальных случаях права доступа к ним не изменяются. Если базовый класс наследуется с ключом private, можно выборочно сделать некоторые элементы доступными в производном классе. Для этого их необходимо объявить в секции public производного класса с помощью оператора доступа к области видимости (::).

*Пример 2*

**class base**

**{**

**...**

**public: void f();**

**};**

**class derived:private base**

**{**

**...**

**public: base::void f();**

**};**

Для различных методов класса при использовании механизма наследования существуют разные правила наследования. Конструкторы и операторы присваивания в производном классе не наследуются. Операторы инкрементации и декрементации – наследуются.

*Пример 3*

*Пример простого наследования*

**class queue**

**{**

**protected:**

**int q[10];**

**int sloc, rloc;**

**public:**

**void init(void);**

**void qput(init i);**

**int qget(void);**

**};**

**class queue1:public queue**

**{**

**int sum;**

**public:**

**int get\_sum(void);**

**void show\_sum(void);**

**};**

**/\* копирование функций-членов класса queue \*/**

**voi queue::init(void)**

**{**

**rloc=sloc=0;**

**}**

**int queue::qget(void)**

**{**

**if(sloac==rloc)**

**{**

**cout<< "очередь пуста";**

**return 0; /\* необходимо вернуть какое-либо значение \*/**

**}**

**return q[rloc++];**

**}**

**void queue::qput(int i)**

**{**

**if(sloc==10)**

**{**

**cout<< "очередь заполнена";**

**return;**

**}**

**q[sloc++]=i;**

**}**

**/\* Описание функций - членов класса queue 1\*/**

**int queue1::get\_sum(void)**

**{**

**sum=0;**

**for(int i = rloc; i<sloc; i++)**

**sum+=q[i];**

**return sum;**

**}**

**void queue1::show\_sum(void)**

**{**

**cout<<"Сумма очереди - "<<sum<<endl;**

**}**

**main(void)**

**{**

**queue1 obj;**

**obj.init();**

**for(int i=0; i<5; i++)**

**{**

**obj.qput(100+i);**

**obj.get\_sum();**

**obj.show\_sum();**

**}**

**for(i=0; i<6; i++)**

**{**

**obj.get\_sum();**

**obj.show\_sum();**

**obj.qget();**

**}**

**return 0;**

**}**

## 6.2. Правила наследования различных методов

Конструкторы не наследуются, поэтому производный класс должен иметь собственный конструктор.

**Порядок вызова конструктора в производном классе осуществляется по следующим правилам:**

1. если в конструкторе производного класса осуществляется явный вызов конструкторов базового класса. Если он отсутствует, автоматически вызывается конструктор базового класса по умолчанию (тот, который можно вызывать без параметров)
2. для иерархии, состоящей из нескольких уровней, конструкторы базовых классов вызываются начиная с верхнего уровня. Потом выполняются конструкторы тех элементов класса, которые являются объектами в порядке их объявления в классе. Затем исполняется конструктор класса.
3. В случае множественного наследования при наличии нескольких базовых классов, их конструкторы вызываются в порядке объявления. Если конструктор базового класса требует указания параметров, он должен быть вызван явным образом в конструкторе производного класса в списке инициализации.
4. Не наследуются операция присваивания, следовательно, её требуется явно определить в производном классе

Вызов функции базового класса предполагает копирование фрагментов кода из функций базового класса в функции производного. При этом осуществляется сокращение объектного кода и достигается упрощение модификации программы.

**Правила наследования деструктора:**

1. Деструкторы не наследуются. Если программист не описал в производном классе деструктор, он формируется по умолчанию и вызывает деструкторы всех базовых классов
2. В отличии от конструктора при описании деструктора производного класса в нем не требуется явно вызывать деструкторы базовых классов, т.к. это будет сделано автоматически.
3. Для иерархии классов, состоящих из нескольких уровней, деструкторы вызываются в порядке строго обратном вызову конструкторов. Сначала вызывается деструктор производного класса, потом деструктор базового класса.

Доступ к переопределенному методу базового касса выполняется через имя, уточненное с помощью операции доступа к области видимости (::).

**Присваивание объектов при наследовании**

Для объектов возможно выполнение операции присваивания. Смысл этой операции заключается в том, что происходит поэлементное присваивание содержимого полей одного объекта полями другого. Данная операция применима как к объектам одного класса, так и объектам классов одной иерархии. Причем разрешается только объектам родительских классов присваивать значение производных классов. Присваивание в обратном порядке не допускается т.к. в объекта потомков объявляются дополнительные поля, а в случае присваивания объекту класса потомка значения объекта родительского класса эти дополнительные поля остаются неопределенными.

Формат записи операции присваивания выглядит следующим образом:

**<объект родительского класса> = <объект производного класса>**

## 6.3. Множественное наследование

Когда механизм наследования осуществляется от нескольких классов, то это множественное наследование. При множественном наследовании производный класс может включать произвольное количество базовых классов. Форма множественного наследования выглядит так:

**class <имя производного класса>:**

**<вид наследования> <имя базового класса>**

**<вид наследования> <имя базового класса>**

**...**

**{...};**

Вид наследования определяет режим доступа к компонентам каждого из базовых классов. Базовые классы создаются в том порядке, в котором они перечислены в списке базовых классов при объявлении производного класса. Если конструкторы базовых классов не имеют аргументов, то производный класс может не иметь конструктора. При наличии у конструктора базового класса одного или нескольких аргументов, каждый производный класс должен иметь конструктор. Чтобы передать аргумент в базовый класс, нужно определить их после объявления конструктора производного класса.

**<имя конструктора производного класса> (<список аргументов>),**

**<имя конструктора базового класса 1> (<список аргументов>),**

**...**

**{<тело конструктора производного класса>}**

Список аргументов, ассоциированных с базовым классом, может состоять из констант, глобальных параметров и/или параметров для конструктора производного класса. Последовательность активации конструктора такая же, как и для случая одиночного наследования:

- активируются конструкторы для базовых классов (в порядке их перечисления в объявлении производного класса)

- затем активируется конструктор класса.

*Пример 1*

**class X**

**{**

**protected:**

**int x;**

**public:**

**x(int i); /\* конструктор с параметром \*/**

**~x(void); /\* деструктор \*/**

**void put\_x(int i) {x=i;}**

**int get\_x(void) {return x;}**

**void show(void);**

**};**

**class Y**

**{**

**protected:**

**int y;**

**public:**

**Y(int i); /\* конструктор с параметром \*/**

**~Y(void); /\* деструктор \*/**

**void put\_y(int i) {y=i;}**

**int get\_y(void) {return y;}**

**void show(void)**

**};**

**class Z: public X, public Y**

**{**

**protected;**

**int z;**

**public:**

**z(int i, int j); /\* конструктор с параметром\*/**

**~z(void); /\* деструктор \*/**

**void make\_z(void);**

**void show(void);**

**};**

**x::x(int i)**

**{**

**x=i;**

**cout<<” конструктор X \n”;**

**}**

**x::~x(void){ cout<<” деструктор X \n”;}**

**void x::show(void) { cout<<” X=”<<x<<endl;}**

**y::y(int i)**

**{**

**y=j;**

**cout<<” конструктор Y \n”;**

**}**

**y::~y(void){ cout<<” деструктор Y \n”;}**

**void y::show(void){ cout<<” Y=”<<y<<endl;}**

**/\* конструктор класса Z передает значение своих параметров \*/**

**z::z(int i, in j):Y(j), X(i) { cout<<” конструктор Z \n”;}**

**z::~x(void){ cout<<” деструктор Z \n”;}**

**void Z::make\_z(void){ z=x\*y;}**

**void Z::show(void){ cout<<z<<” =”<<x<<”\*”<<y<<endl;}**

**main(void)**

**{**

**z zobj(3, 5); /\* создание и инициализация объекта\*/**

**zobj.make\_z();**

**zobj.show();**

**zobj.x::show();**

**zobj.y::show(); /\* осуществление вызова функции из разных классов\*/**

**zobj.put\_x(7);/\* задание новых значений переменных \*/**

**zobj.put\_x(9);**

**zobj.make\_z();**

**zobj.show();**

**zobj.x::show();**

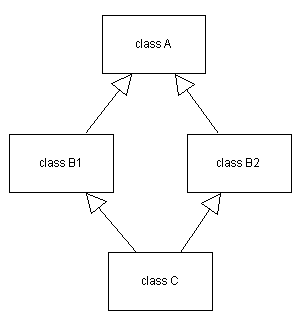
**zobj.y::show();**

**return 0;**

**}**

## 6.4. Виртуальное наследование

При множественном наследовании базовый класс не может быть задан в производном классе более одного раза. На практике возникает ситуация, когда производный класс будет содержать несколько копий одного базового класса. Чтобы избежать многократного включения в производный класс компонент базового класса, используется виртуальное наследование.



При виртуальном наследовании производный класс описывается так:

**сlass <имя производного класса>:**

**virtual <вид наследования> <имя базового класса>**

**{...};**

Включение в производный класс полей базового класса осуществляется один раз, а их инициализация происходит в производном классе, который не является прямым потомком базового класса. Вызов конструкторов при этом происходит в следующем порядке:

1. конструктор виртуально наследованного базового класса
2. конструкторы базовых классов, в порядке их перечисления при объявлении
3. конструкторы объектных полей
4. конструктор производного класса

Деструкторы вызываются в обратном порядке.

Виртуально наследованный класс должен содержать конструктор без параметров. Он активизируется при выполнении конструкторов классов, являющихся прямыми потомками виртуально наследованного класса.

*Пример 1*

**class fixed**

**{**

**protected:**

**int fix;**

**public:**

**fixed(void) /\* конструктор без параметров \*/**

**{ cout<<” вызов конструктора \n”;}**

**fixed(int fix)::fix(fix) /\* конструктор с параметром \*/**

**{ cout<<” вызов конструктора fixed int \n”;}**

**};**

**class derived\_1: virtual public fixed /\* виртуальное наследование \*/**

**{**

**public:**

**int one;**

**derived\_1(void) { cout<<”Вызов конструктора 1 \n”;};**

**};**

**class derived\_2: virtual public fixed /\* виртуальное наследование \*/**

**{**

**public:**

**int two;**

**derived\_2(void) { cout<<” Вызов конструктора 2 \n”;};**

**};**

**class derived: public derived\_1, public derived\_2 /\* объявление производного класса непрямого потомка \*/**

**{**

**public:**

**derived(void) {cout<< “вызов конструктора derived” <<endl;}**

**derived(int fix);**

**fixed(fix) { cout<<”вызов конструктора derived int“<<endl;}**

**void out() { cout<<”Fix = ”<< fix; }**

# Глава 7. Полиморфизм

## 7.1. Полиморфизм

Существует 2 вида полиморфизма: простой полиморфизм (статический), базирующийся на механизме раннего связывания, и сложный полиморфизм (динамический), использующий механизм позднего связывания. Простой полиморфизм поддерживается языком C++ на этапе компиляции (раннее связывание) и реализуется с помощью механизма переопределения функции. Эти полиморфные функции в C++ называются переопределенными. В соответствии с общими правилами они могут различаться типом возвращаемого параметра и сигнатурой ( количеством и типом передаваемых параметров ).

Динамический полиморфизм – выбор функции производится по используемому при выводе имени, типу вызываемого объекта и типу используемый параметров. Если объявленный тип не совпадает с типом использованного объекта, тип этого объекта идентифицируется и проверяется во время исполнения программы, что позволяет осуществить правильный выбор соответствующей функции. Это и называется динамическим полиморфизмом (поздним связыванием). Динамическое связывание приводит к небольшим издержкам памяти и времени выполнения программы, поэтому в C++ его используют только в случае явного указания, посредством ключевого слова virtual. Динамическое связывание используется путем объявления виртуальной функции. Виртуальная функция может реализовываться алгоритмами, которые, как правило, отличаются между собой. Если коды функции различны, то механизм виртуальности не включается. При использовании виртуальной функции, каждый аспект виртуальной функции, вызываемой из базового класса, определяется на этапе выполнения программы, когда известно для какого объекта вызван метод : базового класса или производного. Если вызов функции осуществляется через указатель на базовый класс, нужный аспект функции вызывается в зависимости от того, адрес объекта какого класса будет содержать указатель.

*Пример 1*

**class Window**

**{**

**public:**

**Window(const Point &upper.Dest, const Point &size, const string &title);**

**~Window();**

**void move(const Point &new\_upper\_left);**

**Point upper\_left() const;**

**Point lower\_right() const;**

**virtual void change\_size(const Point fnew\_size);**

**//Если новый размер point меньше минимального, сократить до этого минимального размера**

**int move cursor(contst Point &where);**

**Point cursor() const;**

**Display char character()const;**

**string Tile() const;**

**void add(Display char c);**

**void add(const string &str);**

**void clear();**

**void scroll\_up();**

**void scroll\_down();**

**private:**

**//данные для Window**

**};**

**/\* Функция collapse, которая использует только общие для всех типов окон функции, что обеспечивает возможность её применения для окон любого типа. \*/**

**void collapse(Window &w)**

**{**

**w.clear();**

**w.change\_size(Point(5,5));**

**//метод изменения размера change.size**

**}**

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

**{**

**window w(Point(10,6), Point(30,6), AWIndow);**

**shell\_Window sw(Point(3, [1]), Point(10, 10), "A shell\_window")**

**collapse(w);**

**collapse(sw);**

**return 0;**

**}**

Функция, объявленная виртуальной, остается таковой, сколько бы производных классов не образовывалось. Виртуальная функция обязательно должна быть компонентой некоторого класса. Она может быть объявлена дружественной другому классу, но не может быть статической.

## 7.2. Виртуальные методы

**Раннее связывание**

Работа с объектами чаще всего производится через указатели. Указателю на базовый класс можно присвоить значение адреса объекта любого производного класса.

*Пример 1*

**// Описывается указатель на базовый класс:**

**monstr \*p:**

**// указатель ссылается наобъект производного класса:**

**p = new daemon;**

Вызов методов объекта происходит в соответствии с типом указателя, а не фактическим типом объекта, на который он ссылается, поэтому при выполнении оператора, например, p->draw(1. 1, 1, 1); будет вызван метод класса monstr, а не класса daemon, поскольку ссылки на методы разрешаются во время компоновки программы. Этот процесс называется ранним связыванием или статическим полиморфизмом. Чтобы вызвать метод класса daemon, можно использовать явное преобразование типа указателя: (daemon \* p)->draw(l, 1, 1, 1);

Это не всегда возможно, поскольку в разное время указатель может ссылаться на объекты разных классов иерархии, и во время компиляции программы конкретный класс может быть неизвестен. В качестве примера можно привести функцию, параметром которой является указатель на объект базового класса. На его место во время выполнения программы может быть передан указатель на любой производный класс. Другой пример — связный список указателей на различные объекты иерархии, с которым требуется работать единообразно.

**Позднее связывание**

В C++ реализован механизм позднего связывания, когда разрешение ссылок на метод происходит на этапе выполнения программы в зависимости от конкретного типа объекта, вызвавшего метод. Этот механизм реализован с помощью виртуальных методов и рассмотрен в следующем разделе.

Для определения виртуального метода используется спецификатор virtual.

*Пример 2*

**virtual void draw(int x, int y, int scale, int position);**

Рассмотрим правила описания и использования виртуальных методов.

* Если в базовом классе метод определен как виртуальный, метод, определенный в производном классе с тем же именем и набором параметров, автоматически становится виртуальным, а с отличающимся набором параметров — обычным.
* Виртуальные методы наследуются, то есть переопределять их в производном классе требуется только при необходимости задать отличающиеся действия. Права доступа при переопределении изменить нельзя.
* Если виртуальный метод переопределен в производном классе, объекты этого класса могут получить доступ к методу базового класса с помощью операции доступа к области видимости.
* Виртуальный метод не может объявляться с модификатором static, но может быть объявлен как дружественный.
* Если в классе вводится описание виртуального метода, он должен быть определен хотя бы как чисто виртуальный. Чисто виртуальный метод содержит признак = 0 вместо тела.

*Пример 3*

**virtual void f(int) = 0;**

Чисто виртуальный метод должен переопределяться в производном классе (возможно, опять как чисто виртуальный).

Если определить метод draw в классе monstr как виртуальный, решение о том, ме­тод какого класса вызвать, будет приниматься в зависимости от типа объекта, на который ссылается указатель:

*Пример 4*

**monstr \*r,\*p;**

**r = new monstr;**

**//**

**Создается объект**

**класса monstr**

**р = new daemon;**

**//**

**Создается объект**

**класса daemon**

**r->draw(1, 1, 1, 1); //**

**Вызывается метод**

**monstr::draw**

**p->draw(1, 1, 1, 1); //**

**Вызывается метод**

**daemon::draw**

**p-> monstr::draw(1, 1, 1, 1); // Обход механизма виртуальных методов**

Если объект класса daemon будет вызывать метод draw не непосредственно, а косвенно (то есть из другого метода, определенного в классе monstr), будет вызван метод draw класса daemon.

Итак, виртуальным называется метод, ссылка на который разрешается на этапе выполнения программы (перевод красивого английского слова virtual — в данном значении всего-навсего «фактический», то есть ссылка разрешается по факту вызова).

**Механизм позднего связывания**

Для каждого класса (не объекта!), содержащего хотя бы один виртуальный метод, компилятор создает таблицу виртуальных методов (vtbl), в которой для каждого виртуального метода записан его адрес в памяти. Адреса методов содержатся в таблице в порядке их описания в классах. Адрес любого виртуального метода имеет в vtbl одно и то же смещение для каждого класса в пределах иерархии.

Каждый объект содержит скрытое дополнительное поле ссылки на vtbl, называемое vptr. Оно заполняется конструктором при создании объекта (для этого компилятор добавляет в начало тела конструктора соответствующие инструкции).

На этапе компиляции ссылки на виртуальные методы заменяются на обращения к vtbl через vptr объекта, а на этапе выполнения в момент обращения к методу его адрес выбирается из таблицы. Таким образом, вызов виртуального метода, в отличие от обычных методов и функций, выполняется через дополнительный этап получения адреса метода из таблицы. Это несколько замедляет выполнение программы.

Рекомендуется делать виртуальными деструкторы для того, чтобы гарантировать правильное освобождение памяти из-под динамического объекта, поскольку в этом случае в любой момент времени будет выбран деструктор, соответствующий фактическому типу объекта. Деструктор передает операции delete размер объекта, имеющий тип size\_t. Если удаляемый объект является производным и в нем не определен виртуальный деструктор, передаваемый размер объекта может оказаться неправильным.

Четкого правила, по которому метод следует делать виртуальным, не существует. Можно только дать рекомендацию объявлять виртуальными методы, для которых есть вероятность, что они будут переопределены в производных классах. Методы, которые во всей иерархии останутся неизменными или те, которыми производные классы пользоваться не будут, делать виртуальными нет смысла. С другой стороны, при проектировании иерархии не всегда можно предсказать, каким образом будут расширяться базовые классы (особенно при проектировании библиотек классов), а объявление метода виртуальным обеспечивает гибкость и возможность расширения.

Для пояснения последнего тезиса представим себе, что вызов метода draw осуществляется из метода перемещения объекта. Если текст метода перемещения не зависит от типа перемещаемого объекта (поскольку принцип перемещения всех объектов одинаков, а для отрисовки вызывается конкретный метод), переопределять этот метод в производных классах нет необходимости, и он может быть описан как невиртуальный. Если метод draw виртуальный, метод перемещения сможет без перекомпиляции работать с объектами любых производных классов -даже тех, о которых при его написании ничего известно не было.

Виртуальный механизм работает только при использовании указателей или ссылок на объекты. Объект, определенный через указатель или ссылку и содержащий виртуальные методы, называется полиморфным. В данном случае полиморфизм состоит в том, что с помощью одного и того же обращения к методу выполняются различные действия в зависимости от типа, на который ссылается указатель в каждый момент времени.

## 7.3. Абстрактные классы

Класс, содержащий хотя бы один чисто виртуальный метод, называется абстрактным. Абстрактные классы предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах. Эти классы применяются при определении данных и методов, которые будут общими для различных производных классов. Создание классов, имеющих общим только поведение, осуществляется с помощью базового класса, в котором все функции-члены объявлены в разделе public, а в разделе private данных либо очень мало, либо вовсе нет. Абстрактный класс может использоваться только в качестве базового для других классов — объекты абстрактного класса создавать нельзя, поскольку прямой или косвенный вызов чисто виртуального метода приводит к ошибке при выполнении.

Функции-члены в абстрактных классах определяются через *чистые виртуальные функции.* Чистые виртуальные функции – функции, которые объявлены с ключом virtual, но не определяются. Они должны быть переопределены в производных классах. В базовом классе они инициализируются с помощью идентификатора (=0). Форма записи чистой виртуальной функции:

**virtual <тип возвращаемого значения> <имя функции> ( <список параметров>) = 0;**

*Нельзя создать объект абстрактного класса,* т.к. прямой или косвенный вызов чистого виртуального метода приводит к ошибке при выполнении. Сам абстрактный класс используется только в качестве базового для создания других классов.

При определении абстрактного класса необходимо иметь в виду следующее:

* абстрактный класс нельзя использовать при явном приведении типов, для описания типа параметра и типа возвращаемого функцией значения;
* допускается объявлять указатели и ссылки на абстрактный класс, если при инициализации не требуется создавать временный объект;
* если класс, производный от абстрактного, не определяет все чисто виртуальные функции, он также является абстрактным.

Таким образом, можно создать функцию, параметром которой является указатель на абстрактный класс. На место этого параметра при выполнении программы может передаваться указатель на объект любого производного класса. Это позволяет создавать полиморфные функции, работающие с объектом любого типа в пределах одной иерархии.

*Пример 1*

**class DIsplay\_medium**

**{**

**public:**

**virtual Point size() const = 0;**

**virtual Point cursor() const = 0;**

**virtual int move\_cursor(const Point &p) = 0;**

**virtual Display char character() const = 0;**

**virtual string Tine() caonst = 0;**

**virtual void add (DIsplar char ch) = 0;**

**virtual void add(const String &s) = 0;**

**virtual void clear();**

**};**

**Пример 2**

**Class Gambler {**

**Public:**

**Virtual int move( )=0;**

**};**

# Глава 8. Параметризованные функции и классы

## 8.1. Шаблоны функций.

Простейший шаблон функции имеет форму:

**template <class Type> <заголовок>**

**{**

**<тело функции>**

**}**

В общем случае шаблон функции может содержать несколько параметров, каждый из которых может быть не только типом, но и просто переменной.

*Пример 1*

**template <class A, class B, int i> void f() {...}**

Шаблон функции использует в качестве аргумента тип переменной.

*Пример 2*

**template<class T> Tsgr\_it(T x)**

**{**

**...**

**return x\*x;**

**}**

В шаблоне может использоваться необязательный первый тип как параметр

*Пример 3*

**template<class T1, class T2>**

**max(T1 x, T2 y)**

**{ return (x>y) ? x : y; }**

В C++ вызов шаблона функции, использующий конкретный тип данных, приводит к тому, что компилятор создает код соответствующей версии функции. Этот процесс называется инстанцированием. Конкретный тип для инстанцирования определяется компилятором автоматически, исходя из типов параметров при вызове функции, либо задается явным образом. При повторном вызове с тем же типом данных код заново не генерируется. Если на месте параметра шаблона находится не тип, а переменная, то должно указываться константное выражение.

*Пример 4*

*Явное задание аргументов при вызове:*

**template<class x, class y, class z>void (x, y, z);**

**void g()**

**{**

**f < int, char\***

**}**

Как и обычные функции, шаблоны функций могут быть перегружены как с помощью шаблона, так и с помощью обычной функции. В программе можно предусмотреть специальную обработку.

## 8.2. Шаблоны классов

Шаблоны классов (параметризованные классы) рассматриваются как описание множества классов, моделирующих абстрактные структуры данных и отличающихся типом полей, включенных в эту структуру. Шаблоны классов определяют правила построения каждого отдельного класса из множества разрешенных (допустимых) классов. Синтаксис описания шаблона класса выглядит так:

**Template <список параметров шаблона>. <описание класса>**

В списке параметров шаблона могут быть как параметры, определяющие тип, так и параметры, для которых этот тип фиксирован. Каждый формальный параметр, определяющий тип, обозначается ключевым словом class. Используя шаблоны, можно определить объекты класса. Формат определения объекта одного из классов, порождаемых шаблоном, выглядит так:

**Имя\_парметра\_класса <список параметров шаблона>**

**Имя\_объекта (параметры конструктора)**

*Пример 1*

**Template<class T, int, size>**

**Class queue**

**{**

**T \*g;**

**Int sloc, rloc;**

**Public:**

**queue(void);**

**~queue(void);**

**void qput(T i);**

**T qget(void);**

**};**

**template<class T, int, size>**

**queue<T, size>::queue(void) /\* объединение членов функции класса \*/**

**{**

**if(!(q=newT[size]))**

**{**

**cout<<”not enough memory”;**

**return;**

**}**

**sloc=rloc=0;**

**cout<< “Очередь размера ”<<size<<” инициализирована.”;**

**}**

**template<class T, int, size>**

**queue<T, size>::~queue(void)**

**{**

**delete q;**

**cout<<”очередь разр.”;**

**}**

**template<class T, int, size>**

**void queue<T, size>::qput(T, i)**

**{**

**if(sloc==size)**

**{**

**cout<<”очередь полна”;**

**return;**

**}**

**q[sloc++]=i;**

**}**

**template<class T, int, size>**

**Tqueue<T, size>::qget(void)**

**{**

**if(rloc==sloc)**

**{**

**cout<<”очередь пуста”;**

**}**

**return q[rloc++];**

**}**

**main()**

**{**

**queue<int , 5> a;**

**queue<double, 200> b;**

**/\* объединение двух объектов класса queue массивами разных типов и размеров\*/**

**a: qput(10);**

**b: qput(1.129);**

**a: qput(23);**

**b: qput (5.55);**

**cout<<a.qget()<<” “;**

**cout<<a.qget()<<” “;**

**cout<<b.qget()<<” “;**

**cout<<b.qget()<<”\n“;**

**const int s = 10;**

**queue<long double, s> \*pq;**

**pq=new queue<long double, s>;**

**/\* динамическое создание объекта. именно сейчас вызывается конструктор queue \*/**

**if(!pq)**

**{**

**cout<<”недостаток памяти\n”;**

**return 0;**

**}**

**else**

**cout << “объект создан\n”;**

**for(int i=0; i<s; i++)**

**pq -> qput(i/2.0+i); /\* заполнение очереди \*/**

**for(i=0; i<s; i++) /\* просмотр элементов \*/**

**cout << pq -> qget() << “ “;**

**cout<<”\n”;**

**delete pq;**

**return pq;**

**}**

**Достоинства и недостатки шаблонов классов.**

*Достоинства:*

* мощное средство обращения с различными типами данных, которое называются параметрическим полиморфизмом
* обеспечивает безопасное использование различных типов данных в отличие от макросов препроцессора

*Недостатки:*

* программа, использующая шаблоны, содержит полный код каждого порожденного типа, что может увеличить размеры исполняемого файла
* шаблоны по-разному работают с разными типами данных (следует использовать специализацию шаблонов)

## 8.3. Контейнерные классы

Контейнерные классы — это классы, предназначенные для хранения данных, организованных определенным образом. Примерами контейнеров могут служить массивы, линейные списки или стеки. Для каждого типа контейнера определены методы для работы с его элементами, не зависящие от конкретного типа данных, которые хранятся в контейнере, поэтому один и тот же вид контейнера можно использовать для хранения данных различных типов. Эта возможность реализована с помощью шаблонов классов, поэтому часть библиотеки C++, в которую входят контейнерные классы, а также алгоритмы и итераторы, называют стандартной библиотекой шаблонов (STL - Standard Template Library).

Использование контейнеров позволяет значительно повысить надежность программ, их переносимость и универсальность, а также уменьшить сроки их разработки. Естественно, эти преимущества не даются даром: универсальность и безопасность использования контейнерных классов не могут не отражаться на быстродействии программы. Снижение быстродействия в зависимости от реализации компилятора может быть весьма значительным. Кроме того, для эффективного использования контейнеров требуется затратить усилия на вдумчивое освоение библиотеки.

Контейнерный класс содержит в своем определении несколько объектных полей, указателей на объекты, причем если контейнерный класс использует механизм композиции, то тип и количество управляемых объектов определены типом и количеством объектных полей. В случае, когда контейнерный класс использует механизм управления, подключение реализуется через указатели. Контейнер может управлять как объектами базового, так и всеми потомками этого класса. Обычно контейнерные классы реализуют некоторые типовые структуры (массивы, стеки, списки, очереди) и типовые операции над данными, которые могут записаны в эти структуры и прочитаны из них. Существует два способа реализации контейнерных классов:

Создание специальной процедуры просмотра всех элементов контейнера. В эту процедуру в качестве параметра передается имя функции или процедуры, реализующей алгоритм требуемой обработки элемента контейнера.

Выполнение поэлементной обработки и реализация через определение итератора или класса итераторов, подходящих для данного контейнера. При помощи итератора можно управлять контейнером, не зная фактически типов, используемых для идентификации элементов.

*Пример 1*

**class FixStack { /\* спецификация класса подобъекта \*/};**

**class BracketSyntax { {/\* спецификация контейнерного класса \*/}**

**public: FixStack stack; /\* спецификация подобъекта \*/**

**private: /\* спецификация приватных данных \*/**

**public: /\* спецификация компонентных методов,**

**конструктора и деструктора \*/**

**};**

Контейнеры создают на основе шаблонов класса - STL содержит контейнеры, реализующие основные структуры данных, используемые при написании программ — векторы, двусторонние очереди, списки и их разновидности, словари и множества. В зависимости от используемых шаблонов классов и особенности реализации доступа к данным, контейнеры можно разделить на две группы: последовательные и ассоциативные.

*Последовательные контейнеры* обеспечивают хранение конечного количества однотипных величин в виде непрерывной последовательности. К ним относятся векторы (vector), двусторонние очереди (deque) и списки (list), а также так называемые адаптеры, то есть варианты, контейнеров — стеки (stack), очереди (queue) и очереди с приоритетами (priority queue).

Каждый вид контейнера обеспечивает свой набор действий над данными. Выбор вида контейнера зависит от того, что требуется делать с данными в программе.

Например, при необходимости часто вставлять и удалять элементы из середины последовательности следует использовать список, а если включение элементов выполняется главным образом в конец или начало — двустороннюю очередь.

*Ассоциативные контейнеры* обеспечивают быстрый доступ к данным по ключу. Эти контейнеры построены на основе сбалансированных деревьев. Существует пять типов ассоциативных контейнеров:

* словари (mар),
* словари с дубликатами (multimap),
* множества (set),
* множества с дубликатами (multiset)
* битовые множества (bitset).

Программист может создавать собственные контейнерные классы на основе имеющихся в стандартной библиотеке.

Контейнерные классы обеспечивают стандартизованный интерфейс при их использовании. Смысл одноименных операций для различных контейнеров одинаков, основные операции применимы ко всем типам контейнеров. Стандарт определяет только интерфейс контейнеров, поэтому разные реализации могут сильно отличаться по эффективности.

Практически в любом контейнерном классе определены поля перечисленных ниже типов:

|  |  |
| --- | --- |
| **Поле** | **Пояснение** |
| value type | Тип элемента контейнера |
| size type | Тип индексов, счетчиков элементов и т. д. |
| iterator | Итератор |
| const iterator | Константный итератор |
| reverse iterator | Обратный итератор |
| const\_reverse iterator | Константный обратный итератор |
| reference | Ссылка на элемент |
| const\_reference | Константная ссылка на элемент |
| key\_type | Тип ключа (для ассоциативных контейнеров) |
| key compare | Тип критерия сравнения (для ассоциативных контейнеров) |

Итератор является аналогом указателя на элемент. Он используется для просмотра контейнера в прямом или обратном направлении. Все, что требуется от итератора — уметь ссылаться на элемент контейнера и реализовывать операцию перехода к его следующему элементу. Константные итераторы используются то­гда, когда значения соответствующих элементов контейнера не изменяются.

При помощи итераторов можно просматривать контейнеры, не заботясь о фактических типах данных, используемых для доступа к элементам.

**Итераторы**

Представим себе данные как некую абстрактную последовательность. Вне зависимости от способа ее организации и типа данных нам требуются средства поэлементного просмотра последовательности и доступа к каждому ее элементу. Итератор обеспечивает эти средства просмотра.

Итератор — это обобщение понятия указателя для работы с различными структурами данных стандартным способом. Для того, чтобы можно было реализовать алгоритмы, корректно и эффективно работающие с данными различной структуры, стандарт определяет не только интерфейс, но и требования ко времени доступа с помощью итераторов.

Поскольку итератор является обобщением понятия «указатель», семантика у них одинаковая, и все функции, принимающие в качестве параметра итераторы, могут также работать и с обычными указателями.

В стандартной библиотеке итераторы используются для работы с контейнерными классами, потоками и буферами потоков.

В итераторах используются понятия «текущий указываемый элемент» и «указать на следующий элемент». Доступ к текущему элементу последовательности выполняется аналогично указателям с помощью операций \* и ->. Переход к следующему элементу - с помощью операции инкремента ++. Для всех итераторов определены также присваивание, проверка на равенство и неравенство. Данные могут быть организованы различным образом - например, в виде массива, списка, вектора или дерева. Для каждого вида последовательности требуется свой тип итератора, поддерживающий различные наборы операций. В соответствии с набором обеспечиваемых операций итераторы делятся на пять категорий. Приведенные в таблице операции обеспечиваются за постоянное время.

Пусть i и j — итераторы одного вида, х — переменная того же типа, что и элемент последовательности, п — целая величина. Тогда допустимы выражения:

*1++ ++i i = j i == j i != j*

|  |  |  |
| --- | --- | --- |
| **Категория итератора** | **Операции** | **Контейнеры** |
| входной (input) | х = \*i | все |
| выходной (output) | \*i = х | все |
| прямой (forward) | х = \*i, \*i = x | все |
| двунаправленный (bidirectional) | x = \*i, \*i - x, --i, i-- | все |
| произвольного доступа (random access) | x = \*i, \*i = x, --i, i--, i + n, i - n, i += n, i -= n i < j, i > J, i <= j, i >= J | все, кроме list |

Как видно из таблицы, прямой итератор поддерживает все операции входных и выходных итераторов и может использоваться везде, где требуются входные или выходные итераторы. Двунаправленный итератор поддерживает все операции прямого, а также декремент, и может использоваться везде, где требуется прямой итератор. Итератор произвольного доступа поддерживает все операции двунаправленного, а кроме того, переход к произвольному элементу последовательности и сравнение итераторов.

Можно сказать, что итераторы образуют иерархию, на верхнем уровне которой находятся итераторы произвольного доступа. Чем выше уровень итератора, тем более высокие функциональные требования предъявляются к контейнеру, для которого используется итератор. Например, для списков итераторами произвольного доступа пользоваться нельзя, поскольку список не поддерживает требуемый набор операций итератора.

Итераторные классы и функции описаны в заголовочном файле <iterator>. При использовании стандартных контейнеров этот файл подключается автоматически.

Итераторы могут быть константными. Константные итераторы используются тогда, когда изменять значения соответствующих элементов контейнера нет необходимости.

|  |  |
| --- | --- |
| **Метод** | **Пояснение** |
| iterator beginO, const iterator beginO const | Указывают на первый элемент |
| iterator end(), const iterator end() const | Указывают на элемент, следующий за последним |
| reverse iterator rbeginO, const reversejterator rbeginO const | Указывают на первый элемент в обратной последовательности |
| reverse iterator rendO, const reverse iterator rendO const | Указывают на элемент, следующий за последним, в обратной последовательности |

В каждом контейнере эти типы и методы определяются способом, зависящим от их реализации.

Во всех контейнерах определены методы, позволяющие получить сведения о размере контейнеров:

|  |  |
| --- | --- |
| **Метод** | **Пояснение** |
| size() | Число элементов |
| max\_size() | Максимальный размер контейнера (порядка миллиарда элементов) |
| empty() | Булева функция, показывающая, пуст ли контейнер |

Другие поля и методы контейнеров мы рассмотрим по мере необходимости. STL определяется в 13 заголовочных файлах:

* algorithm
* deque
* functional
* iterator
* list
* map
* memory
* numeric
* queue
* set
* stack
* utility
* vector

Последовательные контейнеры

Векторы (vector), двусторонние очереди (deque) и списки (list) поддерживают разные наборы операций, среди которых есть совпадающие операции. Они могут быть реализованы с разной эффективностью:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Операция** | **Метод** | **vector** | **deque** | **list** |
| Вставка в начало | push\_front | - | + | + |
| Удаление из начала | pop front | - | + | + |
| Вставка в конец | push\_back | + | + | + |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Операция** | **Метод** | **vector** | **deque** | **list** |
| Удаление из конца | pop back | + | + | + |
| Вставка в произвольное место | insert | (+) | (+) | + |
| Удаление из произвольного места | erase | (+) | (+) | + |
| Произвольный доступ к элементу | [ ]. at | + | + | - |

Знак + означает, что соответствующая операция реализуется за постоянное время, не зависящее от количества п элементов в контейнере. Знак (+) означает, что соответствующая операция реализуется за время, пропорциональное n. Для малых n время операций, обозначенных +, может превышать время операций, обо-значенных (+), но для большого количества элементов последние могут оказаться очень дорогими.

Как видно из таблицы, такими операциями являются вставка и удаление произвольных элементов очереди и вектора, поскольку при этом все последующие элементы требуется переписывать на новые места.

Итак, вектор — это структура, эффективно реализующая произвольный доступ к элементам, добавление в конец и удаление из конца.

Двусторонняя очередь эффективно реализует произвольный доступ к элементам, добавление в оба конца и удаление из обоих концов.

Список эффективно реализует вставку и удаление элементов в произвольное место, но не имеет произвольного доступа к своим элементам.

*Пример 2*

*Пример работы с вектором. В файле находится произвольное количество целых чисел. Программа считывает их в вектор и выводит на экран в том же порядке.*

**#include <fstream> #iinclude <vector> using namespace std: int main(){**

**ifstream in ("inpnum.txt");**

**vector<int> v; int x; while ( in » x, lin.eofO)**

**v.push\_back(x);**

**for (vector<int>::iterator i = v.begin(); i != v.end(); ++i)**

**cout « \*i « " ";**

Поскольку файл содержит целые числа, используется соответствующая специализация шаблона vector - vector<int>. Для создания вектора v используется конструктор по умолчанию. Организуется цикл до конца файла, в котором из него считывается очередное целое число. С помощью метода push\_back оно заносится в вектор, размер которого увеличивается автоматически1.

*Примечание 1*

Размер вектора не изменяется каждый раз при добавлении элемента, это было бы нерационально. Он увеличивается по определенному алгоритму, которым можно управлять.

Для прохода по всему вектору вводится переменная i как итератор соответствующего типа (напомню, что операция :: обозначает доступ к области видимо-сти, то есть здесь объявляется переменная i типа «итератор для конкретной специализации шаблона»). С помощью этого итератора осуществляется доступ ко всем по порядку элементам контейнера, начиная с первого. Метод beginO возвращает указатель на первый элемент, метод end() — на элемент, следующий за последним. Реализация гарантирует, что этот указатель определен.

Сравнивать текущее значение с граничным следует именно с помощью операции !=, так как операции < или <= могут быть для данного типа не определены. Операция инкремента (i++) реализована так, чтобы после нее итератор указывал на следующий элемент контейнера в порядке обхода. Доступ к элементу вектора выполняется с помощью операции разадресации, как для обычных указателей.

В данном примере вместо вектора можно было использовать любой последовательный контейнер путем простой замены слова vector на deque или list. При этом изменилось бы внутреннее представление данных и набор доступных операций, а в поведении программы никаких изменений не произошло бы.

Однако если вместо цикла for вставить фрагмент

**for (int i = 0; i<v.size(); i++) cout << v[i] << " ";**

в котором использована операция доступа по индексу [ ], программа не будет работать для контейнера типа list, поскольку в нем эта операция не определена.

# Глава 9. Динамическое выделение памяти

## 9.1. Динамическое выделение памяти

Создание и поддержание динамических структур данных требует динамического распределения памяти: возможности в процессе выполнения программы изменения области памяти для хранения новых узлов и освобождения ресурсов памяти, в которых уже нет необходимости.

Пределы динамического выделения памяти ограничены только объемом доступной физической или виртуальной памяти в системах с виртуальной памятью. Операции new и delete – основные для работы с динамической памятью. Операция new принимает в качестве аргумента тип динамического размещения объекта и возвращает указатель на объект этого типа.

*Пример 1*

**Node \*newptr = new Node[10];**

**// выделяет в памяти sizeof(Node) байтов и сохраняет указатель на область памяти указателем Ptr. Число 10 – число размещенных объектов данных.**

*Пример 2*

**main()**

**{**

**int\* p = new int;**

**cout << "sizeof(int) = " << sizeof(int) "\n";**

**}**

Для освобождения памяти используется оператор delete. Указатель newptr не удаляется, а исчезает область памяти, на которую указывает newptr.

*Пример 3*

**delete[] newptr;**

*Пример 4*

**main() {**

**char \*p = new char[100];**

**char \*q = new char[100];**

**delete p;**

**delete q;**

**}**

Типичные ошибки:

* может быть ссылка на область памяти, которая уже была освобождена
* оператором delete освобождать память, которая не была выделена new
* не осуществляется возвращение динамически выделенной памяти, когда эта память уже не требуется. Это может явиться причиной переполнения памяти (утечка памяти)
* предположение о том, что размер объекта класса является простой суммой объектов его элементов данных. Это не так по причине различных машинно-зависимых требований по выравниванию границ области памяти.
* необходимо проверять, не вернула функция new нулевой указатель

*Пример 5*

**/\* распределить память под 1-мерный массив указателей на беззнаковые целые размером degree**

**по адресу tab используя оператор new \*/**

**tab=new unsigned \*(degree);**

**/\* под каждый указатель полученного массива указателей распределить одномерный массив беззнаковых**

**целых из degree элементов, используя оператор new в цикле \*/**

**for (int i=0; i<degree;i++)**

**tab[i]=new unsigned(degree);**

**/\* освободить память, распределенную под degree одномерных массивов беззнаковых целых (из degree элементов каждый)**

**по адресам от tab[0] до tab[degree-1] \*/**

**for (int i=0; i<degree; i++)**

**delete [degree](tab[i]);**

**/\* освободить память, распределенную под 1-мерный массив указателей на беззнаковые целые,**

**состоящий из degree указателей по адресу tab \*/**

**delete [degree]tab;**

# Глава 10. Перегрузка операций

## 10.1. Перегрузка операций

Функция-оператор (операторная функция, функциональная операция) может быть определена как внутри описания класса, так и вне его. Различают два вида операторных функций:

* простую (определяется вне класса, может быть одноместной или двухместной)
* компонентную (определяется в классе. У нее первый аргумент представляет собой объект класса, заданный неявно. Она может быть одноместной – не имеет явных аргументов, либо двухместной – с одним аргументом)

|  |  |
| --- | --- |
| ***Простая функция*** | ***Компонентная функция*** |
| **Одноместная**: <тип результата> operator@( <аргумент> ) | **Одноместная**: <тип результата> operator@() |
| **Двухместная**: <тип результата> operator@( <аргумент 1>, <аргумент 2> ) | **Двухместная**: <тип результата> operator@( <аргумент 1>, <аргумент 2> ) |

<тип результата> - тип возвращаемого значения (имеет тот же тип, что и класс, но возможен и иной тип значения).

*Следует помнить, что нельзя перегружать следующие операторы: . :: .\* ? sizeof и операторы препроцессора # и ##*

Перегрузка операций подчинена следующим правилам:

* при перегрузке операций сохраняется количество аргументов, приоритеты операций и правила ассоциации, используемые в стандартных типах данных
* для стандартных типов данных переопределить операции нельзя
* функциональные операции не могут иметь аргументов по умолчанию
* функциональные операции наследуются (кроме операции присваивания =)
* функциональные операции не могут определяться как static

Функциональную операцию можно определить тремя способами:

* как метод класса
* как дружественную функцию
* как обычную функцию

Если операторная функция определяется вторым и третьим способами, то она должна принимать хотя бы один аргумент, имеющий тип класса, либо указатель или ссылку на класс.

Особый случай операторной функции: когда первый параметр имеет стандартный тип. В этом случае она не определяется как метод класса.

Формы вызова операторной функции:

|  |  |
| --- | --- |
| ***Стандартная форма*** | ***Операторная форма*** |
| **Простая функция**: operator@( <аргумент> ) | **Простая функция:** @( <аргумент> ) |
| operator@( <аргумент 1>, <аргумент 2>) | <аргумент1>@ <аргумент2> |
| **Компонентная функция:**<аргумент>.operator@() | **Компонентная функция:** @( <аргумент> ) |
| <аргумент1>.operator@( <аргумент2> ) | <аргумент1>@ <аргумент2> |

## 10.2. Перегрузка унарных операций

Унарная операторная функция может быть определена как внутри, так и вне класса. Унарная функциональная операция, определенная внутри класса, должна быть представлена с помощью нестатического метода. При этом операндом является вызывающий её объект.

*Пример 1*

**class monstr**

**{**

**monstr &operator ++()**

**{**

**++health;**

**return \*this;**

**}**

**};**

**monstr Vasia;**

**cout<<(++Vasia).get\_health();**

Если унарная операция определена вне класса, она должна иметь один параметр - типа класса

*Пример 2*

**class monstr**

**{**

**friend monstr &operator ++(monstr &M);**

**};**

**monstr &operator ++ (monstr &M)**

**{**

**++M.health;**

**return M;**

**}**

Если не описывать функцию внутри класса как дружественную функцию, нужно учитывать доступность полей. В данном примере поле health не доступно извне, т.к. описано как private. Для его изменения следует использовать специальные методы. Введем в класс monstr специальный метод change\_health, который будет позволять менять значение health:

*Пример 3*

**void change\_health( int he ) {health =he;}**

При вводе этого метода можно пере+гружать оператор инкрементации с помощью обычной функции, описанной вне класса:

*Пример 4*

**monstr &operator ++ (monstr &M)**

**{**

**int h=M.get\_health();**

**h++;**

**M.change\_health(h);**

**return M;**

**}**

В случае использования оператора постфиксного инкремента и декремента дадим им первый параметр типа int. Он используется для того, чтобы отличить их от префиксной формы.

*Пример 5*

**const monstr operator ++ (int)**

**{**

**monstr M(\*this);**

**health++;**

**/\* Лучше ++(\*this); \*/**

**return M;**

**}**

**monstr Vasia;**

**cout << (Vasia++).get\_health();**

## 10.3. Перегрузка бинарных операций

Бинарная функциональная операция может определятся внутри и вне класса.

Если бинарная операция определена внутри класса, то при этом вызывающий ее объект считается первым операндом.

*Пример 1*

**/\* внутри \*/**

**class monstr**

**{**

**bool operator>(const monstr &M)**

**{**

**if(health > M.get\_health()) return true;**

**return false;**

**}**

**}**

**/\* вне класса\*/**

**bool operator>(const monstr & M1, const monstr &M2)**

**{**

**if(M1.get\_health > M2.get\_health()) return true;**

**return false;**

**}**

## 10.4. Перегрузка операции присваивания

Операция присваивания может быть определена в любом классе по умолчанию как конструктор копирования. Эта операция вызывается каждый раз, когда одному объекту нужно присвоить значение другого. Если класс содержит поля, память под которые выделяется динамически, то необходимо определить собственный оператор присваивания. Операция должна возвращать ссылку на объект для которого она вызывалась и принимать в качестве параметра один аргумент: ссылку на присваиваемый объект.

*Пример 1*

**const monstr &operator = (const monstr &M)**

**{**

**/\* проверка на самоприсвивание \*/**

**if(& == this) return \*this;**

**if(name) delete [] name;**

**if(M.name)**

**{**

**name = new char[strlen(M.name)+1];**

**strcpy(name, M.name)**

**}**

**else name = 0;**

**health = M.health;**

**skin = M.skin;**

**return \*this;**

**}**

*Примечание:* возврат из функции указателя на объект делает возможным цепочку присваиваний.

**monstr A(10), B, C;**

**C=B=A;**

*Операцию присваивания можно определить как метод класса. Операция присваивания не наследуется.*

## 10.5. Перегрузка операций new и delete

Эти операции используются для обеспечения альтернативных вариантов управления памятью. Перегрузка операции new используется для выделения динамической памяти под объекты либо массивы объектов. Перегрузка операции delete используется для освобождения памяти, занятой объектом, или массивом объектов.

Использование функциональных операций new и delete должно соответствовать следующим правилам: этим функциям не требуется передавать параметр типа класса. Первым параметром функциям new и new[] должен передаваться размер объекта типа size. Этот тип, возвращаемый функцией sizeof, определяется в заголовочном файле <stddef.h>. При вызове он передается функции неявным образом. Функции new и delete должны определяться с типом возвращаемого значения void\* даже если return возвращает указатель на другие типы (например, на класс). Операция delete должна иметь тип возврата void и первый аргумент типа \*void. Операции new и delete являются статическими элементами класса. Поведение перегруженной операции должно соответствовать действиям, выполняемым ими по умолчанию. Для операции new это означает, что она должна возвращать правильное значение, корректно обработав запрос на выделение памяти нулевого размера или порождать исключения при невозможности выполнения запроса. Для операции delete следует соблюдать условие: удаление любого указателя должно быть безопасным, следовательно, внутри операции delete необходимо проверять указатель на NULL (при этом должно быть отсутствие каких-либо действий в случае равенства). Стандартные операции выделения и освобождения памяти могут использоваться в области действия класса наряду с перегруженными операциями. Это осуществляется с помощью операции доступа к области видимости для объектов этого класса (::) и непосредственно для всех других.

Перезагрузка операций выделения памяти используется для экономии памяти, либо для размещения данных в некоторой конкретной области, если это необходимо.

*Пример 1*

**class Obj {...};**

**class pObj**

**{**

**private:**

**obj \*p;**

**};**

**PObj \*p = new pObj; /\* Для выделении памяти под объект типа pObj с помощью стандартной функции new \*/**

Для небольших объемов накладные расходы могут оказать весьма значительными. Для экономии памяти можно написать собственную операцию new класса pObj. Она будет выдавать большой блок памяти и размещать в нем указатели на Obj. Для этого в объект pObj вводится статическое поле, в котором хранится указатель на первую свободную ячейку блока для размещения очередного объекта. Неиспользуемые ячейки связываются в список. Чтобы не занимать память под поле связи, можно использовать объединения (union), с помощью которых одна и та же ячейка используется либо для хранения указателя, либо для связи со следующей свободной ячейкой.

*Пример 2*

**class pObj**

**{**

**public:**

**static void \*operator new (size\_t size);**

**private:**

**union**

**{**

**obj p; / \* указатель на объект \*/**

**pObj next /\* указатель на следующую свободную ячейку \*/**

**}**

**static const int BLOCK\_SIZE; /\* размер блока \*/**

**/\* заголовок списка свободных ячеек \*/**

**static pObj \*headOfFree;**

**};**

**void \*pObj::operator new(size\_t size)**

**{**

**/\* перенаправить запрос неверного количества памяти стандартной операции new \*/**

**if(size!=sizeof(pObj)) return::operator new(size);**

**pObj \*p = headOfFree;**

**/\* указатель на первую свободную ячейку \*/**

**/\* переместить указатель списка свободных ячеек \*/**

**if(p) headOfFree = p->next;**

**/\* если свободной памяти нет - выделить очередной блок \*/**

**else**

**pObj \*new block = static\_cast<pObj \*>(::operator new(BLOCK\_SIZE \*sizeof(pObj)));**

**/\* здесь используется явное преобразование типа с помощью операции static\_cast \*/**

**/\* все ячейки кроме первой свободны, связываем их \*/**

**for(int i = 1; i<BLOCK\_SIZE-1; ++i)**

**new block[i].next = &new block[i+1];**

**new block[BLOCK\_SIZE-1.next=0];**

**/\* устанавливаем начало списка свободных ячеек \*/**

**headOfFree = &newblock[1];**

**p = newblock;**

**return p; /\* возвращаем указатель на память \*/**

**}**

**/\* должна присутствовать инициализация статических полей класса pObj \*/**

**pObj\*pObj::headOfFree;**

**/\* устанавливаются нули по умолчанию \*/**

**const int pObj::BLOCK\_SIZE = 1024;**

**/\* операция delete должна добавлять ячейку памяти к списку свободных ячеек \*/**

**void pObj::operator delete(void \* ObjToDie, size\_t size)**

**{**

**if(ObjToDie == 0) return;**

**if(size!=sizeof(pObj))**

**{**

**::operator delete(ObjToDit);**

**return;**

**}**

**pObj p = static\_cast<pObj>(ObjToDie);**

**p->next=headOfFree;**

**headOfFree=p;**

**}**

Если перегруженная операция new наследуется, то она вызывается для производных объектов. Причем если их размер не соответствует размеру базового объекта (это встречается часто), то это приводит к проблемам. Для избежания проблем необходимо осуществлять проверку на размер. Если размер объекта не равен тому, для которого перегружалась new, то запрос на выделение памяти следует осуществлять с помощью не перегруженной new.

В случае, когда стандартная операция delete не найдет в начале объекта верной информации о его размере, это приводит к неверному поведению программы. В рассматриваемом примере операция delete должна добавлять освобождающуюся ячейку памяти к списку свободных ячеек. В операции delete выполняется проверка соответствия размеров объектов по объему выделенной памяти.

## 10.6. Перегрузка операций приведения типа

Преобразования типов бывают сужающими и расширяющими. Сужающие трансформируют величину в тип, который не может содержать всех значений данного типа. Расширяющие преобразования трансформируют величину в тип, который может содержать как минимум все значения исходного типа. Расширяющий тип преобразований безопасен в программировании всегда, сужающий – нет. Преобразование типов может быть как явным, так и неявным. Несмотря на отсутствие в С++ механизмов уточнения подтипов, имеется мощная поддержка перегрузки функций. При проверке соответствия учитывается преобразование типов, (переопределение), как встроенное в язык, так и определенное программистом. Принцип, лежащий в основе применяемых преобразований, состоит в том, что следует отдавать предпочтение преобразованиям наименее опасным с точки зрения возможных ошибок, наименее «сомнительным». Определенные пользователем преобразования рассматриваются только в том случае, если без них вызов разрешить нельзя.

*Пример 1*

**void g (int);**

**double d = 1.234;**

**char \*str = "Строка";**

**g(d); // допустимо, т.к. существует неявное преобразование double к int**

**g(str); // ошибка**

*Пример 2*

**monstr::operator int() {return health;}**

**monstr Vasia;**

**cout<<int(Vasia);**

## 10.7. Правила выбора реализации перегружаемых функций в C++

Далее приведены правила выбора реализации перегружаемых функций в C++ (тип, указанный слева от стрелки, преобразуется к типу, указанному справа).

- преобразования, выполняющиеся по умолчанию

**Array -> pointer**

**T -> const T**

- стандартные преобразования

**int -> double**

**double -> int**

производный тип -> базовый

- расширяющие разрядности преобразования

**int -> double**

**short -> int**

**float -> double**

Формат перегрузки операции приведения типа:

operator <имя нового типа> ();

*Пример 1*

**char c = 256; // из-за потери 8 старших битов c ='\0';**

**double d = 0.999999999999999999;**

**long n = d; // n=0;**

**long m = 32768; // двоичное представление числа 32768 содержит единственную единицу - в 15 разряде**

**int k = m; // k = -32768, т.к. 15-й разряд для int является знаковым**

**unsigned u = m; // m = 32768**

*Пример 2*

**void g (int);**

**double d = 1.234;**

**char \*str = "Строка";**

**g(d); // допустимо, т.к. существует неявное преобразование double к int**

**g(str); // ошибка**

Пример 3

**monstr::operator int() {return health;}**

**monstr Vasia;**

**cout<<int(Vasia);**

## 10.8. Переопределение операций ввода-вывода

При использовании библиотеки ввода-вывода необходимо помнить, что в C++ классы ввода-вывода используют операции: включение в поток, исключение из потока. Эти операции используются только для стандартных типов данных. Поэтому, при разработке программ для определяемых пользователем типов полей класса и для классов целиком при выводе объектов этих классов операции включения/исключения должны переопределяться для каждого класса.

Формат операций включения/исключения из потока:

**ostream &operator << (ostream &out, <новый тип> <имя>)**

**{<тело функции оператора>};**

**istream &operator >> (istream &in, <новый тип> <<мя>)**

**{<тело функции оператора>};**

Если операции включения/исключения используются для полей описания, как private или protected, то эти операции должны описываться как дружественные.

Пример 1

**#include<iostream.h>**

**class Tadress**

**{**

**private:**

**char country[16];**

**char city[16];**

**char street[20];**

**int housenum;**

**public:**

**Tadress(){}**

**~Tadress(){}**

**friend ostream &operator << (ostream &out, Tadress obj);**

**friend istream &operator >> (istream &in, Tadress &obj);**

**};**

**ostream &operator << (ostream &out, Tadress obj) /\* тело функции переопределяет вставки в поток \*/**

**{**

**out << "Address: "<< endl;**

**out << "Country: "<< obj.country<< endl;**

**out << "City:"<< obj.city<< endl;**

**out << "House:"<< obj.housenum<< endl;**

**return out;**

**}**

**istream &operator >> (istream &in, Tadress &obj) /\* тело функции переопределяет исключения из потока \*/**

**{**

**cout<<"введите адрес следующим образом:";**

**cout<<" <<страна>> <<город>> <<улица>> <<номер дома>>"<<endl;**

**in >> obj.country >> obj.city >> obj.street >> obj.housenum ;**

**return in;**

**}**

**void main()**

**{**

**clrscr();**

**Tadress a, b;**

**cin >> a >> b;**

**cout << a << b;**

**getch();**

**}**

## 10.9. Перегрузка операций вызова функций

Класс, в котором переопределен оператор вызова функции, называется функциональным классом. У такого класса не требуется наличие других полей и методов.

*Пример 1*

**class if\_greater**

**{**

**public**

**int operator() (int a, int b) const { return a>b; }**

**};**

Объект данного функционального класса используется так, как если бы он был функцией.

*Пример 2*

**if\_greater x;**

**cout<< x(1, 5)<< endl; /\* 0 \*/**

**cout<< if\_greater() (5, 1)<< endl; /\* 1 \*/**

Поскольку в классе определена операция вызова функции с двумя аргументами, то выражение x(1, 5) является допустимым, или это можно записать так:

*Пример 3*

**\*operator()(1, 5)**

А во втором операторе вызова функции if\_greater используется для вызова конструктор по умолчанию класса if\_greater.

Как и в предыдущем случае для этого объекта вызывается функция с двумя аргументами, записанными в круглые скобки. Операцию круглые скобки можно определить только как метод класса. Можно определить перегруженные операции вызова функции с разным количеством аргументов.

## 10.10. Перегрузка операций индексирования

Операция индексирования обычно перегружается когда тип класса представляет множество значений, для которых индексирование имеет смысл. Операция индексирования должна возвращать ссылку на элемент, содержащийся во множестве.

*Пример 1*

*Пример перегрузки операции индексирования для класса Vect, предназначенного для хранения массива целых чисел и безопасной работы с ним.*

**#include<iostream.h>**

**#include<stdlib.h>**

**class Vect**

**{**

**public:**

**explicit Vect (int n=10);**

**Vect (const int a[], int n); /\* инициализация массива \*/**

**~Vect(){delete[] p;}**

**int &operator[] (int i);**

**void print();**

**private:**

**int \*p;**

**int size;**

**};**

**Vect::Vect(int n):size(n)**

**{**

**p=new int[size];**

**for (int i=0; i<size; i++)**

**p[i]=a[i];**

**}**

**/\* перегрузка операции индексации \*/**

**int &Vect::operator[](int i)**

**{**

**if(i<0 || i>=size)**

**{**

**cout<<"Неверный индекс (i="<<i<<")"<<endl;**

**cout<<"Завершение программы"<<endl;**

**exit(0);**

**}**

**return p[i];**

**}**

**void Vect::Print()**

**{**

**for(int i=0; i<size; i++)**

**cout<<p[i]<<" ";**

**cout<< endl;**

**}**

**int main()**

**{**

**int arr[10]={1,2,3,4,5,6,7,8,9,10};**

**Vect a(arr, 10);**

**a Print();**

**cout<<a[5]<<endl;**

**cout<<a[12]<<endl;**

**return 0;**

**}**

Результаты работы:

1 2 3 4 5 6 7 8 9 10

6

Неверный индекс (i=12)

Завершение программы

Перегруженная операция индексирования получает целый аргумент и проверяет, лежит ли его значение в пределах диапазона массива. Если лежит, то возвращается адрес элемента, что соответствует стандартной семантике операции индексирования. В данном примере конструктор с параметрами по умолчанию объявлен как explicit. Это сделано для того, чтобы он не являлся конструктором преобразования типа, который может быть вызван неявно. В данном случае explicit указывает на то, что этот конструктор будет вызываться явным образом.

Операцию индексирования можно определять только как метод класса.

# Глава 11. Структуры данных

## 11.1. Стек

Со стеком определены только операции добавления и выборки элементов из вершины стека. При осуществлении выборки элемент исключается из стека. В стеке осуществляется принцип LIFO. Возможен также принцип FIFO. Стеки широко применяются в системном программировании при написании компиляторов и различных рекурсивных алгоритмов.

*Пример 1*

*Пример программы, которая формирует стек из пяти целых чисел, и выводит его на экран:*

**#include <iostream>**

**struct Node**

**{**

**int d;**

**Node \*p;**

**}**

**Node \*first(int d);**

**void push(Node \*\*top, int d);**

**void pop(Node \*\*top);**

**int main()**

**{**

**Node \*top = first(1);**

**for(int i=2; i<6; i++) push(&top, i);**

**while(top)**

**cout<<pop(&top)<<" ";**

**return 0;**

**}**

**/\* начальное формирование стека \*/**

**Node \*first (int d)**

**{**

**Node \*Pv = new Node;**

**Pv->d=d;**

**Pv->d=0;**

**return Pv;**

**}**

**/\* занесение в стек \*/**

**void push(Node \*\*top, int d)**

**{**

**Node \*Pv = new Node;**

**Pv -> d = d;**

**Pv -> p = \*top;**

**\*top = Pv;**

**}**

**/\* выборка из стека \*/**

**int pop(Node \*\*top)**

**{**

**int temp = (\*top) -> d;**

**Node \*Pv = \*top;**

**\*top = (\*top) -> p;**

**delete Pv;**

**return temp;**

**}**

**/\* вывод: 5 4 3 2 1 \*/**

## 11.2. Очереди

Очередь – абстрактная структура данных, реализует принцип обслуживания FIFO. Это частный случай однонаправленного списка, добавление элементов в который выполняется в один конец, а выборка идет из другого конца. Таким образом, в структуре «очередь» определены две операции: добавление элементов и выборка. При выборке элемент из очереди исключается. В системном программировании очереди используются для диспетчеризации задач моделирования.

*Пример 1*

*Пример программы, реализующей линейную очередь из пяти элементов, выводя их на экран*

**struct Node**

**{**

**int d;**

**Node \*p;**

**}**

**Node \*first(int d);**

**void add(Node \*\*pend, int d);**

**int del(Node \*\*pbeg);**

**int main()**

**{**

**Node \*pbeg = first(1);**

**Node \*pend = pbeg;**

**for(int i=2; i<6; i++) add (&pend, i);**

**while(pbeg)**

**cout<<del(&pbeg)<<" ";**

**return 0;**

**}**

**/\* начальное формирование очереди\*/**

**Node \*first(ind d)**

**{**

**Node \*pv = new Node;**

**pv -> d = 0;**

**pv -> p = 0;**

**return pv;**

**}**

**/\* добавление в конец \*/**

**void add(Node \*\*pend, int d)**

**{**

**Node \*pv = new Node;**

**pv -> d = d;**

**pv -> p = 0;**

**(\*pend) -> p = pv;**

**\*pend = pv;**

**}**

**/\* выборка \*/**

**int del(Node \*\*pbeg)**

**{**

**int temp = (\*pbeg) -> d;**

**Node \*pv = \*pbeg;**

**\*pbeg = (\*pbeg) -> p;**

**delete pv;**

**return temp;**

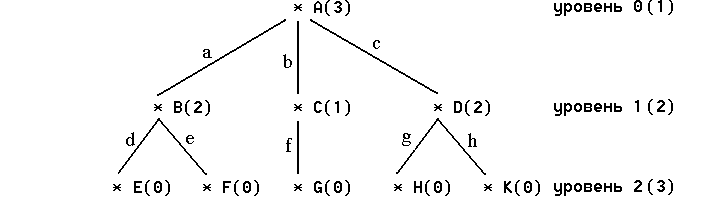
**}**

**результат работы: 1 2 3 4 5**

## 11.3. Деревья

**Понятие корневого дерева.**

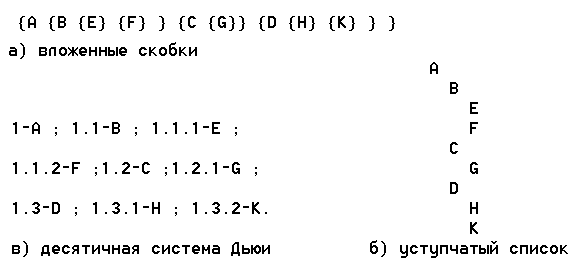
В теории графов конечное корневое дерево формально определяется как непустое конечное множество узлов Т, таких что: существует один, специально выделенный узел, называемый корнем, а остальные узлы образуют попарно непересекающиеся подмножества множества узлов Т, каждое из которых является деревом. Это определение позволяет интерпретировать корневое дерево как рекурсивный объект, который содержит сам себя и определяется с помощью себя самого (т. е. дерево определяется в терминах дерева). Определение корневого дерева как определение любого рекурсивного объекта содержит базисную и рекурсивную части. Базисная часть, определяющая корень дерева, является нерекурсивным утверждением. Рекурсивная часть определения записана так, чтобы она редуцировалась (сводилась) к базе цепочкой повторных применений. В данном случае дерево с числом узлов n>1 индуктивно определяется через деревья с числом узлов меньше, чем n, пока не достигнут базисный шаг, где дерево состоит из единственного узла - корня. Рекурсивное определение корневого дерева позволяет более простым способом формализовать его структуру и алгоритмы обработки. Для неформального описания корневых деревьев часто используется генеалогическая терминология, согласно которой каждая ветвь отражает отношение потомок-предок между инцидентными ей узлами. Корень дерева – это узел, который не имеет предка. Узлы дерева, которые не имеют потомков называются листьями. Остальные узлы (не листья и не корень) называются разветвлениями. Следующий рисунок иллюстрирует классическое изображение корневого дерева средствами теории графов, где вершины и ребра графа представляют узлы и ветви дерева.



*Рис. 1. Изображение корневого дерева в теории графов*

На этом рисунке заглавные буквы латинского алфавита обозначают узлы, а строчные- ветви корневого дерева. Конфигурация ветвей этого корневого дерева такова, что узел А является корнем, узлы В С и D- разветвлениями, а узлы E, F, G, H, и K - листьями.

Следует отметить, что кроме классического изображения, принятого в теории графов, в области информационных технологий применяются альтернативные способы представления корневых деревьев. На следующем рисунке приведены 3 эквивалентных способа представления исходного корневого дерева: с помощью вложенных скобок (а), уступчатого списка (б) и десятичной системы Дьюи (в) соответственно:



*Рис. 2. Альтернативные способы представления корневого дерева*

Приведенные альтернативные способы представления корневых деревьев иллюстрируют возможности практического применения иерархических структур.

Например, десятичная система Дьюи применяется в библиографии, а вложенные скобки - для получения полноскобочной записи при грамматическом разборе арифметических выражений.

Важными метрическими характеристиками корневого дерева является степень и уровень узла. Степенью узла корневого дерева считается число поддеревьев, для которых он является корнем. Для рассмотренного примера корневого дерева: корень А имеет степень 3, степени разветвлений B и D - равны 2, а степень разветвления С равна 1. Степени остальных узлов равны 0, потому что они являются листьями, т. е. не имеют поддеревьев. Уровень узла корневого дерева определяется длиной пути, образованного чередующейся последовательностью узлов и ветвей, который соединяет его с корнем. Длина пути измеряется числом узлов в нем. Для рассмотренного примера корень А имеет уровень 1, разветвления B, C и D имеют уровень 2, а листья E, F, G, H и K - уровень 3.

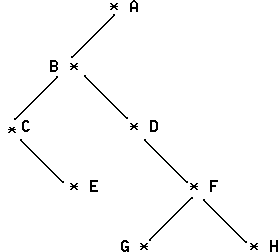
При измерении длины пути числом ветвей в нем, указанные уровни узлов надо уменьшить на 1.

Обобщением понятия корневого дерева является понятие леса. Под лесом понимается упорядоченное множество непересекающихся корневых деревьев. Отражением близости этих понятий является простота преобразований дерева в лес и наоборот, леса в дерево. Исключение корня превращает дерево в лес.

Наоборот, добавление одного узла превращает лес в дерево, где этот узел становится корнем. Чтобы подчеркнуть близость этих понятий, в некоторых работах для обозначения леса из N деревьев употребляют термин: дерево с N -кратным корнем.

Понятие бинарного дерева.

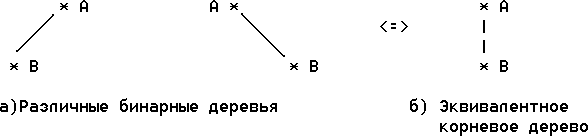
Важной разновидностью корневых деревьев является класс бинарных деревьев, где каждый узел может иметь не более 2-х потомков. В рекурсивном варианте определения бинарное дерево состоит из корня и 2-х бинарных поддеревьев: левого и правого, причем любое из них может быть пустым. Следующий рисунок иллюстрирует изображение бинарного дерева из 8-ми узлов A, B, C, D, E, F, G, H средствами теории графов.



*Рис. 3. Изображение бинарного дерева в теории графов.*

Несмотря на иерархическую структуру, бинарные деревья не являются подмножеством множества деревьев. Например, на следующем рисунке показаны 2 различных бинарных дерева, которые эквивалентны как корневые деревья.

*Рис. 4. Отличие корневых и бинарных деревьев*

Эти бинарные деревья различны по структуре, потому что корень первого из них имеет только левый потомок, а корень второго - только правый. Однако, как деревья они эквивалентны дереву из 2-х узлов А и В, которое изображено на том же рисунке справа. В общем случае различие между корневым деревом и бинарным деревом состоит в том, что каждый узел корневого дерева может иметь произвольное число поддеревьев, в то время как любой узел бинарного дерева может иметь только 0, 1 или 2 поддерева и существует различие между левыми и правыми поддеревьями.

Несмотря на эти отличия бинарные деревья могут быть использованы для представления корневых деревьев. Возможность такого представления устанавливает следующее естественное соответствие между корневыми и бинарными деревьями. Левая ветвь каждого узла соединяет его с первым узлом следующего уровня, а правая - с другими узлами следующего уровня (братьями). Следующий рисунок демонстрирует естественное соответствие 3-х уровневого корневого дерева его бинарному представлению.

|  |
| --- |
| http://bigor.bmstu.ru/?img/?doc=VU/044.mod/?n=5 |

*Рис. 5.  Естественное соответствие между корневым и бинарным деревьями.*

Естественное соответствие между корневыми и бинарными деревьями имеет важное значение в области информационных технологий, где программирование обработки данных может быть более просто реализовано для бинарных деревьев. Классическим примером иерархической обработки данных являются алгоритмы поиска по бинарному дереву и *прохождения бинарного дерева* в различном порядке.

**Динамическая реализация бинарных деревьев.**

Для программной реализации бинарных деревьев чаще всего используют динамические структуры данных, в которых каждый узел управляется записью из 2-х адресных и требуемого числа информационных полей. Адресные поля узла предназначены для хранения адресов левого и правого потомков. При отсутствии левого или правого потомка соответствующим адресным полям узла можно присвоить нулевые значения (NULL). Адресные поля имеют тип указатель на узел бинарного дерева. Информационные поля узлов могут иметь произвольный тип. Они используются для хранения данных и ключей в узлах бинарного дерева. Диаграмма логической структуры узла бинарного дерева представлена на следующем рисунке.

|  |
| --- |
| http://bigor.bmstu.ru/?img/?doc=VU/044.mod/?n=6 |

*Рис. 6.  Логическая структура узла бинарного дерева*

При создании нового узла бинарного дерева в динамической реализации, из *кучи* распределяется область памяти, объем которой необходим для хранения всех полей его логической структуры. Адрес области памяти, выделенной для нового узла, сохраняется в соответствующем (левом или правом) адресном поле предка.

Динамическое распределение узлов по адресным полям предков позволяет конструировать бинарные деревья с желаемой иерархической структурой информационных полей.

Например, на следующем рисунке приведена логическая структура динамической реализации семантического дерева грамматического разбора арифметического выражения: 7+6\*4.

|  |
| --- |
| http://bigor.bmstu.ru/?img/?doc=VU/044.mod/?n=7 |

**Рис. 7.  Логическая структура бинарного дерева**

**Базовый класс бинарных деревьев.**

Для объектно-ориентированной программной обработки [бинарных деревьев](javascript:termInfo(%22%D0%B1%D0%B8%D0%BD%D0%B0%D1%80%D0%BD%D1%8B%D1%85%20%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D1%8C%D0%B5%D0%B2%22)) целесообразно сконструировать базовый [класс](javascript:termInfo(%22%D0%BA%D0%BB%D0%B0%D1%81%D1%81%22)) BiNode с защищенными компонентными данными и общедоступными компонентными методами. Компонентные данные класса BiNode следует использовать для формального описания узлов абстрактного бинарного дерева, в котором узлы имеют только адресные поля для связи с другими узлами, но не имеют каких-либо информационных полей.

Общедоступная часть класса BiNode должна включать инвариантные компонентные методы прохождения и поиска по бинарным деревьям, которые не зависят от информационной природы узлов.

Класс BiNode не должен быть предназначен для самостоятельного применения, т. к. узлы дерева без информационных полей не имеют смысла, потому что нельзя никак идентифицировать, что они найдены или пройдены. По этой причине, класс BiNode следует рассматривать как [абстрактный класс](javascript:termInfo(%22%D0%B0%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D1%8B%D0%B9%20%D0%BA%D0%BB%D0%B0%D1%81%D1%81%22)), который не может быть реализован в объектах. Его следует применять только как базовый класс, который своими адресными полями обеспечивает поддержку иерархической структуры бинарного дерева производных узлов с любыми информационными полями и предоставляет инвариантные базовые методы древовидной обработки данных.

В терминах системы программирования **С++** логическая структура декларации базового класса BiNode должна иметь следующий формат:

*Пример 1*

**class BiNode**

**{**

**protected: /\*спецификация компонентных данных\*/**

**public: /\*спецификация компонентных методов\*/**

**};**

В защищенной (protected) части декларации класса BiNode рекомендуется специфицировать адресную пару компонентных данных:

*Пример 2*

**BiNode \* left; /\*адрес левого потомка\*/**

**BiNode \* right; /\*адрес правого потомка\*/**

Для хранения адресов потомков каждого узла бинарного дерева или нулевых (NULL) указателей, при отсутствии соответствующих потомков. Для инициализации этих адресов нулевыми значениями в классе BiNode рекомендуется предусмотреть и явно описать собственный конструктор без аргументов, который удобно специфицировать как inline-подстановку непосредственно в декларации класса.

В общедоступную (public) часть объявления класса BiNode нужно включить декларацию универсальных базовых компонентных методов, которые программно реализуют алгоритмы прохождения (*preorder*, *inorder* и *postorder*) и поиска по бинарному дереву (search). Эти компонентные методы должны предусматривать инвариантный доступ к узлам бинарных деревьев из объектов различных производных классов, который не зависит от специфики обработки их информационных полей.

Компонентный метод search предназначается для реализации алгоритма поиска по бинарному дереву в соответствии с заданным ключом, который следует передавать ему как аргумент. Компонентный метод search должен возвращать адрес узла бинарного дерева, который соответствует ключу, или нулевой указатель (*NULL*), если объект с заданным ключом не был обнаружен при поиске.

По этой причине естественным типом кода возврата компонентного метода search является указатель на базовый класс (BiNode\*). Передача ключа поиска должна быть организована по адресу. С этой целью у компонентного метода search следует предусмотреть аргумент типа (void\*). Выбор такого типа аргумента обеспечит его независимость от типа данных ключа поиска, потому что по указателю типа (void\*) можно передать адрес данных любого типа, например, адрес строки символов (char\*), если требуется организовать поиск по символическому ключу.

С учетом обозначенных предпочтений по типу аргумента и кода возврата, компонентный метод search может быть декларирован в объявлении класса BiNode следующим образом:

*Пример 3*

**class BiNode**

**{**

**...................**

**BiNode\* search (void\*);**

**};**

Основными элементами спецификации компонентного метода search должны быть: организация направленного перебора существующих узлов бинарного дерева для идентификации по заданному ключу поиска и вставка нового узла, когда ключевой образец поиска не обнаружен. Обеспечивая внешнюю независимость компонентного метода search от типа данных ключа (с помощью аргумента типа void\*), необходимо также выбрать для него инвариантную внутреннюю структуру, которая не зависит от специфики идентификации ключа и вставки нового узла при поиске по бинарному дереву, построенному из объектов любого производного класса. Желаемая инвариантность метода search может быть достигнута путем декларации в базовом классе BiNode виртуальных интерфейсов identify и insert, реализация которых планируется в производных классах. Базовый класс только объявляет *виртуальные интерфейсы*, устанавливая договоренность по типу аргумента и коду возврата.

Приведенное описание виртуальных интерфейсов базового класса, которые необходимы для реализации инвариантного поиска по бинарному дереву, отражает следующая декларация чистых виртуальных функций в классе BiNode:

*Пример 4*

**class BiNode**

**{**

**...................**

**virtual int identify (void\*) = 0;**

**virtual BiNode\* insert (void\*) = 0;**

**...................**

**};**

При обращении к чистой виртуальной функции базового класса выбирается подходящая реализация из производного класса, соответствующего по типу объекту, который инициирует ее вызов. В частности, вызовы компонентного метода search через указатели на объекты различных производных классов будут порождать вызовы различных производных реализаций чистых виртуальных функций identify и insert при обращении к ним через неявно передаваемый указатель this, который при такой схеме обращения сохраняет адрес объекта производного класса. Обращение к чистым виртуальным функциям через адрес объекта базового класса не имеет смысла из-за отсутствия их реализации в нем.

**Производный класс бинарных деревьев.**

Чтобы учесть специфику иерархической обработки для унификации и систематизации символьных строк в программе, целесообразно сформировать класс StrNode, производный от базового класса бинарных деревьев BiNode. Объекты класса StrNode должны описывать иерархию бинарного дерева, узлы которого предназначены для хранения текстовых строк. Производный класс StrNode должен наследовать компоненты базового класса и содержать собственные частные (private) компоненты-данные и общедоступные (public) компонентные методы, учитывающие особенности иерархической обработки символьных строк. В соответствии с правилами реализации принципа наследования в системе программирования **С++**, декларацию класса StrNode можно специфицировать следующим образом:

*Пример 5*

**class StrNode : public BiNode**

**{**

**private : /\*спецификация собственных компонент-данных\*/**

**public : /\*спецификация собственных компонентных методов\*/**

**};**

Категория наследования public выбрана для того, чтобы сохранить в производном классе привилегии доступа к компонентам, установленные в базовом классе.

*Пример 6*

**class BiNode**

**{**

**protected:**

**BiNode \*left;**

**BiNode \*right;**

**public:**

**BiNode() { left = right = NULL; }**

**void inorder (...);**

**void preorder (...);**

**void postorder(...);**

**BiNode\* search(void\*);**

**virtual int identify(void\*)=0;**

**virtual BiNode\* insert(void\*)=0;**

**};**

**class StrNode : public BiNode**

**{**

**char \*str;**

**...**

**public:**

**StrNode(const char \*s);**

**~StrNode()**

**{**

**cerr<<"String:\""<<str<<"\" is deleted."<<endl;**

**delete []str;**

**}**

**virtual int identify(void\* s)**

**{**

**return strcmp(str,(char\*)s);**

**}**

**virtual BiNode\* insert(void\* s)**

**{**

**return(new StrNode((char\*)s));**

**}**

**StrNode\* search(void\* s)**

**{**

**...**

**}**

**....**

**};**

## 11.4. Определение линейного списка. Программирование связных списков в C++

Линейный список представляет последовательность из n>0 узлов x[1], x[2], ... ,x[n], важнейшей структурной особенностью которой является такое расположение элементов списка один относительно другого, как будто они находятся на одной линии. Иначе говоря, в такой структуре следует соблюдать условие: если n>0 и x[1] является первым узлом, а x[n] – последним, то k-тый узел x[k] следует за x[k-1] узлом и перед x[k+1] узлом для любого k лежащего в промежутке от 1 до n.

С линейными списками могут выполняться следующие операции:

* получение доступа к x[k] элементу списка для проверки и/или изменения содержания его полей.
* вставка нового узла сразу до или после x[k] узла.
* удаление x[k] узла.
* объединение в один список двух или более линейных списков.
* разбиение линейного списка на два и более списка.
* создание копии линейного списка.
* определение количества узлов с списке.
* сортировка узлов в порядке возрастания значений в определенных полях этих узлов.
* поиск узла с заданным значением некоторого поля.

Линейные списки, в которых операции вставки, удаления и операции доступа к значению чаще всего выполняются в первом и последнем узле получили специальное название: стек – линейный список, в котором операции вставки и удаления (и как правило доступа к данным) выполняются только на одном из концов списка; очередь (односторонняя очередь) – линейный список, в котором все операции вставки выполняются на одном конце списка, а все операции удаления (и доступа, как правило) – на другом; дек (двухсторонняя очередь) – линейный список, в котором все операции вставки, удаления и доступа выполняются с обоих концов. Различают дек с ограниченным вводом и ограниченным выводом, в котором операция удаления и вставки элементов соответственно выполняется только на одном из концов.

**Программирование связных списков в C++.**

Под связными списками понимается набор элементов, причем каждый из них является часть узла Node, который также содержит ссылку Link.

struct node{Item item; node \*next;};

typedef node\* Link;

Соответственно узлы в связном списке ссылаются на узлы и поэтому связный список называется самоссылочным.

Простые линейные списки можно рассматривать как последовательность, в которой принято одно из следующих соглашений для ссылки последнего узла, а именно:

пустая (NULL) ссылка, не указывает ни на какой узел

ссылка, которая указывает на фиктивный узел, который не содержит элемента

ссылка, которая указывает на первый узел, делает список циклическим

В каждом случае отслеживание ссылок от первого элемента до последнего формирует последовательность расположения элементов.

При программировании связных списков, как только возникает необходимость использовать новый узел в списке, для него необходимо резервировать память.

При объявлении переменной типа node для нее резервируется память во время компиляции. Однако часто приходится организовывать вычисления, связанные с резервирование памяти во время выполнения посредством вызовов системных операторов управления памятью.

*Пример 1*

**link x = new node;**

**Struct node**

**{**

**Item item;**

**node \*next;**

**node (Item x; node \*t)**

**iteam = x;**

**next = t;**

**};**

**link t = new(x,t);**

*Если хотим удалить элемент*

**t = x -> next;**

**x -> next = t -> next;**

**или**

**x -> next = x -> next;**

**x -> next = t;**

*Если хотим добавить элемент*

**t->next = x -> next;**

**t ->next = t;**

*Пример 2*

*Пример циклического списка (задача Иосифа).*

**Struct node**

**{**

**Item item;**

**node \*next;**

**node (Item x; node \*t)**

**item = x;**

**next = t;**

**};**

**typedef node \*link;**

**int main(int argc; char \*argv[])**

**{**

**int i;**

**N = atoi(argv[1]);**

**M = atoi(argv[2]);**

**link t = new node(1,1);**

**t -> next = t;**

**link x = t;**

**for(i=2; i<=N; i++)**

**x=(x -> next = new node a,t);**

**while(x! = x -> next)**

**{**

**for(i=1; i<N; i++)**

**x = x -> next;**

**x -> next + x -> next -> next;**

**}**

**cout << x -> item << endl;**

**}**

**}**

**N=9 M=5**

**517436928**

**Соглашение о ведущем и завершающем узлах связных списков.**

**Список циклический (не бывает пустым)**

|  |  |
| --- | --- |
| Операция | Реализация |
| вставка t после i | t -> next = x -> next; |
| удаление после x | x-> next = t; x -> next = x -> next -> next; |
| цикл обхода | t = head; do {...t= t -> next;} while(t!= head) |
| проверка на наличие минимум одного элемента | if (head -> next == head) |

**Ведущий указатель, Null – указатель завершающего узла.**

|  |  |
| --- | --- |
| инициализация | head = 0 |
| вставка f после x | if ( x==0 || head = t) head -> next =0 else { t -> next = x -> next; x -> next =t; } |
| удаление после x | t = x -> next; x -> next = t -> next; |
| цикл обхода | for( t= head; t!=; t = t -> next) ... |
| проверка на пустоту | if(head == 0) |

**Фиктивный ведущий узел, Null – указатель завершающего узла**

|  |  |
| --- | --- |
| инициализация | head = new node; head -> next =0; |
| вставка t после x | t -> next = x -> next; x -> next = t; |
| удаление после x | t = x -> next; x -> next = t -> next; |
| цикл обхода | for( t = head ->next; t!0; t = t-> next) |
| проверка на пустоту | if( head -> next == 0) |

**Фиктивный ведущий и завершающий узлы.**

|  |  |
| --- | --- |
| инициализация | head = new node; z = new node; head -> next = z; z -> next -> z; |
| вставка t после x | t -> next = x -> next; x -> next = t; |
| удаление после x | x -> next = x -> next -> next; |
| цикл обхода | for (t = head -> next; t!= x; t=t -> next) |
| проверка на пустоту | if( head -> next == x) |

Для связных списков наиболее характерны следующие ошибки:

1) ссылка на неопределенный указатель

2) использование указателя, который изменяется неизвестным образом

Причина этих ошибок – возможное присутствие указателей на один и тот же узел.

## 11.5. Линейные связные списки. Двусвязные списки

Линейный связный список - последовательность ячеек, связанных в некотором порядке. Каждая ячейка имеет два поля: в одном из них находится элемент списка, в другом – указатель, сообщающий положение следующей ячейки.

*Пример 1*

**позиция = 1 2 3 4 5 6 7 8 9 [i]**

**A[i] = x b x d a x c x x**

**Next[i] = x 7 x 0 2 x 4 x x**

**IP = 5**

**признак конца = 0**

**A[3] <- e**

**Next[3] <- Next[2]**

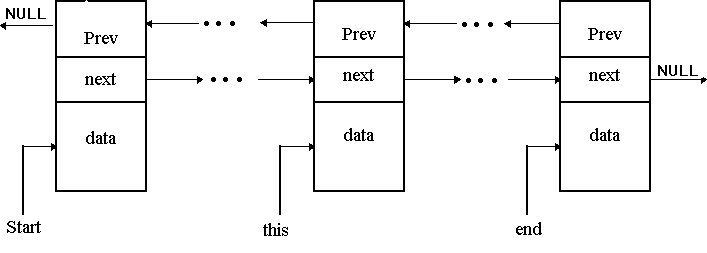
**Next[2] <- 3**

*Если хотим удалить символ 7, то*

**Next[2] <- [7]**

**Двусвязные списки**

Двусвязные (двунаправленные) линейные списки - линейные списки, записи которых связаны посредством пары указателей, хранимых в адресных полях записей списка. Логическую структуру двусвязного списка иллюстрирует следующая диаграмма



**Рис. 1.**

В каждой записи поле prev содержит адрес предыдущей записи, поле - next - адрес последующей записи, а data обозначает информационные поля. Для доступа к списку могут быть использованы адреса начальной (start), конечной (end) и текущей (this) записей списка. Индикатором начальной и конечной записей являются нулевые (NULL) значения адресных полей Start и end, соответственно.

Любая функциональная обработка списка строится на основе процедур модификации и просмотра. *Процедуры модификации списка* должны обеспечивать вставки и исключение записей списка. Частным случаем процедур вставки и исключения являются процедуры добавить и удалить начальную или конечную запись списка. Следующая диаграмма иллюстрирует процедуру вставки новой записи Z после записи Y (или перед записью X)

|  |
| --- |
| http://bigor.bmstu.ru/?img/?doc=VU/040.mod/?n=2 |

**Рис. 2.**

Следующая диаграмма иллюстрирует процедуру исключения текущей записи this.

|  |
| --- |
| http://bigor.bmstu.ru/?img/?doc=VU/040.mod/?n=3 |

**Рис. 3.**

*Процедуры просмотра списка* должны обеспечивать смещение указателя текущей записи (this) на требуемое число позиций в направлении начала или конца списка, как показано на следующей диаграмме.

|  |
| --- |
| http://bigor.bmstu.ru/?img/?doc=VU/040.mod/?n=4 |

**Рис. 4.**

Частным случаем смещения указателя текущей записи является переход к соседней предыдущей или последующей записи (смещение на 1 позицию), а также нефиксированный переход к начальной или конечной записям списка, который позволяет принудительно инициализировать указатели start и end после модификаций в начале и конце списка.

## 11.6. Хранение целых списков

Целый список - линейный список, элементы которого принадлежат множеству натуральных чисел, причем некоторые элементы могут повторяться. В целых списках m – число элементов, n – размах списка, тогда разреженный список тот, в котором m<<n. Для хранения разреженных списков в компактной форме используются массивы.

Если требуется сохранить упорядоченность целых чисел, то включение и исключение элемента из какой-либо позиции вызывает сдвиг всех элементов, находящихся справа от нее. Причем целые списки могут хранить как связные линейные списки, так и кольцевые списки, посредством массива A длины не меньшей m и соответствующего дополнительного массива Next. Этот тип хранения также является компактным.

Альтернативой может служить метод хранения с использованием одного массива A, длина которого соответствует размаху N. Число, хранимое в каждой позиции A дает одновременно значение элемента списка и адрес следующего элемента, в этом случае Next оказывается излишним. Дополнительный указатель IP задает позицию первого числа в списке. Этот тип хранения называется расширенным

# Глава 12. Классы ввода-вывода

## 12.1. Классы стандартной библиотеки

В C++ классы стандартной библиотеки разделяются на группы в соответствии со своим назначением.

1. Потоковые классы – предназначены для управления потоками данных между оперативной памятью и внешними устройствами, а также в пределах оперативной памяти.
2. Строковые классы – предназначены для удобной защищенной от ошибок работы с символьными строками
3. Контейнерные классы – реализуют наиболее распространенные структуры для хранения данных (списки, векторы, очереди). В библиотеки контейнерных классов также входят алгоритмы, использующие эти контейнеры.
4. Итераторы – предназначены для унификации доступа к элементам контейнерных и других классов.
5. Математические классы – поддерживают эффективную обработку массивов численных данных с плавающей точкой и работу с комплексными числами.
6. Диагностические классы – обеспечивают динамическую идентификацию типов и объектно-ориентированную обработку ошибок.
7. Остальные классы – обеспечивают динамическое распределение памяти и адаптацию к локальным особенностям программы, обработку функций, объектов и т.д.

Контейнерные классы, итераторы и алгоритмы классов объединяют в библиотеки шаблонов.

## 12.2. Потоковые классы.

Потоки в C++ отличаются от функций ввода/вывода в C, обеспечивая работу как со стандартными потоками данных, так и с типами данных, определяемыми пользователем, а также обеспечивая единообразный и понятный синтаксис. Чтение данных из потока называется извлечением, а вывод данных в поток – включением. Поток в C++ - это последовательность байтов, независимых от конкретного устройства, с которого производится считывание данных.

Обмен с потоком для увеличения скорости передачи данных производится через специальную область памяти – буфер. Передача данных выполняется при выводе после заполнения буфера и при вводе, если буфер исчерпан. По направлению обмена данных потоки делят на три группы:

* входные потоки (позволяющие вводить данные в память)
* выходные потоки (осуществляющие вывод данных из памяти)
* двунаправленные потоки

В зависимости от вида устройства, с которым работает поток данных, их делят на:

* стандартные потоки
* файловые потоки
* строковые потоки

Стандартные потоки предназначены для передачи данных с клавиатуры на экран (это: stdin - стандартный поток ввода данных, stdout - стандартный поток вывода данных и stderr - стандартный поток ошибок). Файловые потоки – для обмена информацией с файлами. Строковые потоки – для работы с массивами символов в оперативной памяти. Для поддержки этих потоков в C++ стандартная библиотека содержит иерархию классов, построенную на основе двух базовых классов:

ios – базовый класс, содержащий общие для ввода/вывода поля и методы

streambuf – обеспечивает буферизацию потоков и их взаимодействие с физическими устройствами.

От этих базовых классов наследуются классы istream и ostream для входных и выходных потоков соответственно. Эти потоки являются базовыми для iostream, который позволяет реализовывать двунаправленные потоки. Ниже в иерархии находятся файловые и строковые потоки:

* isstrinstream – класс входного строкового потока
* osstringstream – класс выходного строкового потока
* stringstream – класс двунаправленного строкового потока
* ifsteam – класс входных файловых потоков
* ofstream – класс выходных файловых потоков
* fstream – класс двунаправленных файловых потоков

**Стандартный поток**

Чтобы использовать стандартные потоки ввода-вывода нужно включать заголовочный файл <iostream.h>.Заголовочный файл <iostream.h> кроме описания потоков ввода-вывода содержит описание ещё и предопределенных объектов.

|  |  |  |
| --- | --- | --- |
| **Объект** | **Класс** | **Описание** |
| cin | istream | связывается с клавиатурой (со стандартным буфером ввода) |
| cout | ostream | связывается с экраном (со стандартным буфером вывода) |
| cerr | ostream | связывается с экраном (стандартный не буферизованный вывод, куда направляются сообщения об ошибках) |
| clog | ostream | связывается с экраном (стандартный буферизованный вывод, куда направляются сообщения об ошибках) |

Эти объекты создаются при включении в программу файла iostream. При этом становятся доступными средства ввода-вывода. Соответствующие операции << и >> определены путем перегрузки операции сдвига.

*Пример 1*

**#include<iostream.h>**

**int main()**

**{**

**int i;**

**cin>>i;**

**cout << ”Output i = ”<< i;**

**return 0;**

**}**

Операции извлечения и чтения в качестве результата своего выполнения формирует ссылку на объект типа istream для извлечения и на объект типа ostream для чтения.

Файловые потоки.

Под файлом обычно подразумевается поименованная информация на внешнем носителе. Практически в С++ файл рассматривается как последовательный поток байтов. Каждый файл завершается маркером EOF или указанием числа байтов, записанных в служебную структуру данных, поддерживаемую соответствующей системой.

По способу доступа файлы делятся на последовательные и файлы с произвольным доступом. Файлы с последовательным доступом – файлы, в которых чтение и запись производятся сначала, байт за байтом. Файлы с произвольным доступом допускают чтение и запись в произвольной позиции. Стандартная библиотека содержит три класса для работы с файлами:

* Ifstream
* ofstream
* fstream

Наследуют перегруженные операции <<, >>. Использование файловых потоков в программе предполагает выполнение следующих операций:

1) создание потока

2) открытие потока и связывание его с файлом

3) обмен данных (ввод / вывод)

4) уничтожение потока

5) закрытие файла

Каждый класс файловых потоков содержит конструкторы, с помощью которых можно создавать объекты этих классов различными способами. Конструкторы без параметров создают объект соответствующего класса, не связывая его с файлом. Конструктор с параметром создает объект соответствующего класса, открывает файл с указанным именем и связывает файл с объектом.

ifstream(char \*name, int mode = ios::in);

ofstream(char \*name, int mode = ios::out);

fstream(char \*name, int mode = ios::in [ios::out] );

Вторым параметром конструктора является режим открытия файла. Если установленное по умолчанию значение не устраивает программиста, можно указать другой, составив его из битовых масок, определенных в классе ios.

|  |  |
| --- | --- |
| in | = 0x01; //открыт для чтения |
| out | = 0x02; // открыт для записи |
| ate | = 0x04; // установить указатель на конец файла |
| app | = 0x08; // открыть на добавление в конец |
| trune | =0x19; // если файл существует, удалить |
| nocreate | = 0x20; // если файл не существует, выдать ошибку |
| noreplace | = 0x40; // если файл существует, выдать ошибку |
| binary | = 0x80; // открыть в двоичном режиме |

Открыть файл в программе можно с использованием конструктора, либо методом open. В open должны быть такие же параметры, как и в соответствующем конструкторе.

*Пример 2*

**ifstream inpf(‘input.txt’, .ios::n| ios::nocreate); // использование конструктора**

**if(!inpf)**

**{**

**cout << “ невозможно открыть файл для чтения”;**

**return 1;**

**}**

**ofstream f;**

**fopen(“output.txt”); // использование метода open**

**if(!f)**

**{**

**cout<< “ невозможно открыть файл для записи ”;**

**return 1;**

**}**

Чтение и запись выполняются с помощью операций чтения и извлечения аналогичных потоковым классам, либо с помощью методов класса.

*Пример 3*

**#include<fstream.h>**

**int main()**

**{**

**char text[81], buf[81];**

**cout<<” введите имя файла “;**

**cin>>text;**

**ifstream(text, ios::in| ios::nocreate);**

**if(!f)**

**{**

**cout<<” ошибка открытия файла “;**

**return 1;**

**}**

**while(!f, eof())**

**{**

**f.getline(buf, 81);**

**cout<<buf<<endl;**

**}**

**return 0;**

**}**

Для закрытия потока определен метод close, но поскольку он неявно выполняется деструктором, то явный вызов требуется, когда необходимо закрыть поток раньше конца его области видимости.

**Строковые потоки**

Строковые потоки позволяют считывать и записывать информацию из области оперативной памяти так же как из файла. консоли, экрана.

В стандартных библиотеках в потоковых потока определено три класса:

**isstringstream – входной**

**ostringstream – выходной**

**stringstream – двунаправленный**

Эти классы определяются в заголовочном файле <sstream>. Они являются производными от классов istream, ostream, iostream. Соответственно наследуют << и >>, флаги форматирования, манипуляторы.

Строковые классы определяются в соответствии с правилами C++, строковые потоки создаются с соответствующими участками памяти с помощью конструкторов.

**explicit istringstream (int mode = ios::in);**

**explicit istringstream (const string &name, int mode = ios::in);**

**explicit ostringstream (int mode = ios::out);**

**explicit ostringstream (const string & name, int mode = ios::out);**

**explicit stringstream (int mode = ios::in | ios::out );**

**explicit stringstream (const strin &name, int mode = ios::in | ios::out);**

Строковые потоки являются аналогами sprintf, sscanf, и могут использоваться для преобразования данных, когда они заносятся в некоторый участок памяти, а затем считываются в величины требуемых типов. Эти потоки могут применятся для обмена информацией между модулями программы.

В строковых потоках описан метод str, возвращающий копию строки или устанавливающий ее значение.

**string str() const;**

**void str(const string &s);**

Проверять строковый поток на переполнения не требуется, т.к. размер строки изменятся динамически.

*Пример 4*

*Пример программы, в которой строковый поток используется для формирования сообщений, включающих текущее время, и передает в качестве параметра.*

**#include<sstream>**

**#include<string>**

**#include<iostream>**

**#include<time>**

**using namespace std;**

**string message(int i;)**

**{**

**osting stream os;**

**time\_t t;**

**time(&t);**

**os<<”time”<< ctime(&t) << “number:“ << i << endl;**

**return os.str();**

**}**

**int main()**

**{**

**cout << message (22);**

**return 0;**

**}**

**Потоки и типы, определяемые пользователем.**

Для ввода-вывода в потоках, определяемых пользователем, также используется перегрузка для всех стандартных типов операций чтения и извлечения, при этом выбор конкретной операции определяется типом фактических параметров.

Чтобы вводить/выводить величины типов, определяемых пользователем, требуется перегрузить эти операции. Это бинарные операции, левым операндом которых является объект-поток, а правым – объект, который требуется извлечь/поместить в этот поток. Возвращенное значение должно быть ссылкой на поток, чтобы можно было организовать цепочки операций.

*Пример 5*

**class b1**

**{**

**...**

**int x;**

**float y;**

**...**

**//Чтобы вводить/выводить объекты класса b1 требуется определить в классе b1 операции следующего вида**

**friends ostream &operator << (ostream out, b1 &c)**

**{**

**return out << “ x= “ << c.x << “ y= “ << c.y;**

**}**

**friend istream &operator >> (istream in, b1 &c)**

**{**

**cout << “Enter x:”; in>> c.x;**

**cout << “Enter y:”; in>> c.y;**

**return in;**

**}**

**...**

**};**

После того как в программе определены операции ввода/вывода, в данной программе можно использовать объекты класса b1 в операторах ввода/вывода наряду с величинами стандартных типов.

*Пример 6*

**#include<iostream.h>**

**class b1**

**{**

**int x;**

**float y;**

**public:**

**b1(int nx = 1, float ny = 0.01)**

**{**

**x=nx;**

**y=ny;**

**}**

**friends ostream &operator << (ostream out, b1 &c)**

**{**

**return out << “ x= “ << c.x << “ y= “ << c.y;**

**}**

**friend istream &operator >> (istream in, b1 &c)**

**{**

**cout << “Enter x:”; in>> c.x;**

**cout << “Enter y:”; in>> c.y;**

**return in;**

**}**

**};**

**int main()**

**{**

**b1 c;**

**cout << c << endl;**

**b1 c1(100,100);**

**cout << c1 << endl;**

**b1 c2;**

**cin >> c2;**

**cout << c2 << endl;**

**return 0;**

**}**

## 12.3. Форматирование данных

В потоковых классах форматирование данных можно осуществить тремя способами, а именно, с помощью:

* флагов
* манипуляторов
* форматирующих методов

**Форматирование данных с помощью флагов.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Флаги** | **Положение** | **Умолчание** | **Описание действий при установленном бите** |
| skipws | 0x0001 | + | при извлечении пробельные символы игнорируются |
| test | 0x0002 |  | выравнивание по левому краю |
| right | 0x0004 | + | выравнивание по правому краю |
| internal | 0x0008 |  | знак числа выводится по левому краю, а число по правому. Промежуток заполняется символами Х (по умолчанию – пробелами) |
| dec | 0x0010 | + | десятичная система счисления |
| oct | 0x0020 |  | восьмеричная |
| hex | 0x0040 |  | шестнадцатеричная |
| showbase | 0x0080 |  | вывод основания системы счисления: ***0х*** – шестнадцатеричная ***0*** – восьмеричная |
| showpoint | 0x0100 |  | при выводе вещественных чисел печатает десятичную точку и дробную часть |
| uppercase | 0x0200 |  | при выводе использует символы верхнего регистра |
| showpos | 0x0400 |  | печатает знак при выводе положительных чисел |
| scientific | 0x0800 |  | печатает веществ числа в форме мантиссы с порядком |
| fixed | 0x1000 |  | печатает вещественные числа в форме с фиксированной точкой, точность определяется полем precision |
| unitouf | 0x2000 |  | выгружает буферы всех потоков после каждого вывода |
| stdio | 0x4000 |  | выгружает буферы из потоков [stdout](javascript:termInfo(%22stdout%22)) и [stderr](javascript:termInfo(%22stderr%22)) после каждого вывода |

Флаги

1) left, right, internal

2) dec, oct, hex

3) scientific, fixed

взаимно исключают друг друга, то есть в каждый момент времени может быть установлен только один флаг из каждой группы.

Для управления флагами в классе ios есть методы: flags, setf, unsetf.

* longios::flags(); - возвращает текущие флаги потока
* longios::flags(long); - присваивает флагам значения параметров
* longios::setf(long long); - присваивает флагам, биты которых в первом параметре, соответствующие значения во втором
* longios::setf(long); - устанавливает флаги, биты которых установлены в параметре
* longios::unsetf(long); - сбрасывает флаги, биты которых установлены в параметрах

Кроме флагов для форматирования используются следующие поля классов ios:

* int x\_width – минимальная ширина поля ввода
* int x\_precision – количество цифр в дробной части при выводе вещественных чисел с фиксированной точкой, либо общее количество значащих цифр при выводе числа с мантиссой и порядком
* int x\_fill – символ заполнения поля ввода.

Для управления этими полями, используются методы width, precision, fill.

**int ios:: width()**

возвращает значение ширины поля ввода

**int ios:: width(int)**

устанавливает значение ширины поля ввода в соответствии с параметром

**int ios:: precision()**

возвращает значение точности представления при выводе вещественного числа

**int ios:: precision(int)**

устанавливает значение точности представления при выводе вещественного числа и возвращает старое значение точности

**char fill()**

возвращает текущий символ заполнения

**char fill(char)**

устанавливает значение текущего символа заполнения и возвращает старое значение символа.

Перед установкой некоторых флагов требуется сбросить флаги, которые не могут быть установлены одновременно с ними. Для этого удобно воспользоваться вторым параметром метода setf.

**adjustfield(left| right| internal);**

**basefield(dec |oct | hex );**

**floatfield(scientific | fixed );**

*Пример 1*

*Пример форматирования при выводе с помощью флагов и методов:*

**#include<iostream.h>**

**int main()**

**{**

**long a = 1000, b = 077;**

**cout.width(7);**

**cout self(ios::hex| ios::showbase| ios::uppercase);**

**cout << a;**

**cout width(7);**

**cout<< b << endl;**

**double d=0.12, c=1.3\*d-4;**

**cout self(ios::left);**

**cout << d << endl;**

**cout << c;**

**return 0;**

**}**

## 12.4. Манипуляторы

Манипуляторы можно разделить на две группы: простые и параметризированные.

Простые манипуляторы не требуют указания аргументов, а параметризованные – требуют.

*Простые манипуляторы*

* dec – простой манипулятор, устанавливающий при вводе/выводе флаг десятичной системы счисления.
* oct – простой манипулятор, устанавливающий при вводе/выводе флаг восьмеричной системы счисления.
* hex – простой манипулятор, устанавливающий при вводе/выводе флаг шестнадцатеричной системы счисления.
* ws – устанавливает при вводе/выводе извлечение пробельных символов
* endl – при выводе включает в поток символ новой строки и выгружает буфер
* ends – при выводе включает в поток нулевой символ

Изменение системы счисления действует до следующего явного изменения. Параметризованный манипулятор требует указания аргумента, для его использования требуется подключить заголовочный файл <iomanip>

* setbase(int n) – задает основание системы счисления;
* resetiosflags(long) – сбрасывает флаги состояния поток, биты которых установлены в параметре.
* setiosflags(long) – устанавливает флаги состояния потока биты которых в параметре равны единице.
* setfill(int) – устанавливает символ заполнитель с кодом, равным значению параметра.
* setprecision – устанавливает максимальной количество цифр в дробной части для вещественных чисел в
* форме с фиксированной точкой, либо общее число значащих цифр для чисел в форме с мантиссой и
* порядком.
* setw(int) – устанавливает максимальную ширину поля вывода.

*Пример 1*

**#include<iostream.h>**

**#include<iomanip.h>**

**int main()**

**{**

**double d[] = {1.234, -12.34567, 123.456789, -1.234, 0.00001};**

**couy<<setfil('.')<<setprecision(4)<<setioflags(ios::showpoint| ios""fixed);**

**for(int i=0; i<5; i++)**

**cout << setw(12) << d[i] << endl;**

**return 0;**

**}**

*Результаты:*

**1.2340**

**-12.3457**

**123.458**

**-1.2340**

**0.0000**

## 12.5. Методы обмена с потоками

В потоковых классах наряду с операторами извлечения из потока и включения в поток определены также методы для неформатированного чтения (неформатированного ввода) и записи в поток (неформатированного вывода). При этом преобразование данных не выполняется.

Функции чтения, определенные в классе istream:

* gcout – возвращает количество символов, считанных с помощью последней функции неформатированного ввода.
* get() – возвращает код извлеченного из потока символа или EOF.
* get(c) – возвращает ссылку на поток, из которого выполнялось чтение, и записывает извлеченный символ в c.
* get (buf, num, lim= ) – считывает num-1 символов или пока не встретится символ lim и копирует их в символьную строку buf, вместо символа lim в строку записывается признак конца строки. Символ lim остается в потоке. Возвращает ссылку на текущий поток.
* getline(buf, num, lim = ) – аналогична функции get(...) копирует в строку и символ lim.
* ignore(num = 1, lim = EOF) – считывает и пропускает символы до тех пор, пока не будет прочитано num символов или не встретится разделитель, заданный параметром lim. Возвращает ссылку на текущий поток.
* peek() – возвращает следующий символ без удаления его из потока или EOF, если достигнут конец файла
* putback(c) – помещает в поток символ с, который становится текущим при извлечении его из потока.
* read(buf, num) – считывает num символов, или все символы до конца файла, если их меньше num соответственно в символьный массив buf и возвращает ссылку на текущий поток.
* readsome(buf, num) – считывает num символов или все символы, если их меньше num в массив buf и возвращает количество символов.
* seekg (pos) – устанавливает текущую позицию чтения в значение pos.
* seekq (offs, org) – перемещает текущую позицию чтения на offs байтов, считая от одной из трех позиций, определенных вторым параметром (ios::cur – от текущей позиции, ios::end – от конца файла).
* tellq() – возвращает текущую позицию чтения потока.
* unget() – помещает последний прочитанный символ в поток и возвращает ссылку на текущий поток.

В классе ostream определены аналогичные функции для неформатированного вывода.

* flush(c) – описывает содержание потока выводя на физическое устройство.
* put(c) – выводит в поток символ с и возвращает ссылку на поток.
* seek(pos) – устанавливает позицию записи в значение pos.
* seek(offs, org) – перемещает текущую позицию записи на offs байтов, считая от одной из трех позиций, определенных вторым параметром (ios::beg, ios::cur, ios::end).
* tellq() – возвращает текущую позицию записи потока.
* write(buf, num) – записывает в поток num символов из массива buf и возвращает ссылку на поток.

*Пример 1*

*Пример программы, считывающей строки из массива buf и возвращающей ссылку на поток:*

**#include<iostream.h>**

**int main()**

**{**

**const int n=20, len = 100;**

**char str[len][n] int i =0;**

**while(cin, getline(str[i], len, )&&i<n)**

**{**

**...**

**i++;**

**}**

**return 0;**

**}**

*Пример 2*

*Пример программы, записывающей в файл число с плавающей точкой в строку символов. Программа осуществляет считывание из файла и вывод на экран.*

**#include<fstream.h>**

**#include<string.h>**

**int main()**

**{**

**//запись в файл**

**ofstream out(“text”);**

**if(!out)**

**{**

**cout<<”cannot open file test for writing” << endl;**

**return 1;**

**}**

**double num = 100.45;**

**char str[]=” this is a test”;**

**out.write(reinterpret\_cast<char \*>(num), sizeof(double));**

**out.write (str, strlen(str));**

**out.close();**

**//чтение из файла**

**ifstream in(“test”, ios::n| ios::nocreate);**

**if(!in)**

**{**

**cout << “cannot open file test for reading” << endl;**

**return 1;**

**}**

**double check\_num;**

**char check\_str[60];**

**in.read(reinterpret\_cast<char \*> (& check\_num), sizeof(double));**

**in.read(check\_str, 60);**

**int lstr = in.ycout(); // количество прочитанных символов**

**check\_str[lstr]=0; // занести нуль-символ в конец строки**

**cout << check\_num << ‘ ‘ << check\_str << endl;**

**in.close();**

**return 0;**

**}**

Приведение типа reinterpret\_cast <char \*> при вызове функций write и read необходимо в тех случаях, когда параметр не является символьным массивом.

*Пример 3*

*Пример программы, в которой формируется файл test, в который выводятся три строки:*

**#include<fstream.h>**

**#include<string.h>**

**int main()**

**{**

**// запись в файл**

**ofstream out(“test”);**

**if(!out)**

**{**

**cout<<”cannot open file test for writing” << endl;**

**return 1;**

**}**

**char \*str[] = {“line 1”, “line 2”, “line 3”};**

**for(int i=0; i<3; ++i)**

**{**

**out.write(str[i], strlen(str[i]));**

**out.put();**

**}**

**out.close();**

**//чтение файла**

**ifstream in(“test”, ios::in| ios::nocreate);**

**if(!n)**

**{**

**cout << “cannot open file test for reading” << endl;**

**return 1;**

**}**

**char check\_str[3][60];**

**for(i=0; i<3; ++i)**

**{**

**in.get(check.str[], 60);**

**in.get();**

**}**

**//контрольный вывод**

**for(i=0; i<3; ++i)**

**cout << check\_str[i] << endl;**

**in.close();**

**return 0;**

**}**

После выполнения функции символ-разделитель остается во входном потоке, следовательно, необходим вызов функции get для пропуска одного символа. Альтернативный способ – использование функции getline, которая извлекает символ-разделитель из потока.

## 12.6. Классы с самоадресацией

Классы с самоадресацией - классы, которые содержат элемент-указатель, который указывает на объект того же типа класса:

*Пример 1*

**class Node**

**{**

**public:**

**Node(int);**

**void setdata(int);**

**int getdata() const;**

**void setnextptr(const Node \*);**

**const Node \* getnextptr() const;**

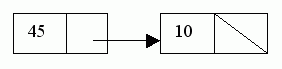
**private:**

**int data;**

**Node \* nextptr;**

**};**

Данное описание определяет тип класса Node – он имеет два закрытых элемента данных: целый элемент data и указатель nextptr, который указывает на объект типа Node, т.е. объект того же тип. Такие классы называются классами с самоадресацией. В них элемент nextptr используется как связывающий, т.е. nextptr может быть использован для связи объекта типа Node с объектом того же типа. У типа Node также имеется пять функций-элементов: конструктор, который принимает данные целого типа, для инициализации данных поля data; функция для установки значения data; функция getdata, возвращающая значение data : функция setnextptr для установки значения указателя nextptr, а также функция getnextptr, которая возвращает значение указателя nextptr. Объекты классов с самоадресацией могут связываться друг с другом, формируя абстрактные структуры данных: связные списки, очереди, стеки, деревья.



На рисунке показано два объекта класса с самоадресацией, связанные вместе для создания структуры «список». Обратный слеш отображает нулевой указатель, помещенный в элемент связи второго объекта, чтобы было ясно, что эта связь не указывает на какой-либо другой объект.

Типичная ошибка: указатель связи в последнем узле не установлен на ноль.

# Глава 13. Взаимосвязь и различия C и C++

## 13.1. Области видимости и время жизни переменных

В языках С и С++ определены правила обзора, которые устанавливают такие понятия, как видимость и время жизни (существования) объектов. В самом общем смысле достаточно различать две области видимости - глобальную и локальную.

Глобальная область видимости существует вне всех других областей. Имя, объявленное в глобальной области, известно всей программе. Например, глобальная переменная доступна для использования всеми функциями в программе. Глобальные переменные существуют на протяжении всего жизненного цикла программы.

Локальная область видимости определяется границами блока, т.е. начинается с открывающей фигурной скобки ({) и оканчивается закрывающей фигурной скобкой (}). Имя, объявленное в локальной области, известно только внутри этой области. Поскольку блоки могут быть вложенными, локальные области также могут быть вложенными. Самой распространенной локальной областью является область, определенная функцией. Локальные переменные создаются при входе в их блок, а разрушаются при выходе из блока. Это означает, что их значения не хранятся между вызовами функций. Чтобы сохранить значения переменных между вызовами, можно использовать модификатор static.

В языке С++ и версии С99 локальные переменные можно объявлять практически в любом месте блока, а в версии С89 они должны быть объявлены в начале блока до выполнения каких-либо инструкций "действия".

*Пример 1*

**void f(int a)**

**{**

**int a;**

**a = 10;**

**int b; // ОК для С++ и С99, но не для С89**

**....**

**}**

Глобальная переменная должна быть объявлена вне всех функций, включая main(). Глобальные переменные, как правило, размещаются в начале файла перед функцией main(), т.к. переменная должна быть объявлена до ее использования; кроме того, размещение глобальных переменных в определенном месте облегчает чтение программы.

Формальные параметры функции также являются локальными переменными, и их можно использовать не только как средство для получения значений аргументов при вызове функции, но и как локальные переменные.

## 13.2. Оператор разрешения области видимости (::). Пространство имён

**Оператор разрешения области видимости (::)**

Оператор разрешения области видимости (::) задает область видимости, которой принадлежит член и имеет следующую форму записи:

**имя::имя\_члена**

Здесь имя задает имя класса или пространства имен, в котором содержится член, заданный элементом имя\_члена. Другими словами, имя определяет область видимости, внутри которой может находиться идентификатор, заданный элементом имя\_члена.

При ссылке на глобальную область видимости элемент имя задавать не нужно. Например, для обращения к глобальной переменной a, которая "скрыта" за локальной переменной с таким же именем a, нужно использовать оператор :: следующим образом: ::a

**Пространства имен**

В языке С++ можно создать локальную область видимости, используя ключевое слово namespace. Пространство имен определяет некоторую декларативную область. Ее назначение - ограничить действие имен. Общая форма задания пространства имен такова:

**namespace <имя\_пространства\_имен> {**

**//.....**

**}**

*Пример 1*

*создается пространство имен MyNameSpace, а внутри него объявляется переменная count*

**namespace MyNameSpace {**

**int count;**

**}**

На имена, объявленные в пространстве имен, могут напрямую ссылаться другие инструкции внутри того же пространства имен. Вне своего пространства имен к именам можно получить доступ двумя путями. Во-первых, использовав оператор разрешения области видимости (::)

*Пример 2*

**MyNameSpace::count = 10;**

Во-вторых, можно использовать инструкцию using, которая привносит заданное имя или пространство имен в текущую область видимости.

*Пример 3*

**using namespace MyNameSpace;**

**count = 100;**

**// В данном случае к переменной можно обращаться напрямую, т.к. теперь она относится к текущей области видимости.**

При появлении языка С++ элементы, объявленные в библиотеке С++, относились к глобальному (т.е. безымянному) пространству имен. Однако ныне действующий стандарт С++ все эти элементы относит к пространству имен std.

## 13.3. Булевы типы данных

В C++ булев тип данных именуется bool, а слова true и false являются ключевыми.

В C булев тип данных именуется \_Bool, но при включении заголовочного файла <stdbool.h> можно пользоваться ключевыми словами bool, true, false.

Объект типа bool может принимать одно из двух значений: true и false.

Например:

*Пример 1*

**// инициализация строки**

**string search\_word = get\_word();**

**// инициализация переменной found**

**bool found = false;**

**string next\_word;**

**while ( cin >> next\_word )**

**if ( next\_word == search\_word ) found = true;**

**// ...**

**// сокращенная запись: if ( found == true )**

**if ( found )**

**cout << "ok, мы нашли слово\n";**

**else cout << "нет, наше слово не встретилось.\n";**

Хотя bool относится к одному из целых типов, он не может быть объявлен как signed, unsigned, short или long, поэтому приведенное определение ошибочно:

*Пример 2*

**// ошибка**

**short bool found = false;**

Объекты типа bool неявно преобразуются в тип int. Значение true превращается в 1, а false – в 0. Например:

*Пример 3*

**bool found = false;**

**int occurrence\_count = 0;**

**while ( /\* mumble \*/ )**

**{ found = look\_for( /\* something \*/ );**

**// значение found преобразуется в 0 или 1**

**occurrence\_count += found; }**

Таким же образом значения целых типов и указателей могут быть преобразованы в значения типа bool. При этом 0 интерпретируется как false, а все остальное как true:

*Пример 4*

**// возвращает количество вхождений**

**extern int find( const string& );**

**bool found = false;**

**if ( found = find( "rosebud" ))**

**// правильно: found == true**

**// возвращает указатель на элемент**

**extern int\* find( int value );**

**if ( found = find( 1024 ))**

**// правильно: found == true**

## 13.4. Модификатор const

В C++ глобальные константы связаны внешним образом, а в C – внутренним

*Пример 1*

**const double PI = 3.1415926 <=> static const double PI = 3.1415926**

(Если оба объявления не включены в состав функции.)

Правило, существующее в C++, нацелено на упрощение использования констант в заголовочных файлах. Если константа связана внутренним образом, тогда каждый файл, включающий заголовочный файл, получает свою копию константы. Если константа связана внешним образом, тогда в одном из файлов должно существовать определенное объявление, в то время как остальные файлы должны содержать ссылочные объявления на файл, содержащий ключевое слово external. В некоторых случаях в C++ можно использовать слово extern, чтобы создавать значения const, имеющие внешнее связывание. Тогда в обоих языках могут быть созданы константы, связанные внешним и внутренним образом.

Различие между C и C++ заключается только в том, какого типа константы используются по умолчанию.

*Пример 2*

**const int APS = 100;**

**double loon[APS]; //допустима в С++, но недопустима в С.**

## 13.5. Константы char

Язык C считает константы char константами целочисленными, а C++ относит их к типу char

*Пример 1*

**char ch = ‘B’;**

В C const B будет отведен код памяти int, такое же численное значение хранится в переменной ch, но здесь оно занимает один байт памяти. В C++ используется один байт памяти как для В, так и для ch ( sizeof(char)<=sizeof(int) ).

*Пример 2*

**char ch = ‘B’;**

**sizeof('B') = 1;**

**sizeof(ch) = 1;**

## 13.6. Комплексные типы данных

В C++ поддерживаются комплексные типы данных с помощью комплексного класса, который определяется в <complex>. В C существуют встроенные комплексные типы данных, а их поддержка осуществляется с помощью заголовочного файла <complex.h>. Эти подходы различны и несовместимы.

Версия C отражает большую зависимость от непосредственных запросов вычислений, производящих числовые расчеты.

*Пример 1*

**# include "complex.h"**

**complex c = 7.4; // инициализация 7.4+0i**

**complex d (8.5); // инициализация 8.5+0i**

**complex e (9.1,10.0);// инициализация 9.1+10.0i**

**complex f ;**

## 13.7. Указатель на void

В C++, как и в C, можно назначать указатель любого типа указателю на void, но в отличие от C, в C++ невозможно присвоить указателю на void указатель любого другого типа, если явно не привести их типы.

*Пример 1*

**int ar[5] = {4,5,6,7,8};**

**int \*pi;**

**void \*pv;**

**pv = ar; // допускается в С и С++**

**pi = pv; // допускается в С, но неверно в С++**

**pi = (int\*)pv; // допускается в С и С++**

В C++ можно присваивать адрес объекта произвольного типа указателю на объект базового класса. В C – нельзя.

## 13.8. Альтернативное правописание

В C++ or или | – ключевые слова. А в стандарте С99 они определены как макросы, следовательно требуется включение заголовочного файла <ISO646.h>.

## 13.9. Поддержка расширенных символов

При работе с C++ имеется встроенный тип данных wchar\_t, wchar\_t – ключевое слово, а в стандарте С99 wchar\_t определен в нескольких заголовочных файлах: <stddef.h> <stdlib.h> <wchar.h> <wctype.h>

В C++ реализуется поддержка ввода-вывода для расширенных символов с помощью заголовочного файла <iostream.h>. Стандарт С99 поддерживает многобайтные символы, что невозможно в C++.

# Глава 14. Исключения и их обработка

Исключение – это некоторое действие, которое выполняется при наступлении нестандартных ситуаций.

Говоря о об исключениях в С++, рассмотрим одну из таких нестандартных ситуаций — деление на 0. Представьте, что мы работаем в программе, где вручную вносим много числовых данных. Далее, в результате каких-то расчетов программы, значение одной из переменных станет равным 0. А в следующей строке кода это значение выступает делителем. Конечно, это действие приведет к неизбежному закрытию программы и выходу из нее. А теперь представьте, какие чувства вы испытаете, когда полчаса вводите данные, работая в этой программе, и на каком-то этапе она просто перестает работать и закрывается, а все данные пропадают.

*Пример 1*

*В переменную num2 необходимо записать значение 0 для понимания сути проблемы.*

**#include <iostream>;**

**using namespace std;**

**int main()**

**{**

**setlocale(LC\_ALL, "rus");**

**int num1;**

**int num2;**

**int var = 2; //управляющая переменная для while**

**while(var--)**

**{**

**cout << "Введите значение num1: ";**

**cin >> num1;**

**cout << "Введите значение num2: ";**

**cin >> num2;**

**cout << "num1 + num2 = " << num1 + num2 << endl;**

**cout << "num1 / num2 = " << num1 / num2 << endl;**

**cout << "num1 - num2 = " << num1 - num2 << endl;**

**cout << "=================================" << endl << endl;**

**}**

**cout << "Программа завершила работу!" << endl << endl;**

**return 0;**

**}**

В программе цикл должен отработать два раза. В строке 12 мы передаем циклу значение переменной var и в этой же строке уменьшаем её на единицу, используя операцию декремента (var--). Но если в переменную num2 будет введено значение 0, мы увидим только результат сложения a + b, и программа завершит работу независимо от того, есть код ниже или нет, отработал цикл или нет.

Используя исключения, мы сможем избежать таких проблем. Чтобы «прикрутить» исключение к этому примеру, надо познакомиться со следующими командами С++: *throw, try, catch.* В исходном коде ниже исключение сработает так: программа получает конкретное указание от программиста — если значение определённой переменной в определённом участке кода (в try-блоке) будет равно 0, то в этом случае пусть генерируется исключение *throw*. Это исключение автоматически передастся *catch*-блоку в виде параметра и выполнится код этого блока.

*Пример 2.*

**#include <iostream>;**

**using namespace std;**

**int main()**

**{**

**setlocale(LC\_ALL, "rus");**

**int num1;**

**int num2;**

**int var = 2;**

**while(var--)**

**{**

**cout << "Введите значение num1: ";**

**cin >> num1;**

**cout << "Введите значение num2: ";**

**cin >> num2;**

**cout << "num1 + num2 = " << num1 + num2 << endl;**

**cout << "num1 / num2 = ";**

**try //код, который может привести к ошибке, располагается тут**

**{**

**if (num2 == 0)**

**{**

**throw 123; //генерировать целое число 123**

**}**

**cout << num1 / num2 << endl;**

**}**

**catch(int i)//сюда передастся число 123**

**{**

**cout << "Ошибка №" << i << " - на 0 делить нельзя!!!!" << endl;**

**}**

**cout << "num1 - num2 = " << num1 - num2 << endl;**

**cout << "=================================" << endl << endl;**

**}**

**cout << "Программа завершила работу!" << endl << endl;;**

**return 0;**

**}**

В этом коде сначала был определен try-блок. В нем располагается код, который потенциально может вызвать ошибку в работе программы, а именно ошибку в случае деления на 0. Задаем условие if — если num2 равно 0, то генерировать целое число 123, к примеру. В этом случае try-блок сразу прекращает выполнение дальнейших команд, а число 123 «падает» в catch. В нашем примере он выводит сообщение об ошибке. При этом программа продолжает работать и выполнять команды, размещенные ниже. Если же число num2 не будет равно нулю, то в try-блоке выполнится команда cout << num1 / num2 << endl; , а catch не сработает.

# Приложение 1.

## Построение магического квадрата

Под магическим квадратом порядка N понимается квадратная матрица размером NxN из N в квадрате последовательных элементов произвольной арифметической прогрессии натуральных чисел, которые размещены так, что суммы элементов любого столбца, строки или главной диагонали одинаковы. Результат вычисления любой из перечисленных сумм принято называть константой магического квадрата. Порядок магического квадрата определяется числом элементов любого столбца или строки. Например, магический квадрат 3-го порядка из 9-ти 1-х натуральных чисел (известный в Китае как талисман ло-шу) представляется следующей матрицей 3x3:

4 9 2

3 5 7

8 1 6

Константа квадрата ло-шу равна 15. Это единственный квадрат 3-го порядка, который можно построить из натуральных чисел от 1 до 9, если не использовать преобразования поворота и отражения. Классический образец магического квадрата 4-го порядка, известный еще в Древней Индии, представляется следующей матрицей 4x4:

1 14 15 4

12 7 6 9

8 11 10 5

13 2 3 16

Константа "индийского" квадрата равна 34. В несколько измененном виде (достигаемом перестановкой строк и столбцов):

16 3 2 13

5 10 11 8

9 6 7 12

4 15 14 1

он известен художникам по философской гравюре А. Дюрера "Меланхолия". Астрологи средних веков приписывали числовым сочетаниям магических квадратов таинственные и волшебные свойства. Современных математиков и программистов интересуют формальные методы составления магических квадратов.

**Составление магических квадратов нечетного порядка**

Наибольший практический интерес представляют универсальные методы, которые не зависят от порядка магического квадрата. Такие методов известны для магических квадратов нечетного порядка. Наиболее наглядный из них удобно рассмотреть на примере составления магического квадрата 5-го порядка из натуральных чисел от 1 до 25. Алгоритм этого метода включает следующие шаги:

1. Сначала исходный пустой квадрат достраивается до симметричной ступенчатой ромбовидной фигуры как показано на следующем рисунке, где ячейки для элементов квадрата обозначены символом #, а достроенные ячейки - символом $.

$

$ $ $

# # # # #

$ # # # # # $

$ $ # # # # # $ $

$ # # # # # $

# # # # #

$ $ $

$

2. Полученная на шаге 1 фигура заполняется по косым рядам сверху-вниз-направо целыми числами от 1 до 25 в натуральном порядке. Результат заполнения показан на следующем рисунке:

1

6 $ 2

11 # 7 # 3

16 # 12 # 8 # 4

21 $ 17 # 13 # 9 $ 5

22 # 18 # 14 # 10

23 # 19 # 15

24 $ 20

25

3. Каждое число, расположенное в фигуре шага 2 вне исходного квадрата, переносится по вертикали или горизонтали внутрь исходного квадрата на число позиций, равное порядку квадрата. В рассматриваемом примере перенос осуществляется на 5 позиций. Таблица переносов имеет следующий вид:

1 - вниз под 13; 2 - вниз под 14; 6 - вниз под 18;

21 - вправо за 13; 22 - вправо за 14; 16 - вправо за 8;

5 - влево перед 13; 4 - влево перед 12; 10 - влево перед 18;

25 - вверх над 13; 24 - вверх над12; 20 - вверх над 8.

Освобождающиеся ячейки, достроенные к исходному квадрату заполняются символом $.

4. После преобразований переноса на шаге 3 освободившиеся ячейки (заполненные символом $) должны быть исключены. Оставшиеся (внутренние) ячейки (заполненные натуральными числами) образуют магический квадрат, представленный следующей матрицей 5x5:

11 24 7 20 3

4 12 25 8 16

17 5 13 21 9

10 18 1 14 22

23 6 19 2 15

константа которого равна 65, что может быть проверено вычислением суммы элементов для столбцов, строк и главных диагоналей.

Рассмотренный метод составления нечетных магических квадратов не является единственным. Не менее известным и не более сложным является следующий алгоритм, предложенный С. Лубером. Правила алгоритма Лубера удобно иллюстрировать на примере магического квадрата порядка 7 из натуральных чисел от 1 до 49, матрица 7x7 которого показана на следующем рисунке:

28 19 10 01 48 39 30

29 27 18 09 07 47 38

37 35 26 17 08 06 46

45 36 34 25 16 14 05

04 44 42 33 24 15 13

12 03 43 41 32 23 21

20 11 02 49 40 31 22

В основе алгоритма Лубера лежит заполнение ячеек квадрата в направлении вверх и влево по диагонали последовательными числами выбранной арифметической прогрессии. Заполнение начинается со среднего элемента верхней строки (01). Если следующая левая диагональная ячейка уже занята числом (ячейка 01 уже занята в момент заполнения ячейки 07), нужно перейти к нижнему соседу (08) текущей заполненной ячейки (07) и продолжить движение по диагонали. Чтобы избежать возможности выхода за границы квадрата при диагональном движении его надо мысленно превратить в тор, соединив верхнюю горизонталь с нижней, а затем соединить основания полученного цилиндра. После свертки строки, столбцы и диагонали квадрата превращаются в замкнутые кривые на поверхности тора и выход за границы квадрата становится невозможным. Превращение квадрата в тор в данном случае обеспечивает возможность диагонального перехода, например, из ячейки 01 в ячейку 02 или из ячейки 45 в ячейку 46.

**Программирование магических квадратов**

Для программирования магических квадратов удобно использовать парадигму инкапсуляции объектно-ориентированного программирования, сосредоточив "магические" структуры данных и методы их обработки в отдельном классе Magic. В класс Magic рекомендуется включить следующие приватные (privat) компоненты - данные:

unsigned degree - порядок квадрата;

unsigned basic, differ - начальный элемент и разность арифметической прогрессии натуральных чисел, заполняющих магический квадрат;

unsigned \*\*tab - указатель на двумерный массив беззнаковых целых чисел, который должен содержать матрицу магического квадрата.

Для обработки приватных данных в классе Magic целесообразно предусмотреть следующие общедоступные (public) компонентные методы:

void magodd( ) - формирует магический квадрат нечетного порядка;

void mageven2( ) - составляет магический квадрат четного порядка, у которого порядок является экспонентой 2;

int chksum( ) - вычисляет контрольные суммы элементов столбцов, строк и главных диагоналей квадрата, возвращая величину константы квадрата или 0, если квадрат не является магическим;

void print( ) - отображает матрицу и константу магического квадрата в потоке стандартного вывода;

void magbuild( ) - вызывает метод составления, соответствующий порядку магического квадрата.

Для инициализации компонент-данных в классе Magic следует предусмотреть конструктор. Конструктор класса Magic должен иметь возможность принимать 3 аргумента, определяющих порядок, начальный элемент и разность арифметической прогрессии элементов магического квадрата. При декларации конструктора рекомендуется указать значения аргументов по умолчанию:

*Пример 1*

**class Magic {**

**//некоторый код**

**public:**

**Magic(unsigned=3, unsigned=1, unsigned=1);**

**//некоторый код**

**};**

Это позволит для создания объектов класса Magic использовать вызов конструктора в сокращенном формате; с одним , двумя или без аргументов. Например, следующий вызов конструктора:

**Magic loshoo;**

является корректным для составления магического квадрата ло-шу, где приватные компоненты-данные degree, basic, differ по умолчанию инициализируются величинами 3, 1, 1, соответственно. Кроме инициализации целочисленных компонент-данных конструктору класса Magic желательно поручить динамическое распределение памяти, необходимое для хранения 2-мерного массива элементов магического квадрата tab. Для динамического распределения памяти под 2-мерный массив рекомендуется использовать следующую 2-х этапную схему:

1. распределить память под 1-мерный массив указателей на беззнаковые целые размером degree по адресу tab используя оператор new:

**tab=new unsigned \*(degree);**

2. под каждый указатель полученного массива указателей распределить одномерный массив беззнаковых целых из degree элементов, используя оператор new в цикле:

**for (int i=0; i<degree;i++)**

**tab[i]=new unsigned(degree);**

Рассмотренный способ формирования 2-мерного массива позволит обращаться к любому j-му элементу i-й строки в компонентных методах класса Magic традиционным образом tab[i][j]. Вызов компонентных методов составления магического квадрата может быть реализован в теле конструктора класса Magic, а обращение к компонентному методу отображения результатов целесообразно организовать в основной функции программы magic.

После завершения требуемой обработки и отображения результатов объекты класса Magic должны быть уничтожены с помощью деструктора ~Magic( ). Деструктор класса Magic должен освободить память, динамически распределенную конструктором по адресу tab для хранения 2-мерного массива элементов магического квадрата. Рекомендуется освобождать память по схеме обратной ее распределению, используя оператор delete:

1. освободить память, распределенную под degree одномерных массивов беззнаковых целых (из degree элементов каждый) по адресам от tab[0] до tab[degree-1]:

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

delete [degree](tab[i]);

2. освободить память, распределенную под 1-мерный массив указателей на беззнаковые целые, состоящий из degree указателей по адресу tab:

**delete [degree]tab;**

Обычно деструктор вызывается неявно при завершении работы с объектом класса, например, после выхода из блока, где объект был определен. При необходимости явного вызова деструктора в обращении к нему должно быть специфицировано имя класса, которому он принадлежит:

**Magic mag(4);**

**mag.Magic::~Magic( );**

В соответствии с принципами структурного программирования следует разместить декларацию класса Magic в отдельном заголовочном h-файле. Исходный код компонентных методов класса Magic и основной функции main( ) программы magic, которая оперирует ими, рекомендуется разместить в 2-х отдельных файлах. Для успеха трансляции в каждый из них нужно включить заголовочный файл класса Magic, используя директиву include.

## Поиск числовых палиндромов

**Описание алгоритма**

Построение алгоритма основано на том, что легче проверить свойство быть палиндромом, чем свойство быть квадратом. Поэтому целесообразно сначала вычислять квадрат числа, а затем проверять, является ли десятичное представление квадрата числа палиндромом. С учетом этого обстоятельства наиболее абстрактный проект алгоритма можно построить следующим образом:

* Задать предельное значение счетчика чисел (100).
* Обнулить счетчик чисел.
* Выполнять цикл
* { Увеличить значение счетчика на 1.
* Возвести значение счетчика в квадрат.
* Представить значение квадрата счетчика в форме строки цифр и получить копию этой строки.
* Реверсировать копию строки цифр, представляющей квадрат счетчика чисел.
* Если прямое и инверсное представления строки квадрата счетчика эквивалентны, то печать палиндрома и значения счетчика.
* } пока счетчик меньше предельного значения.

**Структуры данных**

Исходя из анализа алгоритма решения данной задачи целесообразно объявить следующие переменные:

* count — счетчик целых чисел;
* limit — предельное значение счетчика;
* square — квадрат счетчика целых чисел;
* direct — десятичное представление квадрата счетчика;
* inverse — инверсное десятичное представление квадрата счетчика.

По логике работы алгоритма переменные count, limit и square должны принимать неотрицательные целые значения. Поэтому они должны иметь тип unsigned int. Поскольку предельное значение счетчика постоянно, объявление переменной limit можно дополнить модификатором const.

Для хранения прямого и инверсного представлений квадрата счетчика можно зарезервировать 2 массива символов размером по 6 байт. Такая размерность массивов обеспечивает хранение максимального значения символического представления квадрата счетчика (10000), длиной 5 байт плюс символ ’\0’, который обозначает конец информационной части символьной строки. Размерность массивов может быть задана директивой препроцессора define.

# Приложение 2. Лабораторные работы

## Разработка программы-калькулятора дробей

**Формулировка задания**

Требуется разработать программу-калькулятор дробей: fc (fraction calculator), который обеспечивает выполнение арифметических операций (сложение, вычитание, умножение и деление) для рациональных дробей, заданных в символическом формате. Программа fc должна быть ориентированна на использование следующего формата дробей: a/b, где a и b наборы цифр, отображающие числитель и знаменатель дроби.

Исходными данными программы fc являются последовательности символов, представляющие вычисляемые арифметические выражения, где дробные операнды связывает символ операции(+, -, \*, /). Дробные операнды и символ операции должны передаваться программе fc, через параметры командной строки её вызова. В командной строке операнды и символ операции разделяются с помощью пробелов.

Результатом работы программы fc должно быть дробное число, которое представляет итог вычисления входного выражения и отображается в потоке стандартного вывода (stdout). Для некоторых операций и операндов должен быть предусмотрен вывод соответствующих информационных сообщений в поток протокола стандартной диагностики (stderr).

Программа fc должна быть составлена в системе программирования C++ на основе разработки специального класса, в котором перегружены специальные арифметические операции над дробными числами (выполнена перегрузка операций).

**Свойства дробных чисел**

В некоторых алгоритмах численной обработки данных более полезным является точное выражение результата в виде дроби (например, 1/3), чем приближенное представление с плавающей точкой (0,33333...). Использование дробей позволяет получить нецелочисленный ответ задачи в наглядной форме и исключить ошибку округления, свойственные обработке чисел с плавающей точкой в ограниченной разрядной сетке. Как известно, дробное число образует отношение 2-х целых чисел, которое записывается следующем образом: a/b, где a и b - наборы цифр, представляющие числитель и знаменатель дроби. Относительно числителя и знаменателя дроби приняты следующие договоренности:

* знаменатель дроби должен быть натуральным числом (b>0);
* целое число можно представить дробью с единичным знаменателем (a/1);
* знак дроби определяется знаком числителя;
* если числитель и знаменатель дроби умножить на одно и тоже натуральное число (N), то получиться дробь равная данной (a/b=(N\*a)/(N\*b));
* если числитель и знаменатель дроби имеют общий делитель, то при делении их на него происходит сокращение дроби и образуется дробь равная данной (D\*a)/(D\*b)=a/b);
* если числитель и знаменатель взаимно-простые числа, т. е. не имеют общих делителей, то дробь называется несократимой;
* любая дробь может быть представлена к несократимой, если её числитель сократить на их наибольший общий делитель (Hog) - наибольшее натуральное число, на которое они оба делятся без остатка;
* две любые дроби a/b и c/b считаются равными, если (a\*d)=(b\*c);
* две несократимые дроби считаются равными, если равны их числители и знаменатели (a=c и b=d).

Вычислительная обработка дробных чисел основана на использовании 4-х арифметических операций: сложение, вычитание, умножение и деление. Для выполнения этих операций символическое представление дроби выражается парой целых чисел, соответствующих её числителю и знаменателю. Пусть U/U' и V/V' обозначают дроби-операнды, а W/W'-дроби, результат операции. Тогда числитель и знаменатель результирующей дроби для 4-х арифметических операций выражают следующие соотношения:

Умножение:

(W/W')={(U/U')\*(V/V')}

W=(U/d1)\*(V/d2) и W'=(U'/d2)\*(V'/d1), где d1=Hog(U,V') и d2=Hog(U'/V).

Деление: (W/W')={(U/U')/(V/V')}

W=(U/d1)\*(V'/d2)sign(V) и W'=|(U'/d2)\*(V/d1)|, где d1=Hog(U,V) и d2=Hog(U',V').

Сложение: (W/W')={(U/U')+(V/V')}

Если Hog(U'/V')=1,то

W=(U\*V')+(U'\*V) и W'=(U'\*V')

Если Hog(U'/V')>1,то

W=t/d2 и W'=(U'/d1)\*(V'/d2), где d1=Hog(U',V'), t=U\*(V'/d2)+V\*(U'/d1) и d2=Hog(t,d1).

Вычитание: (W/W')={(U/U')-(V/V')}. Выполняется аналогично сложению, если везде заменить знак плюс на знак минус.

В приведенных соотношениях Hog(m,n) обозначает наибольший общий делитель чисел |m| и |n|. Для эффективного вычисления Hog применяется алгоритм Евклида - самый старый нетривиальный алгоритм, доживший до наших дней. В современной редакции псевдокод алгоритма Евклида выглядит следующим образом:

*Пример 1*

**IF m<0 THEN**

**m <- |m|; /\* перейти к абсолютной величине числа m \*/**

**IF n<0 THEN**

**n <- |n|; /\* перейти к абсолютной величине числа n \*/**

**WHILE n<>0 { /\* цикл уменьшения n \*/**

**r <- m mod n; /\* остаток деления m на n \*/**

**m <- n;**

**n <- r;**

**}**

**RETURN m.**

В классическом варианте алгоритм Евклида используется для поиска наибольшего общего делителя пары не отрицательных целых чисел, при условии, что Hog(0,0) принимается равным 0. Для расширения области применения на поле произвольных целых чисел в предлагаемой версии алгоритма Евклида предусмотрен переход к абсолютным величинам рассматриваемых чисел. На каждом шаге основного цикла алгоритма Евклида происходит последовательное уменьшение обоих чисел, но при этом значение их наибольшего общего делителя остается неизменным, т. к.:

http://bigor.bmstu.ru/?frm/?doc=VU/lab1.mod/?n=1/?k=10.

Цикл уменьшения обоих чисел продолжается, пока на очередной итерации меньшее из чисел не станет равным 0. В этом случае большее из чисел принимается за наибольший общий делитель исходной пары, т. к.:

http://bigor.bmstu.ru/?frm/?doc=VU/lab1.mod/?n=2/?k=10.

**Пример 2**

Hog(259,111) выполняется за 3 шага: Hog(259,111)=Hog(111,37)=Hog(3,0)=3.

Сложность алгоритма Евклида пропорциональна логарифму от максимального по абсолютной величине числа из рассматриваемой пары чисел. эффективность алгоритма Евклида определяет эффективность выполнения арифметических операций над дробями по соотношениям, определенным выше.

На первый взгляд кажется, что арифметические операции над дробями можно реализовать более эффективно, вычисляя наибольший общий делитель в каждой операции только один раз вместо двух. Например, при умножении {(U/U')\*(V/V')}, числитель и знаменатель ответа можно получить по внешне более простым формулам:

http://bigor.bmstu.ru/?frm/?doc=VU/lab1.mod/?n=3/?k=10 где http://bigor.bmstu.ru/?frm/?doc=VU/lab1.mod/?n=4/?k=10.

При сложении 2-х дробей {(U/U')+(V/V')} можно представить результат в виде следующей дроби:

http://bigor.bmstu.ru/?frm/?doc=VU/lab1.mod/?n=5/?k=10

а затем привести результирующую дробь к не сократимому виду, используя свойство: .

Однако, в обоих случаях придется оперировать с относительно большими числами на каждой итерации алгоритма Евклида. Каждая итерация будет выполняться медленнее, чем код на в ней рассматриваются меньшие числа, которые характерны для операций обработки дробей с 2-мя вычислениями наибольшего общего делителя. Таким образом, по быстродействию оба подхода по-крайней мере равнозначны, но в первом исключена возможность переполнения разрядной сетки, что является решающим преимуществом, когда числители и знаменатели дробей являются большими числами.

**Программирование калькулятора дробей.**

При разработке программы калькулятора дробей целесообразно сосредоточить описание дробных структур данных и операции их обработки в отдельном классе Fraction. Класс Fraction должен содержать частные (private) компоненты данные и общедоступные (public) компонентные методы обработки приватных данных. Это позволит оперировать данными класса Fraction только в компонентных методах и исключит возможность непосредственного обращения к ним из любой внешней функции программы. Исходя из этого декларация логической структуры класса Fraction должна иметь следующий формат:

**class Fraction {**

**private: /\* спецификация компонентных данных \*/**

**public: /\* объявление прототипов компонентных методов \*/**

**};**

В приватную часть класса Fraction следует включить спецификацию целочисленных компонентных данных, которые обозначают числитель и знаменатель дроби, как показано в следующем фрагменте класса Fraction:

**class Fraction {**

**private:**

**int nominator; /\* числитель \*/**

**int denominator; /\* знаменатель \*/**

**public:**

**//некоторый код**

**};**

В общедоступную часть декларации класса Fracton нужно включить объявление прототипов компонентных методов, которые реализуют арифметическую обработку дробей и отображение результатов обработки. Кроме того, в общедоступной части следует объявить конструктор класса.

Желаемую арифметическую обработку дробных чисел удобно реализовать с помощью компонентных оператор-функций, которые обеспечат перегрузку арифметических операций (+,-,\*,/) в выражениях с объектами класса Fraction. Наличие оператор-функций позволит конструировать арифметические выражения для объектов класса Fraction в наглядной форме как для объектов простых типов. При наличии соответствующих оператор-функций будет правильно вычисляться, например, следующее выражение программы:

**r=f1@f2;**

где r,f1 и f2 объекты класса Fraction, а @ - знак операции.

По формату определения и декларации оператор-функция идентична обычной функции с предопределенным именем operator@, где @ обозначает знак перегружаемой операции, и специальным соглашением по аргументам. Для декларации 4-х требуемых оператор-функций в классе Fraction может быть рекомендован следующий идентичный формат:

**Fraction operator@( Fraction& );**

который отличается только символом перегружаемой операции @ (+,-,\*,/). Используя эти декларации, компилятор языка C++ будет рассматривать приведенное выше выражение как следующий вызов соответствующей компонентной оператор-функции:

**r=f1 operator@(f2);**

Как видно из приведённой спецификации вызова, аргумент компонентной оператор-функции используется для передачи по ссылке 2-го операнда операции. Первый операнд операции неявно передается через скрытый аргумент, присущий любой компонентной функции и доступный по указателю this. Результат вычислений в теле оператор-функции возвращается в основную программу для присвоения объекту класса Fraction. Следует отметить, что передача аргумента по ссылке в данном случае выбрана из экономических соображений. Для оператор-функции допустима передача аргумента по значению. Однако, в этом случае происходит создание промежуточной копии объекта передачи в стеке оператор-функции.

Кроме компонентных оператор-функций в состав компонентных методов класса Fraction целесообразно включить 3 обычные компонентные функции:

**int euclid(int,int);**

**int sign(int);**

**void ptint(void);**

декларировав их в общедоступной части класса Fraction, указанным образом.

Компонентный метод euclid должен вычислять наибольший общий делитель своих аргументов, используя алгоритм Евклида. Вычисленное значение должно передаваться через код возврат метода. Этот метод предназначается для реализации алгоритмов арифметических операций с дробями в компонентных оператор-функциях класса Fraction.

Компонентный метод sign используется для определения знаков своего аргумента, возвращая (-1), если значение аргумента меньше 0 или +1, если аргумент имеет неотрицательное значение. Этот метод следует использовать для реализации алгоритма деления дробных чисел в соответствующей компонентной оператор-функции класса Fraction.

Компонентный метод print должен отображать в потоке стандартного вывода числитель и знаменатель объекта класса Fraction, от имени которого он вызван. Этот метод следует применять в программе калькулятора дробей для отображения результата операций с дробями.

Для инициализации компонентных данных при создании объектов класса Fraction в нем необходимо предусмотреть конструктор с аргументом типа указатель на строку символов (char\*). Поэтому декларация конструктора в общественной части объявления класса Fraction должен иметь следующий формат:

**Fraction(char\*);**

Через аргумент в этот конструктор должно передаваться представление дроби в виде строки символов, где числитель и знаменатель выражаются выборами цифр, которые разделяет символ '/'. Конструктор должен преобразовывать символьное представление числителя и знаменателя в целочисленные значения компонент-данных класса Fraction, используя библиотечную функцию atoi системы программирования C++.

Рассмотренные общедоступные компоненты класса Fraction можно использовать в основной функции main( int argc,char\*\* argv) программы калькулятора дробей fc. Вычисляемое арифметическое выражение должно передаваться в функцию main через параметры командной строки вызова программы fc, так что 1-й (argv[1]) и 3-й

(argv[3]) параметры соответствуют дробным операндам.

В теле функции main следует задать 3 объекта класса Fraction для хранения операндов и результата операции. Объекты- операнды должны быть инициализированы строками соответствующих параметров командной стоки. Объект результата можно инициализировать произвольной дробью, представленной в символическом формате, например, "0/1".

Знак операции, передаваемый в функцию main через 2-й параметр командной строки, удобно рассматривать в операторе switch системы программирования C++ для выбора одного из 4-х вариантов вычисляемых дробных выражений. После вычисления выбранного выражения с помощью соответствующей компонентной оператор- функции класса Fraction ответ нужно сохранить в зарезервированном объекте-результате. Вызов компонентной функции print класса Fraction от имени объекта результата, позволит отобразить полученную дробь в потоке стандартного вывода.

**Контрольные задания**

1. Расширить класс Fraction перегрузкой операции присваивания (=), в которой должно быть реализовано копирование числителя и знаменателя, а также приведение результирующей дроби к несократимому виду, если это необходимо.

2. Расширить класс Fraction перегрузкой операции проверки равенства 2-х дробей (==).

3. Модифицировать класс Fraction, так чтобы перегрузку операций над его объектами обеспечивали дружественные оператор-функции. При этом необходимо учесть, что прототипы дружественных функций должны быть объявлены в декларации класса с модификатором friend и в них по ссылке должны передаваться оба операнда, т.к. скрытый указатель this на объект класса в них недоступен.

4. Усовершенствовать представление результата операций с дробями, так чтобы в случае, когда числитель больше знаменателя, выделялась целая часть числа.

5. Разработать средства контроля достоверности результатов операций с дробными числами. Например, сложение дробей должно проверяться вычитанием, а умножение - делением.

6. Переориентировать программу-калькулятор для интерактивного ввода вычисляемых арифметических выражений с дробями из потока стандартного ввода.

7. Реализовать операции сложения и вычитания с помощью приведения дробей к общему знаменателю. Для вычисления наименьшего общего кратного (Hok) знаменателей дробей, которое необходимо в этом случае, рекомендуется использовать следующее соотношение:

**U'\*V'=Hog(U',V')\*Hok(U',V').**

**Рекомендуемая литература**

1. Д. Кнут. Искусство программирования для ЭВМ, т.2 Получисленные алгоритмы - М., Мир, 1977 г.

2. П. Лукас. С++ под рукой - Киев, НИПФ "ДиаСофт", 1993 г.

## Разработка программы выполнения теоретико-множественных операций

**Формулировка задания**

Требуется разработать объектно-ориентированную программу для вычисления объединения(суммы) и пересечения(произведения) двух любых подмножеств латинских букв. Заданные подмножества латинских букв должны передаваться программе через аргументы командной строки ее вызова. Программная реализация вычислений должна быть основана на разработке класса множества символов с перегрузкой операторов '+' и '\*' для выполнения соответствующих теоретико-множественных операций.

Результатом работы программы, то есть результат операции должен отображаться символьной строкой в потоке стандартного вывода. Для удобства отображения результата в программе должна быть предусмотрена перегрузка оператора ‘<<’ класса потока стандартного вывода для объектов класса множества латинских букв.

Программа должна быть составлена в системе программирования C++ на основе разработки специального класса, в котором перегружены специальные операции над двумя любыми подмножествами латинских букв.

**Свойства подмножеств**

Вычислительная обработка подмножеств латинских букв основана на использовании 2-х операций: объединения(суммы) и пересечения(произведения). Для выполнения этих операций символическое представление подмножества выражается символьной строкой из латинских букв. Пусть Ml и М2 обозначают подмножества латинских букв, а МЗ-результируюшее подмножество, результат операции. Тогда выполнение теоретико-множественных операций над подмножествами латинских букв выражают следующие соотношения:

* Произведение(пересечение): ABC \* CDA = AC
* Сложение(объединение):ABC + CDA = ABCD

**Программирование операций над подмножествами латинских букв**

При разработке программы выполнения теоретико-множественных операций целесообразно сосредоточить описание подмножеств и операции их обработки в отдельном классе Set. Класс Set должен содержать частные (private) компоненты данные и общедоступные (public) компонентные методы обработки приватных данных. Это позволит оперировать данными класса Set только в компонентных методах и исключит возможность непосредственного обращения к ним из любой внешней функции программы. Исходя из этого, декларация логической структуры класса Set должна иметь следующий формат:

**class Set {**

**private: /\* спецификация компонентных данных \*/**

**public: /\* объявление прототипов компонентных методов \*/**

**};**

В приватную часть класса Set следует включить беззнаковое целое, биты которого соответствуют буквам латинского алфавита. Причем единичный бит означает, что соответствующая его позиция в слове латинских букв (в алфавитном порядке) входит в данное множество. Например, асе представляется следующим образом:

**(е)(с)(а)**

**1 0 1 0 1**

Тогда битовые операции над битами этого поля эквивалентны операциям над элементами множеств.

Таким образом, структура класса множеств Set выглядит следующим образом:

**class Set { private:**

**unsigned wordset; /\*беззнаковое целое\*/ public:**

**................**

**};**

В общедоступную часть декларации класса Set нужно включить объявление прототипов компонентных методов, которые реализуют обработку подмножеств и отображение результатов обработки. Кроме того, в общедоступной части следует объявить конструктор класса с аргументом, где аргумент - это строка символов, обозначающих заданное подмножество и конструктор без аргументов.

Желаемую обработку подмножеств латинских букв удобно реализовать с помощью компонентных оператор-функций, которые обеспечат перегрузку операций (+ ,\*) в выражениях с объектами класса Set. Наличие оператор-функций позволит конструировать выражения для объектов класса Set в наглядной форме как для объектов простых типов. При наличии соответствующих оператор-функций будет правильно вычисляться, например, следующее выражение программы:

**r=f1@f2;**

где r, f1 и f2 объекты класса Set, а @ - знак операции.

По формату определения и декларации оператор-функция идентична обычной функции с предопределенным именем operator@, где @ обозначает знак перегружаемой операции, и специальным соглашением по аргументам. Для декларации 2-х требуемых оператор-функций в классе Set может быть рекомендован следующий идентичный формат:

Set operator@( Set & ); который отличается только символом перегружаемой операции @ (+,\*). Используя эти декларации, компилятор языка C++ будет рассматривать приведенное выше выражение как следующий вызов соответствующей компонентной оператор-функции:

**r=f1.ореrаtor@(f2);**

Как видно из приведённой спецификации вызова, аргумент компонентной оператор-функции используется для передачи по ссылке 2-го операнда операции. Первый операнд операции неявно передается через скрытый аргумент, присущий любой компонентной функции и доступный по указателю this. Результат вычислений в теле оператор-функции возвращается в основную программу для присвоения объекту класса Set. Следует отметить, что передача аргумента по ссылке в данном случае выбрана из экономических соображений. Для оператор-функции допустима передача аргумента по значению. Однако, в этом случае происходит создание промежуточной копии объекта передачи в стеке оператор-функции.

Перегрузка оператора >> используется для вывода результата.

Для инициализации компонентных данных при создании объектов класса Set в нем предусмотрен конструктор с аргументом типа указатель на строку символов (char\*). Поэтому декларация конструктора в общественной части объявления класса Set должен иметь следующий формат:

Set (char\*str){…./\* Преобразование str в битовый формат \*/ }

Через аргумент в этот конструктор должно передаваться представление подмножества латинских букв в виде строки символов. Конструктор должен преобразовывать символьное представление подмножества латинских букв в битовые значения компонент-данных класса Set.

Рассмотренные общедоступные компоненты класса Set можно использовать в основной функции main( int argc,char\*\* argv) программы выполнения теоретико-множественных операций. Вычисляемое выражение должно передаваться в функцию main через параметры командной строки вызова программы.

В теле функции main следует задать 3 объекта класса Set для хранения операндов и результата операции. Объекты- операнды должны быть инициализированы строками соответствующих параметров командной стоки. Объект результата можно инициализировать третьей строкой.

Знак операции, передаваемый в функцию main через 2-й параметр командной строки, удобно рассматривать в операторе switch системы программирования C++ для выбора одного из 2-х вариантов вычисляемых дробных выражений. После вычисления выбранного выражения с помощью соответствующей компонентной оператор-функции класса Set ответ нужно сохранить в зарезервированном объекте-результате. Вызов оператор-функции >> класса Set от имени объекта результата, позволит отобразить полученную дробь в потоке стандартного вывода.

*Пример 1*

*Общий вид кода программы*

**Class Set{**

**private:**

**unsigned wordset;**

**public:**

**Set(char\*str {.../\*Преобразование str в битовый формат\*/})**

**Set() {wordset <- 0 }**

**Set operator +(Set&){...}**

**Set operator \*(Set&){...}**

**};**

**main(int argc, char\*\* argv){**

**SetMl(argv[l]);**

**Set M2(argv[2]);**

**SetM3;**

**МЗ=М1+М2;**

**Cout<<M3;**

**МЗ=М1\*М2;**

**Cout<<M3;**

**}**

**Контрольные задания**

1. Расширить класс Set перегрузкой операций объединения(+) и пересечения(\*) над подмножествами заглавных латинских букв.

2. Расширить класс Set перегрузкой операций объединения(+) и пересечения(\*) над подмножествами цифр.

3. Расширить класс Set перегрузкой операций объединения(+) и пересечения(\*) над подмножествами русских букв.

**Рекомендуемая литература**

1. Д. Кнут. Искусство программирования для ЭВМ, т.2 Получисленные алгоритмы - М, Мир, 1977 г.

2. П. Лукас. C++ под рукой - Киев, НИПФ "ДиаСофт", 1993 г.

## Разработка программы игры

**Цель работы**

Целью лабораторной работы является практическое изучение принципов применения виртуальных функций и абстрактных классов ООП при разработке игровой программы в системе программирования С++.

**Стратегия игры**

Выигрышную стратегию образует последовательность ходов игрока, когда после каждого хода количество R предметов, которое остается в куче и определяется следующей формулой:

**R=N\*L+N+1>0**

В этой формуле константа L обозначает заданное предельно-допустимое количество предметов, которое может быть взято из кучи за один ход игрока, а целочисленная переменная N определяет число выигрышных ходов, оставшихся до конца партии после текущего хода. Таким образом, после последнего выигрышного хода, когда N равно 0, в куче остается один предмет, который вынужден взять проигравший противник. При первом и остальных выигрышных ходах величина N подбирается таким образом, чтобы получить по формуле наибольшее целое число R, которое меньше текущего числа предметов в куче Н. В соответствие с рассмотренным правилом форсированный выигрыш гарантирован игроку, который первым сделает выигрышный ход, потому что после этого независимо от ответов противника он получает возможность делать только выигрышные ходы до конца партии. Пусть, например, в куче первоначально находится Н=11 предметов, а игрокам разрешается брать не более L=3 предметов за один ход. Тогда первый игрок имеет возможность выиграть партию за 3 хода, после которых в куче остаются следующие количества предметов:

**R=2\*3+2+1=9; R=1\*3+1+1=5; R=0\*3+0+1=1**

Чтобы осуществить эту выигрышную стратегию, на начальном ходе первый игрок должен взять из кучи Н- R = 11-9=2 предмета, а на остальных ходах отвечать симметрично ходу противника. Например, если в ответ на начальный ход противник берет 3 из 9-ти предметов, то есть в куче остается 6 предметов, то вторым ходом первый игрок должен взять 1 предмет, чтобы оставить в куче 5 предметов. Если противник на втором ответном ходе берет из 5-ти предметов 3. В результате в куче остается 1 предмет, который вынужден взять второй игрок.

**Компьютерный интерфейс игры**

В компьютерном варианте игры противоборствующими противниками являются компьютер (PEN) и человек (MAN). Право первого хода автоматически предоставляется компьютеру, который должен действовать в соответствии с выигрышной стратегией игры. Если выигрышный ход сделать невозможно, компьютер должен выбрать из кучи 1 предмет. Число предметов, выбранных каждым ходом компьютера, должно отображаться через поток стандартного вывода. Человек может осуществлять произвольные допустимые правилами игры ходы, доказывая число выбранных предметов через поток стандартного ввода. При нарушении правил игры или некорректном (например, нецифровом) вводе партия должна быть блокирована, куча автоматически обнулена, а победителем в игре объявлен компьютер. Каждый ход человека и компьютера должен сопровождаться в потоке стандартного вывода. Запрос должен идентифицировать имя игрока и текущее число предметов в куче. Например, следующий запрос сопровождает первый ход компьютера, когда в куче содержится 11 предметов:

Heap=11 PEN>>

Свой выбор компьютер должен отобразить сразу после запроса. Если компьютер в ответ на этот запрос выбирает 2 предмета, то запрос для ответного хода человека имеет следующий вид:

Heap=9 MAN>>

Чередование запросов для ходов человека и компьютера должно предоставляться до конца партии. После завершения партии в потоке стандартного вывода должно отображаться информационное сообщение, которое объявляет победителя.

**Структура программы игры**

Разработка программной реализации данной игры, как и большинства других игр подобного типа, должна быть основана на использовании принципов наследования и полиморфизма ООП. В большинстве случаев при разработке объектно-ориентированной программной реализации оптимистических игр, где 2 и более предметов, бывает целесообразно применение принципов наследования и полиморфизма. Наследование обеспечивает тиражирование общих свойств игроков, которые передаются им из описания абстрактного игрока. Полиморфизм снабжает игроков виртуальным методом хода, который имеет одинаковый интерфейс, но реализует различные стратегии игроков. В данном случае основу информационной объектно-ориентированной структуры программы игры должны составить: абстрактный базовый класс Gambler, производные от него классы игроков PEN и MAN, а также класс кучи Heap, разделяемый игроками.

Абстрактный базовый класс Gambler предназначен для описания общих свойств и возможностей, которые присущи обоим игрокам. Его спецификацию образуют защищенные (protected) компонентные данные и общедоступные (public) компонентные методы их обработки. Это обеспечивает доступ к его компонентным данным только собственным компонентным методам производных классов игроков, но исключает любое несанкционированное обращение к ним из любых внешних функций программы игры.

В защищенную (protected) часть декларации базового класса Gambler целесообразно включить спецификацию целочисленных полей heap и limit, определяющих текущий размер кучи и предельное число предметов, которые можно выбирать из кучи за один ход. Кроме целочисленных компонентных данных в классе Gambler нужно предусмотреть указатель поля типа (char \*) для идентификации символического имени игрока.

Обработку защищенных компонентных данных класса Gambler должны обеспечивать общедоступные (public) компонентные методы query (запрос) и move (ход) с целочисленным кодом возврата, значение у которого определяет текущий размер кучи.

Компонентный метод query предназначен для отображения в потоке стандартного вывода информационных запросов, сопровождающих ходы игроков по сценарию игры. Для отображения запроса может быть использована библиотечная функция printf системы программирования C/C++ или класс потока стандартного вывода cout.

Компонентный метод swap должен обеспечивать синхронизацию значения размера кучи в производных объектах класса Gambler.

Наиболее простым из них является класс кучи Heap. Он должен содержать закрытые (private) данные о размере кучи (int size), общедоступный (public) компонентный метод (get) , конструктор класса полей данных. Он должен использоваться для хранения текущего размера кучи, который измеряется целочисленным значением. Компонентный метод get должен обеспечить доступ игроков к куче, чтобы извлечь требуемое число предметов или контролировать текущий размер кучи. Причем текущий размер кучи идентифицирует целочисленный код возврата метода get. Таким образом, общедоступный метод get позволяет оценить текущий размер кучи без извлечения предметов. Начальный размер кучи в поле должен устанавливаться конструктором класса Heap по значению своего целочисленного параметра.

Абстрактный класс Gambler предназначен для описания общих свойств и возможностей, которые присущи обоим участникам игры. Его спецификацию образуют защищенные (protected) компонентные данные и общедоступные (public) компонентные методы. В защищенной области спецификации класса Gambler должны быть декларированы 1 целочисленное поле (limit) и поля (heap и name). Компонентные данные в поле limit должны фиксировать ограничении по числу предметов, которые могут быть извлечены из кучи за 1 ход. Компонентное поле name предназначено для хранения указателя типа (char \*) на символьное имя игрока. Компонентные данные в поле heap должны иметь тип указателя на объект класса Heap (Heap \*), чтобы обеспечить адресацию кучи предметов участниками игры. Перечисленные поля компонентных данных должны инициализироваться конструктором класса Gambler, который должен иметь 3 аргумента соответствующих типов. Значения полей компонентных данных должны устанавливаться по аргументам конструктора через список инициализации. Конструктор должен быть специфицирован в общедоступной части класса Gambler. Кроме конструктора в общедоступной части класса Gambler должны быть определены 2 общедоступных компонентных метода query (запрос) и move (ход). Оба метода имеют целочисленный код возврата. Через query (запрос) передается текущий размер кучи. Класс Gambler специфицируется чистой виртуальной функцией с нулевым кодом, которая декларирует только внешний интерфейс хода игрока. В описании класса Gambler она должна быть объявлена следующим образом:

*Пример 1*

**Class Gambler {**

**Public:**

**Virtual int move( )=0;**

**};**

Различные реализации метода хода должны быть даны в классах игроков PEN и MAN, которые являются производными от базового класса Gambler, учитывая особенности ходов человека и компьютера, соответственно. Компоненты базового абстрактного класса Gambler перегружают в виртуальную функцию хода в каждом из производных классов конкретным кодом.

**Рекомендуемая литература**

1. Д. Кнут. Искусство программирования для ЭВМ, т.2 Получисленные алгоритмы - М, Мир, 1977 г.

2. П. Лукас. C++ под рукой - Киев, НИПФ "ДиаСофт", 1993 г.

## Пример программной реализации игры

При разработке данной игровой программы были применены *виртуальные функции* и *абстрактные классы*.

**#include <stdlib.h>**

**#include <iostream.h>**

**class Heap {**

**private:**

**int size;**

**public:**

**Heap(int s) : size(s) {};**

**int get(int n = 0) { size -= n; return(size); };**

**};**

**class Gambler {**

**protected:**

**Heap\* heap;**

**const int limit;**

**const char\* name;**

**public:**

**Gambler(Heap& h, int l) : limit(l) { heap = &h; };**

**virtual int move() = 0;**

**int query();**

**};**

**class Man : public Gambler {**

**public:**

**Man(Heap& h, int l, char\* n) : Gambler(h, l) { name = n; };**

**virtual int move();**

**};**

**class Pen : public Gambler {**

**public:**

**Pen(Heap& h, int l, char\* n) : Gambler(h, l) { name = n; };**

**virtual int move();**

**};**

**int Gambler::query() {**

**cout << "Heap = " << heap->get() << " " << name << " > ";**

**return(heap->get());**

**}**

**int Man::move() {**

**int g;**

**cin >> g;**

**if((g < 1) || (g > (heap->get())) || (g > limit))**

**g = 1;**

**return(heap->get(g));**

**}**

**int Pen::move() {**

**int rest = 0;**

**int n = 0;**

**int h;**

**if((h = heap->get()) == 1)**

**h = heap->get(1);**

**while((rest = (n\*limit + n + 1)) < h)**

**n++;**

**if((rest > h) || (rest == h))**

**rest = (n - 1)\*limit + n;**

**heap->get(h - rest);**

**cout << h - rest << "\n";**

**return(heap->get());**

**}**

**int main(int argc, char\* argv[]) {**

**int i = 0;**

**Heap h(atoi(argv[1]));**

**Gambler\* g[] = { new Pen(h, atoi(argv[2]), "Pen"),**

**new Man(h, atoi(argv[2]), "Man")**

**};**

**while(g[i]->query() != 0) {**

**g[i]->move();**

**if(++i > 1)**

**i = 0;**

**}**

**cout << "Winner\n";**

**delete g[0];**

**delete g[1];**

**return(0);**

**}**

## Варианты лабораторной работы по теме «Наследование»

**Вариант 1++**

Разработать объектно-ориентированную программу для вычисления длины отрезка прямой линии, который задан целочисленными координатами своих концов на плоскости. Координаты концов отрезка должны передаваться программе парами целых чисел через 2 аргумента командной строки ее вызова. При этом первое число в каждой паре должно обозначать координату по оси X, а второе – по оси Y. Для разделения координат в каждой паре должен использоваться символ точка с запятой. Результат вычислений длины отрезка должен отображаться через поток стандартного вывода. Программная поддержка указанных вычислений должна быть основана на разработке контейнерного класса отрезка прямой линии, компонентными данными которого являются объекты класса точек его концов, а компонентный метод обеспечивает вычисление длины отрезка.

**Вариант 2++**

Разработать объектно-ориентированную программу для вычисления длины отрезка прямой линии, который задан целочисленными координатами своих концов в пространстве. Координаты концов отрезка должны передаваться программе тройками целых чисел через 2 аргумента командной строки ее вызова. При этом первое число в каждой тройке должно обозначать координату по оси X, второе – по оси Y, третье – по оси Z. Для разделения координат в каждой тройке должен использоваться символ точка с запятой. Результат вычислений длины отрезка должен отображаться через поток стандартного вывода. Программная поддержка указанных вычислений должна быть основана на разработке контейнерного класса отрезка прямой линии, компонентными данными которого являются объекты класса точек его концов, а компонентный метод обеспечивает вычисление длины отрезка.

**Вариант 3++**

Разработать объектно-ориентированную программу для построения "золотого сечения" отрезка прямой линии заданной длины. Длина отрезка должна передаваться программе целым числом через аргумент командной строки ее вызова. Результат построений "золотого сечения" отрезка заданной длины должен отображаться через поток стандартного вывода. Программная поддержка указанных построений должна быть основана на использовании механизма наследования классов: базового класса – отрезок и производного — "золотое сечение" отрезка с методом вычисления этого сечения.

**Вариант 4++**

Разработать объектно-ориентированную программу вычисления периметра треугольника на плоскости, который задан координатами 3-х своих вершин. Координаты вершин должны передаваться программе парами целых чисел через 3 аргумента командной строки ее вызова. При этом первое число в каждой паре должно обозначать координату вершины по оси Х, а второе по оси Y. Для разделения координат в каждой паре должен использоваться символ точки с запятой. Результаты вычислений периметра треугольника должны отображаться через поток стандартного вывода. Программная реализация указанных вычислений должна быть основана на разработке контейнерного класса треугольника, компонентными данными которого являются объекты класса вершин, компонентные методы обеспечивают вычисление его периметра.

**Вариант 5++**

Разработать объектно-ориентированную программу вычисления площади треугольника на плоскости, который задан координатами 3-х своих вершин. Координаты вершин должны передаваться программе парами целых чисел через 3 аргумента командной строки ее вызова. При этом первое число в каждой паре должно обозначать координату вершины по оси Х, а второе по оси Y. Для разделения координат в каждой паре должен использоваться символ точки с запятой. Результаты вычислений площади треугольника должны отображаться через поток стандартного вывода. Программная реализация указанных вычислений должна быть основана на разработке контейнерного класса треугольника, компонентными данными которого являются объекты класса вершин, компонентные методы обеспечивают вычисление его площади.

**Вариант 6++**

Разработать объектно-ориентированную программу вычисления периметра прямоугольника на плоскости, который задан координатами 2-х своих противоположных вершин, а его стороны параллельны координатным осям. Координаты вершин должны передаваться программе парами целых чисел через 2 аргумента командной строки ее вызова. При этом первое число в каждой паре должно обозначать координату вершины по оси Х, а второе по оси Y. Для разделения координат в каждой паре должен использоваться символ точки с запятой. Результаты вычислений периметра прямоугольника должны отображаться через поток стандартного вывода. Программная реализация указанных вычислений должна быть основана на разработке контейнерного класса изотетичного прямоугольника, компонентными данными которого являются объекты класса противоположных вершин, компонентные методы обеспечивают вычисление его периметра.

**Вариант 7++**

Разработать объектно-ориентированную программу вычисления площади прямоугольника на плоскости, который задан координатами 2-х своих противоположных вершин, а его стороны параллельны координатным осям. Координаты вершин должны передаваться программе парами целых чисел через 2 аргумента командной строки ее вызова. При этом первое число в каждой паре должно обозначать координату вершины по оси Х, а второе по оси Y. Для разделения координат в каждой паре должен использоваться символ точки с запятой. Результаты вычислений площади прямоугольника должны отображаться через поток стандартного вывода. Программная реализация указанных вычислений должна быть основана на разработке контейнерного класса изотетичного прямоугольника, компонентными данными которого являются объекты класса противоположных вершин, компонентные методы обеспечивают вычисление его площади.

**Вариант 8++**

Разработать объектно-ориентированную программу для формирования квадратной матрицы Вандермонда по образующему вектору из необязательно различных десятичных цифр от 1 до 9, столбцы которой являются последовательными натуральными степенями элементов образующего вектора. Размер матрицы Вандермонда определяется длиной ее образующего вектора, который должен передаваться программе в форме цифровой строки через аргумент командной строки ее вызова. Результат построения матрицы Вандермонда должен отображаться через поток стандартного вывода, где все столбцы должны быть выровнены полевой границе своих элементов. Программная реализация указанных построений должна быть основана на разработке класса матрицы Вандермонда, производного от базового класса матрица с динамическим распределением памяти для таблицы целочисленных значений ее элементов с учетом оценки возможности их вычислений для стандартных типов целочисленных данных в системе программирования С++.

**Вариант 9++**

Разработать объектно-ориентированную программу для формирования квадратной матрицы Вандермонда по образующему вектору из необязательно различных десятичных цифр от 1 до 9, столбцы которой являются последовательными натуральными степенями элементов образующего вектора, и построения ее транспонированной матрицы. Размер матрицы Вандермонда определяется длиной ее образующего вектора, который должен передаваться программе в форме цифровой строки через аргумент командной строки ее вызова. Результат построения матрицы Вандермонда и ее транспонированной матрицы должен отображаться через поток стандартного вывода, где все столбцы должны быть выровнены полевой границе своих элементов. Программная реализация указанных построений должна быть основана на разработке класса матрицы Вандермонда, производного от базового класса матрица с динамическим распределением памяти для таблицы целочисленных значений ее элементов с учетом оценки возможности их вычислений для стандартных типов целочисленных данных в системе программирования С++.

**Вариант 10++**

Разработать объектно-ориентированную программу моделирования абсолютно неупругого центрального удара 2-х тел с заданными линейными скоростями и массами, которая вычисляет их общую массу и скорость после удара на основе закона сохранения импульса. Массы и скорости тел до удара должны быть заданы парами целочисленных значений, которые передаются программе через 2 аргумента командной строки ее вызова. При этом каждый аргумент должен быть записан в форме импульса до удара, где первое число определяет массу тела, а второе – его скорость. Указанные значения в каждой паре должны разделяться символом умножения. Результат моделирования должен отображаться через поток стандартного вывода двумя вещественными числами, которые обозначают массу и скорость тела, образовавшегося после удара. При разработке программы моделирования необходимо использовать аппарат множественного наследования классов импульсов тел.

**Вариант 11++**

Разработать объектно-ориентированную программу вычисления периметра и площади прямоугольника на плоскости, который задан координатами 2-х своих противоположных вершин, а его стороны параллельны координатным осям. Координаты вершин должны передаваться программе парами целых чисел через 2 аргумента командной строки ее вызова. При этом первое число в каждой паре должно обозначать координату вершины по оси Х, а второе по оси Y. Для разделения координат в каждой паре должен использоваться символ точки с запятой. Результаты вычислений площади и периметра прямоугольника должны отображаться через поток стандартного вывода. Программная реализация указанных вычислений должна быть основана на разработке контейнерного класса изотетичного прямоугольника, компонентными данными которого являются объекты класса противоположных вершин, компонентные методы обеспечивают вычисление его периметра и площади.

**Вариант 12++**

Разработать объектно-ориентированную программу для вычисления температуры смеси 2-х заданных масс одинаковой жидкости с различной температурой на основе уравнения теплового баланса без учета теплообмена с окружающей средой. Значения масс и температур жидкостей смеси должны задаваться целыми величинами, которые передаются программе через 2 аргумента командной строки ее вызова. При этом каждый аргумент должен задаваться парой целых чисел, первое из которых специфицирует массу, а второе – температуру жидкости. Значения указанных физических параметров в каждой паре должны разделяться символом наклонной черты. Результат вычислений должен отображаться через поток стандартного вывода вещественным числом, которое обозначает температуру смеси. При разработке программы необходимо использовать аппарат множественного наследования для описания состояния смеси.

**Вариант 13++**

Разработать объектно-ориентированную программу для формирования квадратной матрицы A из нулей, главные диагонали которой являются последовательными натуральными числами от 1 до n. Размер матрицы определяется длиной ее образующего вектора, который должен передаваться программе в форме цифровой строки через аргумент командной строки ее вызова. Результат построения матрицы должен отображаться через поток стандартного вывода, где все столбцы должны быть выровнены полевой границе своих элементов. Программная реализация указанных построений должна быть основана на разработке класса матрицы A, производного от базового класса матрица M с динамическим распределением памяти для таблицы целочисленных значений ее элементов с учетом оценки возможности их вычислений для стандартных типов целочисленных данных в системе программирования С++.

**Вариант 14++**

Разработать объектно-ориентированную программу для формирования квадратной матрицы A из нулей, главные диагонали которой являются последовательными натуральными числами от n до 1. Размер матрицы определяется длиной ее образующего вектора, который должен передаваться программе в форме цифровой строки через аргумент командной строки ее вызова. Результат построения матрицы должен отображаться через поток стандартного вывода, где все столбцы должны быть выровнены полевой границе своих элементов. Программная реализация указанных построений должна быть основана на разработке класса матрицы A, производного от базового класса матрица M с динамическим распределением памяти для таблицы целочисленных значений ее элементов с учетом оценки возможности их вычислений для стандартных типов целочисленных данных в системе программирования С++.

**Вариант 15++**

Разработать объектно-ориентированную программу для формирования квадратной матрицы A из нулей, главные диагонали которой являются последовательными натуральными числами: левая — от n до 1, правая – от 1 до n. Размер матрицы определяется длиной ее образующего вектора, который должен передаваться программе в форме цифровой строки через аргумент командной строки ее вызова. Результат построения матрицы должен отображаться через поток стандартного вывода, где все столбцы должны быть выровнены полевой границе своих элементов. Программная реализация указанных построений должна быть основана на разработке класса матрицы A, производного от базового класса матрица M с динамическим распределением памяти для таблицы целочисленных значений ее элементов с учетом оценки возможности их вычислений для стандартных типов целочисленных данных в системе программирования С++.

**Вариант 16++**

Разработать объектно-ориентированную программу для формирования квадратной матрицы A из нулей, главные диагонали которой являются последовательными натуральными числами: левая — от 1 до n, правая – от n до 1. Размер матрицы определяется длиной ее образующего вектора, который должен передаваться программе в форме цифровой строки через аргумент командной строки ее вызова. Результат построения матрицы должен отображаться через поток стандартного вывода, где все столбцы должны быть выровнены полевой границе своих элементов. Программная реализация указанных построений должна быть основана на разработке класса матрицы A, производного от базового класса матрица M с динамическим распределением памяти для таблицы целочисленных значений ее элементов с учетом оценки возможности их вычислений для стандартных типов целочисленных данных в системе программирования С++.

**Вариант 17++**

Разработать объектно-ориентированную программу для формирования квадратной матрицы A из нулей, главные диагонали которой являются последовательными натуральными числами от 1 до n и построения ее транспонированной матрицы. Размер матрицы определяется длиной ее образующего вектора, который должен передаваться программе в форме цифровой строки через аргумент командной строки ее вызова. Результат построения матрицы и ее транспонированной матрицы должен отображаться через поток стандартного вывода, где все столбцы должны быть выровнены полевой границе своих элементов. Программная реализация указанных построений должна быть основана на разработке класса матрицы A, производного от базового класса матрица M с динамическим распределением памяти для таблицы целочисленных значений ее элементов с учетом оценки возможности их вычислений для стандартных типов целочисленных данных в системе программирования С++.

**Вариант 18++**

Разработать объектно-ориентированную программу для формирования квадратной матрицы A из нулей, главные диагонали которой являются последовательными натуральными числами от n до 1 и построения ее транспонированной матрицы. Размер матрицы определяется длиной ее образующего вектора, который должен передаваться программе в форме цифровой строки через аргумент командной строки ее вызова. Результат построения матрицы и ее транспонированной матрицы должен отображаться через поток стандартного вывода, где все столбцы должны быть выровнены полевой границе своих элементов. Программная реализация указанных построений должна быть основана на разработке класса матрицы A, производного от базового класса матрица M с динамическим распределением памяти для таблицы целочисленных значений ее элементов с учетом оценки возможности их вычислений для стандартных типов целочисленных данных в системе программирования С++.

**Вариант 19++**

Разработать объектно-ориентированную программу для формирования квадратной матрицы A из нулей, главные диагонали которой являются последовательными натуральными числами: левая — от 1 до n, правая – от n до 1 и построения ее транспонированной матрицы. Размер матрицы определяется длиной ее образующего вектора, который должен передаваться программе в форме цифровой строки через аргумент командной строки ее вызова. Результат построения матрицы и ее транспонированной матрицы должен отображаться через поток стандартного вывода, где все столбцы должны быть выровнены полевой границе своих элементов. Программная реализация указанных построений должна быть основана на разработке класса матрицы A, производного от базового класса матрица M с динамическим распределением памяти для таблицы целочисленных значений ее элементов с учетом оценки возможности их вычислений для стандартных типов целочисленных данных в системе программирования С++.

**Вариант 20++**

Разработать объектно-ориентированную программу для формирования квадратной матрицы A из нулей, главные диагонали которой являются последовательными натуральными числами: левая — от n до 1, правая – от 1 до n и построения ее транспонированной матрицы. Размер матрицы определяется длиной ее образующего вектора, который должен передаваться программе в форме цифровой строки через аргумент командной строки ее вызова. Результат построения матрицы и ее транспонированной матрицы должен отображаться через поток стандартного вывода, где все столбцы должны быть выровнены полевой границе своих элементов. Программная реализация указанных построений должна быть основана на разработке класса матрицы A, производного от базового класса матрица M с динамическим распределением памяти для таблицы целочисленных значений ее элементов с учетом оценки возможности их вычислений для стандартных типов целочисленных данных в системе программирования С++.

**Вариант 21++**

Разработать объектно-ориентированную программу для формирования квадратной матрицы A из нулей и построения верхней треугольной матрицы, главная диагональ которой заполняется последовательными натуральными числами от n до 1, а строки – это числа из арифметической прогрессии со знаменателем 1 (например, первая – n, n-1, n-2,…2,1) и построения ее транспонированной матрицы. Размер матрицы определяется длиной ее образующего вектора, который должен передаваться программе в форме цифровой строки через аргумент командной строки ее вызова. Результат построения матрицы и ее транспонированной матрицы должен отображаться через поток стандартного вывода, где все столбцы должны быть выровнены полевой границе своих элементов. Программная реализация указанных построений должна быть основана на разработке класса матрицы A, производного от базового класса матрица M с динамическим распределением памяти для таблицы целочисленных значений ее элементов с учетом оценки возможности их вычислений для стандартных типов целочисленных данных в системе программирования С++.

**Вариант 22++**

Разработать объектно-ориентированную программу для формирования квадратной матрицы A из нулей и построения нижней треугольной матрицы, главная диагональ которой заполняется последовательными натуральными числами от 1 до n, а строки – это числа из арифметической прогрессии со знаменателем 1 (например, последняя – n, n-1, n-2,…2,1) и построения ее транспонированной матрицы. Размер матрицы определяется длиной ее образующего вектора, который должен передаваться программе в форме цифровой строки через аргумент командной строки ее вызова. Результат построения матрицы и ее транспонированной матрицы должен отображаться через поток стандартного вывода, где все столбцы должны быть выровнены полевой границе своих элементов. Программная реализация указанных построений должна быть основана на разработке класса матрицы A, производного от базового класса матрица M с динамическим распределением памяти для таблицы целочисленных значений ее элементов с учетом оценки возможности их вычислений для стандартных типов целочисленных данных в системе программирования С++.

**Вариант 23++**

Разработать объектно-ориентированную программу для формирования квадратной матрицы A из нулей и построения нижней треугольной матрицы, главная диагональ которой заполняется последовательными натуральными числами от 1 до n, а строки – это числа из арифметической прогрессии со знаменателем 1 (например, последняя – n, n-1, n-2,…2,1). Размер матрицы определяется длиной ее образующего вектора, который должен передаваться программе в форме цифровой строки через аргумент командной строки ее вызова. Результат построения матрицы должен отображаться через поток стандартного вывода, где все столбцы должны быть выровнены полевой границе своих элементов. Программная реализация указанных построений должна быть основана на разработке класса матрицы A, производного от базового класса матрица M с динамическим распределением памяти для таблицы целочисленных значений ее элементов с учетом оценки возможности их вычислений для стандартных типов целочисленных данных в системе программирования С++.

**Вариант 24++**

Разработать объектно-ориентированную программу для формирования квадратной матрицы A из нулей и построения верхней треугольной матрицы, главная диагональ которой заполняется последовательными натуральными числами от n до 1, а строки – это числа из арифметической прогрессии со знаменателем 1 (например, первая – n, n-1, n-2,…2,1). Размер матрицы определяется длиной ее образующего вектора, который должен передаваться программе в форме цифровой строки через аргумент командной строки ее вызова. Результат построения матрицы должен отображаться через поток стандартного вывода, где все столбцы должны быть выровнены полевой границе своих элементов. Программная реализация указанных построений должна быть основана на разработке класса матрицы A, производного от базового класса матрица M с динамическим распределением памяти для таблицы целочисленных значений ее элементов с учетом оценки возможности их вычислений для стандартных типов целочисленных данных в системе программирования С++.

**Вариант 25++**

Разработать объектно-ориентированную программу для формирования квадратной матрицы A из нулей, главные диагонали которой являются последовательными натуральными числами от 1 до n и вычисления ее определителя. Размер матрицы определяется длиной ее образующего вектора, который должен передаваться программе в форме цифровой строки через аргумент командной строки ее вызова. Результат построения матрицы и вычисления ее определителя должен отображаться через поток стандартного вывода, где все столбцы матрицы должны быть выровнены полевой границе своих элементов. Программная реализация указанных построений должна быть основана на разработке класса матрицы A, производного от базового класса матрица M с динамическим распределением памяти для таблицы целочисленных значений ее элементов с учетом оценки возможности их вычислений для стандартных типов целочисленных данных в системе программирования С++, а метод вычисления определителя матрицы должен быть объявлен в производном классе.

**Вариант 26++**

Разработать объектно-ориентированную программу для формирования квадратной матрицы A из нулей, главные диагонали которой являются последовательными натуральными числами от n до 1 и вычисления ее определителя. Размер матрицы определяется длиной ее образующего вектора, который должен передаваться программе в форме цифровой строки через аргумент командной строки ее вызова. Результат построения матрицы и вычисления ее определителя должен отображаться через поток стандартного вывода, где все столбцы должны быть выровнены полевой границе своих элементов. Программная реализация указанных построений должна быть основана на разработке класса матрицы A, производного от базового класса матрица M с динамическим распределением памяти для таблицы целочисленных значений ее элементов с учетом оценки возможности их вычислений для стандартных типов целочисленных данных в системе программирования С++, а метод вычисления определителя матрицы должен быть объявлен в производном классе.

**Вариант 27++**

Разработать объектно-ориентированную программу для формирования квадратной матрицы A из нулей, главные диагонали которой являются последовательными натуральными числами: левая — от n до 1, правая – от 1 до n и вычисления ее определителя. Размер матрицы определяется длиной ее образующего вектора, который должен передаваться программе в форме цифровой строки через аргумент командной строки ее вызова. Результат построения матрицы и вычисления ее определителя должен отображаться через поток стандартного вывода, где все столбцы должны быть выровнены полевой границе своих элементов. Программная реализация указанных построений должна быть основана на разработке класса матрицы A, производного от базового класса матрица M с динамическим распределением памяти для таблицы целочисленных значений ее элементов с учетом оценки возможности их вычислений для стандартных типов целочисленных данных в системе программирования С++, а метод вычисления определителя матрицы должен быть объявлен в производном классе.

**Вариант 28++**

Разработать объектно-ориентированную программу для формирования квадратной матрицы A из нулей, главные диагонали которой являются последовательными натуральными числами: левая — от 1 до n, правая – от n до 1 и вычисления ее определителя. Размер матрицы определяется длиной ее образующего вектора, который должен передаваться программе в форме цифровой строки через аргумент командной строки ее вызова. Результат построения матрицы и вычисления ее определителя должен отображаться через поток стандартного вывода, где все столбцы должны быть выровнены полевой границе своих элементов. Программная реализация указанных построений должна быть основана на разработке класса матрицы A, производного от базового класса матрица M с динамическим распределением памяти для таблицы целочисленных значений ее элементов с учетом оценки возможности их вычислений для стандартных типов целочисленных данных в системе программирования С++, а метод вычисления определителя матрицы должен быть объявлен в производном классе.

# Приложение 3. Контрольные работы

## Виртуальные методы

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно наращивать сумму, достигнутую на предыдущем ходе противника, выбирая числовое приращение, которое не превышает заданной фиксированной величины N. Игра должна продолжаться, пока значение суммы, набранной обоими игроками, меньше заданной величины S>N. Победителем считается игрок, который на своем ходе смог первым набрать требуемую сумму M. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставлено право первого хода. Человек может делать любые допустимые ходы, сообщая величину приращения суммы через стандартный ввод. При любой ошибке ввода сумма автоматически увеличивается на 1, и ход передается компьютеру. Компьютер должен действовать по выигрышной стратегии, согласно которой остаток до предельной суммы S после каждого его хода должен быть кратен (N+1). Если выигрышный ход сделать невозможно, компьютер должен увеличивать текущую сумму на 1. В любом случае приращение суммы на ходе компьютера должно отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются значение суммы S, которую должны набрать игроки, и предельная величина N ее приращения за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать запрос, который отображает текущую величину суммы и идентифицирует игрока, чей ход ожидается в данный момент. Партию игры должно завершать диагностическое сообщение, которое идентифицирует ее итоговый результат. При разработке программы необходимо реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к значению набранной суммы от абстрактного базового класса.

Вариант 1V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно наращивать сумму, достигнутую на предыдущем ходе противника, выбирая числовое приращение, которое не превышает заданной фиксированной величины N. Игра должна продолжаться, пока значение суммы, набранной обоими игроками, меньше заданной величины S>N. Победителем считается игрок, который на своем ходе смог первым набрать требуемую сумму M. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставлено право первого хода. Человек может делать любые допустимые ходы, сообщая величину приращения суммы через стандартный ввод. При любой ошибке ввода сумма автоматически увеличивается на 1, и ход передается компьютеру. Компьютер должен действовать по выигрышной стратегии, согласно которой остаток до предельной суммы S после каждого его хода должен быть кратен (N+1). Если выигрышный ход сделать невозможно, компьютер должен увеличивать текущую сумму на N. В любом случае приращение суммы на ходе компьютера должно отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются значение суммы S, которую должны набрать игроки, и предельная величина N ее приращения за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать запрос, который отображает текущую величину суммы и идентифицирует игрока, чей ход ожидается в данный момент. Партию игры должно завершать диагностическое сообщение, которое идентифицирует ее итоговый результат. При разработке программы необходимо реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к значению набранной суммы от абстрактного базового класса.

Вариант 2V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно наращивать сумму, достигнутую на предыдущем ходе противника, выбирая числовое приращение, которое не превышает заданной фиксированной величины N. Игра должна продолжаться, пока значение суммы, набранной обоими игроками, меньше заданной величины S>N. Победителем считается игрок, который на своем ходе смог первым набрать требуемую сумму M. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставлено право первого хода. Человек может делать любые допустимые ходы, сообщая величину приращения суммы через стандартный ввод. При любой ошибке ввода сумма автоматически увеличивается на N, и ход передается компьютеру. Компьютер должен действовать по выигрышной стратегии, согласно которой остаток до предельной суммы S после каждого его хода должен быть кратен (N+1). Если выигрышный ход сделать невозможно, компьютер должен увеличивать текущую сумму на 1. В любом случае приращение суммы на ходе компьютера должно отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются значение суммы S, которую должны набрать игроки, и предельная величина N ее приращения за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать запрос, который отображает текущую величину суммы и идентифицирует игрока, чей ход ожидается в данный момент. Партию игры должно завершать диагностическое сообщение, которое идентифицирует ее итоговый результат. При разработке программы необходимо реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к значению набранной суммы от абстрактного базового класса.

Вариант 3V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно наращивать сумму, достигнутую на предыдущем ходе противника, выбирая числовое приращение, которое не превышает заданной фиксированной величины N. Игра должна продолжаться, пока значение суммы, набранной обоими игроками, меньше заданной величины S>N. Победителем считается игрок, который на своем ходе смог первым набрать требуемую сумму M. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставлено право первого хода. Человек может делать любые допустимые ходы, сообщая величину приращения суммы через стандартный ввод. При любой ошибке ввода сумма автоматически увеличивается на N, и ход передается компьютеру. Компьютер должен действовать по выигрышной стратегии, согласно которой остаток до предельной суммы S после каждого его хода должен быть кратен (N+1). Если выигрышный ход сделать невозможно, компьютер должен увеличивать текущую сумму на N. В любом случае приращение суммы на ходе компьютера должно отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются значение суммы S, которую должны набрать игроки, и предельная величина N ее приращения за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать запрос, который отображает текущую величину суммы и идентифицирует игрока, чей ход ожидается в данный момент. Партию игры должно завершать диагностическое сообщение, которое идентифицирует ее итоговый результат. При разработке программы необходимо реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к значению набранной суммы от абстрактного базового класса.

Вариант 4V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно наращивать сумму, достигнутую на предыдущем ходе противника, выбирая числовое приращение, которое не превышает заданной фиксированной величины N. Игра должна продолжаться, пока значение суммы, набранной обоими игроками, меньше заданной величины S>N. Победителем считается игрок, который на своем ходе смог первым набрать требуемую сумму M. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставлено право первого хода. Человек может делать любые допустимые ходы, сообщая величину приращения суммы через стандартный ввод. При любой ошибке ввода игра должна аварийно завершаться соответствующим диагностическим сообщением. Компьютер должен действовать по выигрышной стратегии, согласно которой остаток до предельной суммы S после каждого его хода должен быть кратен (N+1). Если выигрышный ход сделать невозможно, компьютер должен увеличивать текущую сумму на 1. В любом случае приращение суммы на ходе компьютера должно отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются значение суммы S, которую должны набрать игроки, и предельная величина N ее приращения за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать запрос, который отображает текущую величину суммы и идентифицирует игрока, чей ход ожидается в данный момент. Партию игры должно завершать диагностическое сообщение, которое идентифицирует ее итоговый результат. При разработке программы необходимо реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к значению набранной суммы от абстрактного базового класса.

Вариант 5V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно наращивать сумму, достигнутую на предыдущем ходе противника, выбирая числовое приращение, которое не превышает заданной фиксированной величины N. Игра должна продолжаться, пока значение суммы, набранной обоими игроками, меньше заданной величины S>N. Победителем считается игрок, который на своем ходе смог первым набрать требуемую сумму M. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставлено право первого хода. Человек может делать любые допустимые ходы, сообщая величину приращения суммы через стандартный ввод. При любой ошибке ввода игра должна аварийно завершаться соответствующим диагностическим сообщением. Компьютер должен действовать по выигрышной стратегии, согласно которой остаток до предельной суммы S после каждого его хода должен быть кратен (N+1). Если выигрышный ход сделать невозможно, компьютер должен увеличивать текущую сумму на N. В любом случае приращение суммы на ходе компьютера должно отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются значение суммы S, которую должны набрать игроки, и предельная величина N ее приращения за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать запрос, который отображает текущую величину суммы и идентифицирует игрока, чей ход ожидается в данный момент. Партию игры должно завершать диагностическое сообщение, которое идентифицирует ее итоговый результат. При разработке программы необходимо реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к значению набранной суммы от абстрактного базового класса.

Вариант 6V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно наращивать сумму, достигнутую на предыдущем ходе противника, выбирая числовое приращение, которое не превышает заданной фиксированной величины N. Игра должна продолжаться, пока значение суммы, набранной обоими игроками, меньше заданной величины S>N. Победителем считается игрок, который на своем ходе смог первым набрать требуемую сумму M. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставлено право первого хода. Человек может делать любые допустимые ходы, сообщая величину приращения суммы через стандартный ввод. При любой ошибке ввода ему должна предоставляться возможность повторить ход. Компьютер должен действовать по выигрышной стратегии, согласно которой остаток до предельной суммы S после каждого его хода должен быть кратен (N+1). Если выигрышный ход сделать невозможно, компьютер должен увеличивать текущую сумму на 1. В любом случае приращение суммы на ходе компьютера должно отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются значение суммы S, которую должны набрать игроки, и предельная величина N ее приращения за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать запрос, который отображает текущую величину суммы и идентифицирует игрока, чей ход ожидается в данный момент. Партию игры должно завершать диагностическое сообщение, которое идентифицирует ее итоговый результат. При разработке программы необходимо реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к значению набранной суммы от абстрактного базового класса.

Вариант 7V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно наращивать сумму, достигнутую на предыдущем ходе противника, выбирая числовое приращение, которое не превышает заданной фиксированной величины N. Игра должна продолжаться, пока значение суммы, набранной обоими игроками, меньше заданной величины S>N. Победителем считается игрок, который на своем ходе смог первым набрать требуемую сумму M. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставлено право первого хода. Человек может делать любые допустимые ходы, сообщая величину приращения суммы через стандартный ввод. При любой ошибке ввода ему должна предоставляться возможность повторить ход. Компьютер должен действовать по выигрышной стратегии, согласно которой остаток до предельной суммы S после каждого его хода должен быть кратен (N+1). Если выигрышный ход сделать невозможно, компьютер должен увеличивать текущую сумму на N. В любом случае приращение суммы на ходе компьютера должно отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются значение суммы S, которую должны набрать игроки, и предельная величина N ее приращения за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать запрос, который отображает текущую величину суммы и идентифицирует игрока, чей ход ожидается в данный момент. Партию игры должно завершать диагностическое сообщение, которое идентифицирует ее итоговый результат. При разработке программы необходимо реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к значению набранной суммы от абстрактного базового класса.

Вариант 8V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно наращивать сумму, достигнутую на предыдущем ходе противника, выбирая числовое приращение, которое не превышает заданной фиксированной величины N. Игра должна продолжаться, пока значение суммы, набранной обоими игроками, меньше заданной величины S>N. Победителем считается игрок, который на своем ходе смог первым набрать требуемую сумму M. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставлено право первого хода. Человек может делать любые допустимые ходы, сообщая величину приращения суммы через стандартный ввод. При любой ошибке ввода сумма автоматически увеличивается на 1, и ход передается компьютеру. Компьютер должен действовать по выигрышной стратегии, согласно которой остаток до предельной суммы S после каждого его хода должен быть кратен (N+1). Если выигрышный ход сделать невозможно, компьютер должен повторить ход человека. В любом случае величина приращения суммы на ходе компьютера должна отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются значение суммы S, которую должны набрать игроки, и предельная величина N ее приращения за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать запрос, который отображает текущую величину суммы и идентифицирует игрока, чей ход ожидается в данный момент. Партию игры должно завершать диагностическое сообщение, которое идентифицирует ее итоговый результат. При разработке программы необходимо реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к значению набранной суммы от абстрактного базового класса.

Вариант 9V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно наращивать сумму, достигнутую на предыдущем ходе противника, выбирая числовое приращение, которое не превышает заданной фиксированной величины N. Игра должна продолжаться, пока значение суммы, набранной обоими игроками, меньше заданной величины S>N. Победителем считается игрок, который на своем ходе смог первым набрать требуемую сумму M. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставлено право первого хода. Человек может делать любые допустимые ходы, сообщая величину приращения суммы через стандартный ввод. При любой ошибке ввода сумма автоматически увеличивается на N, и ход передается компьютеру. Компьютер должен действовать по выигрышной стратегии, согласно которой остаток до предельной суммы S после каждого его хода должен быть кратен (N+1). Если выигрышный ход сделать невозможно, компьютер должен повторить ход человека. В любом случае величина приращения суммы на ходе компьютера должна отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются значение суммы S, которую должны набрать игроки, и предельная величина N ее приращения за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать запрос, который отображает текущую величину суммы и идентифицирует игрока, чей ход ожидается в данный момент. Партию игры должно завершать диагностическое сообщение, которое идентифицирует ее итоговый результат. При разработке программы необходимо реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к значению набранной суммы от абстрактного базового класса.

Вариант 10V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно выбирать из кучи заданного размера S любое количество предметов, которое не превышает фиксированного значения N<S. Игра должна продолжаться, пока текущий размер кучи больше 0. Победителем считается игрок, который взял последний предмет кучи. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставляется право первого хода. Человек может делать любые допустимые ходы, сообщая число взятых им предметов через стандартный ввод. При любой ошибке ввода размер кучи автоматически уменьшается на 1, и ход передается компьютеру. Компьютер должен действовать по выигрышной стратегии, согласно которой остаток кучи после каждого его хода должен быть кратен (N+1). Если такой выигрышный ход сделать невозможно, компьютер должен взять из кучи 1 предмет. В любом случае число предметов, которое взял из кучи компьютер должно отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются размер кучи S и предельное число предметов N, которое можно взять из кучи за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать информационный запрос, который отображает текущий размер кучи и идентифицирует игрока, чей ход ожидается в данный момент. Игра должна завершаться диагностическим сообщением, которое идентифицирует ее итоговый результат. При разработке программы необходимо реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к куче от абстрактного базового класса.

Вариант 11V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно выбирать из кучи заданного размера S любое количество предметов, которое не превышает фиксированного значения N<S. Игра должна продолжаться, пока текущий размер кучи больше 0. Победителем считается игрок, который взял последний предмет кучи. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставляется право первого хода. Человек может делать любые допустимые ходы, сообщая число взятых им предметов через стандартный ввод. При любой ошибке ввода размер кучи автоматически уменьшается на 1, и ход передается компьютеру. Компьютер должен действовать по выигрышной стратегии, согласно которой остаток кучи после каждого его хода должен быть кратен (N+1). Если такой выигрышный ход сделать нельзя, компьютер должен взять из кучи N предметов. В любом случае число предметов, которое взял из кучи компьютер должно отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются размер кучи S и предельное число предметов N, которое можно взять из кучи за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать информационный запрос, который отображает текущий размер кучи и идентифицирует игрока, чей ход ожидается в данный момент. Игра должна завершаться диагностическим сообщением, которое идентифицирует ее итоговый результат. При разработке программы необходимо реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к куче от абстрактного базового класса.

Вариант 12V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно выбирать из кучи заданного размера S любое количество предметов, которое не превышает фиксированного значения N<S. Игра должна продолжаться, пока текущий размер кучи больше 0. Победителем считается игрок, который взял последний предмет кучи. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставляется право первого хода. Человек может делать любые допустимые ходы, сообщая число взятых им предметов через стандартный ввод. При любой ошибке ввода размер кучи автоматически уменьшается на N, и ход передается компьютеру. Компьютер должен действовать по выигрышной стратегии, согласно которой остаток кучи после каждого его хода должен быть кратен (N+1). Если такой выигрышный ход сделать невозможно, компьютер должен взять из кучи 1 предмет. В любом случае число предметов, которое взял из кучи компьютер должно отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются размер кучи S и предельное число предметов N, которое можно взять из кучи за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать информационный запрос, который отображает текущий размер кучи и идентифицирует игрока, чей ход ожидается в данный момент. Игра должна завершаться диагностическим сообщением, которое идентифицирует ее итоговый результат. При разработке программы необходимо реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к куче от абстрактного базового класса.

Вариант 13V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно выбирать из кучи заданного размера S любое количество предметов, которое не превышает фиксированного значения N<S. Игра должна продолжаться, пока текущий размер кучи больше 0. Победителем считается игрок, который взял последний предмет кучи. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставляется право первого хода. Человек может делать любые допустимые ходы, сообщая число взятых им предметов через стандартный ввод. При любой ошибке ввода размер кучи автоматически уменьшается на N, и ход передается компьютеру. Компьютер должен действовать по выигрышной стратегии, согласно которой остаток кучи после каждого его хода должен быть кратен (N+1). Если такой выигрышный ход сделать нельзя, компьютер должен взять из кучи N предметов. В любом случае число предметов, которое взял из кучи компьютер должно отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются размер кучи S и предельное число предметов N, которое можно взять из кучи за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать информационный запрос, который отображает текущий размер кучи и идентифицирует игрока, чей ход ожидается в данный момент. Игра должна завершаться диагностическим сообщением, которое идентифицирует ее итоговый результат. При разработке программы необходимо реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к куче от абстрактного базового класса.

Вариант 14V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно выбирать из кучи заданного размера S любое количество предметов, которое не превышает фиксированного значения N<S. Игра должна продолжаться, пока текущий размер кучи больше 0. Победителем считается игрок, который взял последний предмет кучи. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставляется право первого хода. Человек может делать любые допустимые ходы, сообщая число взятых им предметов через стандартный ввод. При любой ошибке ввода партия игры должна аварийно завершаться с соответствующей диагностикой. Компьютер должен действовать по выигрышной стратегии, согласно которой остаток кучи после каждого его хода должен быть кратен (N+1). Если такой выигрышный ход сделать невозможно, компьютер должен взять из кучи 1 предмет. В любом случае число предметов, которое взял из кучи компьютер должно отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются размер кучи S и предельное число предметов N, которое можно взять из кучи за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать информационный запрос, который отображает текущий размер кучи и идентифицирует игрока, чей ход ожидается в данный момент. Игра должна завершаться диагностическим сообщением, которое идентифицирует ее итоговый результат. При разработке программы необходимо реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к куче от абстрактного базового класса.

Вариант 15V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно выбирать из кучи заданного размера S любое количество предметов, которое не превышает фиксированного значения N<S. Игра должна продолжаться, пока текущий размер кучи больше 0. Победителем считается игрок, который взял последний предмет кучи. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставляется право первого хода. Человек может делать любые допустимые ходы, сообщая число взятых им предметов через стандартный ввод. При любой ошибке ввода партия игры должна аварийно завершаться с соответствующей диагностикой. Компьютер должен действовать по выигрышной стратегии, согласно которой остаток кучи после каждого его хода должен быть кратен (N+1). Если такой выигрышный ход сделать нельзя, компьютер должен взять из кучи N предметов. В любом случае число предметов, которое взял из кучи компьютер должно отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются размер кучи S и предельное число предметов N, которое можно взять из кучи за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать информационный запрос, который отображает текущий размер кучи и идентифицирует игрока, чей ход ожидается в данный момент. Игра должна завершаться диагностическим сообщением, которое идентифицирует ее итоговый результат. При разработке программы необходимо реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к куче от абстрактного базового класса.

Вариант 16V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно выбирать из кучи заданного размера S любое количество предметов, которое не превышает фиксированного значения N<S. Игра должна продолжаться, пока текущий размер кучи больше 0. Победителем считается игрок, который взял последний предмет кучи. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставляется право первого хода. Человек может делать любые допустимые ходы, сообщая число взятых им предметов через стандартный ввод. При любой ошибке ввода ему должна предоставляться возможность повторить ход. Компьютер должен действовать по выигрышной стратегии, согласно которой остаток кучи после каждого его хода должен быть кратен (N+1). Если такой выигрышный ход сделать невозможно, компьютер должен взять из кучи 1 предмет. В любом случае число предметов, которое взял из кучи компьютер должно отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются размер кучи S и предельное число предметов N, которое можно взять из кучи за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход партии игры должен сопровождать информационный запрос, который отображает текущий размер кучи и идентифицирует игрока, чей ход ожидается в данный момент. Игра должна завершаться диагностическим сообщением, которое идентифицирует ее итоговый результат. При разработке программы необходимо реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к куче от абстрактного базового класса.

Вариант 17V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно выбирать из кучи заданного размера S любое количество предметов, которое не превышает фиксированного значения N<S. Игра должна продолжаться, пока текущий размер кучи больше 0. Победителем считается игрок, который взял последний предмет кучи. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставляется право первого хода. Человек может делать любые допустимые ходы, сообщая число взятых им предметов через стандартный ввод. При любой ошибке ввода ему должна предоставляться возможность повторить ход. Компьютер должен действовать по выигрышной стратегии, согласно которой остаток кучи после каждого его хода должен быть кратен (N+1). Если такой выигрышный ход сделать нельзя, компьютер должен взять из кучи N предметов. В любом случае число предметов, которое взял из кучи компьютер должно отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются размер кучи S и предельное число предметов N, которое можно взять из кучи за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход партии игры должен сопровождать информационный запрос, который отображает текущий размер кучи и идентифицирует игрока, чей ход ожидается в данный момент. Игра должна завершаться диагностическим сообщением, которое идентифицирует ее итоговый результат. При разработке программы необходимо реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к куче от абстрактного базового класса.

Вариант 18V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно выбирать из кучи заданного размера S любое количество предметов, которое не превышает фиксированного значения N<S. Игра должна продолжаться, пока текущий размер кучи больше 0. Победителем считается игрок, который взял последний предмет кучи. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставляется право первого хода. Человек может делать любые допустимые ходы, сообщая число взятых им предметов через стандартный ввод. При любой ошибке ввода размер кучи автоматически уменьшается на 1, и ход передается компьютеру. Компьютер должен действовать по выигрышной стратегии, согласно которой остаток кучи после каждого его хода должен быть кратен (N+1). Если такой выигрышный ход сделать невозможно, компьютер должен повторить ход человека. В любом случае число предметов, которое взял из кучи компьютер должно отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются размер кучи S и предельное число предметов N, которое можно взять из кучи за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать информационный запрос, который отображает текущий размер кучи и идентифицирует игрока, чей ход ожидается в данный момент. Игра должна завершаться диагностическим сообщением, которое идентифицирует ее итоговый результат. При разработке программы необходимо реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к куче от абстрактного базового класса.

Вариант 19V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно выбирать из кучи заданного размера S любое количество предметов, которое не превышает фиксированного значения N<S. Игра должна продолжаться, пока текущий размер кучи больше 0. Победителем считается игрок, который взял последний предмет кучи. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставляется право первого хода. Человек может делать любые допустимые ходы, сообщая число взятых им предметов через стандартный ввод. При любой ошибке ввода размер кучи автоматически уменьшается на N, и ход передается компьютеру. Компьютер должен действовать по выигрышной стратегии, согласно которой остаток кучи после каждого его хода должен быть кратен (N+1). Если такой выигрышный ход сделать нельзя, компьютер должен повторить ход человека. В любом случае число предметов, которое взял из кучи компьютер должно отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются размер кучи S и предельное число предметов N, которое можно взять из кучи за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать информационный запрос, который отображает текущий размер кучи и идентифицирует игрока, чей ход ожидается в данный момент. Игра должна завершаться диагностическим сообщением, которое идентифицирует ее итоговый результат. При разработке программы необходимо реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к куче от абстрактного базового класса.

Вариант 20V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно выбирать из кучи заданного размера S любое количество предметов, которое не превышает фиксированного значения N<S. Игра должна продолжаться, пока текущий размер кучи больше 0. Победителем считается игрок, который вынудил противника взять последний предмет кучи. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставляется право первого хода. Человек может делать любые допустимые ходы, сообщая число взятых предметов через стандартный ввод. При любой ошибке ввода размер кучи автоматически уменьшается на 1 предмет, и ход передается компьютеру. Компьютер должен действовать по выигрышной стратегии, чтобы остаток кучи после каждого его хода был равен (XN+X+1), где X обозначает наибольшее целое число, при котором этот остаток будет меньше текущего размера кучи. Если такой выигрышный ход сделать невозможно, компьютер должен повторить ход человека. В любом случае результат хода компьютера должен отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются размер кучи S и предельное число предметов N, которое можно взять из кучи за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать информационный запрос, который отображает текущий размер кучи и идентифицирует игрока, чей ход ожидается в данный момент. Игра должна завершаться диагностическим сообщением, которое идентифицирует ее итоговый результат. При разработке программы следует реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к куче от абстрактного базового класса.

Вариант 21V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно выбирать из кучи заданного размера S любое количество предметов, которое не превышает фиксированного значения N<S. Игра должна продолжаться, пока текущий размер кучи больше 0. Победителем считается игрок, который вынудил противника взять последний предмет кучи. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставляется право первого хода. Человек может делать любые допустимые ходы, сообщая число взятых предметов через стандартный ввод. При любой ошибке ввода размер кучи автоматически уменьшается на N, и ход передается компьютеру. Компьютер должен действовать по выигрышной стратегии, чтобы остаток кучи после каждого его хода был равен (XN+X+1), где X обозначает наибольшее целое число, при котором этот остаток будет меньше текущего размера кучи. Если такой выигрышный ход сделать невозможно, компьютер должен повторить ход человека. В любом случае результат хода компьютера должен отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются размер кучи S и предельное число предметов N, которое можно взять из кучи за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать информационный запрос, который отображает текущий размер кучи и идентифицирует игрока, чей ход ожидается в данный момент. Игра должна завершаться диагностическим сообщением, которое идентифицирует ее итоговый результат. При разработке программы следует реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к куче от абстрактного базового класса.

Вариант 22V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно выбирать из кучи заданного размера S любое количество предметов, которое не превышает фиксированного значения N<S. Игра должна продолжаться, пока текущий размер кучи больше 0. Победителем считается игрок, который вынудил противника взять последний предмет кучи. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставляется право первого хода. Человек может делать любые допустимые ходы, сообщая число взятых предметов через стандартный ввод. При любой ошибке ввода ему должна предоставляться возможность повторить ход. Компьютер должен действовать по выигрышной стратегии, чтобы остаток кучи после каждого его хода был равен (XN+X+1), где X обозначает наибольшее целое число, при котором этот остаток будет меньше текущего размера кучи. Если такой выигрышный ход сделать невозможно, компьютер должен взять из кучи 1 предмет. В любом случае результат хода компьютера должен отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются размер кучи S и предельное число предметов N, которое можно взять из кучи за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход партии игры должен сопровождать информационный запрос, который отображает текущий размер кучи и идентифицирует игрока, чей ход ожидается в данный момент. Игра должна завершаться диагностическим сообщением, которое идентифицирует ее итоговый результат. При разработке программы следует реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к куче от абстрактного базового класса.

Вариант 23V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно выбирать из кучи заданного размера S любое количество предметов, которое не превышает фиксированного значения N<S. Игра должна продолжаться, пока текущий размер кучи больше 0. Победителем считается игрок, который вынудил противника взять последний предмет кучи. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставляется право первого хода. Человек может делать любые допустимые ходы, сообщая число взятых предметов через стандартный ввод. При любой ошибке ввода ему должна предоставляться возможность повторить ход. Компьютер должен действовать по выигрышной стратегии, чтобы остаток кучи после каждого его хода был равен (XN+X+1), где X обозначает наибольшее целое число, при котором этот остаток будет меньше текущего размера кучи. Если такой выигрышный ход сделать невозможно, компьютер должен взять из кучи N предметов. В любом случае результат каждого хода компьютера должен отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются размер кучи S и предельное число предметов N, которое можно взять из кучи за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать информационный запрос, который отображает текущий размер кучи и идентифицирует игрока, чей ход ожидается в данный момент. Игра должна завершаться диагностическим сообщением, которое идентифицирует ее итоговый результат. При разработке программы следует реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к куче от абстрактного базового класса.

Вариант 24V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно выбирать из кучи заданного размера S любое количество предметов, которое не превышает фиксированного значения N<S. Игра должна продолжаться, пока текущий размер кучи больше 0. Победителем считается игрок, который вынудил противника взять последний предмет кучи. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставляется право первого хода. Человек может делать любые допустимые ходы, сообщая число взятых предметов через стандартный ввод. При любой ошибке ввода игра должна быть аварийно завершена с соответствующей диагностикой. Компьютер должен действовать по выигрышной стратегии, чтобы остаток кучи после каждого его хода был равен (XN+X+1), где X обозначает наибольшее целое число, при котором этот остаток будет меньше текущего размера кучи. Если такой выигрышный ход сделать невозможно, компьютер должен взять из кучи 1 предмет. В любом случае результат хода компьютера должен отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются размер кучи S и предельное число предметов N, которое можно взять из кучи за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход партии игры должен сопровождать запрос, который отображает текущий размер кучи и идентифицирует игрока, чей ход ожидается в данный момент. Игра должна завершаться диагностическим сообщением, которое идентифицирует ее итоговый результат. При разработке программы следует реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к куче от абстрактного базового класса.

Вариант 25V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно выбирать из кучи заданного размера S любое количество предметов, которое не превышает фиксированного значения N<S. Игра должна продолжаться, пока текущий размер кучи больше 0. Победителем считается игрок, который вынудил противника взять последний предмет кучи. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставляется право первого хода. Человек может делать любые допустимые ходы, сообщая число взятых предметов через стандартный ввод. При любой ошибке ввода игра должна быть аварийно завершена с соответствующей диагностикой. Компьютер должен действовать по выигрышной стратегии, чтобы остаток кучи после каждого его хода был равен (XN+X+1), где X обозначает наибольшее целое число, при котором этот остаток будет меньше текущего размера кучи. Если такой выигрышный ход сделать невозможно, компьютер должен взять из кучи N предметов. В любом случае результат каждого хода компьютера должен отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются размер кучи S и предельное число предметов N, которое можно взять из кучи за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать запрос, который отображает текущий размер кучи и идентифицирует игрока, чей ход ожидается в данный момент. Игра должна завершаться диагностическим сообщением, которое идентифицирует ее итоговый результат. При разработке программы следует реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к куче от абстрактного базового класса.

Вариант 26V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно выбирать из кучи заданного размера S любое количество предметов, которое не превышает фиксированного значения N<S. Игра должна продолжаться, пока текущий размер кучи больше 0. Победителем считается игрок, который вынудил противника взять последний предмет кучи. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставляется право первого хода. Человек может делать любые допустимые ходы, сообщая число взятых предметов через стандартный ввод. При любой ошибке ввода размер кучи автоматически уменьшается на N предметов, и ход передается компьютеру. Компьютер должен действовать по выигрышной стратегии, чтобы остаток кучи после каждого его хода был равен (XN+X+1), где X обозначает наибольшее целое число, при котором этот остаток будет меньше текущего размера кучи. Если выигрышный ход сделать невозможно, компьютер должен взять из кучи 1 предмет. В любом случае результат хода компьютера должен отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются размер кучи S и предельное число предметов N, которое можно взять из кучи за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать запрос, который отображает текущий размер кучи и идентифицирует игрока, чей ход ожидается в данный момент. Игру должно завершать диагностическое сообщение, которое идентифицирует ее итоговый результат. При разработке программы следует реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к куче от абстрактного базового класса.

Вариант 27V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно выбирать из кучи заданного размера S любое количество предметов, которое не превышает фиксированного значения N<S. Игра должна продолжаться, пока текущий размер кучи больше 0. Победителем считается игрок, который вынудил противника взять последний предмет кучи. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставляется право первого хода. Человек может делать любые допустимые ходы, сообщая число взятых предметов через стандартный ввод. При любой ошибке ввода размер кучи автоматически уменьшается на N предметов, и ход передается компьютеру. Компьютер должен действовать по выигрышной стратегии, чтобы остаток кучи после каждого его хода был равен (XN+X+1), где X обозначает наибольшее целое число, при котором этот остаток будет меньше текущего размера кучи. Если выигрышный ход сделать невозможно, компьютер должен взять из кучи N предметов. В любом случае результат каждого хода компьютера должен отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются размер кучи S и предельное число предметов N, которое можно взять из кучи за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать запрос, который отображает текущий размер кучи и идентифицирует игрока, чей ход ожидается в данный момент. Игра должна завершаться диагностическим сообщением, которое идентифицирует ее итоговый результат. При разработке программы следует реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к куче от абстрактного базового класса.

Вариант 28V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно выбирать из кучи заданного размера S любое количество предметов, которое не превышает фиксированного значения N<S. Игра должна продолжаться, пока текущий размер кучи больше 0. Победителем считается игрок, который вынудил противника взять последний предмет кучи. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставляется право первого хода. Человек может делать любые допустимые ходы, сообщая число взятых предметов через стандартный ввод. При любой ошибке ввода размер кучи автоматически уменьшается на 1 предмет, и ход передается компьютеру. Компьютер должен действовать по выигрышной стратегии, чтобы остаток кучи после каждого его хода был равен (XN+X+1), где X обозначает наибольшее целое число, при котором этот остаток будет меньше текущего размера кучи. Если выигрышный ход сделать невозможно, компьютер должен взять из кучи N предметов. В любом случае результат хода компьютера должен отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются размер кучи S и предельное число предметов N, которое можно взять из кучи за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать запрос, который отображает текущий размер кучи и идентифицирует игрока, чей ход ожидается в данный момент. Игру должно завершать диагностическое сообщение, которое идентифицирует ее итоговый результат. При разработке программы следует реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к куче от абстрактного базового класса.

Вариант 29V

Разработать объектно-ориентированную программу для числовой игры, где 2 игрока должны поочередно выбирать из кучи заданного размера S любое количество предметов, которое не превышает фиксированного значения N<S. Игра должна продолжаться, пока текущий размер кучи больше 0. Победителем считается игрок, который вынудил противника взять последний предмет кучи. Игра должна быть реализована в варианте, когда двумя противниками являются компьютер и человек, которому предоставляется право первого хода. Человек может делать любые допустимые ходы, сообщая число взятых предметов через стандартный ввод. При любой ошибке ввода размер кучи автоматически уменьшается на 1 предмет, и ход передается компьютеру. Компьютер должен действовать по выигрышной стратегии, чтобы остаток кучи после каждого его хода был равен (XN+X+1), где X обозначает наибольшее целое число, при котором этот остаток будет меньше текущего размера кучи. Если выигрышный ход сделать невозможно, компьютер должен взять из кучи 1 предмет. В любом случае результат каждого хода компьютера должен отображаться через поток стандартного вывода. Исходными данными для любой партии игры являются размер кучи S и предельное число предметов N, которое можно взять из кучи за 1 ход. Эти параметры должны передаваться программе через аргументы командной строки ее вызова. Каждый ход должен сопровождать запрос, который отображает текущий размер кучи и идентифицирует игрока, чей ход ожидается в данный момент. Игра должна завершаться диагностическим сообщением, которое идентифицирует ее итоговый результат. При разработке программы следует реализовать производные классы игроков с виртуальной функцией хода, которые наследуют интерфейс игры и доступ к куче от абстрактного базового класса.

Вариант 30V

## Перегрузка операторов

Разработать объектно-ориентированную программу вычисления разности двух любых подмножеств латинских букв, которые передаются ей аргументами командной строки. Их разность образуют все буквы первого подмножества, которых нет во втором. Полученный результат должен отображаться строкой стандартного вывода, где все буквы перечислены в алфавитном порядке. Программная реализация вычислений разности должна быть основана на разработке класса множества латинских букв с компонентным методом перегрузки оператора '' и полем приватных данных, где состав множества фиксируется двоичными разрядами целого числа. Конструктор класса должен обеспечивать преобразование исходной символьной записи множества в двоичный целочисленный формат. Необходимо также предусмотреть перегрузку оператора класса потока стандартного вывода для отображения объектов класса множества латинских букв.

Вариант 1+(М)

Разработать объектно-ориентированную программу вычисления симметрической разности двух любых подмножеств латинских букв, которые передаются ей аргументами командной строки. Такую разность образуют все различные буквы из двух заданных подмножеств. Полученный результат должен отображаться строкой стандартного вывода, где все буквы перечислены в алфавитном порядке. Программный код должен быть основан на разработке класса множества латинских букв с компонентными методами перегрузки операторов '+' и '' для выполнения теоретико-множественных операций объединения и разности. Они должны быть ориентированы на обработку поля приватных данных, где состав множества фиксируют двоичные разряды целого числа. Конструктор класса должен обеспечивать преобразование исходной символьной записи множества в двоичный целочисленный формат. Необходимо также предусмотреть перегрузку оператора класса потока стандартного вывода для отображения объектов класса множества латинских букв.

Вариант 2+(М)

Разработать объектно-ориентированную программу вычисления дополнения до полного алфавита любого подмножества латинских букв, которое передается ей аргументом командной строки. Результатом этой операции должны быть все латинские буквы, кроме заданных аргументом программы. Они должны отображаться строкой стандартного вывода, где все буквы перечислены в алфавитном порядке. Программная реализация вычислений должна быть основана на разработке класса множества латинских букв с компонентным методом перегрузки оператора '~' и полем приватных данных, где состав множества кодируется двоичными разрядами целого числа. Конструктор класса должен обеспечивать преобразование исходной символьной записи множества в двоичный целочисленный формат. Необходимо также предусмотреть перегрузку оператора класса потока стандартного вывода для отображения объектов класса множества латинских букв.

Вариант 3+(М)

Разработать объектно-ориентированную программу для перечисления всех латинских букв, которые отсутствуют в двух любых символьных строках, заданных аргументами команды ее вызова. Искомый результат должна отображать строка стандартного вывода, где все буквы перечислены в алфавитном порядке. Программная реализация таких перечислений должна быть основана на разработке класса множества латинских букв с компонентными методами перегрузки операторов '+' и '~' для выполнения теоретико-множественных операций объединения и дополнения. Они должны быть ориентированы на обработку поля приватных данных, где состав множества фиксируется двоичными разрядами целого числа. Конструктор класса должен обеспечивать преобразование исходной символьной записи множества в двоичный целочисленный формат. Необходимо также предусмотреть перегрузку оператора класса потока стандартного вывода для отображения объектов класса множества латинских букв.

Вариант 4+(М)

Разработать объектно-ориентированную программу для перечисления всех латинских букв кроме тех, которые совпадают в двух любых символьных строках, заданных аргументами команды ее вызова. Искомый результат должна отображать строка стандартного вывода, где все буквы перечислены в алфавитном порядке. Программная реализация таких перечислений должна быть основана на разработке класса множества латинских букв с компонентными методами перегрузки операторов '\*' и '~' для выполнения операций пересечения и дополнения двух множеств. Они должны быть ориентированы на обработку поля приватных данных, где состав множества фиксируется двоичными разрядами целого числа. Конструктор класса должен обеспечивать преобразование исходной символьной записи множества в двоичный целочисленный формат. Необходимо также предусмотреть перегрузку оператора класса потока стандартного вывода для отображения объектов класса множества латинских букв.

Вариант 5+(М)

Разработать объектно-ориентированную программу вычисления дополнения до полного алфавита разности двух заданных подмножеств латинских букв, которые передаются ей аргументами командной строки. Результатом этой операции должны быть все латинские буквы, кроме тех, которые есть только в первом аргументе. Они должны отображаться строкой стандартного вывода, где все буквы перечислены в алфавитном порядке. Программа должна быть основана на разработке класса множества латинских букв с компонентными методами перегрузки операторов '~' и '' для выполнения операций дополнения и разности. Они должны быть ориентированы на обработку поля приватных данных, где состав множества фиксируется двоичными разрядами целого числа. Конструктор класса должен обеспечивать преобразование исходной символьной записи множества в двоичный целочисленный формат. Необходимо также предусмотреть перегрузку оператора класса потока стандартного вывода для отображения объектов класса множества латинских букв.

Вариант 6+(М)

Разработать объектно-ориентированную программу для исключения всех латинских букв кроме тех, которые совпадают в двух любых символьных строках, заданных аргументами команды ее вызова. Все оставшиеся различные буквы должны отображаться строкой потока стандартного вывода, где все буквы перечислены в алфавитном порядке. Программная реализация таких исключений должна быть основана на разработке класса множества латинских букв с компонентным методом перегрузки оператора '^' и полем приватных данных, где состав множества кодируется двоичными разрядами целого числа. Конструктор класса должен обеспечивать преобразование исходной символьной записи подмножества латинских букв в бинарный целочисленный формат. Необходимо также предусмотреть перегрузку оператора класса потока стандартного вывода для отображения объектов класса множества латинских букв

Вариант 7+(М)

Разработать объектно-ориентированную программу для перечисления всех латинских букв, которые одновременно отсутствуют или совпадают в двух любых символьных строках, заданных аргументами команды ее вызова. Искомый результат должна отображать строка стандартного вывода, где все буквы перечислены в алфавитном порядке. Программная реализация таких перечислений должна быть основана на разработке класса множества латинских букв с компонентными методами перегрузки операторов '^' и '~' для выполнения операций исключения совпадений и дополнения. Они должны быть ориентированы на обработку поля приватных данных, где состав множества фиксируется двоичными разрядами целого числа. Конструктор класса должен обеспечивать преобразование исходной символьной записи множества в двоичный целочисленный формат. Необходимо также предусмотреть перегрузку оператора класса потока стандартного вывода для отображения объектов класса множества латинских букв.

Вариант 8+(М)

Разработать объектно-ориентированную программу вычисления объединения и пересечения двух любых подмножеств латинских букв, которые передаются ей аргументами командной строки. Полученный результат должен отображаться двумя строками стандартного вывода, где все буквы перечислены в алфавитном порядке. Программная реализация вычислений должна быть основана на разработке класса множества латинских букв с компонентными методами перегрузки операторов '+' и '\*' для выполнения теоретико-множественных операций объединения и пересечения. Они должны быть ориентированы на обработку поля приватных данных, где состав множества фиксируется двоичными разрядами целого числа. Конструктор класса должен обеспечивать преобразование исходной символьной записи множества в двоичный целочисленный формат. Необходимо также предусмотреть перегрузку оператора класса потока стандартного вывода для отображения объектов класса множества латинских букв.

Вариант 9+(М)

Разработать объектно-ориентированную программу вычисления разности двух любых подмножеств латинских букв, которые передаются ей аргументами командной строки. Их разность образуют все буквы первого подмножества, которых нет во втором. Полученный результат должен отображаться строкой стандартного вывода, где все буквы перечислены в обратном алфавитному порядке. Программная реализация вычислений разности должна быть основана на разработке класса множества латинских букв с компонентным методом перегрузки оператора '' и полем приватных данных, где состав множества фиксируется двоичными разрядами целого числа. Конструктор класса должен обеспечивать преобразование исходной символьной записи множества в двоичный целочисленный формат. Необходимо также предусмотреть перегрузку оператора класса потока стандартного вывода для отображения объектов класса множества латинских букв.

Вариант 10+(М)

Разработать объектно-ориентированную программу вычисления симметрической разности двух любых подмножеств латинских букв, которые передаются ей аргументами командной строки. Такую разность образуют все различные буквы из двух заданных подмножеств. Полученный результат должен отображаться строкой стандартного вывода, где все буквы перечислены в обратном алфавитному порядке. Программный код должен быть основан на разработке класса множества латинских букв с компонентными методами перегрузки операторов '+' и '' для выполнения теоретико-множественных операций объединения и разности. Они должны быть ориентированы на обработку поля приватных данных, где состав множества фиксируют двоичные разряды целого числа. Конструктор класса должен обеспечивать преобразование исходной символьной записи множества в двоичный целочисленный формат. Необходимо также предусмотреть перегрузку оператора класса потока стандартного вывода для отображения объектов класса множества латинских букв.

Вариант 11+(М)

Разработать объектно-ориентированную программу вычисления дополнения до полного алфавита любого подмножества латинских букв, которое передается ей через аргумент командной строки. Результатом этой операции должны быть все латинские буквы, кроме заданных аргументом программы. Они должны отображаться строкой стандартного вывода, где все буквы перечислены в обратном алфавитному порядке. Программная реализация вычислений должна быть основана на разработке класса множества латинских букв с компонентным методом перегрузки оператора '~' и полем приватных данных, где состав множества кодируется двоичными разрядами целого числа. Конструктор класса должен обеспечивать преобразование исходной символьной записи множества в двоичный целочисленный формат. Необходимо также предусмотреть перегрузку оператора класса потока стандартного вывода для отображения объектов класса множества латинских букв.

Вариант 12+(М)

Разработать объектно-ориентированную программу для перечисления всех латинских букв, которые отсутствуют в двух любых символьных строках, заданных аргументами команды ее вызова. Искомый результат должна отображать строка стандартного вывода, где все буквы перечислены в обратном алфавитном порядке. Программная реализация таких перечислений должна быть основана на разработке класса множества латинских букв с компонентными методами перегрузки операторов '+' и '~' для выполнения теоретико-множественных операций объединения и дополнения. Они должны быть ориентированы на обработку поля приватных данных, где состав множества фиксируется двоичными разрядами целого числа. Конструктор класса должен обеспечивать преобразование исходной символьной записи множества в двоичный целочисленный формат. Необходимо также предусмотреть перегрузку оператора класса потока стандартного вывода для отображения объектов класса множества латинских букв.

Вариант 13+(М)

Разработать объектно-ориентированную программу для перечисления всех латинских букв кроме тех, которые совпадают в двух любых символьных строках, заданных аргументами команды ее вызова. Искомый результат должна отображать строка стандартного вывода, где все буквы перечислены в обратном алфавитному порядке. Программная реализация таких перечислений должна быть основана на разработке класса множества латинских букв с компонентными методами перегрузки операторов '\*' и '~' для выполнения операций пересечения и дополнения двух множеств. Они должны быть ориентированы на обработку поля приватных данных, где состав множества фиксируется двоичными разрядами целого числа. Конструктор класса должен обеспечивать преобразование исходной символьной записи множества в двоичный целочисленный формат. Необходимо также предусмотреть перегрузку оператора класса потока стандартного вывода для отображения объектов класса множества латинских букв.

Вариант 14+(М)

Разработать объектно-ориентированную программу вычисления дополнения до полного алфавита разности двух заданных подмножеств латинских букв, которые передаются ей аргументами командной строки. Результатом являются все латинские буквы, кроме тех, которые есть только в первом аргументе. Они должны отображаться строкой стандартного вывода, где все буквы перечислены в обратном алфавитном порядке. Программа должна быть основана на разработке класса множества латинских букв с компонентными методами перегрузки операторов '~' и '' для выполнения операций дополнения и разности. Они должны быть ориентированы на обработку поля приватных данных, где состав множества фиксируется двоичными разрядами целого числа. Конструктор класса должен обеспечивать преобразование исходной символьной записи множества в двоичный целочисленный формат. Необходимо также предусмотреть перегрузку оператора класса потока стандартного вывода для отображения объектов класса множества латинских букв.

Вариант 15+(М)

Разработать объектно-ориентированную программу для исключения всех латинских букв кроме тех, которые совпадают в двух любых символьных строках, заданных аргументами команды ее вызова. Все оставшиеся различные буквы должны отображаться строкой потока стандартного вывода, где все буквы перечислены в обратном алфавитном порядке. Программная реализация таких исключений должна быть основана на разработке класса множества латинских букв с компонентным методом перегрузки оператора '^' и полем приватных данных, где состав множества кодируется двоичными разрядами целого числа. Конструктор класса должен обеспечивать преобразование исходной символьной записи подмножества латинских букв в бинарный целочисленный формат. Необходимо также предусмотреть перегрузку оператора класса потока стандартного вывода для отображения объектов класса множества латинских букв.

Вариант 16+(М)

Разработать объектно-ориентированную программу для перечисления всех латинских букв, которые одновременно отсутствуют или совпадают в двух любых символьных строках, заданных аргументами команды ее вызова. Искомый результат должна отображать строка стандартного вывода, где все буквы перечислены в обратном алфавитном порядке. Программная реализация таких перечислений должна быть основана на разработке класса множества латинских букв с компонентными методами перегрузки операторов '^' и '~' для выполнения операций исключения совпадений и дополнения. Они должны быть ориентированы на обработку поля приватных данных, где состав множества фиксируется двоичными разрядами целого числа. Конструктор класса должен обеспечивать преобразование исходной символьной записи множества в двоичный целочисленный формат. Необходимо также предусмотреть перегрузку оператора класса потока стандартного вывода для отображения объектов класса множества латинских букв.

Вариант 17+(М)

Разработать объектно-ориентированную программу вычисления объединения и пересечения двух любых подмножеств латинских букв, которые передаются ей аргументами командной строки. Полученный результат должен отображаться двумя строками стандартного вывода, где все буквы перечислены в обратном алфавитном порядке. Программная реализация вычислений должна быть основана на разработке класса множества латинских букв с компонентными методами перегрузки операторов '+' и '\*' для выполнения теоретико-множественных операций объединения и пересечения. Они должны быть ориентированы на обработку поля приватных данных, где состав множества фиксируется двоичными разрядами целого числа. Конструктор класса должен обеспечивать преобразование исходной символьной записи множества в двоичный целочисленный формат. Необходимо также предусмотреть перегрузку оператора класса потока стандартного вывода для отображения объектов класса множества латинских букв.

Вариант 18+(М)

Разработать объектно-ориентированную программу преобразования заданной обыкновенной дроби в конечную десятичную дробь. Обыкновенная дробь должна передаваться программе аргументом командной строки, где записи числителя и знаменателя десятичными цифрами разделены символом '/'. Результат преобразований должна отображать запись вещественного числа с фиксированной точкой строкой стандартного вывода. Программная реализация этого преобразования должна быть основана на разработке класса обыкновенных дробей с приватными полями данных для целочисленных значений числителя и знаменателя, а также оператором преобразования типа (operator double) для получения вещественного числа по результату их деления. Конструктор класса должен обеспечивать представление заданной символьной записи обыкновенной дроби парой взаимно простых целочисленных значений ее числителя и знаменателя. Для приведения обыкновенной дроби к несократимому виду следует предусмотреть компонентный метод, реализующий алгоритм Евклида.

Вариант 19+(Д)

Разработать объектно-ориентированную программу для преобразования заданной конечной десятичной дроби в обыкновенную дробь, которая приведена к несократимому виду. Десятичная дробь должна передаваться программе аргументом командной строки в формате вещественного числа с фиксированной точкой. Результат этого преобразований должен отображаться строкой стандартного вывода, где в записи обыкновенной дроби числитель и знаменатель разделены символом '/'. Программная реализация этого преобразования должна быть основана на разработке класса обыкновенных дробей с приватными полями данных для хранения целочисленных значений числителя и знаменателя. Конструктор класса должен обеспечивать формальный перевод десятичной дроби в обыкновенную дробь и приведение ее к несократимому виду на основе признаков делимости целых чисел на 2 и на 5. Для отображения результатов должна быть предусмотрена дружественная функция перегрузки оператора класса потока стандартного вывода '<<'.

Вариант 20+(Д)

Разработать объектно-ориентированную программу для округления заданной обыкновенной дроби до ближайшего целого числа, значение которого не превосходит ее величину. Запись обыкновенной дроби должна передаваться программе аргументом командной строки, где числитель и знаменатель разделены символом '/'. Результат такого округления должен отображаться через поток стандартного вывода. Программная реализация вычислений должна быть основана на разработке класса обыкновенных дробей с приватными полями данных для целочисленных значений числителя и знаменателя, а также оператором преобразования типа (operator int), который преобразует результат их деления в целое число. Конструктор класса должен обеспечивать представление заданной символьной записи обыкновенной дроби парой взаимно простых целочисленных значений ее числителя и знаменателя. Для приведения полученной обыкновенной дроби к несократимому виду следует предусмотреть компонентный метод, реализующий алгоритм Евклида.

Вариант 21+(Д)

Разработать объектно-ориентированную программу для округления заданной обыкновенной дроби до ближайшего целого числа, значение которого не меньше ее величины. Запись обыкновенной дроби должна передаваться программе аргументом командной строки, где числитель и знаменатель разделены символом '/'. Результат такого округления должен отображаться через поток стандартного вывода. Программная реализация вычислений должна быть основана на разработке класса обыкновенных дробей с приватными полями данных для целочисленных значений числителя и знаменателя, а также оператором преобразования типа (operator int), который преобразует результат их деления в целое число. Конструктор класса должен обеспечивать представление заданной символьной записи обыкновенной дроби парой взаимно простых целочисленных значений ее числителя и знаменателя. Для приведения полученной обыкновенной дроби к несократимому виду следует предусмотреть компонентный метод, реализующий алгоритм Евклида.

Вариант 22+(Д)

Разработать объектно-ориентированную программу представления заданной обыкновенной дроби в виде суммы целой и правильной дробной части. Обыкновенная дробь должна передаваться программе аргументом командной строки, где записи числителя и знаменателя десятичными цифрами разделяет символ '/'. Результат преобразований должен отображаться строкой стандартного вывода, где записи целой и правильной дробной части обыкновенной дроби разделены символом '+'. Программная реализация преобразований должна быть основана на разработке класса обыкновенных дробей с приватными полями данных для целочисленных значений числителя и знаменателя. Получение их значений по заданной символьной записи обыкновенной дроби должен обеспечивать конструктор класса. Их функциональную обработку должен обеспечивать компонентный оператор преобразования типа для вычисления целой части обыкновенной дроби и компонентный оператор '' для перегрузки операции вычитания его из исходной дроби.

Вариант 23+(Д)

Разработать объектно-ориентированную программу представления заданной обыкновенной дроби в виде разности целой и правильной дробной части. Обыкновенная дробь должна передаваться программе аргументом командной строки, где записи числителя и знаменателя десятичными цифрами разделяет символ '/'. Результат преобразований должен отображаться строкой стандартного вывода, где записи целой и правильной дробной части обыкновенной дроби разделены символом ''. Программная реализация преобразований должна быть основана на разработке класса обыкновенных дробей с приватными полями данных для целочисленных значений числителя и знаменателя. Получение их значений по заданной символьной записи обыкновенной дроби должен обеспечивать конструктор класса. Их функциональную обработку должен обеспечивать компонентный оператор преобразования типа для вычисления целой части обыкновенной дроби и компонентный оператор '' для перегрузки операции вычитания его из исходной дроби.

Вариант 24+(Д)

Разработать объектно-ориентированную программу для вычисления суммы обыкновенной дроби и целого числа, которые должны передаваться ей через аргументы командной строки. Результат операции должен отображаться строкой потока стандартного вывода в формате обыкновенной дроби. В обоих случаях для записи обыкновенной дроби должен применяться символьный формат, где ее числитель и знаменатель разделены знаком '/'. Программная реализация вычислений должна быть основана на разработке класса обыкновенных дробей с приватными полями данных для целочисленных значений числителя и знаменателя, а также компонентным методом перегрузки оператора '+'. Конструкторы класса должны выполнять преобразования обоих операндов в указанный числовой формат обыкновенных дробей. Для приведения исходной и результирующей обыкновенных дробей к несократимому виду следует предусмотреть компонентный метод, реализующий алгоритм Евклида, и перегрузку оператора присваивания.

Вариант 25+(Д)

Разработать объектно-ориентированную программу для вычисления разности обыкновенной дроби и целого числа, которые должны передаваться ей через аргументы командной строки. Результат операции должен отображаться строкой потока стандартного вывода в формате обыкновенной дроби. В обоих случаях для записи обыкновенной дроби должен применяться символьный формат, где ее числитель и знаменатель разделены знаком '/'. Программная реализация вычислений должна быть основана на разработке класса обыкновенных дробей с приватными полями данных для целочисленных значений числителя и знаменателя, а также компонентным методом перегрузки оператора ''. Конструкторы класса должны выполнять преобразования обоих операндов в указанный числовой формат обыкновенных дробей. Для приведения исходной и результирующей обыкновенных дробей к несократимому виду следует предусмотреть компонентный метод, реализующий алгоритм Евклида, и перегрузку оператора присваивания.

Вариант 26+(Д)

Разработать объектно-ориентированную программу для перемножения обыкновенной дроби и целого числа, которые должны передаваться ей через аргументы командной строки. Результат операции должен отображаться строкой потока стандартного вывода в формате обыкновенной дроби. В обоих случаях для записи обыкновенной дроби должен применяться символьный формат, где ее числитель и знаменатель разделены знаком '/'. Программная реализация вычислений должна быть основана на разработке класса обыкновенных дробей с приватными полями данных для целочисленных значений числителя и знаменателя, а также компонентным методом перегрузки оператора '\*'. Конструкторы класса должны выполнять преобразования обоих операндов в указанный числовой формат обыкновенных дробей. Для приведения исходной и результирующей обыкновенных дробей к несократимому виду следует предусмотреть компонентный метод, реализующий алгоритм Евклида, и перегрузку оператора присваивания.

Вариант 27+(Д)

Разработать объектно-ориентированную программу для деления обыкновенной дроби на целое число. Оба операнда должны передаваться ей через аргументы командной строки. Результат операции должен отображаться строкой потока стандартного вывода в формате обыкновенной дроби. В обоих случаях для записи обыкновенной дроби должен применяться символьный формат, где ее числитель и знаменатель разделены знаком '/'. Программная реализация вычислений должна быть основана на разработке класса обыкновенных дробей с приватными полями данных для целочисленных значений числителя и знаменателя, а также компонентным методом перегрузки оператора '/'. Конструкторы класса должны выполнять преобразования обоих операндов в указанный числовой формат обыкновенных дробей. Для приведения исходной и результирующей обыкновенных дробей к несократимому виду следует предусмотреть компонентный метод, реализующий алгоритм Евклида, и перегрузку оператора присваивания.

Вариант 28+(Д)

Разработать объектно-ориентированную программу для вычисления суммы обыкновенных дробей, которые должны передаваться ей через два аргумента командной строки. Результат вычислений должен отображаться строкой стандартного вывода в формате обыкновенной дроби. Во всех случаях для записи обыкновенных дроби должен применяться символьный формат, где ее числитель и знаменатель разделены знаком '/'. Программная реализация вычислений должна быть основана на разработке класса обыкновенных дробей с приватными полями данных для целочисленных значений числителя и знаменателя, а также компонентным методом перегрузки оператора '+'. Конструкторы класса должны выполнять преобразования обоих операндов в указанный числовой формат обыкновенных дробей. Для приведения исходной и результирующей обыкновенных дробей к несократимому виду следует предусмотреть компонентный метод, реализующий алгоритм Евклида, и перегрузку оператора присваивания.

Вариант 29+(Д)

Разработать объектно-ориентированную программу для вычисления разности обыкновенных дробей, которые должны передаваться ей через два аргумента командной строки. Результат вычислений должен отображаться строкой стандартного вывода в формате обыкновенной дроби. В обоих случаях для записи обыкновенной дроби должен применяться символьный формат, где ее числитель и знаменатель разделены знаком '/'. Программная реализация вычислений должна быть основана на разработке класса обыкновенных дробей с приватными полями данных для целочисленных значений числителя и знаменателя, а также компонентным методом перегрузки оператора ''. Конструкторы класса должны выполнять преобразования обоих операндов в указанный числовой формат обыкновенных дробей. Для приведения исходной и результирующей обыкновенных дробей к несократимому виду следует предусмотреть компонентный метод, реализующий алгоритм Евклида, и перегрузку оператора присваивания.

Вариант 30+(Д)

Разработать объектно-ориентированную программу для вычисления произведения двух обыкновенных дробей, которые должны передаваться ей через аргументы командной строки. Результат операции должен отображаться строкой потока стандартного вывода в формате обыкновенной дроби. В обоих случаях для записи обыкновенной дроби должен применяться символьный формат, где ее числитель и знаменатель разделены знаком '/'. Программная реализация вычислений должна быть основана на разработке класса обыкновенных дробей с приватными полями данных для целочисленных значений числителя и знаменателя, а также компонентным методом перегрузки оператора '\*'. Конструкторы класса должны выполнять преобразования обоих операндов в указанный числовой формат обыкновенных дробей. Для приведения исходной и результирующей обыкновенных дробей к несократимому виду следует предусмотреть компонентный метод, реализующий алгоритм Евклида, и перегрузку оператора присваивания.

Вариант 31+(Д)

Разработать объектно-ориентированную программу для вычисления отношения двух обыкновенных дробей, которые должны передаваться ей через аргументы командной строки. Результат операции должен отображаться строкой потока стандартного вывода в формате обыкновенной дроби. В обоих случаях для записи обыкновенной дроби должен применяться символьный формат, где ее числитель и знаменатель разделены знаком '/'. Программная реализация вычислений должна быть основана на разработке класса обыкновенных дробей с приватными полями данных для целочисленных значений числителя и знаменателя, а также компонентным методом перегрузки оператора '/'. Конструкторы класса должны выполнять преобразования обоих операндов в указанный числовой формат обыкновенных дробей. Для приведения исходной и результирующей обыкновенных дробей к несократимому виду следует предусмотреть компонентный метод, реализующий алгоритм Евклида, и перегрузку оператора присваивания.

Вариант 32+(Д)

Разработать объектно-ориентированную программу для сравнения обыкновенных дробей, которые должны передаваться ей через 2 аргумента командной строки. Результат сравнения должен отображаться строкой потока стандартного вывода, обе заданные дроби разделяет знак отношения порядка между их величинами. Во всех случаях для записи обыкновенных дробей должен применяться символьный формат, где их числители и знаменатели разделены знаком '/'. Программная реализация сравнения должна быть основана на разработке класса обыкновенных дробей с приватными полями данных для целочисленных значений их числителя и знаменателя, а также компонентным методом перегрузки оператора '=='. Его код возврата должен быть равен '=', '>' или '<' в зависимости от соотношения перекрестных произведений числителей и знаменателей сопоставляемых дробей. Конструктор класса должен обеспечивать преобразование символьной записи обыкновенной дроби в числовой формат.

Вариант 33+(Д)

## Варианты контрольной работы

Вариант 1++

Разработать объектно-ориентированную программу перевода двоичной записи целого числа без знака в его эквивалентную запись в системе счисления по основанию 8. Исходная двоичная запись в форме бинарного вектора произвольной длины, состоящего из символов ‘0’ и ‘1’, должна передаваться программе через аргумент командной строки ее вызова. Результирующая запись этого числа цифрами от ‘0’ до ‘7’в системе счисления по основанию 8 должна отображаться символьной строкой потока стандартного вывода. Программная реализация процедуры указанных преобразований должна быть основана на разработке классов, предоставляющих записи целых чисел без знака в системах счисления по основаниям 2 и 8, которые наследуют стек с фиксированным буфером и используют конструктор копирования-инициализации для выполнения этих преобразований.

Вариант 2++

Разработать объектно-ориентированную программу сравнения 2-х обыкновенных дробей. Обе сравниваемые дроби должны передаваться программе через аргументы командной строки ее вызова, в которых числители и знаменатели дробей разделены символом наклонной черты. Программная реализация процедуры сравнения должна быть основана на разработке класса обыкновенных дробей. В нем необходимо предусмотреть оператор сравнения, который возвращает код: ‘=’, ‘>’, или ‘<’ в зависимости от соотношения перекрестных произведений числителей и знаменателей сопоставляемых дробей. Результат сравнения должен отображаться в форме строки стандартного вывода обоих дробей с указанием знака отношения порядка между их величинами. Для удобства отображения обыкновенных дробей в программе необходимо предусмотреть перегрузку оператора’<<’ класса потока стандартного вывода.

Вариант 3++

Разработать программу поиска натуральных чисел, которые являются цифровым палиндромом, обладающим свойством симметричного расположения своих значащих цифр. Любое натуральное число, для которого необходимо проверить указанное свойство палиндрома, должно передаваться программе через аргумент командной строки ее вызова. Результат проверки свойства палиндрома у заданного числа должен идентифицировать код возврата программы. При этом нулевой код возврата программы должен обозначать, что заданное число не является палиндромом, то код возврата программы должен быть отличен от нуля и равен разности кодов двух первых различных цифр, которые находятся удалении от концов числа. Программная реализация обработки палиндромов должна быть основана на разработке класса символьного стека с фиксированным буфером для размещения цифр заданного числа в обратном порядке.

Вариант 4++

Разработать объектно-ориентированную программу формирования латинского квадрата заданного размера N из последовательных натуральных чисел от 1 до N, в котором все числа любой строки и столбца различны. Для выполнения этого условия каждая следующая строка латинского квадрата должна получаться циклическим сдвигом влево на 1 позицию всех элементов предыдущей строки. При этом начальный элемент предыдущей строки становится последним в текущей строке латинского квадрата, а остальные элементы перемещаются на 1 позицию влево без изменения своего взаимного расположения. Размер латинского квадрата, определяющий его элементный состав, должен передаваться программе целочисленным аргументом командной строки ее вызова. Результат построения латинского квадрата указанного размера должен отображаться через поток стандартного вывода, где все столбцы должны быть выровнены по правой границе значений своих элементов. Программная реализация должна быть основана на разработке класса латинского квадрата, для заполнения которого следует использовать структуру очереди с кольцевым буфером.

Вариант 5++

Разработать объектно-ориентированную программу для вычисления объединения (суммы) и пересечения (произведения) двух любых подмножеств латинских букв. Заданные подмножества латинских букв и знак операции "+" или "\*" должны передаваться программе, через аргументы командной строки ее вызова. Программная реализация вычислений должна быть основана на разработке класса множества символов с перезагрузкой операторов ‘+’ и ‘-‘ для выполнения соответствующих теоретико- множественных операций, которые должны быть реализованы внешними функциями. Результат операции должен отображаться символьной строкой в потоке стандартного вывода. Для удобства отображения результата в программе должна быть предусмотрена перегрузка оператора ‘<<’ класса потока стандартного вывода для объектов класса множества латинских букв.

Вариант 6++

Разработать вычисления симметрической разности 2-х заданных подмножеств латинских букв, которую образуют все их элементы, принадлежащие либо одному, либо другому множеству, но не обоим одновременно. Симметрическая разность должна помещаться в результате объединения дополнений второго множества до первого и первого до второго.

Программная реализация вычислений должна быть основана на разработке класса множества символов с перегрузкой операторов ‘+’ и ‘-‘ для определения объединения и дополнения множеств, соответственно. Результат вычислений должен отображаться символьной строкой в потоке стандартного вывода. Для удобства отображения результата в программе должна быть предусмотрена перегрузка оператора ‘<<’ класса потока стандартного вывода для объектов класса множества символов.

Вариант 7++

Разработать объектно-ориентированную программу для вычисления длины отрезка прямой линии, который задан целочисленными координатами своих концов на плоскости. Координаты концов отрезка должны передаваться программе парами целых чисел через 2 аргумента командной строки ее вызова. При этом первое число в каждой паре должно обозначать координату по оси X, а второе – по оси Y. Для разделения координат в каждой паре должен использоваться символ точка с запятой. Результат вычислений длины отрезка должен отображаться через поток стандартного вывода. Программная поддержка указанных вычислений должна быть основана на разработке контейнерного класса отрезка прямой линии, компонентными данными которого являются объекты класса точек его концов, а компонентный метод обеспечивает вычисление длины отрезка.

Вариант 8++

Разработать объектно-ориентированную программу для формирования квадратной матрицы Вандермонда по образующему вектору из необязательно различных десятичных цифр от 1 до 9, столбцы которой являются последовательными натуральными степенями элементов образующего вектора. Размер матрицы Вандермонда должен отображаться через поток стандартного вывода, где все столбцы должны быть выровнены полевой границе своих элементов. Программная реализация указанных построений должнабыть основана на разработке класса матрицы Вандермонда с динамическим распределением памяти для таблицы целочисленных значений ее элементов с учетом оценки возможности их вычислений для стандартных типов целочисленных данных в системе программирования С++.

Вариант 9++

Разработать объектно-ориентированную программу, реализующую вычисление разности 2- х. заданных подмножеств латинских букв. Заданные подмножества латинских букв. Заданные подмножества латинских букв и знак операции ‘ – ‘ должны передаваться программе через аргумент командной строки ее вызова. Программная реализация вычислений должна быть основана на разработке класса множества с перегрузкой оператора ‘ – ‘, который должен бать реализован компонентным методом класса с использованием бинарных операций: сначала EOR(^) (

этот оператор возвращает 1, если только оба бита не равны), а затем AND(&). Результат операции должен отображаться символьной строкой в потоке стандартного вывода. Для удобства отображения результирующего подмножества в алфавитном порядке в программе должна быть предусмотрена перегрузка оператора ‘<<’ класса потока стандартного вывода для объектов множества латинских букв.

Вариант 10++

Разработать объектно-ориентированную программу, реализующую вычисление суммы по модулю 2 для 2-х. заданных подмножеств латинских букв. Программная реализация вычислений должна быть основана на разработке класса множества с перегрузкой оператора ‘^’, который должен быть реализован компонентным методом класса с использованием бинарной операции ‘^’, для обработки целочисленных компонентных данных. Результат операции должен отображаться символьной строкой в потоке стандартного вывода. Для удобства отображения результирующего подмножества в алфавитном порядке в программе должна быть предусмотрена перегрузка оператора ‘<<’ класса потока стандартного вывода для объектов множества латинских букв.

Вариант 11++

Разработать объектно-ориентированную программу, реализующую сравнение по мощности двух заданных подмножеств латинских букв. Программная реализация сравнений должна быть основана на разработке класса множества с перегрузкой оператора сравнения, который должен возвращать символ ‘<’, ‘=’ или ‘>’ в зависимости от соотношения мощности сравниваемых подмножеств. Результат сравнения должен отображаться в форме строки стандартного вывода обоих подмножеств с указанием знака отношения порядка между их мощностями. Для удобства отображения подмножеств в программе необходимо продумать перегрузку оператора ‘<<’ класса потока стандартного вывода.

Вариант 12++

Разработать объектно-ориентированную программу для перечисления в обратном порядке символов входных строк, которые должны передаваться ей через поток стандартного ввода. Результирующие инвертированные строки с обратным порядком символов должны отображаться через поток стандартного вывода. Указанная обработка входных строк должна продолжаться, пока не введен символ конца потока стандартного ввода, принятый по умолчанию в среде выполнения программы. Программная реализация процедуры инвертирования входного потока должна быть основана на разработке символьного стека с динамическим буфером в форме линейного списка входных символов.

Вариант 13++

Разработать объектно-ориентированную программу, реализующую анализ и последующее перечисление всех латинских букв, не принадлежащих заданному подмножеству латинских букв. Программная реализация анализа должна быть основана на разработке класса множества с перегрузкой оператора, который должен быть реализован компонентным методом класса с использованием бинарной операции’ ‘.Результат операции должен отображаться символьной строкой в потоке стандартного вывода. Для удобства отображения результирующего подмножества в алфавитном порядке в программе должна быть предусмотрена перегрузка оператора ‘<<’ класса потока стандартного вывода для объектов множества латинских букв.

Вариант 14++

Разработать объектно-ориентированную программу, реализующую анализ и последующее перечисление всех латинских букв, не принадлежащих заданному подмножеству латинских букв. Программная реализация анализа должна быть основана на разработке класса множества с перегрузкой оператора, который должен быть реализован компонентным методом класса с использованием бинарной операции’ ‘.Результат операции должен отображаться символьной строкой в потоке стандартного вывода. Для удобства отображения результирующего подмножества в порядке, обратном алфавитному, в программе должна быть предусмотрена перегрузка оператора ‘<<’ класса потока стандартного вывода для объектов множества латинских букв.

Вариант 15++

Разработать объектно-ориентированную программу, реализующую вычисление суммы по модулю 2 для 2-х. заданных подмножеств латинских букв. Программная реализация вычислений должна быть основана на разработке класса множества с перегрузкой оператора ‘^’, который должен быть реализован компонентным методом класса с использованием бинарной операции ‘^’ для обработки целочисленных компонентных данных. Результат операции должен отображаться символьной строкой в потоке стандартного вывода. Для удобства отображения результирующего подмножества в порядке, обратном алфавитному, в программе должна быть предусмотрена перегрузка оператора ‘<<’ класса потока стандартного вывода для объектов множества латинских букв.

Вариант 16++

Разработать объектно-ориентированную программу, реализующую сравнение по мощности двух заданных подмножеств латинских букв. Программная реализация сравнений должна быть основана на разработке класса множества с перегрузкой оператора сравнения, который должен возвращать символ ‘<’, ‘=’ или ‘>’ в зависимости от соотношения мощности сравниваемых подмножеств. Результат сравнения должен отображаться в форме строки стандартного вывода обоих подмножеств с указанием знака отношения порядка между их мощностями. Для удобства отображения подмножеств в порядке, обратном алфавитному в программе необходимо продумать перегрузку оператора ‘<<’ класса потока стандартного вывода.

Вариант 17++

Разработать вычисления симметрической разности 2-х заданных подмножеств латинских букв, которую образуют все их элементы, принадлежащие либо одному, либо другому множеству, но не обоим одновременно. Симметрическая разность должна помещаться в результате объединения дополнений второго множества до первого и первого до второго.

Программная реализация вычислений должна быть основана на разработке класса множества символов с перегрузкой операторов ‘+’ и ‘-‘ для определения объединения и дополнения множеств, соответственно. Результат вычислений должен отображаться символьной строкой в потоке стандартного вывода. Для удобства отображения результата порядке, обратном алфавитному в программе должна быть предусмотрена перегрузка оператора ‘<<’ класса потока стандартного вывода для объектов класса множества символов.

Вариант 18++

Разработать объектно-ориентированную программу вычисления периметра и площади прямоугольника на плоскости, который задан координатами 2-х своих противоположных вершин, а его стороны параллельны координатным осям. Координаты вершин должны передаваться программе парами целых чисел через 2 аргумента командной строки ее вызова. При этом первое число в каждой паре должно обозначать координату вершины по оси Х, а второе по оси Y. Для разделения координат в каждой паре должен использоваться символ точки с запятой. Результаты вычислений площади и периметра прямоугольника должны отображаться через поток стандартного вывода. Программная реализация указанных вычислений должна быть основана на разработке контейнерного класса изотетичного прямоугольника, компонентными данными которого являются объекты класса противоположных вершин, компонентные методы обеспечивают вычисление его периметра и площади.

Вариант 19++

Разработать объектно-ориентированную программу моделирования абсолютно неупругого центрального удара 2-х тел с заданными линейными скоростями и массами, которая вычисляет их общую массу и скорость после удара на основе закона сохранения импульса. Массы и скорости тел до удара должны быть заданы парами целочисленных значений, которые передаются программе через 2 аргумента командной строки ее вызова. При этом каждый аргумент должен быть записан в форме импульса до удара, где первое число определяет массу тела, а второе – его скорость. Указанные значения в каждой паре должны разделяться символом умножения. Результат моделирования должен отображаться через поток стандартного вывода двумя вещественными числами, которые обозначают массу и скорость тела, образовавшегося после удара. При разработке программы моделирования необходимо использовать аппарат множественного наследования классов импульсов тел.

Вариант 20++

Разработать объектно-ориентированную программу для вычисления температуры смеси 2-х заданных масс одинаковой жидкости с различной температурой на основе уравнения теплового баланса без учета теплообмена с окружающей средой. Значения масс и температур жидкостей смеси должны задаваться целыми величинами, которые передаются программе через 2 аргумента командной строки ее вызова. При этом каждый аргумент должен задаваться парой целых чисел, первое из которых специфицирует массу, а второе – температуру жидкости. Значения указанных физических параметров в каждой паре должны разделяться символом наклонной черты. Результат вычислений должен отображаться через поток стандартного вывода вещественным числом, которое обозначает температуру смеси. При разработке программы необходимо использовать аппарат множественного наследования для описания состояния смеси.

Вариант 21++

Разработать объектно-ориентированную программу, которая располагает в алфавитном порядке любые входные слова, составленные из латинских букв без различия регистра. Входные слова должны передаваться программе строками потока стандартного ввода. Результат алфавитного упорядочивания входных слов должен отображаться через поток стандартного вывода, где все слова из текущей входной строки должны быть перечислены в алфавитном порядке как в словаре. Обработка входных строк должна продолжаться, пока не получен признак конца истока стандартного ввода, принятый в операционной среде выполнения программы. Программная реализация процедуры алфавитного упорядочивания входных слов должна быть основана на использовании структуры линейного списка со сторожем, оперативная память для хранения элементов которого должна динамически распределяться при обработке каждой очередной входной строки потока стандартного ввода.

Вариант 22++

Разработать объектно-ориентированную программу для числовой игры, где 2 противника поочередно на своих ходах должны выбирать из кучи заданного размера любое число предметов, которое не превышает фиксированного значения N. Оба эти параметра должны передаваться программе через аргументы командной строки ее вызова.

Победителем считается игрок, который сумел взять последний предмет из кучи. Игра должна быть реализована в варианте, когда одним из игроков является компьютер, а другим – человек. Право первого хода всегда предоставляется компьютеру. При этом ходы компьютера всегда должны осуществляться в соответствии с выигрышной стратегией, согласно которой число предметов в куче после его ходов должно быть кратно (N+1). При невозможности сделать выигрышный ход компьютер должен выбрать из кучи один предмет. В любом случае выбор компьютера должен отображаться через поток стандартного вывода. Человек на своих ходах может выбирать произвольное допустимое число предметов, указывая его через поток стандартного ввода. При любой ошибке ввода человека автоматически считается, что он извлек из кучи 1 предмет. Ходы человека и компьютера должны сопровождаться информационным запросом, который отображает в потоке стандартного вывода текущий размер кучи и имя игрока. Игра должна завершаться диагностическим сообщением, которое идентифицирует победителя и проигравшего. При разработке программы игры необходимо реализовать виртуальную функцию хода игроков для их производных классов, которые наследуютпараметрв и интерфейс игры от абстрактного базового класса.

Вариант 23++

Разработать объектно-ориентированную программу для числовой игры, где 2 противника поочередно на своих ходах должны наращивать сумму, названную в предыдущем ходе значение суммы противника, выбирая числовое приращение, которое не превышает предельной величины N. Оба эти параметра должны передаваться программе через аргументы командной строки ее вызова.

Победителем считается игрок, который сумел взять последний предмет из кучи. Игра должна быть реализована в варианте, когда одним из игроков является компьютер, а другим – человек. Право первого хода всегда предоставляется компьютеру. При этом ходы компьютера всегда должны осуществляться в соответствии с выигрышной стратегией, согласно которой число предметов в куче после его ходов должно быть кратно (N+1). При невозможности сделать выигрышный ход компьютер должен выбрать из кучи один предмет. В любом случае выбор компьютера должен отображаться через поток стандартного вывода. Человек на своих ходах может выбирать произвольное допустимое число предметов, указывая его через поток стандартного ввода. При любой ошибке ввода человека автоматически считается, что он извлек из кучи 1 предмет. Ходы человека и компьютера должны сопровождаться информационным запросом, который отображает в потоке стандартного вывода текущий размер кучи и имя игрока. Игра должна завершаться диагностическим сообщением, которое идентифицирует победителя и проигравшего. При разработке программы игры необходимо реализовать виртуальную функцию хода игроков для их производных классов, которые наследуют параметры и интерфейс игры от абстрактного базового класса.

Вариант 24++

Разработать объектно-ориентированную программу, реализующую вычисление разности 2- х. заданных подмножеств латинских букв. Заданные подмножества латинских букв. Заданные подмножества латинских букв и знак операции ‘ – ‘ должны передаваться программе через аргумент командной строки ее вызова. Программная реализация вычислений должна быть основана на разработке класса множества с перегрузкой оператора ‘ – ‘, который должен бать реализован внешней функцией с использованием бинарных операций: сначала EOR(^) (

этот оператор возвращает 1, если только оба бита не равны), а затем AND(&). Результат операции должен отображаться символьной строкой в потоке стандартного вывода. Для удобства отображения результирующего подмножества в алфавитном порядке в программе должна быть предусмотрена перегрузка оператора ‘<<’ класса потока стандартного вывода для объектов множества латинских букв.

Вариант 25++

Разработать объектно-ориентированную программу, реализующую вычисление разности 2- х. заданных подмножеств латинских букв. Заданные подмножества латинских букв. Заданные подмножества латинских букв и знак операции ‘ – ‘ должны передаваться программе через аргумент командной строки ее вызова. Программная реализация вычислений должна быть основана на разработке класса множества с перегрузкой оператора ‘ – ‘, который должен бать реализован компонентным методом класса с использованием бинарных операций: сначала EOR(^) (

этот оператор возвращает 1, если только оба бита не равны), а затем AND(&). Результат операции должен отображаться символьной строкой в потоке стандартного вывода. Для удобства отображения результирующего подмножества в порядке, обратном алфавитному в программе должна быть предусмотрена перегрузка оператора ‘<<’ класса потока стандартного вывода для объектов множества латинских букв.

Вариант 26++

Разработать объектно-ориентированную программу, реализующую вычисление суммы по модулю 2 для 2-х. заданных подмножеств латинских букв. Программная реализация вычислений должна быть основана на разработке класса множества с перегрузкой оператора ‘^’, который должен быть реализован внешней функцией с использованием бинарной операции ‘^’ для обработки целочисленных компонентных данных. Результат операции должен отображаться символьной строкой в потоке стандартного вывода. Для удобства отображения результирующего подмножества в алфавитном порядке в программе должна быть предусмотрена перегрузка оператора ‘<<’ класса потока стандартного вывода для объектов множества латинских букв.

# Приложение 4. Вопросы к экзамену

1. Организация в C++. Компонентные данные и методы. Доступ к компонентам классов.

2. Статические методы и данные классов.

3. Конструкторы классов. Конструктор по умолчанию. Конструкторы копирования-инициализации.

4. Дружественне функции классов.

5. Деструкторы классов. Явный и неявный вызов деструктора. Виртуальные деструкторы

6. Средства динамического распределения в С и С++.

7. Парадигма наследования в С++. Производные классы. Порядок вызова конструкторов и деструкторов базового и производимого классов. Размещение в памяти производного класса.

8. Множественное наследование в С++. Схема размещения в памяти производного класса при множественном наследовании.

9. Кратное множественное наследование в С++. Виртуальное наследование. Схема размещения в памяти производного класса при кратном множественном и виртуальном наследовании.

10. Полиморфиум. Виртуальные функции. Позднее связывание.

11. Абстрактные классы. Чистые функции.

12. Библиотека функции стандартного ввода-вывода.

13. Библиотека функции обработки файлов.

14. Адресная арифметика при работе с массивами данных.

15. Передача аргументов функции по значению, адресу и ссылке.

16. Явное и неявное преобразование типов. Преобразование типов с помощью конструктора и оператор-функции.

17. Программирование генераторов псевдо-случайных последовательностей в С и С++.

18. Классы памяти системы программирования С++. Статические, внешние и автоматические переменные. Статические функции.

19. Контейнерные классы и объекты.

20. Шаблоны классов.

21. Потоковые классы.

22. Стеки.

23. Очереди.

24. Линейные списки.