

**ЛЕКЦИИ ПО ДИСЦИПЛИНЕ
«ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО
ОБЕСПЕЧЕНИЯ СИСТЕМ УПРАВЛЕНИЯ»**

Библиотека ГГТУ им. П.О.Суворова

Содержание

1. Предмет дисциплины и ее задачи.....	5
1.1 Введение.....	5
1.2. Цели и задачи учебной дисциплины.....	5
1.3. Структура и содержание дисциплины.....	5
1.4. Перечень тем лабораторных занятий.....	9
1.5. Роль объектно-ориентированного программирования, проектирования и анализа при создании систем управления..	10
2. Компиляторы gcc, g++.....	12
2.1. Основные понятия.....	12
2.2. GNU Compiler Collection.....	12
2.3. Компилятор GCC.....	16
2.4. Опции компиляции.....	17
2.5. Утилита make.....	18
2.6. MakeFile.....	19
3. Особенности и синтаксис языков программирования.....	25
3.1. Комментарии.....	25
3.2. Указатели.....	26
3.3. Работа с динамической памятью.....	28
3.4. Динамические массивы.....	30
3.5. Особенности создания многомерных динамических Массивов.....	31
3.6. Тип данных «ссылки».....	34
3.7. Механизм перегрузки функций.....	36
4. Основные принципы объектно-ориентированного программирования.....	39
4.1. Основной элемент ООП – класс. Основные компоненты, объекты.....	39
4.2. Основные принципы ООП.....	40
4.3. Абстрактные типы данных.....	41
4.4. Классы и объекты C++.....	41
4.5. Синтаксис описания класса.....	42
4.6. Управление доступом к членам класса.....	42
4.7. Объявление и определение методов класса. Спецификатор inline.....	44
5. Основной элемент ООП – класс. Основные компоненты, объекты.....	46
5.1. Конструкторы и деструкторы.....	46
5.2. Конструктор умолчания.....	48

5.3. Конструктор преобразования и конструкторы с двумя и более параметрами.....	48
5.4. Конструктор копирования.....	49
5.5. Спецификатор explicit.....	52
5.6. Конструктор копирования и операция присваивания.....	53
5.7. Автоматическая генерация конструкторов и Деструкторов.....	53
5.8. Указатель “this”.....	54
6. Методы класса, способы применения, особенности.....	56
6.1. Дружественные (friend) функции класса.....	56
6.2. Статические методы и данные.....	58
6.3. Методы const, не изменяющие объекты класса.....	60
6.4. Спецификатор mutable.....	61
7. Правила и принципы наследования. Производные и базовые классы.....	62
7.1. Правила наследования.....	62
7.2. Управление уровнем доступа к элементам класса.....	63
7.3. Множественное наследование. Конструктор во множественном наследовании.....	68
7.4. Виртуальный базовый класс.....	71
7.5. Последовательность создания и уничтожения Подобъектов.....	72
8. Виртуальные функции. Позднее связывание.....	74
8.1. Виртуальные функции.....	74
8.2. Виртуальные деструкторы.....	81
9. Переопределение операций.....	83
9.1. Перегрузка операций.....	83
9.2. Многократная перегрузка операций.....	89
9.3. Перегрузка операции присваивания =.....	90
9.4. Перегрузка операций [], () и ->.....	93
10. Шаблоны функций, шаблоны классов.....	97
10.1. Параметры шаблона.....	97
10.2. Отождествление типов аргументов.....	98
10.3. Алгоритм поиска оптимально отождествляемой функции (с учетом шаблонов).....	99
10.4. Шаблоны классов.....	100
10.5. Наследование в шаблонах классов.....	103
11. Другие возможности стандартных библиотек.....	106
11.1. Указатели на компоненты класса.....	106

11.2. Понятие абстрактного класса. “Чистые” виртуальные Функции.....	110
12. Библиотека потоков данных.....	114
12.1. Простое внесение.....	115
12.2. Выражение извлечения.....	117
12.3. Создание собственных функции внесения и извлечения...	117
12.4. Функции библиотеки iostream.....	119
12.5. Манипуляторы ввода-вывода.....	121
12.6. Файловые и строковые потоки.....	123
13. Исключительные ситуации, обработка, основные операции.....	126
13.1. Возбуждение исключения.....	126
13.2. Перехват исключений.....	133
13.3. Объекты-исключения.....	135
13.4. Раскрутка стека.....	138
13.5. Повторное возбуждение исключения.....	139
13.6. Перехват всех исключений.....	141
13.7. Спецификации исключений.....	143
13.8. Спецификации исключений и указатели на функции.....	146
14. Особенности построения графического интерфейса с использованием библиотек Qt.....	148
14.1. Класс QObject.....	148
14.2. Класс QWidget.....	149
14.3. Механизм сигналов и слотов.....	152
14.4. Сигналы.....	153
14.5. Слоты.....	155
14.6. Соединение объектов.....	156
14.7. Разъединение объектов.....	159
14.8. Переопределение сигналов.....	159
15. Стандартные библиотеки и проектирование основных интерфейсов приложения.....	161
15.1. Преимущества использования стандартных библиотек....	161
15.2. Иерархия классов стандартных библиотек.....	163
16. Поточная многозадачность.....	172
16.1. Многозадачность, основанная на процессах.....	172
16.2. Типы псевдопараллельной многозадачности.....	173
16.3. Многозадачность, основанная на процессах и потоках.....	174
16.4. Потоки в стандартных библиотеках.....	175
16.5. Мьютексы.....	180

16.6. Поддержка потоков в Qt.....	1851.
Предмет дисциплины и ее задачи.....	5
1.1 Введение.....	5
1.2. Цели и задачи учебной дисциплины.....	5
1.3. Структура и содержание дисциплины.....	5
1.4. Перечень тем лабораторных занятий.....	10
1.5. Роль объектно-ориентированного программирования, проектирования и анализа при создании систем управления..	11
2. Компиляторы gcc, g++.....	13
2.1. Основные понятия.....	13
2.2. GNU Compiler Collection.....	13
2.3. Компилятор GCC.....	16
2.4. Опции компиляции.....	17
2.5. Утилита make.....	18
2.6. MakeFile.....	20
3. Особенности и синтаксис языков программирования.....	26
3.1. Комментарии.....	26
3.2. Указатели.....	27
3.3. Работа с динамической памятью.....	30
3.4. Динамические массивы.....	31
3.5. Особенности создания многомерных динамических Массивов.....	33
3.6. Тип данных «ссылки».....	35
3.7. Механизм перегрузки функций.....	38
4. Основные принципы объектно-ориентированного программирования.....	41
4.1. Основной элемент ООП – класс. Основные компоненты, объекты.....	41
4.2. Основные принципы ООП.....	42
4.3. Абстрактные типы данных.....	43
4.4. Классы и объекты C++.....	43
4.5. Синтаксис описания класса.....	44
4.6. Управление доступом к членам класса.....	44
4.7. Объявление и определение методов класса. Спецификатор inline.....	46
5. Основной элемент ООП – класс. Основные компоненты, объекты.....	48
5.1. Конструкторы и деструкторы.....	48
5.2. Конструктор умолчания.....	50

5.3. Конструктор преобразования и конструкторы с двумя и более параметрами.....	50
5.4. Конструктор копирования.....	52
5.5. Спецификатор explicit.....	55
5.6. Конструктор копирования и операция присваивания.....	55
5.7. Автоматическая генерация конструкторов и Деструкторов.....	56
5.8. Указатель “this”.....	56
6. Методы класса, способы применения, особенности.....	59
6.1. Дружественные (friend) функции класса.....	59
6.2. Статические методы и данные.....	61
6.3. Методы const, не изменяющие объекты класса.....	63
6.4. Спецификатор mutable.....	64
7. Правила и принципы наследования. Производные и базовые классы.....	66
7.1. Правила наследования.....	66
7.2. Управление уровнем доступа к элементам класса.....	67
7.3. Множественное наследование. Конструктор во множественном наследовании.....	73
7.4. Виртуальный базовый класс.....	76
7.5. Последовательность создания и уничтожения Подобъектов.....	76
8. Виртуальные функции. Позднее связывание.....	79
8.1. Виртуальные функции.....	79
8.2. Виртуальные деструкторы.....	86
9. Переопределение операций.....	89
9.1. Перегрузка операций.....	89
9.2. Многократная перегрузка операций.....	96
9.3. Перегрузка операции присваивания =.....	96
9.4. Перегрузка операций [], () и ->.....	99
10. Шаблоны функций, шаблоны классов.....	104
10.1. Параметры шаблона.....	104
10.2. Отождествление типов аргументов.....	105
10.3. Алгоритм поиска оптимально отождествляемой функции (с учетом шаблонов).....	106
10.4. Шаблоны классов.....	107
10.5. Наследование в шаблонах классов.....	110
11. Другие возможности стандартных библиотек.....	113
11.1. Указатели на компоненты класса.....	113

11.2. Понятие абстрактного класса. “Чистые” виртуальные Функции.....	117
12. Библиотека потоков данных.....	122
12.1. Простое внесение.....	122
12.2. Выражение извлечения.....	125
12.3. Создание собственных функции внесения и извлечения...	125
12.4. Функции библиотеки iostream.....	127
12.5. Манипуляторы ввода-вывода.....	129
12.6. Файловые и строковые потоки.....	132
13. Исключительные ситуации, обработка, основные операции.....	134
13.1. Возбуждение исключения.....	134
13.2. Перехват исключений.....	142
13.3. Объекты-исключения.....	143
13.4. Раскрутка стека.....	147
13.5. Повторное возбуждение исключения.....	148
13.6. Перехват всех исключений.....	150
13.7. Спецификации исключений.....	152
13.8. Спецификации исключений и указатели на функции.....	155
14. Особенности построения графического интерфейса с использованием библиотек Qt.....	157
14.1. Класс QObject.....	157
14.2. Класс QWidget.....	158
14.3. Механизм сигналов и слотов.....	161
14.4. Сигналы.....	163
14.5. Слоты.....	165
14.6. Соединение объектов.....	166
14.7. Разъединение объектов.....	168
14.8. Переопределение сигналов.....	169
15. Стандартные библиотеки и проектирование основных интерфейсов приложения.....	171
15.1. Преимущества использования стандартных библиотек....	171
15.2. Иерархия классов стандартных библиотек.....	173
16. Поточная многозадачность.....	183
16.1. Многозадачность, основанная на процессах.....	183
16.2. Типы псевдопараллельной многозадачности.....	184
16.3. Многозадачность, основанная на процессах и потоках.....	185
16.4. Потоки в стандартных библиотеках.....	186
16.5. Мьютексы.....	191
16.6. Поддержка потоков в Qt.....	197

1. Предмет дисциплины и ее задачи

1.1. Введение

Изучение учебной дисциплины «Технология разработки программного обеспечения систем управления» осуществляется в соответствии с требованиями к формированию академических, социально-личностных и профессиональных компетенций специалиста в сфере программирования.

1.2. Цели и задачи учебной дисциплины

Цель учебной дисциплины – ознакомить студентов с основными технологиями, применяемыми в настоящее время для разработки программного обеспечения современных автоматизированных и автоматических систем, научить студентов способам использования обширного инструментария разработчика программного обеспечения.

Задачи дисциплины:

- подготовка специалистов в области информационных технологий и управления, имеющих достаточный объем знаний и практические навыки создания управляющих программ и программного обеспечения для автоматических и автоматизированных систем;
- приобретение знаний в области разработки программного обеспечения для систем управления;
- формирование навыков разработки управляющих программ и программных интерфейсов для автоматических и автоматизированных систем;
- изучение принципов объектно-ориентированного программирования и создание приложений под различные операционные системы,
- овладение методами создания программного обеспечения для систем управления.

1.3. Структура и содержание дисциплины

Данная дисциплина содержит 10 разделов и 29 тем.

Раздел 1. Интегрированная среда разработки Qt Creator. Особенности языков C++

Тема 1.1. Предмет дисциплины и ее задачи

Структура, содержание дисциплины, ее связь с другими дисциплинами учебного плана. Роль объектно-ориентированного

программирования, проектирования и анализа при создании систем управления.

Тема 1.2. Компиляторы gcc, g++

Работа с консольными приложениями. Создание нового проекта. Добавление к проекту файлов с исходным кодом. Компиляция, компоновка и выполнение проекта. Конфигурация проекта.

Тема 1.3. Особенности и синтаксис языков программирования

Комментарии. Соккрытие имен и унарная операция привязки. Операции new и delete для работы с динамической памятью. Особенности создания многомерных динамических массивов. Тип данных «ссылки»: псевдонимы имен и передача ссылок в функции в качестве аргументов. Ввод/вывод потоками. Передача аргументов функции по умолчанию. Механизм перегрузки функций (раннее связывание).

Раздел 2. Введение в ООП, понятие класса и объекта. Классы, дополнительные возможности методов классов

Тема 2.1. Основные принципы объектно-ориентированного программирования

Класс. Объект. Инкапсуляция. Внешние и внутренние проявления класса, абстрагирование. Ограничение доступа. Модульность. Иерархия классов и наследование. Полиморфизм, достоинства и недостатки объектно-ориентированного программирования.

Тема 2.2. Основной элемент ООП – класс. Основные компоненты, объекты

Характеристика элементов данных класса. Характеристика элементов- функций класса. Операция привязки. Встроенные функции. Полное имя компонентов класса. Указатель “this”. Методы-конструкторы объектов. Конструктор копирования.

Тема 2.3. Методы класса, способы применения, особенности

Дружественные (friend) функции класса. Способы передачи аргументов в функции-друзья класса. Методы const, не изменяющие объекты класса. Статические методы и данные. Особенности работы со статическими элементами класса.

Раздел 3. Наследование. Виртуальные функции. Переопределение операций

Тема 3.1. Правила и принципы наследования. Производные и базовые классы

Правила наследования. Управление уровнем доступа к элементам класса. Множественное наследование. Последовательность создания и

уничтожения подобъектов. Инициализация объектов при наследовании. Указатели на базовый и производные классы. Виртуальный базовый класс. Конструктор во множественном наследовании.

Тема 3.2. Виртуальные функции. Позднее связывание
Механизм переопределения метода класса (позднее связывание).
Виртуальные методы. Виртуальные деструкторы.

Тема 3.3. Переопределение операций
Общие положения и определения. Перегрузка методами и функциями-друзьями.

Раздел 4. Шаблоны (параметризованные типы). Другие возможности стандартных библиотек

Тема 4.1. Шаблоны функций, шаблоны классов
Требования к фактическим параметрам шаблона. Отождествление типов аргументов. Шаблоны классов. Шаблоны классов: не только для типов. Наследование в шаблонах классов.

Тема 4.2. Другие возможности стандартных библиотек
Указатели на компоненты класса. Понятие абстрактного класса. “Чистые” виртуальные функции. Шаблоны функций и классов. Особенности перегрузки шаблонов.

Раздел 5. Библиотека потоков данных. Обработка исключительных ситуаций

Тема 5.1. Библиотека потоков данных
Простое внесение. Выражение извлечения. Создание собственных функции внесения и извлечения. Функции библиотек разных языков программирования. Манипуляторы ввода-вывода. Файловые и строковые потоки.

Тема 5.2. Исключительные ситуации, обработка, основные операции
Операции try, catch, throw. Синтаксис операций, примеры использования.

Раздел 6. Разработка графического интерфейса

Тема 6.1. Особенности построения графического интерфейса с использованием библиотек Qt

Базовые классы Qjbest и QWidget. Система сигналов и слотов. Наследование от QWidget. Создание собственных компонентов на основе базовых классов.

Тема 6.2. Стандартные библиотеки и проектирование основных интерфейсов приложения

Преимущества использования стандартных библиотек. Иерархия классов стандартных библиотек. Описание классов. Понятие объекта приложения.

Тема 6.3. Аппаратно-независимая графика, файловая система и ресурсы приложения

Основные принципы работы и эффективное использование файловой системы. Сохранение и восстановление состояния объектов. Применение методов классов стандартных библиотек. Обеспечение сериализации данных приложения системы управления. Оптимизация вывода графики на экран. Графическое устройство и его контекст. Реализация принципа эффективной перерисовки рабочей области приложения. Главное меню приложения. Управление созданием, загрузкой и заменой меню. Динамически изменяющееся меню. Панель управления приложением. Ресурсы, форма и режимы работы панели управления. Расширение панели управления. Панель состояния работы приложения. Изменение характеристик индикаторов. Использование дополнительных возможностей панели состояния.

Тема 6.4. Поточная многозадачность

Многозадачность, основанная на процессах и потоках. Потоки стандартных библиотек. Интерфейсные и рабочие потоки приложения. Создание, запуск, остановка и возобновление выполнения потоков. Организация управления приоритетами потоков. Синхронизация потоков. Объекты синхронизации и классы стандартных библиотек. Механизм синхронизации. Общая концепция управления доступом к разделяемым ресурсам приложения. Синхронизация на основе семафоров. Работа с объектами событий. Использование критических секций.

Тема 6.5. Приложение, основанное на диалоге

Мастера проектов. Создание шаблона приложения. Главный класс приложения. Класс диалоговой панели.

Тема 6.6. Диалоговые панели

Мастера проектов. Модальная диалоговая панель. Диалоговая панель - главное окно приложения. Немодальная диалоговая панель.

Раздел 7. Динамически подключаемые библиотеки

Тема 7.1. Создание динамической библиотеки на основе стандартных библиотек. Создание приложения, обращающегося к dll. Основные понятия о библиотеке динамической загрузки (DLL). Технология функционирования

DLL. Различные типы DLL. Динамическая загрузка и выгрузка DLL. Создание DLL с использованием стандартных библиотек. Экспорт класса. Экспортирование объектов и переменных. Экспорт функции.

Тема 7.2. Эффективное использование библиотек динамической загрузки

Использование DLL в приложении. Использование DLL с библиотекой импорта. Использование DLL без библиотеки импорта. Эффективная загрузка DLL. Загрузка динамических расширений стандартных библиотек.

Раздел 8. Технология Qt Quick

Тема 8.1. Применение визуальных элементов, фиксаторов и пользовательский ввод

Общие положения Qt Quick. Язык описания графического интерфейса QML. Базовые элементы. Управление размещением элементов на форме. Компоненты для ввода и вывода информации.

Тема 8.2. Создание собственных элементов QML

Создание собственных графических компонентов. Компоновка элементов в пользовательские библиотеки.

Раздел 9. Поддержка баз данных

Тема 9.1. Доступ к данным БД

Поддержка СУБД: Oracle, Microsoft SQL Server, Sybase Adaptive Server, IBM DB2, PostgreSQL, MySQL и ODBC-совместимые базы данных.

Тема 9.2. Поддержка баз данных в стандартных библиотеках

Классы для работы с базами данных. Ошибки при операциях с базами данных. Создание приложения, работающего с базой данных.

Тема 9.3. Использование языка SQL в библиотеке Qt

Слой драйверов: QSqlDriver, QSqlDriverCreator, QSqlDriverCreatorBase, QSqlDriverPlugin и QSqlResult.

Слой SQL API: QSqlDatabase, QSqlQuery, QSqlError, QSqlField, QSqlIndex и QSqlRecord

Раздел 10. Разработка служб. Поддержка сети

Тема 10.1. Применение стандартных протоколов HTTP и FTP

Описание протоколов. Авторизация. Применение стандартных библиотек для работы с ними.

Тема 10.2. Сокетные соединения. TCP и UDP

Основы работы с SOCKET. Технология клиент-сервер. Поддержка сокетного соединения в Qt

Тема 10.3. Разработка служб

Понятие системной службы. Особенности разработки служб для Windows (Windows Service Applications). Особенности разработки служб (daemons) для Linux. Интеграция службы в операционную систему.

1.4. Перечень тем лабораторных занятий

1. Компилятор gcc и интегрированная среда Qt Creator
2. Работа с динамической памятью, указателями на массивы данных, передача указателя на массив в качестве аргумента в функцию.
3. Использование классов, встроенных методов, доступ к классу через объект, указатель на объект и через динамическое выделение памяти под класс.
4. Использование методов классов: конструктора, деструктора и дружественной функции.
5. Наследование классов, механизм виртуальных функций, переопределение операций.
6. Программирование шаблонов функций и классов.
7. Файлового ввод - вывод с применением файловых потоков.
8. Функции обработки исключительных ситуаций.
9. Основы построения графического интерфейса. Базовый класс Qwidget. Система сигналов и слотов.
10. Управление компоновкой элементов на форме.
11. Работа с элементами отображения.
12. Элементы ввода, управления и выбора.
13. Управление событиями.
14. Компоненты для работы с мультимедиа.
15. Многопоточность в приложении. Синхронизация потоков.
16. Работа с диалогами. Класс QDialog.
17. Построения основного окна приложения на основе QMainWindow.
18. Разработка и использование динамических библиотек.
19. Основы QML. Построение графического интерфейса.
20. Разработка собственных компонентов на основе QML.
21. Разработка приложений с поддержкой БД.
22. Работа с сетями.
23. Разработка системных служб (часть 1).
24. Разработка системных служб (часть 2).

Роль объектно-ориентированного программирования, проектирования и анализа при создании систем управления

От любого метода программирования мы ждем, что он поможет нам в решении наших проблем. Но одной из самых значительных проблем в программировании является сложность. Чем больше и сложнее программа, тем важнее становится разбить ее на небольшие, четко очерченные части. Чтобы побороть сложность, мы должны абстрагироваться от мелких деталей. В этом смысле классы представляют собой весьма удобный инструмент.

- Классы позволяют проводить конструирование из полезных компонентов, обладающих простыми инструментами, что дает возможность абстрагироваться от деталей реализации.
- Данные и операции вместе образуют определенную сущность, и они не «размазываются» по всей программе, как это нередко бывает в случае процедурного программирования.
- Локализация кода и данных улучшает наглядность и удобство сопровождения программного обеспечения.
- Инкапсуляция информации защищает наиболее критичные данные от несанкционированного доступа.

ООП дает возможность создавать расширяемые системы (extensible systems). Это одно из самых значительных достоинств ООП и именно оно отличает данный подход от традиционных методов программирования. Расширяемость (extensibility) означает, что существующую систему можно заставить работать с новыми компонентами, причем без внесения в нее каких-либо изменений. Компоненты могут быть добавлены на этапе выполнения.

Расширение типа (type extension) и вытекающий из него полиморфизм переменных оказываются полезными преимущественно в следующих ситуациях.

Обработка разнородных структур данных. Программы могут работать, не утруждая себя изучением вида объектов. Новые виды могут быть добавлены в любой момент.

Изменение поведения во время выполнения. На этапе выполнения один объект может быть заменен другим. Это может привести к изменению алгоритма, в котором используется данный объект.

Реализация родовых компонент. Алгоритмы можно обобщать до такой степени, что они уже смогут работать более, чем с одним видом объектов.

Доведение полуфабрикатов. Компоненты нет надобности подстраивать под определенное приложение. Их можно сохранять в библиотеке в виде полуфабрикатов (semifinished products) и расширять по мере необходимости до различных законченных продуктов.

Расширение каркаса. Независимые от приложения части предметной области могут быть реализованы в виде каркаса и в дальнейшем расширены за счет добавления частей, специфичных для конкретного приложения.

Многоразового использования программного обеспечения на практике добиться не удастся из-за того, что существующие компоненты уже не отвечают новым требованиям. ООП помогает этого достичь без нарушения работы уже имеющихся клиентов, что позволяет нам извлечь максимум из многоразового использования компонент.

- Мы сокращаем время на разработку, которое с выгодой может быть отдано другим проектам;
- Компоненты многоразового использования обычно содержат гораздо меньше ошибок, чем вновь разработанные, ведь они уже не раз подвергались проверке;
- Когда некая компонента используется сразу несколькими клиентами, то улучшения, вносимые в ее код, одновременно оказывают свое положительное влияние и на множество работающих с ней программ;
- Если программа опирается на стандартные компоненты, то ее структура и пользовательский интерфейс становятся более унифицированными, что облегчает ее понимание и упрощает ее использование.

2. Компиляторы gcc, g++

Основные понятия

Компиляция - трансляция программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке, близком машинному коду (абсолютный код, объектный модуль, иногда на язык ассемблера). Входной информацией для компилятора (исходный код) является описание алгоритма или программа на объектно-ориентированном языке, а на выходе компилятора — эквивалентное описание алгоритма на машинно-ориентированном языке (объектный код).

Компоновщик (также редактор связей или линкер, отангл. link editor, linker) — это инструментальная программа, которая производит компоновку («линковку»): принимает на вход один или несколько объектных модулей и собирает по ним исполнимый модуль.

GNU Compiler Collection (обычно используется сокращение GCC) — набор компиляторов для различных языков программирования, разработанный в рамках проекта GNU. GCC является свободным программным обеспечением, распространяется фондом свободного программного обеспечения (FSF) на условиях GNU GPL и GNU LGPL и является ключевым компонентом GNU toolchain. Он используется как стандартный компилятор для свободных UNIX-подобных операционных систем.

Изначально названный GNU C Compiler поддерживал только язык Си. Позднее GCC был расширен для компиляции исходных кодов на таких языках программирования, как C++, Objective-C, Java, Фортран, Ada и Go.

Начало GCC было положено Ричардом Столлманом, который реализовал первый вариант GCC в 1985 году на нестандартном и непереносимом диалекте языка Паскаль; позднее компилятор был переписан на языке Си Леонардом Тауэром (англ. Leonard H. Tower Jr.) и Ричардом Столлманом и выпущен в 1987 году как компилятор для проекта GNU, который сам по себе являлся свободным программным обеспечением. Разработка GCC курируется Free Software Foundation.

В настоящее время GCC поддерживается группой программистов со всего мира. GCC является лидером по количеству процессоров и операционных систем, которые он поддерживает.

Будучи официальным компилятором системы GNU, GCC также является главным компилятором для сборки ряда других операционных

систем; среди них - различные варианты Linux и BSD, а также ReactOS, Mac OS X, OpenSolaris, NeXTSTEP, BeOS и Haiku.

GCC часто выбирается для разработки программного обеспечения, которое должно работать на большом числе различных аппаратных платформ. Различия между «родными» для каждой из аппаратных платформ компиляторами приводят к трудностям при разработке кода, который бы корректно компилировался разными компиляторами, а кроме того, при использовании различных компиляторов сильно усложняются сборочные скрипты, которые должны собирать ПО для всех аппаратных платформ. При использовании GCC для компиляции кода под разные платформы, будет использован один и тот же синтаксический анализатор. Поэтому, если удалось собрать программу для одной из целевых платформ, то велика вероятность, что программа нормально соберётся и для других платформ.

Поддерживаемые языки:

- Ada (GCC для Ada, или GNAT)
- Си
- C++ (C++ для GCC, или G++)
- Фортран (GCC для Fortran, или gfortran)
для Java, или GCJ)
- Objective-C (GCC для Objective-C, или gobjc)
- Objective-C++ (GCC для Objective-C++, или gobjc++)
- Go (GCC для Go, или gccgo)

Поддерживаемые архитектуры:

- Alpha
- ARM
- Atmel AVR
- Blackfin
- HC12
- H8/300
- x86 (IA-32 и x86-64)
- IA-64 («Itanium»)
- m68k
- Motorola 88000
- MIPS
- Texas Instruments MSP430
- PA-RISC
- PDP-11
- PowerPC

- R8C/M16C/M32C
- SPU B Cell
- System/370, System/390
- SuperH
- SPARC
- VAX
- A29K
- ARC
- ETRAX CRIS
- D30V
- DSP16xx
- FR-30
- FR-V
- Intel i960
- IP2000
- M32R
- 68HC11
- MCORE
- MMIX
- MN10200
- MN10300
- Motorola 88000
- NS32K
- ROMP
- Stormy16
- V850
- Xtensa
- AVR32
- D10V
- MeP
- MicroBlaze
- TI MSP430
- TI C6X
- Nios II и Nios
- PDP-10
- TIGCC (вариация Motorola 68000)
- Z8000
- PIC24/dsPIC

- OpenRISC 1000

Внешний интерфейс GCC является стандартом для компиляторов на платформе UNIX. Пользователь вызывает управляющую программу, которая называется `gcc`. Она интерпретирует аргументы командной строки, определяет и запускает для каждого входного файла свои компиляторы нужного языка.

Компилятор каждого языка является отдельной программой, которая получает исходный текст и порождает вывод на языке ассемблера. Все компиляторы имеют общую внутреннюю структуру: `front end`, который производит синтаксический разбор и порождает абстрактное синтаксическое дерево, и `back end`, который конвертирует дерево в Register Language[en] (RTL), выполняет различные оптимизации, затем порождает программу на языке ассемблера, используя архитектурно-зависимое сопоставление с образцом.

Компилятор GCC

GCC - это свободно доступный оптимизирующий компилятор для языков C, C++.

Программа `gcc`, запускаемая из командной строки, представляет собой надстройку над группой компиляторов. В зависимости от расширений имен файлов, передаваемых в качестве параметров, и дополнительных опций, `gcc` запускает необходимые препроцессоры, компиляторы, линкеры.

Файлы с расширением `.cc` или `.C` рассматриваются, как файлы на языке C++, файлы с расширением `.c` как программы на языке C, а файлы с расширением `.o` считаются объектными.

Чтобы откомпилировать исходный код C++, находящийся в файле `F.cc`, и создать объектный файл `F.o`, необходимо выполнить команду:

Опция `-c` означает «только компиляция».

Чтобы скомпоновать один или несколько объектных файлов, полученных из исходного кода `-F1.o, F2.o, ...` - в единый исполняемый файл `F`, необходимо ввести команду:

Опция `-o` задает имя исполняемого файла.

Можно совместить два этапа обработки - компиляцию и компоновку - в один общий этап с помощью команды:

```
gcc -o F <compile-and-link-options> F1.cc ... -lg++ <other-libraries>
```

k `–options`> - возможные дополнительные опции компиляции и компоновки. Опция `–lg++` указывает на необходимость подключить стандартную библиотеку языка C++, `<other-libraries>` - возможные дополнительные библиотеки.

После компоновки будет создан исполняемый файл F, который можно запустить с помощью команды `./F <arguments>`. Строка `<arguments>` определяет аргументы командной строки Вашей программы.

В процессе компоновки очень часто приходится использовать библиотеки. Библиотекой называют набор объектных файлов, сгруппированных в единый файл и проиндексированных. Когда команда компоновки обнаруживает некоторую библиотеку в списке объектных файлов для компоновки, она проверяет, содержат ли уже скомпонованные объектные файлы вызовы для функций, определенных в одном из файлов библиотек. Если такие функции найдены, соответствующие вызовы связываются с кодом объектного файла из библиотеки. Библиотеки могут быть подключены с помощью опции вида `-lname`. В этом случае в стандартных каталогах, таких как `/lib`, `/usr/lib`, `/usr/local/lib` будет проведен поиск библиотеки в файле с именем `libname.a`. Библиотеки должны быть перечислены после исходных или объектных файлов, содержащих вызовы к соответствующим функциям.

Опции компиляции

Самые распространенные опции компиляции приведены в таблице

Таблица 2.1. Основные опций компиляции

Опция	Назначение
	Эта опция означает, что необходима только компиляция. Из исходных файлов программы создаются объектные файлы в виде <code>name.o</code> . Компоновка не производится.
	Определить имя <code>name</code> в компилируемой программе, как значение <code>value</code> . Эффект такой же, как наличие строки в начале программы. Часть <code>=value</code> может быть опущена, в этом случае значение по умолчанию равно 1.
	Использовать <code>file-name</code> в качестве имени для создаваемого файла.
	Использовать при компоновке библиотеку <code>libname.so</code>

	Добавить к стандартным каталогам поиска библиотек и заголовочных файлов пути <code>lib-path</code> и <code>include-path</code> соответственно.
	Поместить в объектный или исполняемый файл отладочную информацию для отладчика <code>gdb</code> . Опция должна быть указана и для компиляции, и для компоновки. В сочетании <code>-g</code> рекомендуется использовать опцию отключения оптимизации <code>-O0</code>
	Вывести зависимости от заголовочных файлов, используемых в Си или C++ программе, в формате, подходящем для утилиты <code>make</code> . Объектные или исполняемые файлы не создаются.
	Поместить в объектный или исполняемый файл инструкции профилирования для генерации информации, используемой
	Вывод сообщений о всех предупреждениях или ошибках, возникающих во время компиляции программы.
	Различные уровни оптимизации.
	Не оптимизировать. Если вы используете многочисленные <code>-O</code> опции с номерами или без номеров уровня, действительной является последняя такая опция.
	Используется для добавления ваших собственных каталогов для поиска заголовочных файлов в процессе сборки
	Передается компоновщику. Используется для добавления ваших собственных каталогов для поиска библиотек в процессе сборки.

Утилита `make`

— утилита, автоматизирующая процесс преобразования файлов из одной формы в другую. Чаще всего это компиляция исходного кода в объектные файлы и последующая компоновка в исполняемые файлы или библиотеки.

Утилита использует специальные `make`-файлы, в которых указаны зависимости файлов друг от друга и правила для их удовлетворения. На основе информации о времени последнего изменения каждого файла `make`

определяет и запускает необходимые программы. Основные параметры утилиты make приведены в таблице 2.2.

Таблица 2.2. Параметры утилиты make

Опция	Описание
	Игнорируется для совместимости.
	Без условий обрабатывать все цели.
-C КАТАЛОГ, --directory=КАТАЛОГ	Перейти в КАТАЛОГ перед выполнением действий.
	Выводить массу отладочных сообщений.
--debug[=ФЛАГИ]	Выводить различные типы отладочной информации.
	Переменные окружения заменяют значения
--eval=СТРОКА	Вычислить СТРОКУ, как предложение makefile.
-f ФАЙЛ, --file=ФАЙЛ, --makefile=ФАЙЛ	Использовать ФАЙЛ в качестве makefile.
	Показать справку и выйти.
	Игнорировать ошибки способов.
-I КАТАЛОГ --include-dir=КАТАЛОГ	Искать включаемые make-файлы в КАТАЛОГЕ.
	Запускать одновременно до N заданий; если N не указано, число заданий неограничено.
	Продолжать работу, даже если некоторые цели не могут быть достигнуты.
-l [N], --load-average[=N],	Не запускать несколько заданий, если загрузка больше N.
	Показать информацию о версии и выйти.
	Напечатать текущий каталог.
	Отменить ключ -w, даже если он был явно указан.

Простой make-файл состоит из "правил" (rules) следующего вида:

цель ... : пререквизит ...
команда

Обычно, цель (target) представляет собой имя файла, который генерируется в процессе работы утилиты make. Примером могут служить объектные и исполняемый файлы собираемой программы. Цель также может быть именем некоторого действия, которое нужно выполнить (например, 'clean' — очистить).

Пререквизит (prerequisite) - это файл, который используется как исходные данные для порождения цели. Очень часто цель зависит сразу от нескольких файлов.

Команда - это действие, выполняемое утилитой make. В правиле может содержаться несколько команд - каждая на своей собственной строке. Важное замечание: строки, содержащие команды обязательно должны начинаться с символа табуляции.

Обычно, команды находятся в правилах с пререквизитами и служат для создания файла-цели, если какой-нибудь из пререквизитов был модефицирован. Однако, правило, имеющее команды, не обязательно должно иметь пререквизиты. Например, правило с целью 'clean' ("очистка"), содержащее команды удаления, может не иметь пререквизитов.

Правило (rule) описывает, когда и каким образом следует обновлять файлы, указанные в нем в качестве цели. Для создания или обновления цели, make исполняет указанные в правиле команды, используя пререквизиты в качестве исходных данных. Правило также может описывать, каким образом должно выполняться некоторое действие.

Помимо правил, make-файл может содержать и другие конструкции, однако, простой make-файл может состоять и из одних лишь правил. Правила могут выглядеть более сложными, чем приведенный выше шаблон, однако все они более или менее соответствуют ему по структуре.

Пример простого make-файла, в котором описывается, что исполняемый файл edit зависит от восьми объектных файлов, которые, в свою очередь, зависят от восьми соответствующих исходных файлов и трех заголовочных файлов.

В данном примере, заголовочный файл 'defs.h' включается во все файлы с исходным текстом. Заголовочный файл 'command.h' включается

только в те исходные файлы, которые относятся к командам редактирования, а файл `buffer.h` - только в "низкоуровневые" файлы, непосредственно оперирующие буфером редактирования.

```
edit : main.o kbd.o command.o display.o \  
      insert.o search.o files.o utils.o  
      cc -o edit main.o kbd.o command.o display.o \  
          insert.o search.o files.o utils.o
```

```
main.o : main.c defs.h  
        cc -c main.c  
kbd.o : kbd.c defs.h command.h  
        cc -c kbd.c  
command.o : command.c defs.h command.h  
           cc -c command.c  
display.o : display.c defs.h buffer.h  
           cc -c display.c  
insert.o : insert.c defs.h buffer.h  
           cc -c insert.c  
search.o : search.c defs.h buffer.h  
           cc -c search.c  
files.o : files.c defs.h buffer.h command.h  
          cc -c files.c  
utils.o : utils.c defs.h  
          cc -c utils.c  
clean :  
        rm edit main.o kbd.o command.o display.o \  
          insert.o search.o files.o utils.o
```

Для повышения удобочитаемости, строки разбиты на две части с помощью символа обратной косой черты, за которым следует перевод строки.

В приведенном примере, целями, в частности, являются объектные файлы `main.o` и `kbd.o`, а также исполняемый файл `edit`. К пререквизитам относятся такие файлы, как `main.c` и `defs.h`. Каждый объектный файл, фактически, является одновременно и целью и пререквизитом. Примерами команд могут служить `cc -c main.c` и `cc -c kbd.c`.

В случае, если цель является файлом, этот файл должен быть перекомпилирован или перекомпонован всякий раз, когда был изменен

какой-либо из его пререквизитов. Кроме того, любые пререквизиты, которые сами генерируются автоматически, должны быть обновлены первыми. В нашем примере, исполняемый файл `edit` зависит от восьми объектных файлов; объектный файл `main.o` зависит от исходного файла `main.c` и заголовочного файла `defs.h`.

За каждой строкой, содержащей цель и пререквизиты, следует строка с командой. Эти команды указывают, каким образом надо обновлять целевой файл. В начале каждой строки, содержащей команду, должен находиться символ табуляции. Именно наличие символа табуляции является признаком, по которому `make` отличает строки с командами от прочих строк `make`-файла.

Цель `clean` является не файлом, а именем действия. Поскольку, при обычной сборке программы это действие не требуется, цель `clean` не является пререквизитом какого-либо из правил. Следовательно, `make` не будет "трогать" это правило, пока не произойдет принудительный вызов. Это правило не только не является пререквизитом, но и само не содержит каких-либо пререквизитов. Таким образом, единственное предназначение данного правила - выполнение указанных в нем команд. Цели, которые являются не файлами, а именами действий называются абстрактными целями (phony

В приведенном выше примере, в правиле для `edit` дважды перечисляется список объектных файлов программы:

```
edit : main.o kbd.o command.o display.o \  
      insert.o search.o files.o utils.o  
cc -o edit main.o kbd.o command.o display.o \  
    insert.o search.o files.o utils.o
```

Подобное дублирование чревато ошибками. При добавлении в проект нового объектного файла, можно добавить его в один список и забыть про другой. Можно устранить подобный риск, и, одновременно, упростить `make`-файл, используя переменные. Переменные (variables) позволяют, один раз, определив текстовую строку, затем использовать ее многократно в нужных местах.

Обычной практикой при построении `make`-файлов является использование переменной с именем `objects`, `OBJECTS`, `objs`, `OBJS`, `obj`, или `OBJ`, которая содержит список всех объектных файлов программы. Можно определить подобную переменную с именем `objects` таким образом:

```
objects = main.o kbd.o command.o display.o \  
          insert.o search.o files.o utils.o
```

Далее, всякий раз, когда нужен будет список объектных файлов, можно использовать значение этой переменной с помощью записи '\$(objects)'. Вот как будет выглядеть предыдущий пример с использованием переменной для хранения списка объектных файлов:

```
objects = main.o kbd.o command.o display.o \  
         insert.o search.o files.o utils.o
```

```
edit : $(objects)  
      cc -o edit $(objects)  
main.o : main.c defs.h  
      cc -c main.c  
kbd.o : kbd.c defs.h command.h  
      cc -c kbd.c  
command.o : command.c defs.h command.h  
      cc -c command.c  
display.o : display.c defs.h buffer.h  
      cc -c display.c  
insert.o : insert.c defs.h buffer.h  
      cc -c insert.c  
search.o : search.c defs.h buffer.h  
      cc -c search.c  
files.o : files.c defs.h buffer.h command.h  
      cc -c files.c  
utils.o : utils.c defs.h  
      cc -c utils.c  
clean :  
      rm edit $(objects)
```

На самом деле, нет необходимости явного указания команд компиляции отдельно для каждого из исходных файлов. Утилита `make` сама может "догадаться" об использовании нужных команд, поскольку у нее имеется, так называемое, неявное правило (*implicit rule*) для обновления файлов с расширением `.o` из файлов с расширением `.c`, с помощью команды `cc -c main.c -o main.o` для преобразования файла `main.c` в файл `main.o`. Таким образом, можно убрать явное указание команд компиляции из правил, описывающих построение объектных файлов.

Когда файл с расширением `.c` автоматически используется подобным образом, он также автоматически добавляется в список пререквизитов "своего" объектного файла. Таким образом, мы вполне можем убрать файлы с расширением `.c` из списков пререквизитов объектных файлов.

Пример можно модифицировать следующим образом:

```
objects = main.o kbd.o command.o display.o \  
insert.o search.o files.o utils.o
```

```
edit : $(objects)  
cc -o edit $(objects)
```

```
main.o : defs.h  
kbd.o : defs.h command.h  
command.o : defs.h command.h  
display.o : defs.h buffer.h  
insert.o : defs.h buffer.h  
search.o : defs.h buffer.h  
files.o : defs.h buffer.h command.h  
utils.o : defs.h
```

```
.PHONY : clean
```

Примерно так и выглядят `make`-файлы в реальной практике. Если для создания объектных файлов используются только неявные правила, то можно использовать другой стиль написания `make`-файлов. В таком `make`-файле записи группируются по их пререквизитам, а не по их целям. Вот как может выглядеть подобный `make`-файл:

```
objects = main.o kbd.o command.o display.o \  
insert.o search.o files.o utils.o
```

```
edit : $(objects)  
cc -o edit $(objects)
```

```
$(objects) : defs.h  
kbd.o command.o files.o : command.h  
display.o insert.o search.o files.o : buffer.h
```

Здесь, заголовочный файл `defs.h` объявляется пререквизитом для всех объектных файлов программы. Файлы `command.h` и `buffer.h` являются пререквизитами для перечисленных объектных файлов.

Часто, в make-файле указывается, каким образом можно выполнить некоторые другие действия, не относящиеся к компиляции программы. Таким действием, например, может быть удаление все объектных и исполняемых файлов программы для очистки каталога. Вот как можно было бы написать правило для очистки каталога:

Однако на практике, это правило реализуется чуть более сложным способом, предполагающим возможность непредвиденных ситуаций:

```
.PHONY : clean
```

```
clean :
```

```
    -rm edit $(objects)
```

Такая запись предотвратит возможную путаницу, если, вдруг, в каталоге будет находится файл с именем `clean`, а также позволит make продолжить работу, даже если команда `g`.

3. Особенности и синтаксис языков программирования

Комментарии

В С все комментарии начинаются с пары символов /* и заканчиваются парой */ (в случае многострочных комментариев) или // (в случае однострочного). Между слэшем и звездочкой не должно быть пробелов. Компилятор игнорирует любой текст между данными парами символов. Например, следующая программа выводит на экран только hello:

Комментарии могут находиться в любом месте программы, за исключением случая, когда комментарий разбивает на части ключевое слово или идентификатор. Таким образом, следующий комментарий абсолютно корректен:

```
x = 10 + /* сложение чисел */ 5;  
в то время как  
swi/* не работает */tch(c) { . . .
```

некорректно, поскольку ключевое слово не может содержать комментарий. Тем не менее комментарии, как правило, не принято помещать в середину выражения, поскольку в таких случаях труднее разобраться с самим выражением.

Комментарии не могут быть вложенными, т.е. один комментарий не может содержать другой комментарий. Например, следующий фрагмент, когда вызовет ошибку при компиляции:

```
/* внешний комментарий  
x = y / a;  
/* внутренний комментарий вызывает ошибку */
```

Комментарии следует использовать, когда необходимо объяснить какую-либо операцию кода. Все функции, за исключением самых

очевидных, должны содержать комментарии в начале их объявления, где следует писать, что функция делает, какие параметры она получает и что возвращает.

Указатели

Указатель это переменная, которая содержит адрес другой переменной. Для объявления указателя нужно перед именем переменной поставить символ '*'.
Библиотека ГГТУ им. О.Сумского

При объявлении указателя, желательно, сразу же его проинициализировать. Это значительно упрощает отладку программы. Если начальное значение указателя неизвестно, то ему нужно присвоить значение NULL, которое будет явно указывать, что указатель пока не используется. Например,

Адресация бывает:

- прямая
- косвенная

Прямой доступ:

```
printf("Address: %x\n",&count);
```

Косвенный доступ:

```
int count=154;  
int*p=&count;  
(*p)++;  
printf("Address: %x\n",p);
```

где *p* - переменная-указатель, используемая для хранения адреса переменной *count*.

Так же существует косвенный доступ второго порядка, т. е. указатель на указатель. Например:

где

- указатель на тип `int`;
- указатель на указатель на тип `int`;
- доступ к значению `p`;
- доступ к значению `count`.

Применение указателей:

- передача параметров в функцию;
- манипуляции адресами;
- повышение эффективности кода;
- организация динамических массивов;
- формирование связей между объектами.

Особенности работы с указателями:

- указатель всегда объявляется на определенный тип;
- размер указателя определяется разрядностью ОС;
- при работе с разнотипными указателями необходимо использовать явное преобразование типа.

Можно объявить указатель на:

- обычную скалярную переменную;
- массив;
- функцию;
- указатель.

Объявление и присваивание:

```
int val1=0;
char ch='#';
double pi=3.14159265358;
```

```
int*p1;
char* p2;
double*p3;
```

```
p1=&val1;
p2=&ch;
```

Объявление и инициализация

с
н
е
п
в
н
р
р
р
т
н
н
н
р

Примеры работы с указателями:

```
printf("A=%d, B=%d\n", *pa, *pb);  
printf("A=%d, B=%d\n", a, b);  
temp=pa;  
pa=pb;  
pb=temp;  
printf("A=%d, B=%d\n", *pa, *pb);  
printf("A=%d, B=%d\n", a, b);
```

2)

```
int a=10,b=20,temp;  
int * pa=&a;  
int * pb=&b;
```

```
printf("A=%d, B=%d\n", *pa, *pb);  
printf("A=%d, B=%d\n", a, b);  
temp=*pa;  
*pa=*pb;  
*pb=temp;  
printf("A=%d, B=%d\n", *pa, *pb);  
printf("A=%d, B=%d\n", a, b);
```


Работа с динамической памятью

Динамическое выделение памяти необходимо для эффективного использования памяти компьютера. Например, написана программа, которая обрабатывает массив. При написании данной программы необходимо было объявить массив, то есть задать ему фиксированный размер (к примеру, от 0 до 100 элементов). Тогда данная программа будет не универсальной, ведь может обрабатывать массив размером не более 100 элементов. А если понадобятся всего 20 элементов, но в памяти выделится место под 100 элементов, ведь объявление массива было статическим, а такое использование памяти крайне неэффективно.

В C++ операции `new` и `delete` предназначены для динамического распределения памяти компьютера. Операция `new` выделяет память из области свободной памяти, а операция `delete` высвобождает выделенную память. Выделяемая память, после её использования должна высвободиться, поэтому операции `new` и `delete` используются парами.

```
// пример использования операции new
```

```
//где ptrvalue – указатель на выделенный участок памяти типа int  
//new – операция выделения свободной памяти под создаваемый объект.
```

Операция `new` создает объект заданного типа, выделяет ему память и возвращает указатель правильного типа на данный участок памяти. Если память невозможно выделить, например, в случае отсутствия свободных участков, то возвращается нулевой указатель, то есть указатель вернет значение 0. Выделение памяти возможно под любой тип данных: `int`, `float`, `double`, `char` и т. д.

```
// пример использования операции delete:
```

```
// где ptrvalue – указатель на выделенный участок памяти типа int  
// delete – операция высвобождения памяти
```

```
using namespace std;
```

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

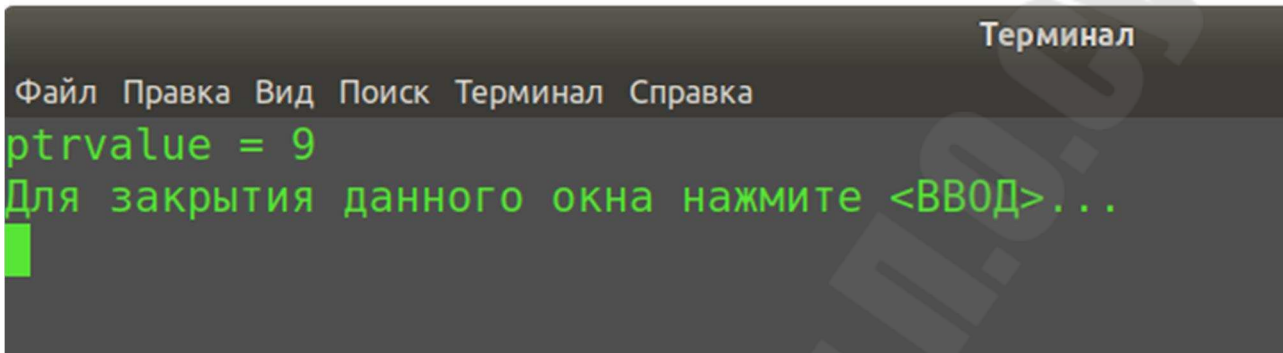
```
int *ptrvalue; // указатель на объект типа int
```

```
//int *ptrvalue = new int (9); инициализация может выполняться сразу при  
объявлении динамического объекта
```

```
cout << "ptrvalue = " << *ptrvalue << endl;
```

```
delete ptrvalue; // высвобождение памяти
```

Результат выполнения данной программы приведен на рисунке 3.1.



```
Терминал
Файл Правка Вид Поиск Терминал Справка
ptrvalue = 9
Для закрытия данного окна нажмите <ВВОД>...
```

Рис. 3.1. Вывод результатов работы программы.

Динамические массивы

Чаще всего операции `new` и `delete` применяются для создания динамических массивов, а не для создания динамических переменных. Рассмотрим фрагмент кода создания одномерного динамического массива.

```
// объявление одномерного динамического массива на 10 элементов:
```

```
// где ptrarray – указатель на выделенный участок памяти под массив вещественных чисел типа float
```

```
// в квадратных скобках указываем размер массива
```

После того как динамический массив стал ненужным, нужно освободить участок памяти, который под него выделялся.

```
// высвобождение памяти, отводимой под одномерный динамический массив:
```

После оператора `delete` ставятся квадратные скобки, которые говорят о том, что высвобождается участок памяти, отводимый под одномерный массив. Пример программы, в которой создается одномерный динамический массив, заполненный случайными числами.

```
// в заголовочном файле <ctime> содержится прототип функции time()
```

```

// в заголовочном файле <iomanip> содержится прототип функции

#include <iomanip>

using namespace std;

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

    srand(time(0)); // генерация случайных чисел
    float *ptrarray = new float [10]; // создание динамического массива
    вещественных чисел на десять элементов
    for (int count = 0; count < 10; count++)
        ptrarray[count] = (rand() % 10 + 1) / float((rand() % 10 + 1));
//заполнение массива случайными числами с масштабированием от 1 до 10
cout << "array = ";
    for (int count = 0; count < 10; count++)
        cout << setprecision(2) << ptrarray[count] << " ";
    delete [] ptrarray; // высвобождение памяти

```

Созданный одномерный динамический массив заполняется случайными вещественными числами, полученными с помощью функций генерации случайных чисел, причём числа генерируются в интервале от 1 до 10, интервал задается так — $\text{rand()} \% 10 + 1$. Чтобы получить случайные вещественные числа, выполняется операция деления, с использованием явного приведения к вещественному типу знаменателя — $\text{float}((\text{rand()} \% 10 + 1))$. Чтобы показать только два знака после запятой используем функцию `setprecision(2)`, прототип данной функции находится в заголовочном файле `<iomanip>`. Функция `time(0)` заполняет генератор случайных чисел временным значением, таким образом, получается, воспроизводить случайность возникновения чисел. Вывод результатов выполнения данного примера приведен на рисунке 3.2.

```
Терминал
Файл Правка Вид Поиск Терминал Справка
array = 1.6 0.5 2.3 1 0.75 0.75 0.2 1 1 0.17
Для закрытия данного окна нажмите <ВВОД>...
```

Рис. 3.2. Вывод результатов работы программы.

По завершению работы с массивом, он удаляется, таким образом, высвобождается память, отводимая под его хранение.

Особенности создания многомерных динамических массивов

Основной способ работы с динамическими матрицами базируется на использовании двойных указателей. Рассмотрим следующий фрагмент программы.

С помощью оператора `new` создан массив из N элементов, каждый из которых является адресом, где хранится указатель на `float`. Осталось определить значение этого указателя. Для этого организуется цикл от 0 до $N-1$, в котором каждый указатель будет адресовать участок памяти, где хранится M элементов типа `float`.

```
for (i=0; i<N; i++)
    a[i]=new float [M];
```

После этого определен массив N указателей, каждый из которых адресует массив из M вещественных чисел (типа `float`). Фактически создана динамическая матрица размера $N \times M$. Обращение к элементу динамической матрицы проходит так же, как и к элементу статической матрицы. Для обращения к элементу $a_{i,j}$ в программе на C++ необходимо указать ее имя, и в квадратных скобках номер строки и столбца (`a[i][j]`).

Пример программы, которая реализует перемножение двух матриц:

```

#include <iostream>

using namespace std;

int main()
{
    setlocale (LC_ALL, "RUS");

    //описание двойных указателей для матриц A, B, C

    //ввод размерности матриц
    cout<<"N = "; cin>>N;
    cout<<"M = "; cin>>M;

    //выделение памяти для массива a, каждый элемент массива -
    //указатель на double
    a=new double *[N];

    //выделение памяти для каждого элемента
    //a[i], a[i] адресует M элементов типа double
    a[i]=new double [M];

    //выделение памяти для массива b, каждый элемент массива -
    //указатель на double
    b=new double *[M];

    //выделение памяти для каждого элемента
    //b[i], b[i] адресует M элементов типа double
    b[i]=new double [L];

    //выделение памяти для массива c, каждый элемент массива -
    //указатель на double
    c=new double *[N];

    //выделение памяти для каждого элемента
    //c[i], c[i] адресует L элементов типа double
    c[i]=new double [L];

    //ввод матриц A и B
    cout<<"введите матрицу A"<<endl;
    for (i=0; i<N; i++)
        for (j=0; j<M; j++)

```

```

        cin>>a[i][j];
    cout<<"введите матрицу B"<<endl;
    for (i=0; i<M; i++)
        for (j=0; j<L; j++)

//умножение матриц A и B
for (i=0; i<N; i++)
    for (j=0; j<L; j++)
        //цикл для умножения i-й строки матрицы A на j-й столбец матрицы B
for (c[i][j]=0, k=0; k<M; k++)
    c[i][j]+=a[i][k]*b[k][j];
//вывод матрицы C
cout<<"Матрица C "<<endl;
for (i=0; i<N; cout<<endl, i++)
    for (j=0; j<L; j++)
        cout<<c[i][j]<<"\t";
for (i=0; i<N; i++)
    //освобождение памяти для каждого элемента a[i]

//освобождение памяти для a
d
€
B//освобождение памяти для b
delete [] b;
f
ö
M//освобождение памяти c
delete []c;
//освобождение памяти для каждого элемента b[i]
return 0; }
N
i
3.6. Тип данных «ссылки»
//освобождение памяти для каждого элемента c[i]

```

В C++ введен ссылочный тип данных.

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

быть изменена. Так же, как и указатель, ссылка является типизированной. Синтаксис определения ссылки:

тип& идентификатор = инициализатор;

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

Пример:

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

Ссылка может быть определена и для динамического объекта:

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

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

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

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

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

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

Пример:

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

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

Механизм перегрузки функций

Под перегрузкой функции понимается, определение нескольких функций (две или больше) с одинаковым именем, но различными параметрами. Наборы параметров перегруженных функций могут отличаться порядком следования, количеством, типом. Таким образом перегрузка функций нужна для того, чтобы избежать дублирования имён функций, выполняющих сходные действия, но с различной программной логикой. Например, рассмотрим функцию `areaRectangle()`, которая вычисляет площадь прямоугольника.

Функция, вычисляющая площадь прямоугольника с двумя параметрами `a(см)` и `b(см)`.

Это функция с двумя параметрами типа `float`, причём аргументы, передаваемые в функцию, должны быть в сантиметрах, возвращаемое значение типа `float` — тоже в сантиметрах.

Например, исходные данные (стороны прямоугольника) заданы в метрах и сантиметрах: `a = 2м 35 см`; `b = 1м 86 см`. В таком случае, удобно было бы использовать функцию с четырьмя параметрами. То есть, каждая длинна сторон прямоугольника передаётся в функцию по двум параметрам: метры и сантиметры.

```
float areaRectangle(float a_m, float a_sm, float b_m, float b_sm)
{
    return (a_m * 100 + a_sm) * (b_m * 100 + b_sm);
}
```

В теле функции значения, которые передавались в метрах (`a_m` и `b_m`) переводятся в сантиметры и суммируются с значениями `a_sm` `b_sm`, после чего перемножаем суммы и получаем площадь прямоугольника в см.

Т.е. в данном случае имеется уже две функции с разной сигнатурой, но одинаковыми именами (перегруженные функции). Сигнатура — это комбинация имени функции с её параметрами. Вызов перегруженных функций ничем не отличается от вызова обычных функций:

```
// будет вызвана функция, вычисляющая площадь прямоугольника с двумя
параметрами a(см) и b(см)
```

// будет вызвана функция, вычисляющая площадь прямоугольника с 4-мя параметрами a(м) a(см); b(м) b(см)

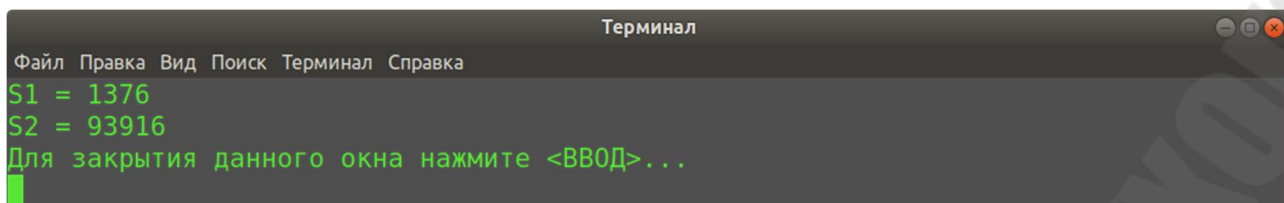
Компилятор самостоятельно выберет нужную функцию, анализируя только лишь сигнатуры перегруженных функций. Минус перегрузки функций, можно было бы просто объявить функцию с другим именем, и она бы хорошо справлялась со своей задачей. Однако, когда подобных функций становится много, то их использовать становится крайне неудобно. Пример программы с использованием данных функций:

```
// прототипы перегруженных функций
float areaRectangle(float a, float b);
float areaRectangle(float a_m, float a_sm, float b_m, float b_sm);
int main()
{
    cout << "S1 = " << areaRectangle(32,43) << endl; // вызов перегруженной
    функции 1
    cout << "S2 = " << areaRectangle(4, 43, 2, 12) << endl; // вызов
    перегруженной функции 2
    return 0;
}

float areaRectangle(float a, float b)
{
    return a * b;
}

float areaRectangle(float a_m, float a_sm, float b_m, float b_sm) /
{
    return (a_m * 100 + a_sm) * (b_m * 100 + b_sm);
}
```

Результат выполнения программы приведен на рисунке 3.3.

A screenshot of a terminal window titled "Терминал". The window has a menu bar with "Файл", "Правка", "Вид", "Поиск", "Терминал", and "Справка". The terminal content shows the following text:

```
S1 = 1376  
S2 = 93916  
Для закрытия данного окна нажмите <ВВОД>...
```

A green cursor is visible at the end of the last line.

Рис. 3.3. Вывод результатов работы программы.

4. Основные принципы объектно-ориентированного программирования

4.1. Основной элемент ООП – класс. Основные компоненты, объекты

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

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

Таким образом, можно выделить две основные сущности процесса обработки информации: код, как совокупность инструкций, и данные.

Все программы в соответствии с выбранной технологией программирования концептуально организованы вокруг своего кода или вокруг своих данных. Рассмотрим основные на сегодняшний день парадигмы программирования:

- 1) Процессно-ориентированная парадигма, при которой программа представляет собой ряд последовательно выполняемых операций – модель фон-Неймана. При этом код воздействует на данные. Языки, реализующие эту парадигму, называются процедурными или императивными. Такими языками являются, например, С, Pascal и др.
- 2) Объектно-ориентированная парадигма, при которой программа рассматривается как совокупность фрагментов кода, обрабатывающих отдельные совокупности данных – объекты. Эти объекты взаимодействуют друг с другом посредством так называемых интерфейсов. При этом данные управляют доступом к коду.

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

4.2. Основные принципы ООП

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

При создании программного комплекса необходимо разработать определенные абстракции.

Пример: Задача составления расписания занятий.

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

Операции:

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

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

- Инкапсуляция;
- Наследование;
- Полиморфизм.

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

Доступ к коду и данным жестко контролируется интерфейсом.

Основой инкапсуляции при ООП является класс.

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

2) Наследование – механизм, с помощью которого один объект (производного класса) приобретает свойства другого объекта (родительского, базового класса). При использовании наследования новый объект не обязательно описывать, начиная с нуля, что существенно упрощает работу программиста. Наследование позволяет какому-либо

объекту наследовать от своего родителя общие атрибуты, а для себя определять только те характеристики, которые делают его уникальным внутри класса.

Наследование есть очень важное понятие, поддерживающее концепцию иерархической классификации.

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

Общая концепция полиморфизма: один интерфейс – много методов.

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

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

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

4.3. Абстрактные типы данных

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

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

4.4. Классы и объекты С++

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

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

4.5. Синтаксис описания класса

Имя_класса { определение_членов_класса };

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

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

4.6. Управление доступом к членам класса

Принцип инкапсуляции обеспечивается вводом в класс областей доступа:

(закрытый, доступный только собственным методам);

(открытый, доступный любым функциям);

(защищенный, доступный только собственным методам и методам производных классов).

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

Имя_класса {

определение_закрытых_членов_класса

определение `_открытых_членов_класса`

определение `_защищенных_членов_класса`

...

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

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

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

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

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

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

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

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


...

Объекты класса можно определять совместно с описанием класса:

```
class Y {...} y1, y2;
```

4.7. Объявление и определение методов класса. Спецификатор

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

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

Пример:

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

int x::func1(){ ... return ia1; }
```

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

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

Такие функции называются встроенными.

Спецификатор `inline` указывает компилятору, что необходимо по возможности генерировать в точке вызова код функции, а не команды вызова

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

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

5. Основной элемент ООП – класс. Основные компоненты, объекты

5.1. Конструкторы и деструкторы

Конструкторы и деструкторы являются специальными методами класса

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

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

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

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

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

Отличия и особенности описания конструктора от обычной функции:

Имя конструктора совпадает с именем класса

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

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

Конструкторы можно классифицировать разными способами:

по наличию параметров:

- без параметров,
- с параметрами;

2) по количеству и типу параметров:

- конструктор умолчания,
- конструктор преобразования,
- конструктор копирования,
- конструктор с двумя и более параметрами.

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

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

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

Имя деструктора: `~имя_класса`

У деструкторов нет параметров и возвращаемого значения.

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

Пример: Описание класса.

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

```

    }
    box(int s){
        len = wid = hei = s;
    }
    box(){ len = 2; wid = hei = 1;
}

int volume(){
    return len * wid * hei;
}

```

Конструктор умолчания

Конструктор без параметров называется конструктором умолчания.

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

```
X{ ... }; X x1;
```

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

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

Конструктор преобразования и конструкторы с двумя и более параметрами

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

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

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

- X(T);
- X(T&);
- X
(

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

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

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

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

Пример:

```
class X { int  
x1; public:
```

Для такого класса будут верны следующие объявления объектов:

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

5.4. Конструктор копирования

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

Для такого создания объекта используется конструктор копирования.

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

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

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

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

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

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

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

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

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

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

Замечания по работе конструктора копирования:

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

к закрытым членам не только данного объекта, но и к закрытым членам любого объекта данного типа.

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

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

- `X(X&);`
- `X(const X&).`

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

Пример:

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

Спецификатор `explicit`

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

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

оно эквивалентно следующему:

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

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

В этом случае при объявлении переменной:

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

Конструктор копирования и операция присваивания

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

Пример:

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

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

Автоматическая генерация конструкторов и деструкторов

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

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

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

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

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

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

Указатель “this”

Ключевое слово `this` представляет собой неявно определенный указатель на сам объект. С его помощью метод определяет, с данными какого объекта ему предстоит работать. Каждый метод класса неявно содержит в качестве член-данного указатель:

```
ИмяКласса *this;
```

При вызове метода ему передается неявный аргумент, содержащий адрес объекта, для которого этот метод вызывается.

```

class example {
    int m;
public:
    int readm() { return m; } // return this->m
};
void f() {
    example aa, bb;
    int a = aa.readm(); // this указывает на aa
    int b = bb.readm(); // this указывает на bb
}

```

В первом случае функции readm() неявно передается указатель на объект aa, а во втором случае – bb.

Использование this необходимо в функциях, которые непосредственно работают с указателем на объект:

- this – указатель на объект (адрес объекта);
- *this – разыменованный указатель (сам объект).

Внутри тела класса могут использоваться ссылки или указатели на тот же класс. Это позволяет строить рекурсивные структуры класса.

Указатель this может быть использован только для нестатического метода.

```

#include <iostream>
using namespace std;
class example {
private:
    char c1, c2;
public:
    void init(char b) {c1=b; c2=b+1;}
    example &increment() { c1++; c2++; return(*this);}
    example *where() {return(this);}
    void print() {cout << c1 << » « << c2 << endl;}
};
int main() {
    example a, b;
    a.init('A'); // a.c1='A', a.c2='B'
    b.init('B'); // b.c1='B', b.c2='C'
    a.print(); // A B
}

```

```
cout << «&a=» << a.where() << endl; // &a=#####  
a.increment().print(); // B C  
b.increment().print(); // C D
```

Библиотека ГГТУ им. П.О.Сухого

6. Методы класса, способы применения, особенности

6.1. Дружественные (friend) функции класса

Принцип инкапсуляции и ограничения доступа к данным запрещает функциям, не являющимся методами соответствующего класса, доступ к скрытым (private) или защищенным данным объекта. Политика этих принципов такова, что, если функция не является членом объекта, она не может пользоваться определенным рядом данных. Тем не менее, есть ситуации, когда такая жесткая дискриминация приводит к значительным неудобствам.

Допустим, что необходимо, чтобы функция работала с объектами двух разных классов. Например, функция будет рассматривать объекты двух классов как аргументы и обрабатывать их скрытые данные. В такой ситуации спасет лишь friend-функция.

```
class beta; //нужно для объявления frifunc
class alpha
{
private:
    int data;

    alpha() : data(3) { } //конструктор без
    //аргументов
    friend int frifunc(alpha, beta); //дружественная
    //функция
};
////////////////////////////////////
class beta
{
private:
    int data;

    beta() : data(7) { } //конструктор без
    //аргументов
    friend int frifunc(alpha, beta); //дружественная
    //функция
};
```

```

int frifunc(alpha a, beta b) //определение функции
{
    return( a.data + b.data );
}

int main()
{
    alpha aa;
    beta bb;
    cout << frifunc(aa, bb) << endl; //вызов функции
}

```

В этой программе два класса — `alpha` и `beta`. Конструкторы этих классов задают их единственные элементы данных в виде фиксированных значений (3 и 7 соответственно). Необходимо, чтобы функция `frifunc()` имела доступ и к тем, и к другим скрытым данным, поэтому мы делаем ее дружественной функцией. Этой цели в объявлениях внутри каждого класса служит ключевое слово `friend`:

```
friend int frifunc(alpha, beta)
```

Это объявление может быть расположено где угодно внутри класса (нет разницы в `public` или `private`). Объект каждого класса передается как параметр функции `frifunc()`, и функция имеет доступ к скрытым данным обоих классов посредством этих аргументов. Кроме того, к классу нельзя обращаться до того, как он объявлен в программе.

Важное отличие дружественных функций состоит в том, что для них не используется указатель `this`. При объявлении дружественной функции-оператора должны передаваться два аргумента: для бинарных операций и один для унарных. Есть ситуации, в которых использование дружественных функций обязательно. Перегрузку операции умножения объекта класса на целое можно записать в виде функции-члена и в виде дружественной функции. В то время, как операцию умножения целое на объект класса можно определить только через дружественную функцию.

Методы могут быть превращены в дружественные функции одновременно с определением всего класса как дружественного.

```

#include <iostream>
using namespace std;

class alpha
{

```

```

private:
    int data1;
public:
    alpha() : data1(99) { } //конструктор
    friend class beta; //beta P дружественный класс

    { //все методы имеют доступ
public: //к скрытым данным alpha
    void func1(alpha a) { cout << "\ndata1=" << a.data1;}
    void func2(alpha a) { cout << "\ndata1=" << a.data1;}
};

int main()
{
    alpha a;
    beta b;
    b.func1(a);
    b.func2(a);
}

```

.2. Статические методы и данные

Некоторые члены класса могут быть объявлены с модификатором класса памяти `static`. Это статические члены класса. Статические члены-данные класса являются общими для всех объектов данного класса. Изменив значение статического члена класса в одном объекте, мы получим изменившееся значение во всех других объектах класса.

Статические члены данные класса можно использовать для подсчета количества созданных объектов класса или существующих в данный момент объектов класса.

Методы класса также могут быть объявлены статическими. Статические методы класса не получают указатель `this`, соответственно, эти функции не могут обращаться к нестатическим членам класса. К статическим членам класса статические методы класса обращаются посредством операции точка или `->`. Статический метод класса не может быть виртуальным. К статическим методам класса можно обращаться, даже

если не создано ни одного объекта данного класса, нужно только использовать полное имя члена класса. Если метод func1() является статическим методом класса А, то ее можно вызвать: А::func1();

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

class gamma
{
private:
    s    //(только объявление)
    t    int id; //ID текущего объекта
    a
    t    gamma() //конструктор без аргументов
    i
    c        total++; //добавить объект
    i        id = total; //id равен текущему значению total
    n
    t
    t    ~gamma() //деструктор
    o
    t
    a        cout << "Удаление ID " << id << endl;
    l
    всего объектов класса
    s
    t        cout << "Всего: " << total << endl;
    a
    t
    i
    v        cout << "ID: " << id << endl;
    e
    v    }
    d.
    showid() // Нестатическая функция
    -----
    d
    int gamma::total = 0; // определение total
    showtotal() // статическая функция

int main()
```

```

g //без static обращение к методу только через объект
a gamma g2, g3;
m gamma::showtotal();
m g1.showid();
a g2.showid();
s
h
g cout << "конец программы\n";
w
p
r
o

```

Методы const, не изменяющие объекты класса

Const-метод — это обычный метод, который, помимо своих прямых обязательств, дает гарантию того, что он не изменит атрибуты объекта. Любая попытка нарушить эту гарантию будет пресекаться компилятором. Определяется такой метод наличием ключевого слова const в конце сигнатуры метода. Рассмотрим на примере.

Есть некоторый класс Foo, метод doSomething() которого присваивает переменной члену _x значение 5. Методы const, не изменяющие объекты класса

```

class Foo
{
public:
    Foo(): _x(0) {}
    void doSomething();

private:
    int _x;
};

void Foo::doSomething()
{

```

В данном классе метод doSomething изменяет значение переменной X. В таком случае класс отработает и никаких проблем не возникнет. А в

случае, если данный метод будет объявлен с модификатором `const`, дело обстоит по иному:

```
class Foo
{
public:
    Foo(): _x(0) {}
    void doSomething()const;

private:
    int _x;
};

void Foo::doSomething()const
{
```

То компилятор ругнется на строку, и напишет (в случае `g++`):
x
для const-метода изменение атрибутов запрещено
test.cpp: In member function 'void Foo::doSomething() const':
test.cpp:15:10: error: assignment of member 'Foo::_x' in read-only object

6.4. Спецификатор `mutable`

В C++ есть ключевое слово `mutable`, которое может ставиться перед атрибутами класса (естественно, кроме констант и статических атрибутов). Атрибут помеченный как `mutable` может изменяться из `const`-методов.

```
{
public:
    Foo(): _x(0) {}
    void doSomething()const;

private:
    mutable int _x;
};

void Foo::doSomething()const
```

```
{  
  _x = 5; // все в порядке! :)
```

Библиотека ГГТУ им. П.О.Суворова

7. Правила и принципы наследования. Производные и базовые классы

7.1. Правила наследования

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

Пример:

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

```
struct B: A {int z};
```

```
A a1;
```

```
B b1;
```

```
b1.x =1;
```

```
b1.y =2;
```

```
b1.z =3;
```

```
a1 = b1;
```

Объект типа B наследует свойства объекта типа A.

Таким образом, объект типа-наследника содержит внутри себя члены базового типа. Принцип наследования структур изображен на рисунке 1.7.

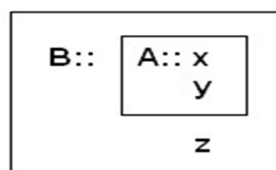


Рис. 1.7. Наследование структур

Наследование — процесс создания новых классов, называемых наследниками или производными, из уже существующих, базовых классов. Производный класс получает все возможности базового класса, но может быть усовершенствован. Основная форма наследования:

```
class имя_производного_класса : режим_доступа
    имя_базового_класса
```

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

Не наследуются:

- Конструкторы;
- Деструктор;
- Операция присваивания.

Управление уровнем доступа к элементам класса

Наследование свойств и поведения могут контролироваться с помощью квалификаторов доступа, задаваемых при наследовании: `public`, `protected`, `private`. Названия квалификаторов доступа совпадают с названиями квалификаторов доступа к методам и членам класса. Квалификаторы доступа ограничивают видимость полностью или частично для полностью или частично открытых членов-данных и методов. Закрытые члены-данных и методы всегда остаются закрытыми. При наследовании можно уменьшить видимость, но не расширить.

Кратко вид доступа в типе-наследнике для членов базового типа представлен в таблице 7.1.

Таблица 7.1. Области видимости при наследовании

Область доступа базового типа	Квалификатор доступа			

Данная таблица показывает вид доступа для членов-данных и методов в типе наследнике для типа наследника следующего уровня. Закрытый вид доступа в типе-наследнике для закрытых членов базового типа имеет особый статус.

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

Пример:

```
struct A { int x; };
class C: A {};

i
n
t
m
a
i
n
    c.x = 1; // ошибка: в классе C из-за закрытого по
            // умолчанию наследования поле x
            // становится закрытым.
```

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

Пример:

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

c.x = 1; // ошибки нет, т.к. наследование – открытое.
```

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

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

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

Пример:

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

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

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

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

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

```
struct A {int x; int y;};
struct B: A {int z;};
```



```

class C: protected A {int z;};

int main(){
A a;
A* pa;
B b;
C c;
C* pc = &c;
b.x = 1;
pc -> z; // ошибка: доступ к закрытому полю

pc -> x; // ошибка: доступ к закрытому полю
pa = (A*)pc; // см. примечание далее.

pa -> x=4; // правильно: поле A::x - открытое

```

Так как в данном примере наследование – защищенное, то при присвоении указателя производного типа указателю базового типа требуется явное преобразование ($pa=(A*)pc$). При открытом наследовании возможно простое присвоение указателей ($pa=pc$). Это связано с тем, что указатель кроме адреса содержит информацию об объекте. При защищенном наследовании изменяется не только состав членов класса, но и права доступа.

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

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

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

Таким образом, приведенная таблица показывает вид доступа для членов в типе наследнике для типа наследника следующего уровня. Но, для

текущего типа-наследника доступность зависит от вида доступа в базовом типе.

Пример:

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

class Y1: protected X1 {
    ...
};

class Z1: public Y1 {
    ...
};

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

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

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

подпрограммы, только для функций, унаследованных из класса Y2, так как в классе Y2 она находится в закрытой области.

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

Пример:

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

```
#include <iostream>
using namespace std;
class twom {
    double x,y;
public:
    twom (double x1=1, double y1=1): x(x1), y(y1) {}
    double sq(){ return x*y; }
};

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

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

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

7.3. Множественное наследование. Конструктор во множественном наследовании

Класс может быть производным не только от одного базового класса, а от многих. Этот случай называется множественным наследованием. Форма наследования в этом случае следующая:

```
class имя_производного_класса: список базовых классов {
```

Список базовых классов содержит перечисленные через запятую базовые классы с соответствующими режимами доступа к каждому из базовых классов (рисунок 1).

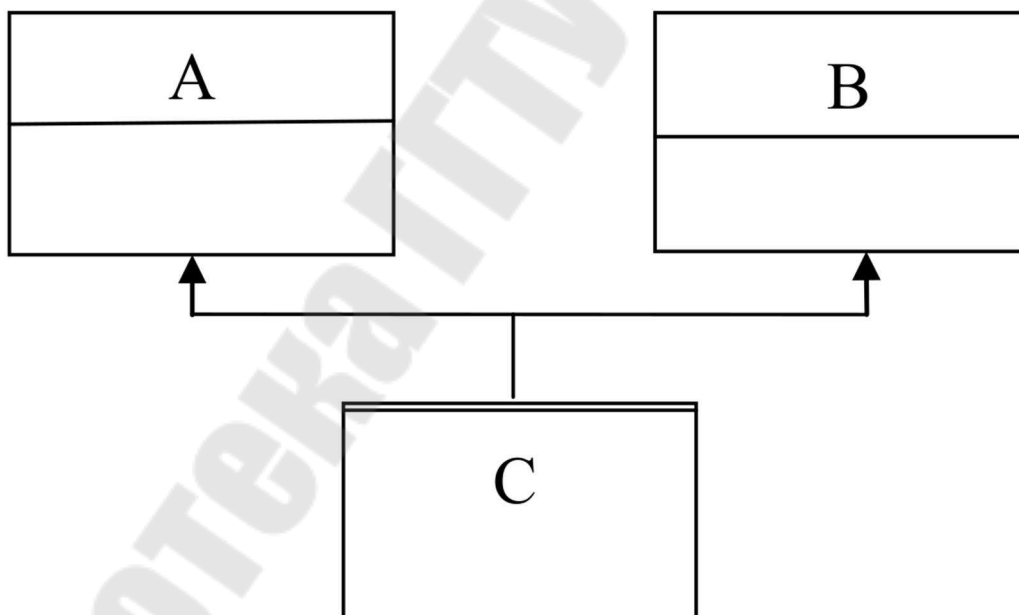


Рисунок 7.2. Диаграмма классов при множественном наследовании.

Пока конструкторы базовых классов не имеют аргументов, то производный класс может не иметь функцию-конструктор. Если же конструктор базового класса имеет один или несколько аргументов, каждый производный класс обязан иметь конструктор. Чтобы передать аргументы в

базовый класс, нужно определить их после объявления конструктора базового класса:

```
конструктор_производного_класса(список аргументов):  
базовый класс1(список аргументов){}
```

```
базовый классN(список аргументов){}
```

Здесь базовый класс1...базовый классN — это имена конструкторов базовых классов, которые наследуются производным классом. Двоеточие отделяет имя конструктора производного класса от списка аргументов базового класса. Список аргументов, ассоциированный с базовым классом, может состоять из констант, глобальных параметров. Так как объект инициализируется во время выполнения программы, можно в качестве параметров использовать переменные.

В определенных ситуациях могут возникнуть некоторые проблемы, связанные со множественным наследованием. Допустим, что в обоих классах существуют методы с одинаковыми именами, а в производном классе метода с таким именем нет. Проблема решается путем использования оператора разрешения, определяющего класс, в котором находится метод. Процесс направления к версии метода конкретного класса называется устранением неоднозначности.

Пример:

```
#include <iostream>  
using namespace std;  
////////////////////////////////////  
class A  
{  
public:  
void show ( ) { cout << "Класс A\n"; }  
};  
class B  
{  
public:  
void show ( ) { cout << "Класс B\n"; }  
};  
class C : public A, public B  
{
```

```

};
////////////////////////////////////
int main ( )
{
C objC; // объект
// objC.show ( );
не скомпилируется
objC.A::show ( );
objC.B::show ( );

```

Другой вид неопределенности возникает, если создается производный класс от двух базовых классов, которые, в свою очередь, являются производными одного класса. Это создает дерево наследования в форме ромба.

Пример:

```

using namespace std;
////////////////////////////////////
class A
{
public:
void func ( ){cout << "A";};
};
class B : public A
{
};
class C : public A
{
};
class D : public B, public C

objD.func ( ); // неоднозначность: программа не ском-
пилируется

```

```
objD.B::func ( ); //так можно
```

Классы В и С являются производными класса А, а класс D — производный классов В и С. Трудности начинаются, когда объект класса D пытается воспользоваться методом класса А. В этом примере объект D использует метод func(). Однако классы В и С содержат копии метода func(), унаследованные от класса А. Компилятор не может решить, какой из методов использовать, и сообщает об ошибке.

7.4. Виртуальный базовый класс

При множественно наследовании базовый класс не может быть задан в производном классе более одного раза.

В то же время базовый класс может быть передан производному классу более одного раза косвенно:

```
class X: public Base {...};  
class Y: public Base {...};  
class Derive: public X, public Y {...};
```

Каждый объект класса Derive будет иметь два подобъекта класса Base. Чтобы избежать неоднозначности при обращении к членам базового объекта Base, можно объявить этот класс виртуальным. Для этого используется ключевое слово virtual.

```
class Y: virtual public Base {...};  
class Derive: public X, public Y {...};
```

7.5. Последовательность создания и уничтожения подобъектов

При наследовании в родительский класс не вносятся никаких изменений, в следствии чего при создании объекта на основе класса наследника создается и объект на основе класса родителя.

При создании объекта, сначала создается объект на основе родительского класса (отрабатывает конструктор родительского класса), а затем создается дочерний. При удалении все происходит наоборот, сначала удаляется дочерний, а только потом родительский.

Пример.

```
#include <iostream>

using namespace std;

class parent
{
public:
    parent()
    {
        cout<<"Create parent"<<endl;
    }

    ~parent()
    {
        cout<<"Destroy parent"<<endl;
    }
};

class child:parent
{
public:
    child() : parent()
    {
        cout<<"Create child"<<endl;
    }

    ~child()
    {
        cout<<"Destroy child"<<endl;
    }
};

int main()
{
    cout << "Create object" << endl;
    child *ptrChild = new child();
}
```



```
cout << "Destroy object" << endl;  
delete ptrChild;
```

В данном примере описано два класса: parent и child. Класс child является наследником класса parent (квалификатор видимости в данном примере не имеет значения). В конструктор дочернего класса жестко связан с конструктором родительского child() : parent(). В данном примере это не критично, т. к. в обоих классах реализовано по одному конструктору. Однако, если конструкторов более одного, бывает необходимо производить их жесткое связывание чтобы дочерний класс отработывал как положено.

Результат вывода на экран при выполнении программы:

```
Create object  
Create parent  
Create child  
Destroy object  
Destroy child  
Destroy parent
```

8. Виртуальные функции. Позднее связывание

8.1. Виртуальные функции

Полиморфизм времени исполнения обеспечивается за счет использования производных классов и виртуальных функций. Виртуальная функция — это функция, объявленная с ключевым словом `virtual` в базовом классе и переопределенная в одном или в нескольких производных классах. Виртуальные функции являются особыми функциями, потому что при вызове объекта производного класса с помощью указателя или ссылки на него C++ определяет во время исполнения программы, какую функцию вызвать, основываясь на типе объекта. Для разных объектов вызываются разные версии одной и той же виртуальной функции. Класс, содержащий одну или более виртуальных функций, называется полиморфным классом.

Виртуальная функция объявляется в базовом классе с использованием ключевого слова `virtual`. Когда же она переопределяется в производном классе, повторять ключевое слово `virtual` нет необходимости, хотя и в случае его повторного использования ошибки не возникнет.

Пример.

```
#include <iostream>

using namespace std;

class Base {
public:
    virtual void who() {
        // определение виртуальной функции
        cout << "Class Base" << endl; }
};

class first_d: public Base {

    // определение who() применительно к first_d
    cout << "Class first_d" << endl; }
};

class second_d: public Base {
```

```

        // определение who() применительно к second_d
        cout << "Class second_d" << endl; }
};

```

```

int main() {
    Base *p;
    Base base_obj;
    first_d first_obj;
    second_d second_obj;
    p = &base_obj;

    p = &first_obj;
    w
    h    p = &second_obj;
    o    p->who(); // доступ к who класса second_d
    return 0;
}

```

В объекте Base функция who() объявлена как виртуальная. Это означает, что эта функция может быть переопределена в производных классах. В каждом из классов first_d и second_d функция who() переопределена. В функции main() определены три переменные. Первой является объект base_obj, имеющий тип Base. После этого объявлен указатель p на класс Base, затем объекты first_obj и second_obj, относящиеся к двум производным классам. Далее указателю p присвоен адрес объекта base_obj и вызвана функция who(). Поскольку эта функция объявлена как виртуальная, то C++ определяет на этапе исполнения, какую из версий функции who() употребить, в зависимости от того, на какой объект указывает указатель p. В данном случае им является объект типа Base, поэтому исполняется версия функции who(), объявленная в классе Base. Затем указателю p присвоен адрес объекта first_obj. (Как известно, указатель на базовый класс может быть использован для любого производного класса.) После того, как функция who() была вызвана, C++ снова анализирует тип объекта, на который указывает p, для того, чтобы определить версию функции who(), которую необходимо вызвать. Поскольку p указывает на объект типа first_d, то используется соответствующая версия функции who(). Аналогично, когда указателю p присвоен адрес объекта second_obj, то используется версия функции who(), объявленная в классе second_d.

Рассмотрим еще один более наглядный пример:

```

#include <iostream>

using namespace std;

class classVirtual
{
public:
    classVirtual() { }
    void f1(void) {
        cout<<"parent f1"<<endl; }

    virtual void f2(void) {
        cout<<"parent f2"<<endl; }
};

class classVirtualChild : public classVirtual
{
public:
    classVirtualChild() { }
    void f1(void) {
        cout<<"child f1"<<endl; }

    void f2(void) {
        cout<<"child f2"<<endl; }
};

int main(int argc, char *argv[])
{
    cout << "Start programm" << endl;

    classVirtual *virt = new classVirtual();
    virt->f1();
    virt->f2();

    cout<<"-----"<<endl;

    classVirtualChild *virt2 = new classVirtualChild();
    virt2->f1();

```

```

virt2->f2();

cout<<"-----"<<endl;

delete virt;
virt = new classVirtualChild();
virt->f1();

```

В данном примере реализовано два класса: родительский класс `classVirtual` и дочерний класс `classVirtualChild`. В обоих классах есть два метода `f1()` и `f2()`. В данных методах реализован вывод на экран соответствующей надписи, которая указывает какой метод был вызван и из какого класса. В родительском классе метод `f1(void)` является обычным, а метод `f2(void)` является виртуальным.

Результат выполнения программы:

```

-----
child f1
child f2
-----
parent f1

```

Наиболее распространенным способом вызова виртуальной функции служит использование параметра функции.

Пример.

```

#include <iostream>
using namespace std;

class Base {
public:
    virtual void who() {
        // определение виртуальной функции
        cout << "Base" << endl;
    }
}

```

```

};

class first_d: public Base {

    // определение who() применительно к first_d
    cout << "First derivation" << endl;
    }
};

class second_d: public Base {

    // определение who() применительно к second_d

// использование в качестве параметра ссылки на базовый класс
void show_who (Base &r) {
    r.who();
}

int main()
{
    Base base_obj;
    first_d first_obj;
    second_d second_obj;
    show_who (base_obj) ; // доступ к who класса Base
    show_who(first_obj); // доступ к who класса first_d
    show_who(second_obj); // доступ к who класса second_d
}

```

В данном примере функция `show_who()` имеет параметр типа ссылки на класс `Base`. В функции `main()` вызов виртуальной функции осуществляется с использованием объектов типа `Base`, `first_d` и `second_d`. Вызываемая версия функции `who()` в функции `show_who()` определяется типом объекта, на который ссылается параметр при вызове функции.

Ключевым моментом в использовании виртуальной функции для обеспечения полиморфизма времени исполнения служит то, что используется указатель именно на базовый класс. Полиморфизм времени исполнения достигается только при вызове виртуальной функции с использованием указателя или ссылки на базовый класс. Однако, ничто не мешает вызывать виртуальные функции, как и любые другие «нормальные» функции, однако достичь полиморфизма времени исполнения на этом пути не удастся.

На первый взгляд переопределение виртуальной функции в производном классе выглядит как специальная форма перегрузки функции. Но это не так, и термин перегрузка функции не применим к переопределению виртуальной функции, поскольку между ними имеются существенные различия. Во-первых, функция должна соответствовать прототипу. Как известно, при перегрузке обычной функции число и тип параметров должны быть различными. Однако при переопределении виртуальной функции интерфейс функции должен в точности соответствовать прототипу. Если же такого соответствия нет, то такая функция просто рассматривается как перегруженная и она утрачивает свои виртуальные свойства. Кроме того, если отличается только тип возвращаемого значения, то выдается сообщение об ошибке. (Функции, отличающиеся только типом возвращаемого значения, порождают неопределенность.) Другим ограничением является то, что виртуальная функция должна быть членом, а не дружественной функцией класса, для которого она определена. Тем не менее виртуальная функция может быть другом другого класса.

Если функция была объявлена как виртуальная, то она и остается таковой вне зависимости от количества уровней в иерархии классов, через которые она прошла. Например, если класс `second_d` получен из класса `first_d`, а не из класса `Base`, то функция `who()` останется виртуальной и будет вызываться корректная ее версия, как показано в следующем примере:

```
// порождение от first_d, а не от Base
class second_d: public first_d {
```

определение `who()` применительно к `second_d`

Если в производном классе виртуальная функция не переопределяется, то тогда используется ее версия из базового класса. Например, запустим следующую версию предыдущей программы:

```
class Base {
public:
    virtual void who() {
        cout << "Base" << endl;
    }
};
```

```
class first_d: public Base {
public:
    void who() {
        cout << "First derivation" << endl;
    }
};
```

```
class second_d: public Base {
// who() не определяется
```

```
Base base_obj;
```

```
Base *p;
```

```
first_d first_obj;
```

```
second_d second_obj;
```

```
p = &base_obj;
```

```
p = &first_obj;
```

```
p->who(); /* доступ к who() класса Base, поскольку second_d не переопределяет */
```

```
h
```

```
w
```

```
доступ к who класса Base
```

```
o
```

```
доступ к who класса first_d
```


Эта программа выдаст следующий результат:

Надо иметь в виду, что характеристики наследования носят иерархический характер. Чтобы проиллюстрировать это, предположим, что в предыдущем примере класс `second_d` порожден от класса `first_d` вместо класса `Base`. Когда функцию `who()` вызывают, используя указатель на объект типа `second_d` (в котором функция `who()` не определялась), то будет вызвана версия функции `who()`, объявленная в классе `first_d`, поскольку этот класс — ближайший к классу `second_d`. В общем случае, когда класс не переопределяет виртуальную функцию, C++ использует первое из определений, которое он находит, идя от потомков к предкам.

8.2. Виртуальные деструкторы

В C++ деструктор полиморфного базового класса должен объявляться виртуальным. Только так обеспечивается корректное разрушение объекта производного класса через указатель на соответствующий базовый класс.

Пример.

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

// Вспомогательный класс
Object() { cout << "Object::ctor()" << endl; }
~Object() { cout << "Object::dtor()" << endl; }
};

// Базовый класс
class Base {
public:
    Base() { cout << "Base::ctor()" << endl; }
```

```

    virtual ~Base() { cout << "Base::~dtor()" << endl; }
    virtual void print() = 0;
};

// Производный класс
class Derived: public Base {
public:
    Derived() { cout << "Derived::ctor()" << endl; }
    ~Derived() { cout << "Derived::~dtor()" << endl; }
    void print() {}
    Object obj;
};

int main ()
{
    Base * p = new Derived;
    delete p;
}

```

В функции main указателю на базовый класс присваивается адрес динамически создаваемого объекта производного класса Derived. Затем через этот указатель объект разрушается. При этом наличие виртуального деструктора базового класса обеспечивает вызовы деструкторов всех классов в ожидаемом порядке, а именно, в порядке, обратном вызовам конструкторов соответствующих классов.

Вывод данной программы:

```

Base::ctor()
Object::ctor()
Derived::ctor()
Derived::~dtor()
Object::~dtor()
Base::~dtor()

```

Уничтожение объекта производного класса через указатель на базовый класс с неvirtуальным деструктором дает неопределенный результат. На практике это выражается в том, что будет разрушена только часть объекта, соответствующая базовому классу. Если в коде выше убрать ключевое слово virtual перед деструктором базового класса, то вывод программы будет уже иным.

В данном случае член данных obj класса Derived также не разрушается:

```
Base::ctor()  
Object::ctor()  
Derived::ctor()
```

Если базовый класс предназначен для полиморфного использования, то его деструктор должен объявляться виртуальным. Для реализации механизма виртуальных функций каждый объект класса хранит указатель на таблицу виртуальных функций vptr, что увеличивает его общий размер. Обычно, при объявлении виртуального деструктора, такой класс уже имеет виртуальные функции, и увеличения размера соответствующего объекта не происходит.

Если же базовый класс не предназначен для полиморфного использования (не содержит виртуальных функций), то его деструктор не должен объявляться виртуальным.

9. Переопределение операций

9.1. Перегрузка операций

Примером полиморфизма в языке C++ является перегрузка (overload) операций. Она позволяет манипулировать объектами классов, используя обычный синтаксис языка C. Для обеспечения такой возможности в перегружаемых операциях должно сохраняться количество аргументов соответствующей операции. Например, операция деления, обозначаемая "/", в C имеет два аргумента, поэтому перегружаемая одноименная операция "/" также должна принимать два аргумента. Как правило, в качестве аргументов перегружаемых операций и возвращаемых ими значений выступают собственные классы. Так, если перегрузить операцию сложения для класса complex, то можно использовать естественный синтаксис языка C для их сложения:

```
class complex
{
double re, im;
public:
.....
}
.....
complex a, b, c;
```

Все перегружаемые операции имеют тот же приоритет и правила ассоциативности, что и предопределенные операции языка. По этой причине их можно использовать для построения цепочек операций, например:

Отметим, что в C++ нельзя перегрузить операции .:: .* и ?. Язык C++ не позволяет отличать постфиксную и префиксную формы перегруженных операций, поэтому, например, для перегруженной операции ++ выражения

++x и x++

имеют одно и то же значение. В общем случае предполагается, что перегруженные унарные операции используются в префиксной форме, поэтому для выражения "x++" компилятор выдаст предупреждение. Нельзя также определять новые лексические символы операций.

Язык C++ обеспечивает достаточную гибкость в перегрузке операций и их наследовании. Так, все перегруженные операции базовых классов, за

исключением операции присваивания, наследуются производным классом. Перегруженная операция базового класса может затем вновь перегружаться в производном классе и т.д. Операции перегружаются с помощью функций-операторов, имеющих следующий формат:

```
<тип> operator <символ> (<список_аргументов>) {...}
```

где тип - это тип возвращаемого операцией значения, а символ - символьное обозначение перегруженной операции.

Функции-операторы могут вызываться таким же образом, как и любая другая функция. Использование операции - это лишь сокращенная запись явного вызова функциооператора, например, для комплексных чисел сокращенная запись

эквивалентна вызову функции

В отношении действий, выполняемых перегружаемыми операциями, C++ не накладывает никаких ограничений. Можно, например, перегрузить операцию "+" для выполнения вычитания комплексных чисел, а "-" - для их сложения, но подобное использование механизма перегрузки операций не рекомендуется с точки зрения здравого смысла.

Сложные операции C++, равносильные комбинации нескольких операций, автоматически не раскрываются и, если пользователь их не определил, остаются компилятору неизвестными. Например, если для комплексных чисел перегружены операции сложения "+" и присваивания "=", то запись вида

совсем не означает, что будет выполняться

Чтобы можно было применять операцию "+=" с комплексными числами, она должна быть соответствующим образом перегружена в классе

Функции-операторы, являющиеся "друзьями" класса, всегда имеют число аргументов, равное числу аргументов перегружаемой операции. Для унарных операций - это один аргумент, для бинарных - два и т.д.

```
class complex
{
    double re, im;
public:
    complex (double r=0, double i=0)
        { re = r; im = i; }
```

```

    void print(char *s)
        {...};
}

friend complex operator + (complex, complex);
};

complex operator + (complex a, complex b)
{ return complex(a.re + b.re, a.im + b.im); }

```

или :

```

class complex {...}
complex operator + (complex a, complex b)
{
    complex c;
    c.re = a.re + b.re;
    c.im = a.im + b.im;
return c;
}

```

Теперь для сложения комплексных чисел можно использовать естественный синтаксис языка C++:

```

complex x(2.0,1.0), y(3.14,1.0), z;
z = x + y;

```

Если попытаться для класса `complex` аналогичным образом перегрузить унарную операцию инкремента следующим образом:

```

class complex
{
    .....
public:
    .....
friend complex operator ++(complex);
};
complex operator ++(complex a)
{ return complex(a.re++, a.im++); }

```

или :

```

class complex {...};

```

```

complex operator ++(complex a)
{
    a.re++,
    a.im++;
return a;

```

то можно убедиться, что перегруженная операция работать не будет. Дело в том, что в C++

аргументы передаются по значению и, следовательно, возвращаемая величина не изменяется своего значения. Можно использовать в качестве аргумента указатель:

```

complex operator + (complex *p)
{
    p->re++,

```

Однако в этом случае компилятор выдаст ошибку, поскольку язык C++ требует, чтобы операнд операции "++" имел тип класса объекта. Выходом из этой ситуации является использование ссылок в качестве аргумента функции-оператора:

```

class complex
{
    .....
public:
friend complex operator ++(complex &);
};
complex operator + (complex &r)
{
    r.re++,

```

Невозможность перегрузки унарных операций функциями "друзьями" послужило одной из причин введения ссылок в язык C++. Ссылки в качестве аргументов перегружаемых операций позволяют передавать реализующей ее функции не копии объектов, а их адреса, что позволяет изменять значения этих аргументов. Кроме того, использование ссылок в качестве аргументов для больших объектов, позволяет значительно сократить объем копируемой информации, что приводит к повышению эффективности программы.

Рассмотрим теперь вопрос каким должно быть возвращаемое значение функции оператора. Оно не может быть ссылкой на автоматическую переменную, так как после выхода из функции возвращаемое значение указывает на несуществующую переменную. Оно

также не может быть статической переменной. Допускается ссылка на переменную в "куче", однако это усложняет программирование. Лучшим решением, с точки зрения простоты реализации и эффективности, считается копирование возвращаемого значения. Можно также возвращать ссылку, но только на существующий объект, например, когда эта ссылка передается функции-оператору в качестве параметра:

```
class complex
{
    .....
public:
    .....
    friend complex& operator ++(complex& r);
};
complex& operator ++(complex& r)
{
    r.re++;
}
```

Для функций-операторов, являющихся компонентами класса, первым аргументом по умолчанию является указатель `this` на тот объект, к которому она относится. Поэтому число аргументов для таких операторов на единицу меньше, по сравнению с числом аргументов перегружаемой операции. Например, для бинарных операций - один аргумент, для унарных - аргументы отсутствуют и т.д. Перегрузка операций `+` и `++` для класса `complex`, реализованная с помощью операторовкомпонент имеет вид:

```
class complex
{
    .....
```



```

public:
    complex operator + (complex b);
    complex operator ++ (void);
};
complex complex:: operator + (complex b)
    { return complex(re + b.re, im + b.im); };
complex complex:: operator ++()
    { return complex(re++, im++); };

```

Отметим, что при перегрузке операций с помощью компонент функций широко используется указатель `this`. Однако при этом следует соблюдать особую осторожность. Соблазнительной может показаться идея использовать для получения результата вместо промежуточной переменной первый аргумент, например:

```

class complex
{
    .....
public:
    complex operator +(complex b);
    complex operator ++(void);
};
complex complex:: operator + (complex b)
{
    re += b.re;
    im += b.im;
return *this;
};
complex complex:: operator ++ ()
{

```

В последнем примере операция инкремента работает правильно, а операция сложения имеет побочный эффект, увеличивая значение первого операнда.

Напомним, что с помощью функций-операторов, являющихся компонентами класса, можно перегружать операции `=`, `()`, `[]` и `->`, которые запрещены для перегрузки функциями "друзьями".

Может возникнуть вопрос в каких случаях при перегрузке операций следует использовать функции-"друзья", а в каких - функции-компоненты? Рассмотрим пример:

```
main(void)
{
    complex a(2.0,1.0), b(3.14,1.0), c;
    double d = 1.5;
```

Выражение $c = a + b$ будет верным, поскольку a и b оба типа `complex`, для которого переопределена операция "+". Выражение $c = a + d$ также будет верным, поскольку оно интерпретируется как `a.operator+(complex(d))`. Однако выражение $c = d + b$ вызовет ошибку, если операция "+" перегружена с помощью функции-компоненты. Дело в том, что $d + b$ по определению эквивалентно `d.operator+(b)`, но d не является объектом класса и не имеет компонент! В случае же перегрузки "+" как функции-друга выражение верно, поскольку здесь будет вызван конструктор инициализации и $d + b$ будет эквивалентно `complex(d) + b`. Отсюда можно сделать следующий вывод: в том случае, если предполагается неявное преобразование операндов перегружаемой операции, то реализующую ее функцию лучше сделать "другом". К таким операциям можно отнести арифметические и логические операции (+, -, || и т.д.).

Если же операция изменяет состояние объекта, а это можно сделать только для ранее созданного объекта, то реализующая операцию функция должна быть компонентой класса. К подобным операциям относятся =, *=, ++ и др.

При прочих равных условиях для перегрузки операций чаще используются функции-компоненты. Это связано с тем, что они имеют на один аргумент меньше, допускают использование неявного параметра `this`, не чувствительны к модификациям функций преобразования типа и, наконец, описание функции-компоненты, как правило, короче описания функции-друга.

9.2. Многократная перегрузка операций

C++ позволяет несколько раз перегружать одну и ту же операцию в пределах одного класса. Важно только, чтобы однозначно определялся вызов необходимой операции, т.е. функции-операторы должны различаться количеством или типом своих аргументов, например:

```
class String
{
    char *str;
public:
    String (char * s = "\0")
    // конструктор
    { str = new char [strlen (s) +1]; strcpy (str,s); }
    String operator + (String t)
    // первая перзагрузка +
    { return String(strcat(str,t.str)); }
    String operator + (char *s)
    // вторая перзагрузка +
    { return String(strcat(str,s)); }
    void print(char* s)
    { cout << s << ": " << str << "\n"; }
};
main(void)
{
    String a("Минск"), b(" - город"), c;
    c = a + b + " -герой!";
    c.print("c");
}
```

В данном примере дважды перегружена операция + для класса String. В первом случае она позволяет нам связывать объекты типа String между собой, а во втором - тип String со строкой символов. Это дает возможность в одном выражении записывать как объекты типа String, так и обычные строки типа (char *).

9.3. Перегрузка операции присваивания =

Операция присваивания перегружается обычным образом. Особенностью этой операции является то, что она не может быть

унаследована производным классом. Кроме того, в C++ операцию "=" можно перегружать только с помощью функции-компоненты.

Например:

```
class complex
{
    .....
public:
    complex& operator =(complex &);
};
complex& complex:: operator =(complex& b)
{
    re = b.re;
    im = b.im;
```

Здесь первый операнд передается через указатель `this`, принимает значение второго операнда, полученного в качестве аргумента, и возвращается с помощью выражения `*this`.

Не смотря на то, что в этом примере изменяется значение первого аргумента, это не является ошибкой, поскольку его значение как раз и нужно изменить. Данный пример также демонстрирует те немногие случаи, когда в программе необходимо явно использовать указатель `this`.

Если для некоторого класса `X` операция присваивания не определена, то при необходимости она формируется компилятором C++ автоматически в виде:

```
X & X:: operator = (const X & <источник>)
```

```
// покомпонентное присваивание
```

Рассмотрим на примере класса `String` более сложный случай перегрузки операции "=", чем простое покомпонентное присваивание. Первый вариант перегрузки операции "=" для класса `String` на первый взгляд вообще не возвращает никакого значения

```
class String
{
    .....
public:
    void operator = (String &t){
```

```

if (this == &t)

// копирование в ту же строку

// удаляется старая строка
str = new char[strlen(t.str)+1]; // образуется новая

// в нее копируется значение

```

Здесь вначале проверяется, чтобы строка не дублировала саму себя. Затем уничтожается старое значение строки, отводится память необходимого объема и туда копируется принимаемый аргумент.

Более корректной реализацией перегрузки операции "=" будет явный возврат сформированного объекта с помощью указателя this:

```

class String
{
    .....
public:
    String& operator = (String &t)
    {
        if (this == &t)
            return *this;
        delete str;
        // удаляется старая строка
        str = new char[strlen(t.str)+1]; // образуется новая

        // в нее копируется значение

```

Если сравнить последнюю реализацию с конструктором класса String, то можно увидеть их значительное сходство. Поэтому можно воспользоваться уже готовым конструктором для формирования возвращаемого значения:

```

class String
{
    .....
public:

```

```
String operator = (String &t)
{
    if (this == &t) return *this;
    delete str;
    // удаляется старая строка
return String(t.str);
}
}
```

Заметим, что здесь изменился тип возвращаемого значения, поскольку конструктор `String::String()` возвращает объект, а не ссылку на него. Таким образом, для простых типов (например, `complex`) операцию присваивания можно не перегружать, в этом случае она формируется компилятором автоматически, осуществляя покомпонентное присваивание. Если объекты сложные и покомпонентное присваивание для них не работает, следует операцию присваивания перегружать явно. В качестве возвращаемого значения перегруженной операции присваивания чаще используется ссылка на объект. Возможно в возвращаемом выражении обращаться к конструктору класса. В последнем случае возвращаемым значением будет сам объект.

9.4. Перегрузка операций `[]`, `()` и `->`

Прежде всего отметим, что эти операции перегружаются только с помощью компонент-функций и их нельзя перегрузить функциями-"друзьями".

Начнем с рассмотрения перегрузки операции `[]`. Предположим, что мы хотим обращаться к действительной и мнимой частям некоторого комплексного числа `h` как к элементам массива, т.е. `h.re` соответствует `h[0]` и `h.im` соответствует `h[1]`. Для этого можно перегрузить операцию `[]` выделения элемента массива для класса `complex` следующим образом:

```
class complex
{
    double re, im;
public:
    .....
    double & operator [] (int i)
        { return *(&re + i);}
};
```

```

main(void)
{
    complex one(1.0, 0.0);
    cout << "one.re=" << one[0] << "one.im=" << one[1] << "\n";
}

```

Перегрузка операции [] для класса X позволяет всякое выражение вида ob[i], где ob - объект класса X, интерпретировать как обращение к функции ob.operator[](i). Операция вызова функции вида:

```
<имя>(<список_аргументов>);
```

в языке C++ рассматривается как бинарная операция, где первым операндом является <имя>, а вторым - <список_аргументов>. При перегрузке операции () список аргументов вычисляется и проверяется в соответствии с обычными правилами передачи аргументов. В общем случае список аргументов может быть пуст.

Перегрузка операции () позволяет рассматривать выражение вида ob(<список_аргументов>) как обращение к функции ob.operator()(<список_аргументов>).

Рассмотрим перегрузку операции () для класса Array. Иногда целесообразно считать, что индекс массива начинается не с нуля, а с единицы (подобно тому, как это реализовано в языке PL/1). Для этого перегрузим операцию () :

```

class Array
{
    .....
public:
    int operator()(int i) { return m[i-1]; }
};
void main()
{
    Array x(5);
    cout << "x= ";
    for (int i=1; i<=5; i++)
        cout << x(i) << " ";
}

```

Операцию () обычно перегружают для операций, требующих большого числа операндов, для классов с единственно возможной

операцией, а также когда некоторая операция используется особенно часто. Перегружаемая операция () не может быть статической компонентой-функцией класса.

C++ предоставляет пользователю возможность выполнять некоторую предварительную обработку до обращения к компонентам классов. С этой целью используется перегрузка операций -, * и &. Операция → рассматривается как унарная операция и ее перегрузка позволяет выражение вида

трактовать как

Причем функция-оператор, реализующая операцию -> должна либо возвращать указатель на объект данного класса, либо возвращать объект этого класса, т.е. ее описание для класса X должно иметь вид:

```
X *operator -> () {...}
```

или

```
X operator -> () {...}.
```

Рассмотрим случай, когда возвращаемым значением перегуженной операции -> является тип этого же класса:

```
class X
{
public:
    int a;
    X(int i) { a = i; }
    X operator ->()

    cout << "Доступ к компонентам класса X\n";
return *this;
}
};
main(void)
{
    X x(5);

"; // перегуженная операция << "\n"; // predefined operation
```

В данном примере перед обращением к компоненте в стандартный поток выводится сообщение. Наибольший интерес представляет случай,

когда возвращаемым значением перегруженной операции `->` класса `X` является указатель на некоторый другой класс `Y`. В этом случае операция `->` вначале применяется к своему левому операнду для получения указателя `p` на класс `Y`, а затем указатель `p` используется как левый операнд бинарной операции `->` для доступа к компоненте класса `Y`, например:

```
class Y
{
public:
    int b;
    Y(int j) { b = j; }
};
class X
{
    int a;
public:
    Y *p;
    X(int i, int j)
    {
        a = i;
        p = new Y(j);
    }
Y* operator ->()

    cout << "Доступ к компонентам класса Y\n";
return p;
}
};
main(void)
{
    X x(3,5);
    cout << "b= " << x->b << "\n"; // перегруженная операция
    cout << "b= " << x.p->b << "\n"; // базовая операция
}
```

Если класс `Y` в свою очередь имеет перегруженную операцию `->`, то `p` будет использован в качестве левого операнда унарной операции `->` и вся процедура повторится для класса `Y`.

Унарные операции `*` и `&` перегружаются аналогично, причем сохраняется соответствующая семантика между операцией `->` и операциями `*` и `&`.

Библиотека ГГТУ им. П.О.Сухого

10. Шаблоны функций, шаблоны классов

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

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

10.1. Параметры шаблона

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

В соответствии со Стандартом ISO 1998 C++ параметром шаблона может быть:

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

Интегральные типы:

- знаковые и беззнаковые целые типы;

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

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

. Отождествление типов аргументов

Так как компилятор генерирует экземпляры шаблонов функций согласно типам, заданным при их вызовах, то критическим моментом является передача корректных типов, особенно если шаблон функции имеет два или более параметров. Хорошим примером является классическая функция `maxE()`:

```
{  
    return a > b ? a : b;
```

Функция `maxE()` будет работать правильно, если оба ее аргумента имеют один и тот же тип:

```
int i = maxE (1, 2);  
double d = maxE (1.2, 3.4);
```

Однако, если аргументы различных типов, то вызов `max()` приведет к ошибке, так как компилятор не сможет понять, что ему делать.

Один из возможных способов для разрешения неоднозначности состоит в использовании приведения типов, чтобы прояснить наши намерения:

```
int i = maxE ((int)'a', 100);
```

Вторая возможность - это явно объявить версию экземпляра шаблона функции перед ее вызовом:

```
int maxE (int, int);  
int j = maxE ('a', 100);
```

Третий способ решить проблему состоит в создании шаблона функций, который имеет параметры различных типов.

```
template <class T1, class T2> T1 maxE (T1 a, T2 b) {  
return a > (T1)b ? a : (T1)b;
```

Использование этой новой версии `max()` не приведет к неоднозначности в случае использования двух различных типов.

Например, если написать

то компилятор будет использовать два заданных (посредством аргументов типа) и построит версию функции `max()` с заголовком

Далее компилятор, перед выполнением сравнения, приведет тип второго аргумента к типу первого аргумента. Такой способ допустим, однако использование двух типовых параметров в шаблоне функции, которая должна была бы работать только с одним типом, часто лишь затрудняет жизнь. Довольно тяжело помнить, что

дает значение типа `char`, в то время как

передает в вызывающую программу `int`.

Алгоритм поиска оптимально отождествляемой функции (с учетом шаблонов)

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

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

1. Для каждого шаблона, подходящего по набору формальных параметров, осуществляется формирование специализации, соответствующей списку фактических параметров.
2. Если могут быть два шаблона функции и один из них более специализирован, то на следующих этапах рассматривается только он (порядок специализаций описан далее).
3. Осуществляется поиск оптимально отождествляемой функции из полученного набора функций, включая определения обычных функций, подходящие по количеству параметров. При этом, если параметры некоторого шаблона функции были определены путем выведения по типам фактических параметров вызова функции, то при дальнейшем поиске оптимально отождествляемой функции к параметрам данной специализации шаблона нельзя применять никаких описанных выше преобразований, кроме преобразований точного отождествления
4. Если обычная функция и специализация подходят одинаково хорошо, то выбирается обычная функция.
5. Так же, как и при поиске оптимально отождествляемой функции для обычных функций, если полученное множество подходящих вариантов состоит из одной функции, то вызов разрешим. Если множество пусто или содержит более одной функции, то генерируется сообщение об ошибке.

Шаблоны классов

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

Объявление шаблона класса:

```
template <Список_параметров шаблона> class Имя класса { . . . };
```

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

```
Имя класса <Список_параметров шаблона> Идентификатор ;
```

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

Пример описания стека для хранения величин разных типов данных:

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

int main(){
    stack<int>
    S1( 20);
    stack<char>
```

S
s
t
d
o
u
b
l

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

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

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

Пример:

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

Статические члены создаются для каждого инстанцирования класса.

Пример:

```
template <class T> class X {
    static T x1;
```



```

// ...
};

int X <int> :: x1 = 0;
double X <double> :: x1 = 1.5;

int main() {
    X <int> xx1;
    X <double> xx2;
}

```

Ограничений на наследование нет — разрешаются все разновидности:
Наследование в шаблонах классов

- Шаблон класса наследует обычный класс;
- Обычный класс наследует шаблон класса;
- Шаблон класса наследует другой шаблон класса.

В разработке классов на базе шаблона, отличающемся от одиночного наследования, множественное наследование было бы более частым, чем многоуровневое, гибридное или виртуальное наследование.

Шаблонный класс — это шаблон для класса, который инстанцирован в зависимости от типа (типов) и других полученных шаблоном аргументов. Инстанциация сгенерирует шаблонный класс, или специализацию этого класса. Этот процесс известен как инстанциация, и выходной результат известен как специализация.

Когда происходит наследование, может быть два варианта: наследуется шаблон класса (Item< T >), или специализация (Item< int >).

Эти две модели появляются одинаково, однако полностью разные.

Пример:

```
class ItemExt : public Item< int >
```

Здесь обычный класс ItemExt наследуется из специализации шаблона (Item< int >), и не упрощает никакое другое инстанцирование Item.

Рассмотрим следующее: пустой класс ItemExt, сам по себе, может быть классифицирован как:

Так или иначе (через typedef или наследование) при использовании emExt не нужно указывать тип:

int_item всего лишь объект, наследуемый от Item< int >. Это означает, что нельзя создать объект другого нижележащего типа, используя наследованный класс ItemExt. Экземпляр ItemExt всегда будет Item< int >, даже если будут добавлены новые методы / члены в унаследованный класс. Новый класс может предоставить новые возможности наподобие печати значения, или сравнения с другими типами и т. д., но класс не позволит использовать гибкость шаблонов. Под этим подразумевается, что нельзя сделать следующее:

Поскольку ItemExt не шаблон класса, а просто обычный класс.

Другой тип наследования - это наследованием шаблона, где наследуется сам шаблон класса. Пример:

```
t
e
h   Класс SmartItem это другой шаблон класса, который наследуется из
шаблона Item. При инстанцировании SmartItem< > с некоторым типом T
тот же тип передается в шаблон класса Item. И все это происходит на этапе
компиляции. Например, если инстанцировать SmartItem с типом char, то
инстанцируются Item< char > и SmartItem< char >.
```

```
m   В качестве другого примера наследования шаблона, можно привести
наследование из шаблона класса Array:
```

```
u
p   template< size_t SIZE >
c   class IntArray : public Array< int, SIZE >
n   {
e
m
p
T
b
l
.
```

```
};
```

```
int main()  
{  
    IntArray< 20 > Arr;  
    Arr[0] = 10;
```

В качестве первого аргумента шаблона используется `int` и `SIZE`, в качестве второго аргумента шаблона на базе шаблона класса `Array`. Аргумент `SIZE` это только аргумент для `IntArray`, и второй аргумент для базового класса `Array`. Это допустимая, интересная особенность, упрощающая автоматическую генерацию кода с помощью компилятора. Однако `IntArray` был бы всегда массивом из `int`, но программист может указать размер массива.

Похожим образом можно наследовать из `Array` и так:

```
template< typename T >  
class Array64 : public Array< T, 64 >  
{  
};  
  
int main()  
{  
    Array64< float > Floats;
```

Другие возможности стандартных библиотек

Указатели на компоненты класса

Рассмотрим пример описания класса XXX:

```
class XXX
{
public:
    long x1;

    /* Данные-члены класса.*/
i
n
long getVal1() {return x1;} /*Функции-члены класса без параметров.*/
x
    char* getVal4(char *str) {return str;}
int getVal3(int param) {return x2*param;} /*Функции-члены класса с параметрами.*/

    static int f1(int n) {return n*10;}
    static int f3(int param) {return param;}
    /* Определение различных статических функций*/

X
X
X
    /*Конструктор.*/
```

Поскольку нестатические функции-члены формально, а нестатические данные-члены фактически не существуют без объекта-представителя класса, определение указателя на компонент класса (член класса или функцию-член) отличается от определения указателя на объект или обычную функцию.

Для объявления указателя на нестатическую функцию используется специальная синтаксическая конструкция, состоящая из спецификатора объявления и заключённого в скобки квалифицированного имени указателя, состоящего из имени класса, операции доступа к члену класса ::, разделителя * , собственно имени указателя, закрывающей скобки и списка параметров:

l
x
v
a
l
x
v
a

Подобный указатель может быть проинициализирован инициализатором, состоящим из операции присвоения, операции взятия адреса и квалифицированного имени соответствующей функции-члена:

```
int (XXX::*fp_3) (int) = &XXX::getVal1;
```

Как известно, значение унарного выражения, состоящего из операции взятия и операнда, который является именем функции и первичного выражения, состоящего из имени функции эквивалентны. Это адрес данной функции. Поэтому поэтому в качестве инициализирующего выражения для указателя на функцию-член класса также может быть использовано первичное выражение, представляющее собой квалифицированное имя функции-члена:

Класс - это не объект. И не совсем понятно, какое значение имеет адрес нестатического члена класса. Значение проинициализированного указателя на нестатическую компоненту остаётся неопределённым.

Оно определяется лишь в результате выполнения операций обращения к членам класса .* и ->* .

При этом функция-член класса вызывается по указателю на компоненту относительно конкретного объекта или указателя на объект-представитель класса. Первым операндом операций обращения к членам класса является l-выражение, ссылающееся на объект (возможно, что имя объекта) или указатель на объект, вторым операндом является ссылка на указатель на компоненту класса:

```
int val = (q.*fp)(6);  
char val = (pq->*fp4)("new string");
```

Аналогичным образом осуществляется объявление и инициализация указателей на данные-члены класса. При этом структура объявления указателя на член класса проще (нет спецификации возвращаемого значения, не нужно указывать список параметров). Это не функция, здесь дело обходится спецификацией объявления и квалифицированными именами указателей:

```
long (XXX::*px1) = &XXX::x1;
```

// Определение и инициализация указателя на член класса XXX типа

p - объект-представитель класса XXX.

// pq - указатель на объект-представитель класса XXX.

Основные приёмы работы с указателями на функции-члены демонстрируются на следующих примерах:

```
class XXX
{
public:
    long x1;

    /*Данные-члены класса.*/
i   long getVal1() {return x1;}
n   long getVal2() {return x2*x1;}
t   /*Функции-члены класса без параметров.*/
X   int  getVal3(int param) {return x2*param;}
    char* getVal4(char *str) {return str;}
    /*Функции-члены класса с параметрами.*/
static int f1() {return 100;}
    static int f2() {return 10;}
    static int f3(int param) {return param;}
    /* Определение различных статических функций*/

X   /*Конструктор.*/
X
X
l
o
n,2);/*XXX:определение(объекта)*/
g   pq->x1 = 100;
v
a   /*Определение и инициализация объекта по указателю.*/
l
i   /*Указатель на функцию-член класса.*/
n   long (XXX::*fp_1) () = &XXX::getVal1;
t
v
a
l}
[Проинициализированный указатель на функцию-член
X
v
a
l
```

класса. Его значение является относительной величиной и равняется значению смещения функции-члена относительно первого члена класса. */

Инициализация первого указателя. Один и тот же указатель можно настраивать на различные функции-члены класса. Главное, чтобы у всех этих функций-членов совпадали списки параметров и возвращаемые значения функций. */

/*Вызов функции-члена класса по указателю из объекта.*/

Вызов функции-члена класса по указателю с помощью указателя на объект. */

i к
Проинициализированный указатель на функцию-член
t а
X с
Вызов функции-члена класса по указателю из объекта с
X а
f . класса с параметрами типа int. */
Проинициализированный указатель на функцию-член
p C char val_4 = (pq->*fp4)("new string");
a
X
X
Вызов функции-члена класса по указателю с помощью
X П указателя на объект. */
X а
й р
е а
h Указатель на объект, указатель на функцию-член и указатель. */
у М спецификации класса. Явная спецификация класса
р С /*Вызов статической функции по указателю.*/
X р
X а
X м
г и
р т
b и
V п

требование — возвращаемого указателю.*/ класса класс XXX	Перенастройка статического указателя. Главное совпадение списков параметров и типа, значения. */ int (*fp_6) (int) = &XXX::f3; /*Указатель на статическую функцию с параметрами.*/ /*Вызов статической функции с параметрами по указателю.*/ long (XXX::*px1) = &XXX::x1; /*Определили и проинициализировали указатель на член long*/ /*Используя указатель на компоненту класса, изменили значение переменной x1 объекта q, представляющего XXX.*/ /*Используя указатель на компоненту класса, изменили значение переменной x1 объекта, представляющего класс и расположенного по адресу pq.*/
---	---

Вызов статических функций-членов класса не требует никаких объектов и указателей на объекты. От обычных функций их отличает лишь специфическая область видимости.

Понятие абстрактного класса. “Чистые” виртуальные функции

Практика объектно-ориентированного программирования показывает, что иногда при разработке сложных иерархий классов целесообразно описывать классы, которые выражают некоторую общую концепцию, описывают основной интерфейс для использования в производных классах. Такой класс включает определение данных и методов, которые будут общими для различных производных классов, и составляет некоторую базу, от которой наследуются другие классы иерархии. Смыслового определения виртуальной функции в подобном базовом классе может не быть, но без ее объявления корректная реализация сложной иерархии затруднена. Такие функции в языке C++ описывают как абстрактные виртуальные.

Абстрактной виртуальной называется функция, объявленная в базовом классе как виртуальная, но не содержащая описания выполняемых

действий. Для объявления абстрактной виртуальной функции используется следующая форма:

```
virtual <Тип><Имя_функции>(<Список параметров>) =0;
```

Здесь присваивание нулю – признак абстрактной виртуальной функции. При этом производный класс должен определить свою собственную версию функции, так как в базовом классе не существует версии, которую можно было бы использовать в производном.

Класс, который содержит, по крайней мере, одну абстрактную виртуальную функцию, принято называть абстрактным. Этот класс может использоваться только как базовый для создания других классов, причем если класс, производный от абстрактного, не содержит аспекта виртуальной функции, то он также является абстрактным.

Создание объектов абстрактного класса запрещено. Однако можно создавать указатели на объект абстрактного базового класса и ссылки того же типа и применять их для реализации механизма полиморфизма.

Пример использования абстрактного класса при работе с полиморфными объектами:

Пусть нам необходимо организовать массив, в котором хранятся указатели на объекты двух классов: целые числа и строки. По условию задачи объекты этого массива необходимо выводить на экран.

Для того, чтобы собрать в одном массиве указатели на объекты разных классов, необходимо, чтобы эти классы имели общего предка, т.е. нам потребуется организовать иерархию классов, по типу представленной на рисунке 11.1.

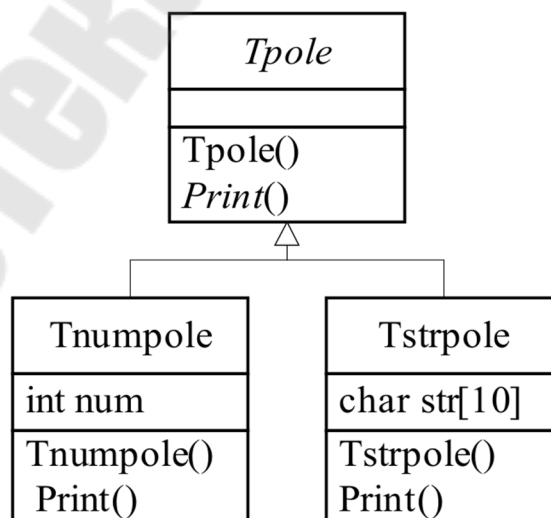


Рис. 11.1. Пример иерархии с абстрактным классом

Класс `TPole` – общий базовый, его существование позволит нам объявить массив указателей на объекты этого класса. Сами объекты создавать не будем: в процессе работы в этот массив будут помещены указатели на объекты класса `Число` или на объекты класса `Строка`.

Для вывода на экран содержимого хранимых объектов в программе должен быть организован цикл, в котором для каждого объекта вызывается метод `Print()`. При этом вызов метода происходит через указатель на объекты базового класса. Если бы такой метод в базовом классе отсутствовал, то для вызова этого метода объектами производных классов тип указателя пришлось бы явно переопределять. Это связано с тем, что указатель на объекты базового класса «не видит» полей и методов, появившихся в переопределенном классе. При наличии абстрактного метода `Print()` в базовом классе этой проблемы не возникает. Естественно в базовом классе метод `Print()` объявляется виртуальным абстрактным, поскольку в объектах базового класса на экран выводить нечего. Соответственно и класс `Tpole` получается абстрактным.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

class Tpole {
public:
    Tpole() {}
    virtual void Print(void)=0; // абстрактная функция
};

class Tnumpole:public Tpole { // класс Число
private:
    int num;
public:
    Tnumpole(int n):num(n) {}

    /* аспект виртуальной функции*/
    printf("Число = %5d\n",num);
};
```

```

    } };

class Tstrpole:public Tpole { // класс Строка
private:
    char str[10];
public :
    Tstrpole(char *st) { strcpy(str,st); }
    void Print(void) { printf("Строка = %s\n",str); } /* аспект
    виртуальной функции*/
};
void main()
{
    setlocale(0,"russian");
    Tpole *a[10];
    int n,i; char st[80];
    // массив указателей на объекты класса Tpole

    printf("\n Введите целое число или строку: ");
    scanf_s("%s",st,80);
    if ((n=atoi(st))!=0||(strlen(st)==1 && st[0]=='0'))
        a[i]=new Tnumpole(n); // Число
    else
        a[i]=new Tstrpole(st); // Строка
    }

    for(i=0;i<10;i++)
        a[i]->Print();
    for(i=0;i<10;i++) delete a[i];
}

```

Программа создает 10 полиморфных объектов разных производных классов: если введено число, то объект класса Число, иначе – объект класса Строка. Затем, при выводе содержимого массива осуществляется вызов нужного аспекта виртуальной функции.

12. Библиотека потоков данных

Библиотека потокового ввода-вывода `iostream` может быть использована в качестве альтернативы известной стандартной библиотеки ввода-вывода языка C – `stdio`. Для нас библиотека `iostream` интересна как прекрасный пример объектно-ориентированного проектирования, так как содержит многие характерные приемы и конструкции. В основе ООП подхода, реализуемого средствами `iostream`, лежит предположение о том, что объекты обладают знанием того, какие действия следует предпринимать при вводе и выводе.

При пользовании библиотекой `iostream` ошибки, связанные с "перепутыванием" типов данных, исключены. Если вы используете в операции ввода-вывода переменную типа `unsigned long`, то вызывается подпрограмма, ответственная именно за этот тип.

Библиотека `stdio` поддерживает средства языка C, позволяющее использовать переменное число параметров. Но такая гибкость дается не даром – на этапе компиляции проверка соответствия между спецификацией формата, как в функциях `printf()` и `scanf()`, не выполняется.

В библиотеке `iostream` применен другой подход. Операторы ввода-вывода оформляются в виде выражений с применением переопределяемых функций-операторов для каждого из типов данных, встречающихся в выражении. Если вам необходимо использовать новый тип данных в операциях ввода-вывода, вы можете расширить библиотеку `iostream` своими функциями-операторами.

Библиотека `iostream` более медленная, чем `stdio`, но это небольшая плата за надежность и расширяемость, базирующиеся на возможностях объектно-ориентированных средств вывода.

12.1. Простое внесение

В данном примере реализовано две программы, первая с использованием библиотеки `stdio.h`, а вторая с использованием `iostream.h`.

Вариант на стандартном C	Вариант на C++
<pre>#include <stdio.h> int i; char buff[80]; p в printf ("Введите символьную строку:");</pre>	<pre>#include <iostream.h> int i; cout << "Введите число и символьную строку:";</pre>

```
printf ("Вы ввели число: %d\n Вы  
ввели строку: %s\n", i, buff);
```

```
cout << "Вы ввели число:" << i << "\n"  
<< "Вы  
ввели строку:" << buff << "\n";
```

Сообщения, выводимые программой, будут иметь вид:

Вы ввели число: 12

Вы ввели строку: My string

В первом варианте программы для вывода информации на экран применена функция `printf()`. В качестве параметра ей передается строка форматирования и переменные, значения которых необходимо вывести. Для того чтобы вывести числовое значение применяется спецификатор `%d`, а для вывода строки применяется спецификатор `%s`. Как видно из примера спецификатор в строке форматирования должен строго совпадать с типом данных, иначе произойдет ошибка или результат вывода на экран будет не таким как ожидается.

Во второй программе для вывода такой же информации применяется библиотека `iostream`. Данная программа выводит абсолютно то же самое, но отличается во всем остальном. Вывод в ней осуществляется в результате выполнения выражения, а не вызова отдельной функции типа `printf()`. Приведенное выражение называется выражением внесения, так как текст в данном случае вносится в выходной поток.

Для того чтобы понять, как работает данная технология необходимо более подробно разобраться как компилятор интерпретирует данную команду. Первым шагом к пониманию операции внесения будет расстановка скобок в выражении так, чтобы можно было понять, каким образом оно интерпретируется компилятором:

```
(((((cout<<"Вы ввели число:")<<i)<<"\n Вы ввели строку:")<< buff)<<"\n");
```

Объект `cout` – это предопределенный объект класса `iostream`, который используется для вывода. Кроме него существуют еще

`cin` – стандартный поток ввода

`cerr` – поток для вывода сообщений об ошибках.

Когда компилятор разбирает приведенное выше выражение, он начинает с самого высокого уровня вложенности скобочной структуры и на этом уровне находит:

```
cout<<"Вы ввели число:"
```

В процессе анализа этого выражения компилятор пытается отыскать функцию – `operator<<()`, имеющую в качестве левого операнда объект класса `ostream`, а в качестве правого – целое. Описание переопределяемой функции `operator << (ostream &, char*)` содержится в заголовочном файле `iostream.h`. Здесь компилятор C++ преобразует исходное выражение в более пригодное для дальнейшей обработки. В результате получается следующее:

```
operator<<(cout, "Вы ввели число:")
```

Когда функция `operator << (ostream &, char*)` будет выполнена, она выведет свой аргумент (строку) и примет значение объекта `cout` (ее первый аргумент). "Вы ввели число:" выведется на экран, и подвыражением, находящимся на самом глубоком уровне вложенности, станет

```
cout << i
```

Теперь компилятор ищет функцию `operator <<(ostream &, int)`. Объект `iostream` содержит функции операторы для всех встроенных типов. Процесс продолжается до тех пор, пока выражение вхождения не будет сведено к набору вызовов функции `operator << ()`.

В итоге, выполнение этой строчки приведет к последовательному вызову нескольких функций–операторов. Последовательность вызова операторов приведена в таблице 12.1.

Таблица 12.1. Последовательность вызова операторов

Функции	Левый операнд	Правый операнд	Возвращаемое значение
		"Вы ввели число:"	
		"\n Вы ввели строку:"	

Библиотека `iostream` чрезвычайно устойчива к ошибкам, связанным с неправильным использованием типов переменных, так как каждый тип при

вводе-выводе обслуживается свой собственной функцией извлечения и внесения.

Выражение извлечения

Как и внесение, извлечение выполняется в C++ переопределяемыми функциями–операторами, обращения к которым подставляются компилятором в зависимости от типов данных, используемых в выражении. Рассмотрим еще раз предыдущий пример.

Выражение извлечения в данной программе – это выражение, которое использует `cin`, предопределенный объект оператора `iostream`.

При анализе этой конструкции компилятор ведет себя так же, как и при анализе выражения внесения, подставляя вызовы функции в соответствии с типами используемых переменных.

Если на вход поступит "строчечка 12", то программа будет в большом затруднении при попытке интерпретировать "строчечка" как число. Однако библиотека `iostream`, в отличие от `scanf()`, производит контроль ошибок после ввода каждого значения. Кроме того, `iostream` может быть расширена введением операторов для новых типов.

Одна из наиболее распространенных ошибок при использовании `scanf()` – это задание вместо адресов аргументов их значений. Другая распространенная ошибка – путаница в использовании модификаторов форматов. При работе с `iostream` такого не бывает, так как проверка соответствия типов – неотъемлемая часть процесса ввода-вывода. Компилятор обеспечивает вызов функций-операторов, строго соответствующих используемым типам.

Создание собственных функции внесения и извлечения

Операторы `>>` и `<<` можно перегружать, причем, можно создавать собственные операторы извлечения и внесения для собственных типов данных. В общем виде, операция внесения имеет следующую форму:

```
o
s
t          stream << ...// вывод элементов объекта obj
r          // в поток stream, используя готовые функции внесения
e
a
m
o
p
e
r
```

Аналогичным образом может быть определена функция извлечения:

```
istream& operator >> (istream& stream, имя_класса& obj)
```

```
    stream >> ...// ввод элементов объекта obj  
    // из потока stream, используя готовые функции внесения
```

Функции внесения и извлечения возвращают ссылку на соответствующие объекты, являющиеся, соответственно, потоками вывода или ввода. Первые аргументы тоже должны быть ссылкой на поток. Второй параметр – ссылка на объект, выводящий или получающий информацию.

Эти функции не могут быть методами класса, для работы с которым они создаются. Если бы это было так, то левым аргументом, при вызове операции << (или >>), должен стоять объект, генерирующий вызов этой функции.

```
cout << "Моя строка"; a cout.operator << ("Моя строка"); // ошибка
```

Получается, что функция внесения, определяемая для объектов вашего класса, должна быть методом объекта cout.

Однако при этом можно столкнуться с серьезным ограничением, в случае, когда необходимо вывести (или ввести) значения защищенных членов класса. По этой причине, как правило, функции внесения и извлечения объявляют дружественными к создаваемому вами классу.

```
class _3d  
{  
    double x, y, z;  
public:  
    _3d ();  
    _3d (double initX, double initY, double initZ);  
    double mod () {return sqrt (x*x + y*y +z*z);};  
    double projection (_3d r) {return (x*r.x + y*r.y + z*r.z) / mod();}  
    _3d& operator + (_3d& b);  
    _3d& operator = (_3d& b);  
    friend ostream& operator << (ostream& stream, _3d& obj);
```



```

    friend ostream& operator >> (ostream& stream, _3d& obj);
};

ostream& operator << (ostream& stream, _3d& obj)
{
    stream << "x=" << obj.x << "y=" << obj.y << "z=" << obj.z;
    return stream;
}

istream& operator << (istream& stream, _3d& obj)
{
    stream >> obj.x >> obj.y >> obj.z;
    return stream;
}

main()
{
    _3d vecA;
    cin >> vecA;
    cout << "My vector: " << vecA << "\n";
}

```

Хотя в функции внесения (или извлечения) может выполняться любая операция, лучше ограничить ее работу только выводом (или вводом) информации.

Функции библиотеки `iostream`

Основное отличие профессионального программиста от любителя заключается в том, что он проверяет все входные данные. В библиотеке `iostream` для этого используются специальные методы класса `ios` и двух его наследников `istream` и `ostream`. Всего в этих классах находится порядка 25 методов, позволяющих получить информацию о состоянии объектов и управлять их поведением. Некоторые из этих методов приведены в таблице

Таблица 12.2. Методы для получения информации об объекте

Имя функции	Действие
	возвращает 1, если ошибок не обнаружено
	возвращает 1, если поток находится в состоянии "коней файла"
	возвращает 1, если обнаружена восстанавливаемая ошибка ввода-вывода (обычно, ошибка преобразования данных)
	возвращает 1, если обнаружена невосстанавливаемая ошибка ввода-вывода
	сбрасывает состояние ошибки ввода-вывода
	устанавливает точность вывода чисел с плавающей точкой
	устанавливает ширину поля вывода

С учетом данных функций предыдущую программу можно переработать следующим образом:

```

#include <iostream.h>
int i;
char buff[80];
do{
    if (cin.fail()) cin.clear(); // сброс состояния ошибки
    cout << "Введите число и символьную строку:";
cin >> i;
    if (cin.fail())

        cout << "Нужно ввести число";
}

cin >> buff;
if (cin.fail())

    cout << "Нужно ввести строку";
}

```

```
} while (cin.fail() && !cin.bad());
```

```
cout << "Вы ввели число:" << i << "\n"  
<< "Вы ввели строку:" << buff << "\n";
```

Условие повторение цикла означает, что нужно повторить цикл, если произошла восстанавливаемая ошибка (как правило, это ошибка преобразования), но только в том случае, если есть возможность восстановления. Единственная ошибка, которая может вызвать аварийное завершение – это переполнение 80-символьного буфера для ввода строки. Можно решить и эту проблему, указав объекту `cin` размер буфера, используя метод `width()` класса `ios`.

```
cin.width(sizeof(buff));  
cin >> buff;
```

Манипуляторы ввода-вывода

Кроме представленных ранее методов, существует и другой способ управления процессом ввода-вывода, основанный на использовании манипуляторов ввода-вывода. Манипуляторы позволяют менять режимы операций ввода-вывода непосредственно в выражениях внесения или извлечения. Для того чтобы использовать их в своей программе, в исходные тексты нужно включить заголовочный файл `iomanip.h`.

В предыдущем пункте было использовано ограничение на размер буфера `cin`, применяя метод `width()`:

```
cin.width(sizeof(buff));  
cin >> buff;
```

Используя манипулятор `setw()`, можно определить размер буфера так:

```
cin >> setw (sizeof (buff)) >> buff;
```

Существует два вида манипуляторов ввода-вывода: параметризованные и непараметризованные. Список стандартных манипуляторов библиотеки `iostream()` приведены в таблице 12.3, а список флагов приведен в таблице 12.4.

Таблица 12.3. Основные манипуляторы

манипулятор	описание
	устанавливает базу для преобразования чисел к 10
	устанавливает базу для преобразования чисел к 8
	устанавливает базу для преобразования чисел к 16
	пропуск начальных пробелов
	выводит нулевой символ
	Выводит признак конца строки
	Устанавливает символ-заполнитель
	Устанавливает точность вывода чисел с плав. точкой
	устанавливает базу для преобразования числовых значений
	устанавливает отдельные флаги потока
	сбрасывает отдельные флаги потока

Таблица 12.4. Некоторые из флагов потока:

Флаг	Описание
	выравнивание по левому краю
	выравнивание по правому краю
	устанавливают базу для ввода-вывода
	выводить показатель базы
	выводить десятичную точку для чисел с плав точкой
	16-ричные большие буквы

	показать "+" для положительных целых чисел
	установить экспон форму для чисел с плав. точкой
	fixed установить формат чисел с фиксированной плав точкой

Кроме того, вы можете создать свой собственные манипуляторы. Этому могут быть, по крайней мере, две причины. Во-первых, это может быть манипулятор, заключающий в себе некоторую часто встречающуюся в программе последовательность манипуляторов. Во-вторых, для управления нестандартными устройствами ввода-вывода.

Создание непараметризованных манипуляторов не очень сложное занятие. Все, что для этого нужно, это определить функцию, имеющую прототип, похожий на следующий:

Если создается манипулятор только для вывода, `ios` заменяется на `ostream`, если для ввода – на `istream`.

В теле функции можно, что-то сделать с потоком, переданным в качестве параметров, а затем передать его в вызывающую программу на переданный функции аргумент типа поток.

Вот пример манипулятора для вывода с выравниванием по правому краю:

```
ostream & right(ostream &)
{
    s << resetiosflags(ios::left) << setiosflags(ios::right);
}
```

Функция манипулятор изменяет биты флагов потока, переданного ей в качестве аргумента. После того, как `right` установит биты флагов потока, последний будет выводиться в таком режиме до тех пор, пока флаги потока не будут изменены.

Таким образом, выражение:

означает следующее. Манипулятор `setw(20)` установит поле вывода шириной 20 символов, манипулятор `right` установит выравнивание по правой границе поля вывода, выведется число, а манипулятор `endl` вызовет переход к следующей строке.

Параметризованные манипуляторы более сложны в написании, так как в этом случае необходимо создать класс, содержащий указатель на функцию – манипулятор и указатель на аргументы функции манипулятора. Макроопределения, которые облегчат процесс разработки параметризованных манипуляторов, содержатся в заголовочном файле

Файловые и строковые потоки

Одной из самых привлекательных возможностей библиотеки `iostream` является то, что она одинаково совершенно работает с различными источниками данных для ввода и вывода: клавиатурой и экраном, файлами и строками. Главное, что вы должны сделать, чтобы использовать строку или файл с библиотекой `iostream`, - это создать объект определенного типа.

Первостепенная задача при создании таких объектов - это организация буфера и связывание его с потоком.

При создании выходных файловых потоков вначале создается объект для буферизации типа `filebuf`, а затем этот объект связывается с новым объектом типа `ostream` для последующего вывода в этот новый объект.

```
filebuf mybuff; // создаем буферный объект
// увязываем этот буфер с файлом output для вывода
mybuff.open("output", ios::out);
// новый потоковый объект
```

Аналогичные операции можно проделать при реализации извлечения из файла-буфера. Но можно поступить и по-другому, используя буферизированные файловые потоки трех видов:

```
ifstream    – для ввода,
ofstream    – для вывода,
fstream     – для ввода и вывода.
```

Например,

```
// сразу создается буферизованный потоковый объект, связанный
// с файлом input
ifstream mycout("input");
```

Для выходных строковых потоков вы можно выделить буфер в программе или предоставить потоку возможность самому создать буфер динамически, но в таком случае придется заботиться о доступе к буферу, удалении и высвобождении выделенной под буфер памяти.

```
char mybuff[128];  
ostream mycout (mybuff, sizeof(mybuff));
```

При этом mybuff примет значение "123".

Аналогично для ввода,

```
char mybuff[128]= "123";  
istream mycin (mybuff, sizeof(mybuff));
```

После этого i примет значение 123. Для входных строковых потоков можно использовать символьный массив с завершающим нулем, либо указав точный размер.

13. Исключительные ситуации, обработка, основные операции

Обработка исключений – это механизм, позволяющий двум независимо разработанным программным компонентам взаимодействовать в аномальной ситуации, называемой исключением.

13.1. Возбуждение исключения

Исключение – это аномальное поведение во время выполнения, которое программа может обнаружить, например: деление на 0, выход за границы массива или истощение свободной памяти. Такие исключения нарушают нормальный ход работы программы, и на них нужно немедленно отреагировать. В C++ имеются встроенные средства для их возбуждения и обработки. С помощью этих средств активизируется механизм, позволяющий двум несвязанным (или независимо разработанным) фрагментам программы обмениваться информацией об исключении.

Когда встречается аномальная ситуация, та часть программы, которая ее обнаружила, может сгенерировать, или возбудить, исключение. Чтобы понять, как это происходит, реализуем класс `iStack`, используя исключения для извещения об ошибках при работе со стеком. Определение класса

```
#include
class iStack {
public:
    iStack( int capacity )
        : _stack( capacity ), _top( 0 ) { }

    bool pop( int &top_value );
    bool push( int value );

    bool full();
    bool empty();
    void display();

    int size();
private:
    int _top;
    vector< int > _stack;
```


iStack выглядит следующим образом:

Стек реализован на основе вектора из элементов типа `int`. При создании объекта класса `iStack` его конструктор создает вектор из `int`, размер которого (максимальное число элементов, хранящихся в стеке) задается с помощью начального значения. Например, следующая инструкция создает объект `myStack`, который способен содержать не более 20 элементов типа `int`:

При манипуляциях с объектом `myStack` могут возникнуть две ошибки:

- запрашивается операция `pop()`, но стек пуст;
- запрашивается операция `push()`, но стек полон.

Вызвавшую функцию нужно уведомить об этих ошибках посредством исключений. Во-первых, должны быть определены, какие именно исключения могут быть возбуждены. В C++ они чаще всего реализуются с помощью классов. Эти определения мы поместим в заголовочный файл

```
// stackExp.h
class popOnEmpty { /* ... */ };
```

с

l Затем надо изменить определения функций-членов `pop()` и `push()` так, чтобы они возбуждали эти исключения. Для этого предназначена инструкция `throw`, которая во многих отношениях напоминает `return`. Она состоит из ключевого слова `throw`, за которым следует выражение того же типа, что и тип возбуждаемого исключения.

P Попробуем такой вариант:

u

s

h

O

n Исключение – это объект, и функция `pop()` должна генерировать объект класса соответствующего типа. Выражение в инструкции `throw` не может быть просто типом. Для создания нужного объекта необходимо вызвать конструктор класса. Инструкция `throw` для функции `pop()` будет выглядеть так:

```
// инструкция является вызовом конструктора
```

```
{
```

```
/
```

```
*
```

```
.
```

```
.
```

Эта инструкция создает объект исключения типа `popOnEmpty`.

Методы `pop()` и `push()` были определены как возвращающие значение типа `bool`: `true` означало, что операция завершилась успешно, а `false` – что произошла ошибка. Поскольку теперь для извещения о неудаче `pop()` и `push()` используют исключения, возвращать значение необязательно. Поэтому мы будем считать, что эти методы имеют тип `void`:

```
// больше не возвращают значения  
  
void pop( int &top_value );  
void push( int value );
```

Теперь функции, пользующиеся нашим классом `iStack`, будут предполагать, что все хорошо, если только не возбуждено исключение; им больше не надо проверять возвращенное значение, чтобы узнать, как завершилась операция.

```
#include "stackExcp.h"  
void iStack::pop( int &top_value )  
{  
    if ( empty() )  
        throw popOnEmpty();  
  
    top_value = _stack[ --_top ];  
  
    cout << "iStack::pop(): "<< top_value << endl;  
}  
void iStack::push( int value )  
{  
    cout << "iStack::push( "<< value << " )\n";  
  
    if ( full() )  
        throw pushOnFull( value );  
}
```

Хотя исключения чаще всего представляют собой объекты типа класса, инструкция `throw` может генерировать объекты любого типа. Например, функция `mathFunc()` в следующем примере возбуждает исключение в виде объекта-перечисления:

```
enum EHstate { noErr, zeroOp, negativeOp, severeError };

int mathFunc( int i ) {
    if ( i == 0 )
        throw zeroOp; // исключение в виде объекта-перечисления

    // в противном случае продолжается нормальная обработка
}

Тру-блок
#include
#include "iStack.h"

int main() {
    iStack stack( 32 );

    stack.display();
    for ( int ix = 1; ix < 51; ++ix )
    {
        if ( ix % 3 == 0 )
            stack.push( ix );

        if ( ix % 4 == 0 )
            stack.display();

        if ( ix % 10 == 0 ) {
            int dummy;
            stack.pop( dummy );
        }
    }
}
```

В данной программе тестируется определенный в предыдущем разделе класс `iStack` и его методы `pop()` и `push()`. Выполняется 50 итераций цикла `for`. На каждой итерации в стек помещается значение, кратное 3: 3, 6, 9 и т.д. Если значение кратно 4 (4, 8, 12...), то выводится текущее содержимое стека, а если кратно 10 (10, 20, 30...), то с вершины снимается один элемент, после чего содержимое стека выводится снова.

Инструкции, которые могут возбуждать исключения, должны быть заключены в `try`-блок. Такой блок начинается с ключевого слова `try`, за которым идет последовательность инструкций, заключенная в фигурные скобки, а после этого – список обработчиков, называемых `catch`-предложениями. `Try`-блок группирует инструкции программы и ассоциирует с ними обработчики исключений.

```
for ( int ix = 1; ix < 51; ++ix ) {
    try { // try-блок для исключений pushOnFull

        stack.push( ix );
    if ( ix % 3 == 0 )
        catch ( pushOnFull ) { ... }

        if ( ix % 4 == 0 )
            stack.display();

        try { // try-блок для исключений popOnEmpty

            if ( ix % 10 == 0 ) {
                stack.pop( dummy );
            }
        }
    }
}
```

В таком виде программа выполняется корректно. Однако обработка исключений в ней перемежается с кодом, используемым при нормальных обстоятельствах, а такая организация несовершенна. В конце концов, исключения – это аномальные ситуации, возникающие только в особых

s
k
d
i
s
p
l

случаях. Желательно отделить код для обработки аномалий от кода, реализующего операции со стеком. Следующая схема облегчает чтение и сопровождение программы:

```
f    {
o        if ( ix % 3 == 0 )
r            stack.push( ix );
i
n        if ( ix % 4 == 0 )
t            stack.display();
i
x        if ( ix % 10 == 0 ) {
i            int dummy;
x            stack.pop( dummy );
i            stack.display();
x                }
            }
        }
    }
catch ( pushOnFull ) { ... }
catch ( popOnEmpty ) { ... }
```

С try-блоком ассоциированы два catch-предложения, которые могут обработать исключения pushOnFull и popOnEmpty, возбуждаемые функциями-членами push() и pop() внутри этого блока. Каждый catch-обработчик определяет тип "своего" исключения. Код для обработки исключения помещается внутрь составной инструкции (между фигурными скобками), которая является частью catch-обработчика. (Подробнее catch-предложения мы рассмотрим в следующем разделе.)

Исполнение программы может пойти по одному из следующих путей:

- если исключение не возбуждено, то выполняется код внутри try-блока, а ассоциированные с ним обработчики игнорируются. Функция main() возвращает 0;
- если функция-член push(), вызванная из первой инструкции if внутри цикла for, возбуждает исключение, то вторая и третья инструкции if игнорируются, управление покидает цикл for и try-блок, и выполняется обработчик исключений типа pushOnFull;

- если функция-член `pop()`, вызванная из третьей инструкции `if` внутри цикла `for`, возбуждает исключение, то вызов `display()` игнорируется, управление покидает цикл `for` и `try`-блок, и выполняется обработчик исключений типа `popOnEmpty`.

Когда возбуждается исключение, пропускаются все инструкции, следующие за той, где оно было возбуждено. Исполнение программы возобновляется в `catch`-обработчике этого исключения. Если такого обработчика не существует, то управление передается в функцию `terminate()`, определенную в стандартной библиотеке C++. `Try`-блок может содержать любую инструкцию языка C++: как выражения, так и объявления. Он вводит локальную область видимости, так что объявленные внутри него переменные недоступны вне этого блока, в том числе и в `catch`-обработчиках. Например, функцию `main()` можно переписать так, что объявление переменной `stack` окажется в `try`-блоке. В таком случае обращаться к этой переменной в `catch`-обработчиках нельзя:

```
iStack stack( 32 ); // правильно: объявление внутри try-блока

stack.display();
for ( int ix = 1; ix < 51; ++ix )

// то же, что и раньше

// здесь к переменной stack обращаться нельзя

// здесь к переменной stack обращаться нельзя

// и здесь к переменной stack обращаться нельзя
```

Можно объявить функцию так, что все ее тело будет заключено в `try`-блок. При этом не обязательно помещать `try`-блок внутрь определения функции, удобнее заключить ее тело в функциональный `try`-блок. Такая

организация поддерживает наиболее чистое разделение кода для нормальной обработки и кода для обработки исключений. Например:

```
iStack stack( 32 ); // правильно: объявление внутри try-блока

stack.display();
for ( int ix = 1; ix < 51; ++ix )

// то же, что и раньше

// здесь к переменной stack обращаться нельзя

// здесь к переменной stack обращаться нельзя
```

Ключевое слово `try` находится перед фигурной скобкой, открывающей тело функции, а `catch`-обработчики перечислены после закрывающей его скобки. Как видим, код, осуществляющий нормальную обработку, находится внутри тела функции и четко отделен от кода для обработки исключений. Однако к переменным, объявленным в `main()`, нельзя обратиться из обработчиков исключений.

Функциональный `try`-блок ассоциирует группу `catch`-обработчиков с телом функции. Если инструкция возбуждает исключение, то поиск обработчика, способного перехватить это исключение, ведется среди тех, что идут за телом функции. Функциональные `try`-блоки особенно полезны в сочетании с конструкторами классов.

Перехват исключений

В языке C++ исключения обрабатываются в предложениях `catch`. Когда какая-то инструкция внутри `try`-блока возбуждает исключение, то просматривается список последующих предложений `catch` в поисках такого, который может его обработать.

`Catch`-обработчик состоит из трех частей: ключевого слова `catch`, объявления одного типа или одного объекта, заключенного в круглые скобки (оно называется объявлением исключения), и составной инструкции. Если для обработки исключения выбрано некоторое `catch`-предложение, то выполняется эта составная инструкция. Рассмотрим `catch`-обработчики исключений `pushOnFull` и `popOnEmpty` в функции `main()` более подробно:

```
catch ( pushOnFull ) {
    cerr << "trying to push value on a full stack\n";
    return errorCode88;
}
catch ( popOnEmpty ) {
    cerr << "trying to pop a value on an empty stack\n";
```

В обоих `catch`-обработчиках есть объявление типа класса; в первом это `pushOnFull`, а во втором – `popOnEmpty`. Для обработки исключения выбирается тот обработчик, для которого типы в объявлении исключения и в возбужденном исключении совпадают. (В главе 19 мы увидим, что типы не обязаны совпадать точно: обработчик для базового класса подходит и для исключений с производными классами.) Например, когда функция-член `pop()` класса `iStack` возбуждает исключение `popOnEmpty`, то управление попадает во второй обработчик. После вывода сообщения об ошибке в `cerr`, функция `main()` возвращает код `errorCode89`.

После завершения обработчика выполнение возобновляется с инструкции, идущей за последним `catch`-обработчиком в списке. В нашем примере оно продолжается с инструкции `return` в функции `main()`. После того как `catch`-обработчик `popOnEmpty` выведет сообщение об ошибке, `main()` вернет 0.

```
int main() {
    iStack stack( 32 );

    try {
        stack.display();
        for ( int x = 1; ix < 51; ++ix )
```



```

        // то же, что и раньше
    }
    catch ( pushOnFull ) {
        cerr << "trying to push value on a full stack\n";
    }
    catch ( popOnEmpty ) {
        cerr << "trying to pop a value on an empty stack\n";
    }

    // исполнение продолжается отсюда

```

Говорят, что механизм обработки исключений в C++ невозвратный: после того как исключение обработано, управление не возобновляется с того места, где оно было возбуждено. В нашем примере управление не возвращается в функцию-член pop(), возбуждвшую исключение.

Объекты-исключения

Объявлением исключения в catch-обработчике могут быть объявления типа или объекта. В каких случаях это следует делать тогда, когда необходимо получить значение или как-то манипулировать объектом, созданным в выражении throw. Если классы исключений спроектированы так, что в объектах-исключениях при возбуждении сохраняется некоторая информация и если в объявлении исключения фигурирует такой объект, то инструкции внутри catch-обработчика могут обращаться к информации, сохраненной в объекте выражением throw.

Изменим реализацию класса исключения pushOnFull, сохранив в объекте-исключении то значение, которое не удалось поместить в стек. Catch-обработчик, сообщая об ошибке, теперь будет выводить его в cerr. Для этого мы сначала модифицируем определение типа класса pushOnFull следующим образом:

```

// новый класс исключения:
// он сохраняет значение, которое не удалось поместить в стек
class pushOnFull {
public:

```

```

    pushOnFull( int i ) : _value( i ) { }
    int value { return _value; }
private:

```

Новый закрытый член `_value` содержит число, которое не удалось поместить в стек. Конструктор принимает значение типа `int` и сохраняет его в члене `_data`. Вот как вызывается этот конструктор для сохранения значения из выражения `throw`:

```

void iStack::push( int value )

```

```

    // значение, сохраняемое в объекте-исключении

```

У класса `pushOnFull` появилась также новая функция-член `value()`, которую можно использовать в `catch`-обработчике для вывода хранящегося в объекте-исключении значения:

```

catch ( pushOnFull eObj ) {
    cerr << "trying to push value << "eObj.value()
    << "on a full stack\n";

```

Обратите внимание, что в объявлении исключения в `catch`-обработчике фигурирует объект `eObj`, с помощью которого вызывается функция-член `value()` класса `pushOnFull`.

Объект-исключение всегда создается в точке возбуждения, даже если выражение `throw` – это не вызов конструктора и, на первый взгляд, не должно создавать объекта. Например:

```

enum EHstate { noErr, zeroOp, negativeOp, severeError };
enum EHstate state = noErr;

```

```

int mathFunc( int i ) {

```

```
// иначе продолжается обычная обработка
// создан объект-исключение
```

В этом примере объект `state` не используется в качестве объекта-исключения. Вместо этого, выражением `throw` создается объект-исключение типа `EHstate`, который инициализируется значением глобального объекта `state`. Как программа может различить их? Для ответа на этот вопрос мы должны присмотреться к объявлению исключения в `catch`-обработчике более внимательно.

Это объявление ведет себя почти так же, как объявление формального параметра. Если при входе в `catch`-обработчик исключения выясняется, что в нем объявлен объект, то он инициализируется копией объекта-исключения. Например, следующая функция `calculate()` вызывает определенную выше `mathFunc()`. При входе в `catch`-обработчик внутри `calculate()` объект `eObj` инициализируется копией объекта-исключения, созданного выражением

```
void calculate( int op ) {
    try {
        mathFunc( op );
    }
    catch ( EHstate eObj ) {
```

Объявление исключения в этом примере напоминает передачу параметра по значению. Объект `eObj` инициализируется значением объекта-исключения точно так же, как переданный по значению формальный параметр функции – значением соответствующего фактического аргумента. `eObj` – копия сгенерированного объекта-исключения. Как и в случае параметров функции, в объявлении исключения может фигурировать ссылка. Тогда `catch`-обработчик будет напрямую ссылаться на объект-исключение, сгенерированный выражением `throw`, а не создавать его локальную копию:

```
void calculate( int op ) {
```

```

try {
    mathFunc( op );
}
catch ( EHstate &eObj ) {
    // eObj ссылается на сгенерированный объект-
исключение

```

Для предотвращения ненужного копирования больших объектов применять ссылки следует не только в объявлениях параметров типа класса, но и в объявлениях исключений того же типа.

В последнем случае catch-обработчик сможет модифицировать объект-исключение. Однако переменные, определенные в выражении throw, остаются без изменения. Например, модификация eObj внутри catch-обработчика не затрагивает глобальную переменную state, установленную в выражении throw:

```

void calculate( int op ) {
    try {
        mathFunc( op );
    }
    catch ( EHstate &eObj ) {
        // исправить ошибку, вызвавшую исключение
        eObj = noErr; // глобальная переменная state не
изменилась

```

Catch-обработчик переустанавливает eObj в noErr после исправления ошибки, вызвавшей исключение. Поскольку eObj – это ссылка, можно ожидать, что присваивание модифицирует глобальную переменную state. Однако изменяется лишь объект-исключение, созданный в выражении throw, поэтому модификация eObj не затрагивает state.

Раскрутка стека

Поиск catch-обработчика для возбужденного исключения происходит следующим образом. Когда выражение throw находится в try-блоке, все ассоциированные с ним предложения catch исследуются с точки зрения того,

могут ли они обработать исключение. Если подходящее предложение `catch` найдено, то исключение обрабатывается. В противном случае поиск продолжается в вызывающей функции. Предположим, что вызов функции, выполнение которой прекратилось в результате исключения, погружен в `try`-блок; в такой ситуации исследуются все предложения `catch`, ассоциированные с этим блоком. Если один из них может обработать исключение, то процесс заканчивается. В противном случае переходим к следующей по порядку вызывающей функции. Этот поиск последовательно проводится во всей цепочке вложенных вызовов. Как только будет найдено подходящее предложение, управление передается в соответствующий обработчик.

В нашем примере первая функция, для которой нужен `catch`-обработчик, – это метод `pop()` класса `iStack`. Поскольку выражение `throw` внутри `pop()` не находится в `try`-блоке, то программа покидает `pop()`, не обработав исключение. Следующей рассматривается функция, вызвавшая `pop()`, то есть `main()`. Вызов `pop()` внутри `main()` находится в `try`-блоке, и далее исследуется, может ли хотя бы одно ассоциированное с ним предложение `catch` обработать исключение. Поскольку обработчик исключения `popOnEmpty` имеется, то управление попадает в него.

Процесс, в результате которого программа последовательно покидает составные инструкции и определения функций в поисках предложения `catch`, способного обработать возникшее исключение, называется раскруткой стека. По мере раскрутки прекращают существование локальные объекты, объявленные в составных инструкциях и определениях функций, из которых произошел выход. C++ гарантирует, что во время описанного процесса вызываются деструкторы локальных объектов, хотя они исчезают из-за возбужденного исключения.

Если в программе нет предложения `catch`, способного обработать исключение, оно остается необработанным. Но исключение – это настолько серьезная ошибка, что программа не может продолжать выполнение. Поэтому, если обработчик не найден, вызывается функция `terminate()` из стандартной библиотеки C++. По умолчанию `terminate()` активизирует функцию `abort()`, которая аномально завершает программу.

Выражение `throw` ведет себя аналогично вызову, а предложение `catch` чем-то напоминает определение функции. Основная разница между этими двумя механизмами заключается в том, что информация, необходимая для вызова функции, доступна во время компиляции, а для обработки исключений – нет. Обработка исключений в C++ требует языковой поддержки во время выполнения. Например, для обычного вызова функции компилятору в точке активизации уже известно, какая из перегруженных функций будет вызвана. При обработке же исключения компилятор не знает, в какой функции находится `catch`-обработчик и откуда возобновится

выполнение программы. Функция `terminate()` предоставляет механизм времени выполнения, который извещает пользователя о том, что подходящего обработчика не нашлось.

Повторное возбуждение исключения

Может оказаться так, что в одном предложении `catch` не удалось полностью обработать исключение. Выполнив некоторые корректирующие действия, `catch`-обработчик может решить, что дальнейшую обработку следует поручить функции, расположенной "выше" в цепочке вызовов. Передать исключение другому `catch`-обработчику можно с помощью повторного возбуждения исключения. Для этой цели в языке используется конструкция:

Которая вновь генерирует объект-исключение. Повторное возбуждение возможно только внутри составной инструкции, являющейся частью `catch`-обработчика:

```
catch ( exception eObj ) {
    if ( canHandle( eObj ) )
        // обработать исключение

    // повторно возбудить исключение, чтобы его перехватил другой
    // catch-обработчик
```

При повторном возбуждении новый объект-исключение не создается. Это имеет значение, если `catch`-обработчик модифицирует объект, прежде чем возбудить исключение повторно.

```
enum EHstate { noErr, zeroOp, negativeOp, severeError };

void calculate( int op ) {
    try {
        // исключение, возбужденное mathFunc(), имеет значение zeroOp
    }
    catch ( op );
```

```
catch ( EHstate eObj ) {  
    // что-то исправить  
  
    // пытаемся модифицировать объект-исключение  
  
    // предполагалось, что повторно возбужденное исключение будет  
    // иметь значение severeErr
```

Так как `eObj` не является ссылкой, то `catch`-обработчик получает копию объекта-исключения, так что любые модификации `eObj` относятся к локальной копии и не отражаются на исходном объекте-исключении, передаваемом при повторном возбуждении. Таким образом, переданный далее объект по-прежнему имеет тип `zeroOp`.

Чтобы модифицировать исходный объект-исключение, в объявлении исключения внутри `catch`-обработчика должна фигурировать ссылка:

```
// модифицируем объект-исключение  
  
// повторно возбужденное исключение имеет значение severeErr
```

Теперь `eObj` ссылается на объект-исключение, созданный выражением `throw`, так что все изменения относятся непосредственно к исходному объекту. Поэтому при повторном возбуждении исключения далее передается модифицированный объект.

Таким образом, другая причина для объявления ссылки в `catch`-обработчике заключается в том, что сделанные внутри обработчика модификации объекта-исключения в таком случае будут видны при повторном возбуждении исключения.

13.6. Перехват всех исключений

Иногда функции нужно выполнить определенное действие до того, как она завершит обработку исключения, даже несмотря на то, что обработать его она не может. К примеру, функция захватила некоторый ресурс, скажем открыла файл или выделила память из хипа, и этот ресурс необходимо освободить перед выходом:

```
void manip() {
    resource res;
    res.lock();    // захват ресурса

    // использование ресурса
    // действие, в результате которого возбуждено исключение

    // не выполняется, если возбуждено исключение
```

Если исключение возбуждено, то управление не попадет на инструкцию, где ресурс освобождается. Чтобы освободить ресурс, не пытаясь перехватить все возможные исключения, воспользуемся специальной конструкцией, позволяющей перехватывать любые исключения. Это не что иное, как предложение `catch`, в котором объявление исключения имеет вид (...) и куда управление попадает при любом исключении. Например:

```
// управление попадает сюда при любом возбужденном исключении

// здесь размещаем наш код
```

Конструкция `catch(...)` используется в сочетании с повторным возбуждением исключения. Захваченный ресурс освобождается внутри составной инструкции в `catch`-обработчике перед тем, как передать исключение по цепочке вложенных вызовов в результате повторного возбуждения:

```
void manip() {
    resource res;
    res.lock();

    // использование ресурса
```



```
// действие, в результате которого возбуждено исключение
```

```
res.release(); // не выполняется, если возбуждено исключение
```

Чтобы гарантировать освобождение ресурса в случае, когда выход из `map()` происходит в результате исключения, мы освобождаем его внутри `try` до того, как исключение будет передано дальше. Можно также управлять захватом и освобождением ресурса путем инкапсуляции в класс всей работы с ним. Тогда захват будет реализован в конструкторе, а освобождение – в автоматически вызываемом деструкторе.

Предложение `catch(...)` используется самостоятельно или в сочетании с другими `catch`-обработчиками. В последнем случае следует позаботиться о правильной организации обработчиков, ассоциированных с `try`-блоком.

`Catch`-обработчики исследуются по очереди, в том порядке, в котором они записаны. Как только найден подходящий, просмотр прекращается. Следовательно, если предложение `catch(...)` употребляется вместе с другими `catch`-обработчиками, то оно должно быть последним в списке, иначе компилятор выдаст сообщение об ошибке:

```
try {  
    stack.display();  
    for ( int ix = 1; ix < 51; ++x )  
        // то же, что и выше  
}  
catch ( pushOnFull ) { }  
catch ( popOnEmpty ) { }  
catch ( ... ) { } // должно быть последним в списке catch-  
обработчиков
```

Спецификации исключений

По объявлениям методов `pop()` и `push()` класса `iStack` невозможно определить, что они возбуждают исключения. Можно, конечно, включить в объявление подходящий комментарий. Тогда описание интерфейса класса в заголовочном файле будет содержать документацию возбуждаемых исключений:

```
class iStack {
public:
    // ...

    void pop( int &value ); // возбуждает popOnEmpty
    void push( int value ); // возбуждает pushOnFull
```

Но такое решение несовершенно. Неизвестно, будет ли обновлена документация при выпуске следующих версий `iStack`. Кроме того, комментарий не дает компилятору достоверной информации о том, что никаких других исключений функция не возбуждает. Спецификация исключений позволяет перечислить в объявлении функции все исключения, которые она может возбуждать. При этом гарантируется, что другие исключения функция возбуждать не будет.

Такая спецификация следует за списком формальных параметров функции. Она состоит из ключевого слова `throw`, за которым идет список типов исключений, заключенный в скобки. Например, объявления методов класса `iStack` можно модифицировать, добавив спецификации исключений:

```
class iStack {
public:
    // ...

    void pop( int &value ) throw(popOnEmpty);
    void push( int value ) throw(pushOnFull);
```

Гарантируется, что при обращении к pop() не будет возбуждено никаких исключений, кроме popOnEmpty, а при обращении к push()—только

Объявление исключения – это часть интерфейса функции, оно должно быть задано при ее объявлении в заголовочном файле. Спецификация исключений – это своего рода "контракт" между функцией и остальной частью программы, гарантия того, что функция не будет возбуждать никаких исключений, кроме перечисленных.

Если в объявлении функции присутствует спецификация исключений, то при повторном объявлении этой же функции должны быть перечислены точно те же типы. Спецификации исключений в разных объявлениях одной и той же функции не суммируются:

```
// два объявления одной и той же функции
extern int foo( int parm = 0 ) throw(string);
```

```
// ошибка: опущена спецификация исключений
```

e

x Исключения возбуждаются только при обнаружении определенных аномалий в поведении программы, и во время компиляции неизвестно, встретится ли то или иное исключение во время выполнения. Поэтому нарушения спецификации исключений функции могут быть обнаружены только во время выполнения. Если функция возбуждает исключение, не указанное в спецификации, то вызывается unexpected() из стандартной библиотеки C++, а та по умолчанию вызывает terminate().

t

f Необходимо уточнить, что unexpected() не вызывается только потому, что функция возбудила исключение, не указанное в ее спецификации. Все нормально, если она обработает это исключение самостоятельно, внутри функции. Например:

i

n

t

p

a

r

m

```
void recoup( int op1, int op2 ) throw(ExceptionType)
{
    try {
        // ...
        throw string("we're in control");
```

```
// обрабатывается возбужденное исключение
```

```
// сделать все необходимое
```

```
} // все хорошо, unexpected() не вызывается
```

Функция `getResource()` возбуждает исключение типа `string`, несмотря на его отсутствие в спецификации. Поскольку это исключение обработано в теле функции, `unexpected()` не вызывается.

Нарушения спецификации исключений функции обнаруживаются только во время выполнения. Компилятор не сообщает об ошибке, если в выражении `throw` возбуждается исключение неуказанного типа. Если такое выражение никогда не выполнится или не возбудит исключения, нарушающего спецификацию, то программа будет работать, как и ожидалось, и нарушение никак не проявится:

```
extern void doit( int, int ) throw(string, exceptionType);
```

```
void action ( int op1, int op2 ) throw(string) {  
    doit( op1, op2 ); // ошибки компиляции не будет
```

Функция `doit()` может возбудить исключение типа `exceptionType`, которое не разрешено спецификацией `action()`. Однако функция компилируется успешно. Компилятор при этом генерирует код, гарантирующий, что при возбуждении исключения, нарушающего спецификацию, будет вызвана библиотечная функция `unexpected()`.

Пустая спецификация показывает, что функция не возбуждает никаких исключений:

```
extern void no_problem () throw();
```

Если же в объявлении функции спецификация исключений отсутствует, то может быть возбуждено исключение любого типа.

Между типом возбужденного исключения и типом исключения, указанного в спецификации, не разрешается проводить никаких преобразований:

```
int convert( int parm ) throw(string)  
{  
    //...
```

```

if ( somethingRather )
    // ошибка программы:
    // convert() не допускает исключения типа const char*

```

Выражение `throw` в функции `convert()` возбуждает исключение типа строки символов в стиле языка C. Созданный объект-исключение имеет тип `const char*`. Обычно выражение типа `const char*` можно привести к типу `string`. Однако спецификация не допускает преобразования типов, поэтому если `convert()` возбуждает такое исключение, то вызывается функция `unexpected()`. Для исправления ошибки выражение `throw` можно модифицировать так, чтобы оно явно преобразовывало значение выражения в тип `string`:

Спецификации исключений и указатели на функции

Спецификацию исключений можно задавать и при объявлении указателя на функцию. Например:

```

v
o
i
f
i
n
t
t
h
r

```

В этом объявлении говорится, что `pf` указывает на функцию, которая способна возбуждать только исключения типа `string`. Как и для объявлений функций, спецификации исключений в разных объявлениях одного и того же указателя не суммируются, они должны быть одинаковыми:

```

extern void (*pf) ( int ) throw(string);
// ошибка: отсутствует спецификация исключения

```

При работе с указателем на функцию со спецификацией исключений есть ограничения на тип указателя, используемого в качестве инициализатора или стоящего в правой части присваивания. Спецификации исключений обоих указателей не обязаны быть идентичными. Однако на указатель-инициализатор накладываются такие же или более строгие ограничения, чем на инициализируемый указатель (или тот, которому присваивается значение). Например:

```

n
g

```

```

void recoup( int, int ) throw(exceptionType);
void no_problem() throw();
void doit( int, int ) throw(string, exceptionType);

// правильно: ограничения, накладываемые на спецификации
// исключений recoup() и pf1, одинаковы
void (*pf1)( int, int ) throw(exceptionType) = &recoup;

// правильно: ограничения, накладываемые на спецификацию
// исключений no_problem(), более строгие,
// чем для pf2
void (*pf2)( ) throw(string) = &no_problem;

// ошибка: ограничения, накладываемые на спецификацию
// исключений doit(), менее строгие, чем для pf3
//
void (*pf3)( int, int ) throw(string) = &doit;

```

Третья инициализация не имеет смысла. Объявление указателя гарантирует, что pf3 адресует функцию, которая может возбуждать только исключения типа string. Но doit() возбуждает также исключения типа exceptionType. Поскольку она не подходит под ограничения, накладываемые спецификацией исключений pf3, то не может служить корректным инициализатором для pf3, так что компилятор выдает ошибку.

14. Особенности построения графического интерфейса с использованием библиотек Qt

К базовым классам Qt относятся:

Класс `QObject` является родительским для все классов библиотеки Qt, как графических так и обычных. За счет данного класса реализуются все основные механизмы взаимодействия, например, система сигналов и слотов а так же система родитель-потомок.

Класс `QWidget` является родительским классом для все графических классов библиотеки Qt и представляет собой пустое графическое поле. Так же с помощью данного класса и его инструментария можно создавать и свои графические классы.

Класс `QObject`

Как уже было сказано `QObject` является базовым классом для почти всех классов Qt. Исключением являются только классы, которые должны быть достаточно "лёгкими" (экземпляры которых должны занимать как можно меньше памяти) и классы, объекты которых должны копироваться (`QObject` не поддерживает копирования), а также контейнерные классы. Все виджеты Qt наследуют `QObject` (класс `QWidget` является потомком `QObject`). `QObject` реализует все базовые особенности, которыми обладают классы Qt:

- мощный механизм взаимодействия между объектами с помощью сигнально-слотовых соединений;
- иерархические взаимосвязи между объектами, позволяющие объединять их в объектные деревья;
- управление памятью;
- "умные" указатели, позволяющие отслеживать уничтожение объекта;
- поддержка свойств;
- таймеры;
- обработка событий и фильтры событий;
- метаянформация о типе объекта, его свойства и т. п.

Каждый объект типа `QObject` обладает метаданными, которые хранятся внутри специального метаобъекта. Этот метаобъект создаётся для каждого потомка `QObject` и сохраняет различную информацию об объекте

(так называемые метаданные). Среди доступных для программиста метаданных есть:

- имя класса (метод `const char *QObject::className()`);
- наследование (метод `bool QObject::inherits(const char *className)`);
- информация о свойствах;
- информация о сигналах и слотах;
- общая информация о классе (`QObject::classInfo`).

Метаданные собираются во время компиляции (предварительной обработки проекта с помощью `qmake`) метаобъектным компилятором `moc`, который анализирует содержание заголовочных файлов программы. Эти метаданные позволяют получить информацию о любом потомке `QObject`. Эта информация может быть полезна как для отладки программы, так и для создания различных механизмов взаимодействия между объектами в программе. Для того, чтобы ваш класс был обработан метаобъектным компилятором, после объявления имени класса и открытия операторных скобок необходимо указать ключевое слово `Q_OBJECT`. Иначе ваш класс будет пропущен метаобъектным компилятором в процессе обработки исходного кода.

При разработке с использованием Qt часто возникает необходимость наследовать класс `QObject` непосредственно или его потомка. Объекты, которые наследуют `QObject`:

- имеют имя (`QObject :: objectName ()`), которое используется в Qt для реализации различных возможностей (стили, QML и т.д.);
- могут занимать место в иерархии других объектов `QObject`;
- могут иметь сигнально-слотовые соединения с другими объектами

14.2. Класс `QWidget`

Класс `QWidget` является фундаментальным для всех классов виджетов. Его интерфейс содержит 254 метода, 53 свойства и массу определений, необходимых каждому из виджетов, например, для изменения размеров, местоположения, обработки событий и др. Сам класс `QWidget`, как видно из рис. 14.1, унаследован от класса `QObject`, а значит, может использовать механизм сигналов/слотов и механизм объектной иерархии. Благодаря этому виджеты могут иметь потомков, которые отображаются внутри предка. Это очень важно, так как каждый виджет может служить контейнером для других виджетов, — то есть в Qt нет разделения между элементами управления и контейнерами. Виджеты в контейнерах могут

выступать в роли контейнеров для других виджетов, и так до бесконечности. Например, диалоговое окно содержит кнопки Ok и Cancel (Отмена) — следовательно, оно является контейнером. Это удобно еще и потому, что если виджет-предок станет недоступным или невидимым, то виджеты-потомки автоматически примут его состояние.

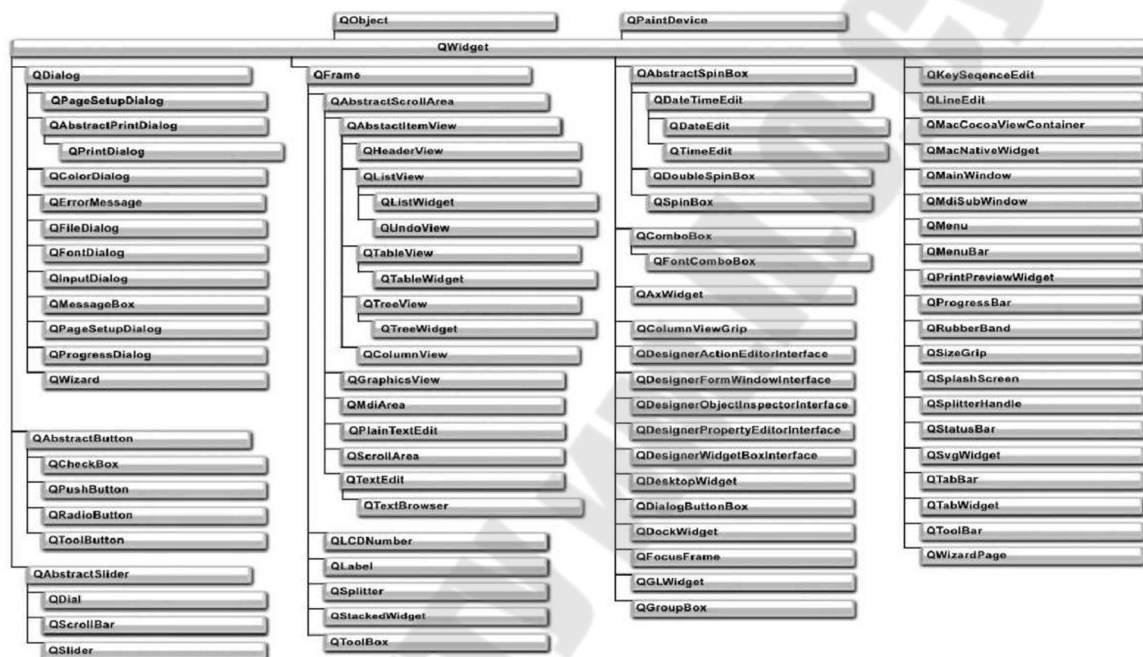


Рис. 14.1. Иерархия классов виджетов

Виджеты без предка называются виджетами верхнего уровня (top-level widgets) и имеют свое собственное окно. Все виджеты без исключения могут быть виджетами верхнего уровня. Позиция виджетов-потомков внутри виджета-предка может изменяться методом `setGeometry()` вручную или автоматически, с помощью специальных классов компоновки (layouts). Для отображения виджета на экране вызывается метод `show()`, а для скрытия — метод `hide()`.

После создания виджета верхнего уровня, чтобы показать его на экране, нужно вызвать метод `show()`, иначе и его окно, и виджеты-потомки будут невидимыми.

Класс `QWidget` и большинство унаследованных от него классов имеют конструктор с двумя параметрами:

```
QWidget(QWidget* pwgt = 0, Qt::WindowFlags f = 0)
```

Из определения видно, что не обязательно передавать параметры в конструктор, так как они равны нулю по умолчанию. А это значит, что если конструктор вызывается без аргументов, то созданный виджет станет виджетом верхнего уровня. Вторым параметром Qt: :WindowFlags служит для задания свойств окна, и с его помощью можно управлять внешним видом окна и режимом отображения. Чтобы изменить внешний вид окна, необходимо во втором параметре конструктора передать значения модификаторов, объединенные с типом окна (рис. 5.2) побитовой операцией ИЛИ. Аналогичного результата можно добиться вызовом метода

Например:

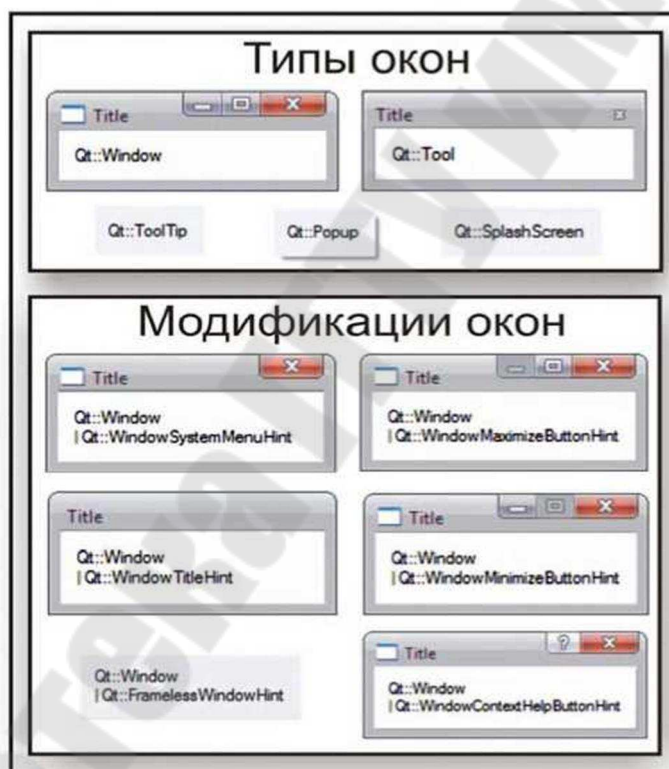


Рис. 14.2. Вид окон виджетов верхнего уровня

Значение Qt: :WindowStaysOnTopHint не изменяет внешний вид окна, а лишь рекомендует, чтобы окно всегда находилось на переднем плане и не перекрывалось другими окнами. Внешний вид окон верхнего уровня приведен на рисунке 14.2.

При помощи слота `setWindowTitle()` устанавливается надпись заголовка окна. Но это имеет смысл только для виджетов верхнего уровня.

Например:

```
wgt.setWindowTitle("My Window");
```

Слот `setEnabled ()` устанавливает виджет в доступное (`enabled`) или недоступное (`disabled`) состояние. Параметр `true` соответствует доступному, а `false` — недоступному состоянию. Чтобы узнать, в каком состоянии находится виджет, вызывается метод `isEnabled ()`.

При создании собственных классов виджетов важно, чтобы виджет был в состоянии обрабатывать события. Например, для обработки событий мыши необходимо перезаписать хотя бы один из следующих методов: `mousePressEvent ()`, `mouseMoveEvent ()`, `mouseReleaseEvent ()` ИЛИ

Механизм сигналов и слотов

Элементы графического интерфейса определенным образом реагируют на действия пользователя и посылают сообщения. Существует несколько вариантов такого решения.

Проблема расширения языка C++ решена в Qt с помощью специального препроцессора МОС (Meta Object Compiler, метаобъектный компилятор). Он анализирует классы на наличие в их определении специального макроса `Q_OBJECT` и внедряет в отдельный файл всю необходимую дополнительную информацию. Это происходит автоматически, без непосредственного участия разработчика. Подобная операция автоматического создания кода не противоречит привычному процессу программирования на C++ — ведь стандартный препроцессор перед компиляцией самой программы тоже создает промежуточный код, содержащий исполненные команды препроцессора. Подобным образом действует и МОС, записывая всю необходимую дополнительную информацию в отдельный файл, содержимое которого не требует внимания разработчика. Макрос `Q_OBJECT` должен располагаться сразу на следующей строке после ключевого слова `class` с определением имени класса. После макроса не должно стоять точки с запятой. Внедрять макрос в определение класса имеет смысл в тех случаях, когда созданный класс использует механизм сигналов и слотов или если ему необходима информация о свойствах.

Механизм сигналов и слотов полностью замещает старую модель функций обратного вызова, он очень гибок и полностью объектно-ориентирован. Сигналы и слоты — это краеугольный концепт программирования с использованием Qt, позволяющий соединить вместе

несвязанные друг с другом объекты. Каждый унаследованный от QObject класс способен отправлять и получать сигналы.

Использование механизма сигналов и слотов дает программисту следующие преимущества:

- каждый класс, унаследованный от QObject, может иметь любое количество сигналов и слотов;
- сообщения, посылаемые посредством сигналов, могут иметь множество аргументов любого типа;
- сигнал можно соединять с различным количеством слотов. Отправляемый сигнал поступит ко всем подсоединенным слотам;
- слот может принимать сообщения от многих сигналов, принадлежащих разным объектам;
- соединение сигналов и слотов можно производить в любой точке приложения;
- сигналы и слоты являются механизмами, обеспечивающими связь между объектами. Более того, эта связь может выполняться между объектами, которые находятся в различных потоках;
- при уничтожении объекта происходит автоматическое разъединение всех сигнально- слотовых связей. Это гарантирует, что сигналы не будут отправляться к несуществующим объектам.

Нельзя не упомянуть и о недостатках, связанных с применением сигналов и слотов:

- сигналы и слоты не являются частью языка C++, поэтому требуется запуск дополнительного препроцессора перед компиляцией программы;
- отправка сигналов происходит немного медленнее, чем обычный вызов функции, который осуществляется при использовании механизма функций обратного вызова;
- существует необходимость в наследовании класса QObject;
- в процессе компиляции не производится никаких проверок: имеется ли сигнал или слот в соответствующих классах или нет; совместимы ли сигнал и слот друг с другом и могут ли они быть соединены вместе. Об ошибке станет известно лишь тогда, когда приложение будет запущено в отладчике или на консоли. Вся эта информация выводится на консоль, поэтому, для того чтобы увидеть ее в Windows, в проектном файле необходимо в секции config добавить опцию console (для Mac OS X и Linux никаких дополнительных изменений проектного файла не требуется).

14.4. Сигналы

В программировании с использованием Qt под понятием сигналы подразумеваются методы, которые в состоянии осуществлять пересылку сообщений.

Причиной для появления сигнала может быть сообщение об изменении состояния управляющего элемента — например, перемещение ползунка. На подобные изменения присоединенный объект, отслеживающий такие сигналы, может соответственно отреагировать. Соединяемые объекты могут быть абсолютно независимы и реализованы отдельно друг от друга. Такой подход позволяет объекту, отправляющему сигналы, не беспокоиться о том, что впоследствии будет происходить с этими сигналами. Благодаря такому разделению, можно разбить большой проект на компоненты, которые будут разрабатываться разными программистами по отдельности, а потом соединяться при помощи сигналов и слотов вместе. Это делает код очень гибким и легко расширяемым — если один из компонентов устареет или должен будет реализован иначе, то все другие компоненты и сам проект в целом, не изменятся. Новый компонент после разработки встанет на место старого и будет подключен к основной программе при помощи тех же самых сигналов и слотов. Это делает библиотеку Qt особенно привлекательной для реализации компонентно-ориентированных приложений. Однако не забывайте, что большое количество взаимосвязей приводит к возникновению сильно связанных систем, в которых даже незначительные изменения могут привести к непредсказуемым последствиям.

Сигналы определяются в классе, как и обычные методы, только без реализации. С точки зрения программиста они являются лишь прототипами методов, содержащихся в заголовочном файле определения класса. Всю дальнейшую заботу о реализации кода для этих методов берет на себя МОС. Методы сигналов не должны возвращать каких-либо значений, и поэтому перед именем метода всегда должен стоять возвращаемый параметр `void`.

Сигнал не обязательно соединять со слотом. Если соединения не произошло, то он просто не будет обрабатываться. Подобное разделение отправляющих и получающих объектов исключает возможность того, что один из подсоединенных слотов каким-то образом сможет помешать объекту, отправившему сигналы.

Библиотека предоставляет большое количество уже готовых сигналов для существующих элементов управления. В основном, для решения поставленных задач хватает этих сигналов, но иногда возникает необходимость реализации новых сигналов в своих классах.

Листинг Определение сигнала:

```
Q_OBJECT
```

```
signals:
```

```
void dolt();
```

/*Обратите внимание на метод сигнала `doit ()`. Он не имеет реализации — эту работу принимает на себя МОС, обеспечивая примерно такую реализацию:

```
void MySignal::dolt()
```

```
{
```

```
QMetaObject::activate(this, sstaticMetaObject, 0, 0);
```

Не имеет смысла определять сигналы как `private`, `protected` или `public`, поскольку они играют роль вызывающих методов.

Выслать сигнал можно при помощи ключевого слова `emit`. Ввиду того, что сигналы играют роль вызывающих методов, конструкция отправки сигнала `emit doit ()` приведет к обычному вызову метода `doit ()`. Сигналы могут отправляться из классов, которые их содержат. Например, в листинге сигнал `doit ()` может отсылаться только объектами класса `MySignal`. Чтобы иметь возможность отослать сигнал программно из объекта этого класса, следует добавить метод `sendsignal ()`, вызов которого заставит объект класса `MySignal` отправлять сигнал `doit ()`.

Листинг Реализация сигнала

```
Q_OBJECT
```

```
public:
```

```
void sendSignal() {
```

```
    emit dolt ();
```

```
}
```

```
signals:
```

```
void dolt();
```

Сигналы также имеют возможность высылать информацию, передаваемую в параметре. Например, если возникла необходимость передать в сигнале строку текста, то можно реализовать это, как показано в листинге.

Листинг Реализация сигнала с параметром

```

{
Q_OBJECT
public:
void sendSignal()
{
emit sendString("Information");
}
signals:
void sendString(const QString&);

```

Слоты

Слоты (slots) — это методы, которые присоединяются к сигналам. По сути, они являются обычными методами. Самое большое их отличие состоит в возможности принимать сигналы. Как и обычные методы, они определяются в классе как `public`, `private` или `protected`. Если необходимо сделать так, чтобы слот мог соединяться только с сигналами сторонних объектов, но не вызываться как обычный метод со стороны, то слот нужно объявить, как `protected` или `private`. Во всех других случаях слоты объявляются как `public`. В объявлениях перед каждой группой слотов должно стоять соответственно:

- `private slots:`
 - `protected slots:`
 - `public slots:`

Слоты могут быть и виртуальными.

Соединение сигнала с виртуальным слотом примерно в десять раз медленнее, чем с не виртуальным. Поэтому не стоит делать слоты виртуальными, если нет особой необходимости.

Правда, есть небольшие ограничения, отличающие обычные методы от слотов. В слотах нельзя использовать параметры по умолчанию — например: `slotMethod(int n = 8)`, или определять слоты как `static`.

Реализация слота

```

class MySlot : public QObject { Q_OBJECT public:
MySlot();
public slots:

```

```
void slot()
```

Внутри слота вызовом метода `sender ()` можно узнать, от какого объекта был выслан сигнал. Он возвращает указатель на объект типа `QObject`. Например, в этом случае на консоль будет выведено имя объекта, выславшего сигнал:

```
void slot ()  
{  
    qDebug() « sender()->objectName();
```

Соединение объектов

Соединение объектов осуществляется при помощи статического метода `connect ()`, который определен в классе `QObject`. В общем виде вызов метода `connect ()` выглядит следующим образом:

```
const char* signal,  
const QObject* receiver,  
const char* slot,  
Qt::ConnectionType type = Qt::Autoconnection );
```

Ему передаются пять следующих параметров:

- `sender` — указатель на объект, отправляющий сигнал;
- `signal` — это сигнал, с которым осуществляется соединение. Прототип (имя и аргументы) метода сигнала должен быть заключен в специальный макрос `signal (method ())`;
- `receiver` — указатель на объект, который имеет слот для обработки сигнала;
- `slot` — слот, который вызывается при получении сигнала. Прототип слота должен быть заключен в специальный макрос `slot (method ())`;
- `type` — управляет режимом обработки. Имеется три возможных значения:
 - сигнал обрабатывается сразу вызовом соответствующего метода слота;
 - сигнал преобразуется в событие и ставится в общую очередь для обработки;
 - это автоматический режим, который действует следующим образом: если отсылающий сигнал объект находится в одном потоке с принимающим его объектом, то устанавливается режим

Qt: : DirectConnection, В противном случае — режим

— этот режим (Qt: Autoconnection) определен в методе connection «вручную», но полезно знать, что такая возможность есть.

Существует альтернативный вариант метода connect (), преимущество которого заключается в том, что все ошибки соединения сигналов со слотами выявляются на этапе компиляции программы, а не при ее исполнении, как это происходит в классическом варианте метода connect (). Прототип альтернативного метода выглядит так:

```
QObject::connect(const QObject* sender, const QMetaMethods signal,
const QObject* receiver, const QMetaMethods slot,
Qt::ConnectionType type = Qt::Autoconnection );
```

Параметры этого метода полностью аналогичны предыдущему, за исключением тех параметров, которые были объявлены в предыдущем методе как const char*. Вместо них используются указатели на методы сигналов и слотов классов напрямую. Благодаря именно этому, если вы вдруг ошибетесь с названием сигнала или слота, ваша ошибка будет выявлена сразу в процессе компиляции программы. К недостаткам альтернативного метода можно отнести то, что при каждом соединении нужно явно указывать имена классов для сигнала и слота и следить за совпадением их параметров.

Следующий пример демонстрирует то, как может быть осуществлено соединение объектов в программе с помощью первого метода connect ().

```
void main()
QObject::connect(pSender, SIGNAL(signalMethod()), pReceiver,
SLOT(slotMethod()))
```

А вот так выглядит аналогичное соединение при помощи альтернативного метода

```
connect():
QObject::connect(pSender, &SenderClass::signalMethod, pReceiver,
&ReceiverClass::slotMethod );
```

В случае если слот содержится в классе, из которого производится соединение, то можно воспользоваться сокращенной формой метода connect (), опустив третий параметр (pReceiver), указывающий на объект-получатель.

```
MyClass::MyClass() : QObject()
{
```

```

connect(pSender, SIGNAL(signalMethod()), SLOT(slot())) ;
}
void MyClass::slot()
{
qDebugO « "I'm a slot";

```

Метод `connect ()` после вызова возвращает объект класса `Connection`, с помощью которого можно определить, произошло соединение успешно или нет. Этим обстоятельством можно воспользоваться, например, для того, чтобы в случае, если в методе будет допущена какая-либо ошибка, то аварийно завершить программу вызовом макроса `q assert ()`.

Класс `Connection` содержит оператор неявного преобразования к типу `bool`, поэтому используем в примере переменную `bool` этого типа. Подобный код может выглядеть следующим образом.

Иногда возникают ситуации, когда объект не обрабатывает сигнал, а просто передает его дальше. Для этого необязательно определять слот, который в ответ на получение сигнала (при помощи `emit`) отправляет свой собственный. Можно просто соединить сигналы друг с другом. Отправляемый сигнал должен содержаться в определении класса:

```

MyClass::MyClass() : QObject()
{
connect(pSender, SIGNAL(signalMethod()), SIGNAL(mySignal()));

```

Отправку сигналов можно на некоторое время заблокировать, вызвав метод `blocksignals()` с параметром `true`. Объект будет «молчать», пока блокировку не снимут тем же методом `blockSignals()` с параметром `false`. При помощи метода `signalsBlocked()` можно узнать текущее состояние блокировки сигналов.

Разъединение объектов

В Qt при уничтожении объекта все связанные с ним соединения уничтожаются автоматически, но в редких случаях может возникнуть необходимость в уничтожении этих соединений «вручную». Для этого существует статический метод `disconnect()`, параметры которого аналогичны параметрам статического метода `connect()`. В общем виде этот метод выглядит таким образом:

Следующий пример демонстрирует, как может быть выполнено разъединение объектов в программе:

```

void main()
QObject::disconnect(pSender, SIGNAL(signalMethod()), pReceiver,
SLOT(slotMethod()));

```

Существуют два сокращенных, не статических варианта: `disconnect (signal, receiver, slot)` и `disconnect (receiver, slot)`.

Переопределение сигналов

Если есть необходимость сократить в классе количество слот-методов и выполнить действия на разные сигналы в одном слоте, то следует воспользоваться классом `QSignalMapper`. С его помощью можно переопределить сигналы и сделать так, чтобы в слот отправлялись значения типов `int`, `QString` или `QWidget`.

```

MyClass::MyClass(QWidget* pwidget)
{
    QSignalMapper* pMapper = new QSignalMapper(this);
    connect(pMapper, SIGNAL(mapped(const
QString&)),this,SLOT(slotShowAction(const QString&)));
    QPushButton* pcmd1 = new QPushButton("Button1");
    connect(pcmd1, SIGNAL(clicked()), pMapper, SLOT(map()));
    pMapper->setMapping(pcmd1, "Button1 Action");
    QPushButton* pcmd2 = new QPushButton("Button2");
    connect(pcmd2, SIGNAL(clicked()), pMapper, SLOT(map()));
    pMapper->setMapping(pcmd2, "Button2 Action");
}

void MyClass::slotShowAction(const QString& str)

```

В примере мы создаем объект класса `QSignalMapper` и соединяем его сигнал `mapped ()` с единственным слотом `slotShowAction ()`, который принимает объекты `QString`. Класс `QSignalMapper` предоставляет слот `map ()`, с которым должен быть соединен каждый объект, сигнал которого должен быть переопределен. При помощи метода `QSignalMapper:: setMapping()` мы устанавливаем конкретное значение, которое должно быть направлено в слот при получении сигнала.

15. Стандартные библиотеки и проектирование основных интерфейсов приложения

Преимущества использования стандартных библиотек

Qt является уроссплатформенным набором библиотек и классов, однако для того чтобы реализовать по-настоящему кроссплатформенное приложение необходимо использовать только стандартные библиотеки и классы. Иначе при использовании различных сторонних библиотек или системных вызовов, написанное приложение сможет быть скомпилировано только под определенную платформу.

Реализация некоторых классов, например, классов работающих с различными устройствами, невозможна без использования системных вызовов. В таких случаях класс для всех платформ реализуется с единым интерфейсом, но с различной реализацией программного кода. Такая задача может быть реализована с помощью следующей конструкции:

```
//Код который выполнится при выполнении условия
```

```
//Код который выполнится при невыполнении условия
```

Рассмотрим пример программы, которая выводит название операционной системы в которой она была собрана:

```
#include <QCoreApplication>
#include <QDebug>

QString getOSName()
{
    #if defined(Q_OS_ANDROID)
        return QLatin1String("android");
    #elif defined(Q_OS_BLACKBERRY)
        return QLatin1String("blackberry");
    #elif defined(Q_OS_IOS)
        return QLatin1String("ios");
    #elif defined(Q_OS_MACOS)
        return QLatin1String("macos");
    #elif defined(Q_OS_TVOS)
        return QLatin1String("tvos");
```

```

#elif defined(Q_OS_WATCHOS)
return QLatin1String("watchos");
#elif defined(Q_OS_WINCE)
return QLatin1String("wince");
#elif defined(Q_OS_WIN)
return QLatin1String("windows");
#elif defined(Q_OS_LINUX)
return QLatin1String("linux");
#elif defined(Q_OS_UNIX)
return QLatin1String("unix");
#else
return QLatin1String("unknown");
#endif
}

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    qDebug()<<getOSName();
}

```

Как видно из исходного кода — это консольное приложение с использованием Qt, Класс `QCoreApplication` является основным классом консольного приложения, который содержит в себе основной цикл программы, обработчики событий и т.д. Функция `getOSName()` возвращает имя операционной системы в которой оно было собрано. В теле функции содержится набор условий, которые выполняются в случае совпадения типа операционной системы.

В случае выполнения данной программы в терминал будет выведено название операционной системы (рисунок 15.1).

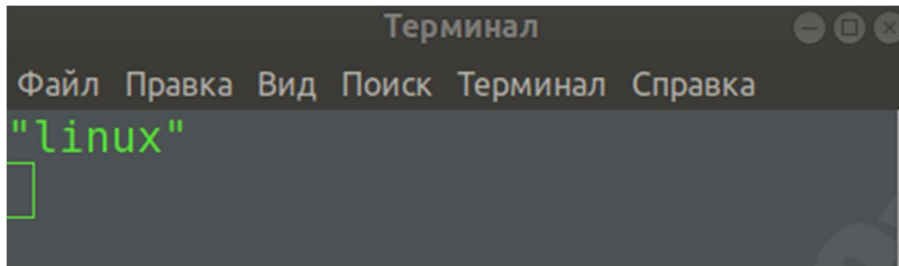


Рис. 15.1. Результат выполнения программы в ОС Ubuntu 18.04

Таким образом платформозависимый код должен помещаться в структуру условия с соответствующим типом операционной системы:

```
#if defined(Q_OS_WIN)
    /* Код который будет выполняться в операционных
       системах семейства Windows*/
#elif defined(Q_OS_LINUX)
    /* Код который будет выполняться в операционных
       системах семейства Linux*/

    /* Код который будет выполняться в операционных
       системах семейства MacOS*/
```

15.2. Иерархия классов стандартных библиотек

У программистов, начинающих изучение классов новой библиотеки, из-за большого объема информации, которую надо усвоить, зачастую создается ощущение перенасыщения. Но иерархия классов Qt имеет четкую внутреннюю структуру, которую важно сразу понять, чтобы потом уметь хорошо и интуитивно в этой библиотеке ориентироваться.

Библиотека Qt — это множество классов (более 500), которые охватывают большую часть функциональных возможностей операционных систем, предоставляя разработчику мощные механизмы, расширяющие и, вместе с тем, упрощающие разработку приложений. При этом не нарушается идеология операционной системы. Qt не является единым целым, она разбита на модули. Список основных модулей приведен в таблице 15.1.

Таблица 15.1. Основные модули

Библиотека	Обозначение в проектном файле	Назначение
QtCore	core	Основополагающий модуль, состоящий из классов, не связанных с графическим интерфейсом
QtGui	gui	Модуль базовых классов для программирования графического интерфейса
QtWidgets	widgets	Модуль, дополняющий QtGui «строительным материалом» для графического интерфейса в виде виджетов на C++
QtQuick	quick	Модуль, содержащий описательный фреймворк для быстрого создания графического интерфейса
QtQML	qml	Модуль, содержащий движок для языка QML и JavaScript
QtNetwork	network	Модуль для программирования сети
QtOpenGL	opengl	Модуль для программирования графики OpenGL
QtSql	sql	Модуль для программирования баз данных
QtSvg	svg	Модуль для работы с SVG (Scalable Vector Graphics, масштабируемая векторная графика)
QtXml	xml	Модуль поддержки XML, классы, относящиеся к SAX и DOM
QtXmlPatterns	xmlpatterns	Модуль поддержки XPath, XQuery, XSLT и XmlSchemaValidator
QtScript	script	Модуль поддержки языка сценариев
QtScriptTools	scripttools	Модуль дополнительных возможностей поддержки языка сценария. В настоящее время предоставляет отладчик
QtMultimedia	multimedia	Модуль мультимедиа
QtMultimediaWidgets	multimediawidgets	Модуль с виджетами для QtMultimedia
QtWebKit	webkit	Модуль для создания Web-приложений

QtWebKitWidgets	webkitwidgets	Модуль с виджетами для QtWebKit
QPrintSupport	printsupport	Модуль для работы с принтером
QtTest	test	Модуль, содержащий классы для тестирования кода

Любая Qt-программа так или иначе должна использовать хотя бы один из модулей QtCore, QtGui и Qtwidgets. Они присутствуют во всех программах с графическим интерфейсом и поэтому определены в программе создания make-файлов по умолчанию. Для использования других модулей в своих проектах необходимо перечислить их в проектном файле. Например, чтобы добавить модули, нужно написать:

Q

А чтобы исключить модуль из проекта:

Q

Наиболее значимый из перечисленных в таблице 1 модулей — это QtCore, так как он является базовым для всех остальных модулей. Далее идут модули, которые непосредственно зависят от QtCore, это: QtNetwork, QtSql и Qtxml. И, наконец, модули, зависящие от только что упомянутых модулей: QtOpenGL И QtSvg.

Для каждого модуля Qt предоставляет отдельный заголовочный файл, содержащий заголовочные файлы всех классов этого модуля. Название такого заголовочного файла соответствует названию самого модуля. Например, для включения модуля Qtwidgets нужно добавить в программу строку:

Пространство имен Qt

Пространство имен Qt содержит ряд типов перечислений и констант, которые часто применяются при программировании. Если вам необходимо получить доступ к какой-либо константе этого пространства имен, то вы должны указать префикс Qt (например, не red, а Qt:: red). Если вы все-таки хотите опускать префикс Qt, то необходимо в начале файла с исходным кодом добавить следующую директиву:

Модуль QtCore

Базовым является модуль QtCore. При этом он является базовым для приложений и не содержит классов, относящихся к интерфейсу

g
n
a
m
e
s

пользователя. Если необходимо реализовать консольное приложение, то, вполне возможно ограничиться одним этим модулем. В модуль QtCore входят более 200 классов, вот некоторые из них:

- контейнерные классы: QList, QVector, QMap, QVariant, QString и т. д.;
- классы для ввода и вывода: QiODevice, QTextstream, QFile ;
- к
- Q
- классы для работы с таймером: QBasicTimer и QTimer;
- классы для работы с датой и временем: QDate и QTime;
- класс QObject, являющийся краеугольным камнем объектной модели

- базовый класс событий QEvent;
- класс для сохранения настроек приложения QSettings;
- класс приложения QApplication, из объекта которого, если требуется, можно запустить цикл событий;
- классы поддержки анимации: QAbstractAnimation, QVariantAnimation и т. д.;
- классы для машины состояний: QStateMachine, QState и т. д.;
- к

Модуль содержит так же механизмы поддержки файлов ресурсов.

Модуль QtGui

Этот модуль предоставляет классы интеграции с оконной системой, с OpenGL и OpenGL ES. Он содержит класс QWindow, который является элементарной областью с возможностью получения событий пользовательского ввода, изменения фокуса и размеров, а также позволяющей производить графические операции и рисование на своей поверхности.

Модуль QtWidgets

Этот модуль содержит в себе классы виджетов, представляющие собой «строительный материал» для программирования графического интерфейса пользователя. В этот модуль входят около 300 классов. Вот некоторые из них:

- класс QWidget — это базовый класс для всех элементов управления библиотеки Qt. По своему внешнему виду он не что иное, как заполненный четырехугольник, но за этой внешней простотой скрывается большой потенциал непростых функциональных возможностей. Этот класс насчитывает 254 метода и 53 свойства. В главе 5 ему уделено особое внимание;

Q

A

b

s

t

r

a

- классы для автоматического размещения элементов: `QVBoxLayout`,
- классы элементов отображения: `QLabel`, `QLCDNumber`);
- классы кнопок: `QPushButton`, `QCheckBox`, `QRadioButton`;
- классы элементов установок: `QSlider`, `QScrollBar`;
- классы элементов ввода: `QLineEdit`, `QSpinBox`;
- классы элементов выбора: `QComboBox`, `QToolBox`;
- классы меню: `QMainWindow` и `QMenu`;
- классы окон сообщений и диалоговых окон: `QMessageBox`, `QDialog`;
- классы для рисования: `QPainter`, `QBrush`, `QPen`, `QColor`;
- классы для растровых изображений: `QImage`, `QPixmap`;
- классы стилей (см. главу 26) — отдельному элементу, так и всему приложению может быть присвоен определенный стиль, изменяющий их внешний облик.

Модули `QtQuick` и `QtQML`

Это альтернатива виджетам — модули, представляющие собой набор технологий для быстрой разработки графических интерфейсов нового поколения на базе описательного языка `QML`, языка программирования `JavaScript` и всех остальных возможностей библиотеки `Qt`.

Модуль `QtNetwork`

Сетевой модуль `QtNetwork` предоставляет инструментарий для программирования TCP- и UDP-сокетов (классы `QTcpSocket` и `QUdpSocket`), а также для реализации программ-клиентов, использующих HTTP- и FTP-протоколы (класс `QNetworkAccessManager`).

Модули `QtXml` и `QtXmlPatterns`

Модуль `QtXml` предназначен для работы с базовыми возможностями XML посредством SAX2- и DOM-интерфейсов, которые определяют классы `Qt`. А модуль `QtXmlPatterns` идет дальше и предоставляет поддержку для дополнительных технологий XML — таких как: `XPath`, `XQuery`, `XSLT` и

Модуль `QtSql`

Этот модуль предназначен для работы с базами данных. В него входят классы, предоставляющие возможность для манипулирования значениями баз данных.

Модуль `QtOpenGL`

Модуль QtOpenGL делает возможным использование OpenGL в Qt-программах для двух- и трехмерной графики. Основным классом этого модуля является QGLWidget, который унаследован от QWidget.

Модули QtWebKit и QtWebKitWidgets

Модуль QtWebKit позволяет очень просто интегрировать в приложение возможности Web. А модуль QtwebKitewidgets предоставляет готовые к интеграции в приложение элементы в виде виджетов с возможностью также расширять элементы Web своими собственными виджетами.

Модули QtMultimedia и QtMultimediaWidgets

Модуль QtMultimedia обладает всем необходимым для создания приложений с поддержкой мультимедиа. Он поддерживает как низкий уровень, необходимый для более детальной специализированной реализации, так и высокий уровень, делающий возможным проигрывать видео- и звуковые файлы при помощи всего нескольких строк программного кода. Модуль QtMultimediaWidgets содержит полезные элементы в виде виджетов, которые позволяют экономить время для реализации.

Модули QtScript и QtScriptTools

Модуль QtScript предоставляет возможности расширения и изменения уже написанных приложений, при помощи языка сценариев JavaScript. Модуль QtScriptTools обеспечивает средства отладки для программ сценариев. Эти модули подробно описывает часть VII.

Модуль QtSvg

Модуль поддержки графического векторного формата SVG, базирующегося на XML. Этот формат предоставляет возможность не только для вывода одного кадра векторного изображения, но может быть использован и для векторной анимации.

Понятие объекта приложения.

Объект класса приложения QCoreApplication можно образно сравнить с сосудом, содержащим объекты, подсоединенные к контексту операционной системы. Срок жизни объекта класса QCoreApplication соответствует продолжительности работы всего приложения, и он остается доступным в любой момент работы программы. Объект класса QCoreApplication должен создаваться в приложении только один раз. К задачам этого объекта можно отнести:

- управление событиями между приложением и операционной системой;

- передачу и предоставление аргументов командной строки.

Кроме того, `QCoreApplication` можно унаследовать, чтобы перезаписать некоторые методы, а также задействовать сам объект для дополнительных глобальных данных, используемых внутри приложения. Такой подход может избавить вас от нежелательного использования шаблона проектирования Singleton.

Класс `QApplication` является наследником `QCoreApplication`, поэтому все, что было сказано про `QCoreApplication`, относится и к `QApplication`. Объект класса `QApplication` представляет собой центральный контрольный пункт Qt-приложений, имеющих пользовательский интерфейс на базе виджетов. Этот объект используется для получения событий клавиатуры, мыши, таймера и других событий, на которые приложение должно реагировать соответствующим образом. Например, окно даже самого простого приложения может быть изменено по величине или быть перекрыто окном другого приложения, и на все подобные события необходима правильная реакция. Основные методы `QCoreApplication` приведены в таблице 15.2.

Класс `QApplication` напрямую унаследован от `QGuiApplication` и дополняет его следующими возможностями:

- установка стиля приложения. Таким способом можно устанавливать виды и поведения (Look & Feel) приложения, включая и свои собственные;
- получение указателя на объект рабочего стола;
- управление глобальными манипуляциями с мышью (например, установка интервала двойного щелчка кнопкой мыши) и регистрация движения мыши в пределах и за пределами окна приложения;
- обеспечение правильного завершения работающего приложения при завершении работы операционной системы.

Основные методы `QGuiApplication` приведены в таблице 15.3.

Таблица 15.2. Основные методы `QCoreApplication`.

Метод	Описание
<code>QCoreApplication(int &argc, char **argv)</code>	Конструктор класса. В качестве входных параметров передается количество входных аргументов и собственно сам список аргументов
<code>void quit()</code>	Выход из приложения(завершение основного цикла приложения)

<code>void addLibraryPath(const QString &path)</code>	Добавления пути к библиотекам. Необходимо при изменении стандартного пути к динамическим библиотекам
<code>QString applicationDirPath()</code>	Возвращает путь к папке в которой содержится исполняемый файл
<code>QString applicationFilePath()</code>	Возвращает путь к исполняемому файлу
<code>QString applicationName()</code>	Возвращает имя исполняемого файла
<code>qint64 applicationPid()</code>	Взвращает ID процесса данного приложения
<code>QString applicationVersion()</code>	Возвращает версию приложения
<code>QStringList arguments()</code>	Возвращает список входных аргументов
<code>int exec()</code>	Запуск основного цикла приложения
<code>bool installTranslator(QTranslator *translationFile)</code>	Установка перевода
<code>QStringList libraryPaths()</code>	Возвращает пути к библиотекам
<code>void removeLibraryPath(const QString &path)</code>	Удаляет указанный путь из списка путей к библиотекам
<code>bool removeTranslator(QTranslator *translationFile)</code>	Удаляет перевод интерфейса
<code>void setApplicationName(const QString &application)</code>	Устанавливает имя приложения
<code>void setApplicationVersion(const QString &version)</code>	Устанавливает версию приложения
<code>void setLibraryPaths(const QStringList &paths)</code>	Устанавливает список путей к библиотекам

Таблица 15.3. Основные методы `QGuiApplication`

Метод	Описание
<code>QGuiApplication(int &argc, char **argv)</code>	Конструктор класса. В качестве входных параметров передается количество входных аргументов и собственно сам список аргументов

QWindowList allWindows()	Возвращает список окон приложения
QString applicationDisplayName()	Возвращает отображаемое имя приложения
Qt::ApplicationState applicationState()	Возвращает текущее состояние приложения
QClipboard *clipboard()	Возвращает указатель на объект буфера обмена
int exec()	Запуск основного цикла приложения
QObject *focusObject()	Возвращает указатель на объект, который сейчас находится в фокусе
QWindow *focusWindow()	Возвращает указатель на окно, находящееся в фокусе
QFont font()	Возвращает шрифт приложения
Qt::MouseButton mouseButtons()	Возвращает состояние кнопок мыши
QScreen *screenAt(const QPoint &point)	Возвращает указатель на объект экрана по точке
QList<QScreen *> screens()	Возвращает список экранов
void setApplicationDisplayName(const QString &name)	Устанавливает отображаемое имя приложения
void setFont(const QFont &font)	Устанавливает шрифт приложения
void setWindowIcon(const QIcon &icon)	Устанавливает иконку основного окна приложения
QIcon windowIcon()	Возвращает установленную иконку основного окна приложения

Таблица 15.4. Основные сигналы QApplication

Сигнал	Описание
void applicationDisplayNameChanged()	Отображаемое имя приложения изменено
void applicationStateChanged(Qt::ApplicationState state)	Состояние приложения изменено
void focusObjectChanged(QObject *focusObject)	Изменился объект в фокусе

void focusWindowChanged(QWindow *focusWindow)	Изменился фокус окна
void fontChanged(const QFont &font)	Изменился шрифт
void primaryScreenChanged(QScreen *screen)	Изменился основной экран
void screenAdded(QScreen *screen)	Экран добавлен
void screenRemoved(QScreen *screen)	Экран удален

. Потокковая многозадачность

.1. Многозадачность, основанная на процессах

Многозадачность (англ. multitasking) — свойство операционной системы или среды выполнения обеспечивать возможность параллельной (или псевдопараллельной) обработки нескольких задач. Истинная многозадачность операционной системы возможна только в распределённых вычислительных системах.

Существует 2 типа многозадачности:

- Процессная многозадачность (основанная на процессах — одновременно выполняющихся программах). Здесь программа — наименьший элемент управляемого кода, которым может управлять планировщик операционной системы. Более известна большинству пользователей (работа в текстовом редакторе и прослушивание музыки);
- Поточная многозадачность (основанная на потоках). Наименьший элемент управляемого кода — поток (одна программа может выполнять 2 и более задачи одновременно).

Многопоточность — специализированная форма многозадачности. Свойства многозадачной среды. Прimitивные многозадачные среды обеспечивают чистое «разделение ресурсов», когда за каждой задачей закрепляется определённый участок памяти, и задача активизируется в строго определённые интервалы времени.

Более развитые многозадачные системы проводят распределение ресурсов динамически, когда задача стартует в памяти или покидает память в зависимости от её приоритета и от стратегии системы. Такая многозадачная среда обладает следующими особенностями:

- Каждая задача имеет свой приоритет, в соответствии с которым получает процессорное время и память;
- Система организует очереди задач так, чтобы все задачи получили ресурсы, в зависимости от приоритетов и стратегии системы;
- Система организует обработку прерываний, по которым задачи могут активироваться, деактивироваться и удаляться;
- По окончании положенного кванта времени ядро временно переводит задачу из состояния выполнения в состояние готовности, отдавая ресурсы другим задачам. При нехватке памяти страницы невыполняющихся задач могут быть вытеснены на диск (своппинг), а

потом, через определённое системой время, восстанавливаться в памяти;

- Система обеспечивает защиту адресного пространства задачи от несанкционированного вмешательства других задач;
- Система обеспечивает защиту адресного пространства своего ядра от несанкционированного вмешательства задач;
- Система распознаёт сбои и зависания отдельных задач и прекращает их;
- Система решает конфликты доступа к ресурсам и устройствам, не допуская тупиковых ситуаций общего зависания от ожидания заблокированных ресурсов;
- Система гарантирует каждой задаче, что рано или поздно она будет активирована;
- Система обрабатывает запросы реального времени;
- Система обеспечивает коммуникацию между процессами.

Типы псевдопараллельной многозадачности

Простое переключение

Тип многозадачности, при котором операционная система одновременно загружает в память два или более приложений, но процессорное время предоставляется только основному приложению.

Преимущества: можно задействовать уже работающие программы, написанные без учёта многозадачности.

Недостатки: невозможно в неинтерактивных системах, работающих без участия человека. Взаимодействие между программами крайне ограничено.

Совместная или кооперативная многозадачность.

Тип многозадачности, при котором следующая задача выполняется только после того, как текущая задача явно объявит себя готовой отдать процессорное время другим задачам.

Преимущества кооперативной многозадачности: отсутствие необходимости защищать все разделяемые структуры данных объектами типа критических секций и мьютексов, что упрощает программирование, особенно перенос кода из однозадачных сред в многозадачные.

Недостатки: неспособность всех приложений работать в случае ошибки в одном из них, приводящей к отсутствию вызова операции «отдать процессорное время». Крайне затрудненная возможность реализации многозадачной архитектуры ввода-вывода в ядре ОС, позволяющей

процессору исполнять одну задачу в то время, как другая задача инициировала операцию ввода-вывода и ждет её завершения.

Вытесняющая, или приоритетная, многозадачность.

Вид многозадачности, в котором операционная система сама передает управление от одной выполняемой программы другой в случае завершения операций ввода-вывода, возникновения событий в аппаратуре компьютера, истечения таймеров и квантов времени, или же поступлений тех или иных сигналов от одной программы к другой. В этом виде многозадачности процессор может быть переключен с исполнения одной программы, на исполнение другой без всякого пожелания первой программы и, буквально, между любыми двумя инструкциями в её коде. Распределение процессорного времени осуществляется планировщиком процессов. К тому же каждой задаче может быть назначен пользователем или самой операционной системой определенный приоритет, что обеспечивает гибкое управление распределением процессорного времени между задачами (например, можно снизить приоритет ресурсоёмкой программе, снизив тем самым скорость её работы, но повысив производительность фоновых процессов). Этот вид многозадачности обеспечивает более быстрый отклик на действия пользователя.

Преимущества: возможность полной реализации многозадачного ввода-вывода в ядре ОС, когда ожидание завершения ввода-вывода одной программой позволяет процессору тем временем исполнять другую программу; сильное повышение надежности системы в целом, в сочетании с использованием защиты памяти — идеал в виде «ни одна программа пользовательского режима не может нарушить работу ОС в целом» становится достижимым хотя бы теоретически, вне вытесняющей многозадачности он не достижим, даже в теории. Возможность полного использования многопроцессорных и многоядерных систем.

Недостатки: необходимость особой дисциплины при написании кода, особые требования к его реентерабельности, к защите всех разделяемых и глобальных данных объектами типа критических секций и мьютексов.

Многозадачность, основанная на процессах и потоках.

Множественные нити исполнения в одном процессе называют потоками и это базовая единица загрузки ЦПУ, состоящая из идентификатора потока, счетчика, регистров и стека. Потоки внутри одного процесса делят секции кода, данных, а также различные ресурсы: описатели открытых файлов, учетные данные процесса сигналы, значения `umask`, `nice`,

таймеры и прочее. На рисунке 16.1 приведены структуры однопоточного и многопоточного приложения.

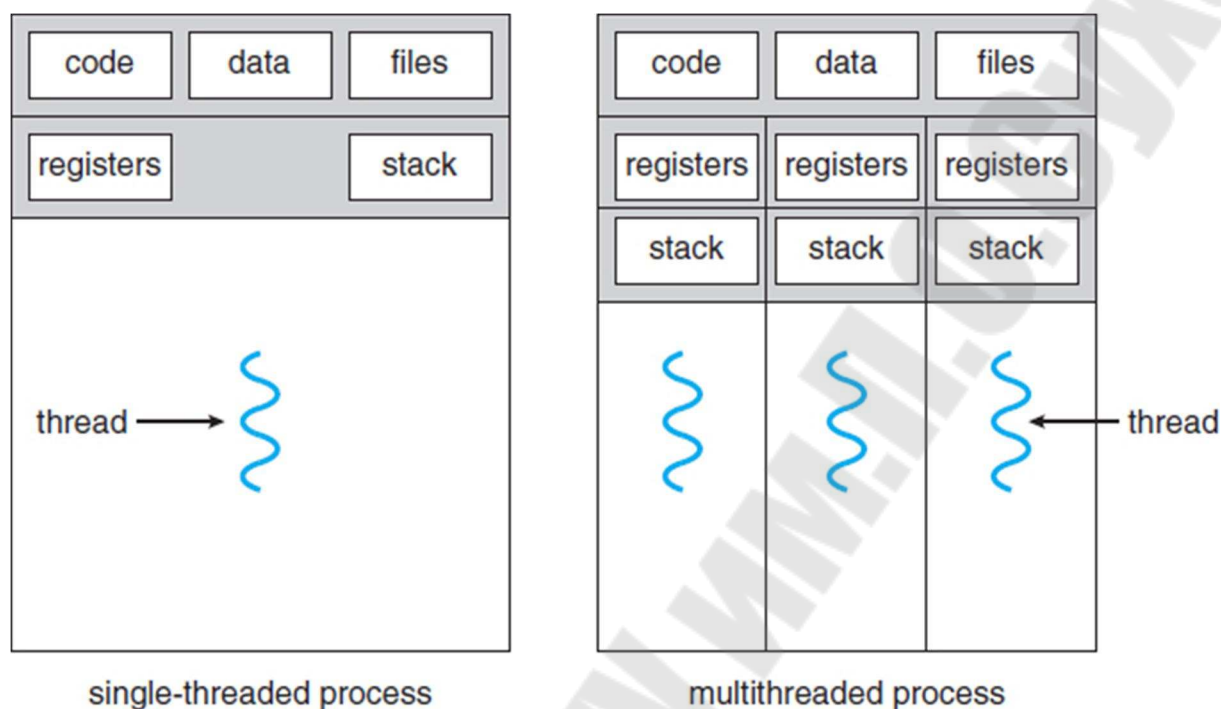


Рисунок 16.1. Однопоточное и многопоточное приложение.

У всех исполняемых процессов есть как минимум один поток исполнения. Некоторые процессы этим и ограничиваются, в тех случаях, когда дополнительные нити исполнения не дают прироста производительности, но только усложняют программу. Однако таких программ с каждым днем становится относительно меньше.

Для наглядного примера стоит рассмотреть http сервер, с которым одновременно работают несколько клиентов.

Если бы сервер создавал отдельный процесс для обслуживания каждого http запроса, мы бы ожидали вечно пока загрузится наша страница. Создания нового процесса — дорогостоящее удовольствие для ОС. Даже учитывая оптимизацию за счет копирования при записи.

Потоки в стандартных библиотеках

Потоки POSIX.

В конце 1980-х и начале 1990-х было несколько разных API, но в 1995 г. POSIX.1c стандартизовал потоки POSIX, позже это стало частью спецификаций SUSv3.

Pthreads определяет набор типов и функций на Си:

- `pthread_t` — идентификатор потока;
- `pthread_mutex_t` — мютекс;
- `pthread_mutexattr_t` — объект атрибутов мютекса;
- `pthread_cond_t` — условная переменная;
- `pthread_condattr_t` — объект атрибута условной переменной;
- `pthread_key_t` — данные, специфичные для потока;
- `pthread_once_t` — контекст контроля динамической инициализации;
- `pthread_attr_t` — перечень атрибутов потока.

В начале создается потоковая функция. Затем новый поток создается функцией `pthread_create()`, объявленной в заголовочном файле `pthread.h`. Далее, вызывающая сторона продолжает выполнять какие-то свои действия параллельно потоковой функции.

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void *(*start)(void *), void *arg);
```

При удачном завершении `pthread_create()` возвращает код 0, ненулевое значение сигнализирует об ошибке.

Параметры функции `pthread_create()`:

- Первый параметр является адресом для хранения идентификатора создаваемого потока типа `pthread_t`;
- Аргумент `start` является указателем на потоковую `void *` функцию, принимающей бестиповый указатель в качестве единственной переменной;
- Аргумент `arg` — это бестиповый указатель, содержащий аргументы потока. Чаще всего `arg` указывает на глобальную или динамическую переменную, но если вызываемая функция не требует наличия аргументов, то в качестве `arg` можно указать `NULL`;
- Аргумент `attr` также является бестиповым указателем атрибутов потока `pthread_attr_t`. Если этот аргумент равен `NULL`, то поток создается с атрибутами по умолчанию;

Пример много поточного приложения:

```
int count; /* общие данные для потоков */
int atoi(const char *nptr);
```

```

void *potok(void *param); /* потоковая функция */

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

    pthread_t tid; /* идентификатор потока */
    pthread_attr_t attr; /* атрибуты потока */

    if (argc != 2) {
        fprintf(stderr, "usage: progtest <integer value>\n");
        return -1;
    }

    if (atoi(argv[1]) < 0) {
        fprintf(stderr, "Аргумент %d не может быть
отрицательным                числом\n", atoi(argv[1]));

        /* получаем дефолтные значения атрибутов */
        pthread_attr_init(&attr);

        /* создаем новый поток */
        pthread_create(&tid, &attr, potok, argv[1]);

        /* ждем завершения исполнения потока */

        printf("count = %d\n", count);

        /* Контроль переходит потоковой функции */

        {
            int i, upper = atoi(param);
            count = 0;

            if (upper > 0) {
                for (i = 1; i <= upper; i++)
                    count += i;
            }
        }
    }

```

```
    pthread_exit(0);  
}
```

Чтобы подключить библиотеку Pthread к программе, нужно передать компоновщику опцию `-lpthread`.

```
gcc -o progtest -std=c99 -lpthread progtest.c
```

Строка `pthread_t tid` задает идентификатор потока. Атрибуты функции задает `pthread_attr_t attr`. Так как они не заданы явно, будут использованы значения по умолчанию.

Поток завершает выполнение задачи в следующих случаях:

- потоковая функция выполняет `return` и возвращает результат произведенных вычислений;
- в результате вызова завершения исполнения потока `pthread_exit()`;
- в результате вызова отмены потока `pthread_cancel()`;
- одна из нитей совершает вызов `exit()`
- основная нить в функции `main()` выполняет `return`, и в таком случае все нити процесса резко сворачиваются.

Синтаксис функции `pthread_exit`:

```
#include <pthread.h>  
void pthread_exit(void *retval);
```

Функция `pthread_join()` ожидает завершения потока обозначенного `D_ID`. Если этот поток к тому времени был уже завершен, то функция немедленно возвращает значение. Смысл функции в том, чтобы синхронизировать потоки. Она объявлена в `pthread.h` следующим образом:

```
#include <pthread.h>  
int pthread_join (pthread_t THREAD_ID, void ** DATA);
```

При удачном завершении `pthread_join()` возвращает код 0, ненулевое значение сигнализирует об ошибке.

Если указатель `DATA` отличается от `NULL`, то туда помещаются данные, возвращаемые потоком через функцию `pthread_exit()` или через инструкцию `return` потоковой функции. Несколько потоков не могут ждать завершения одного. Если они пытаются выполнить это, один поток завершается успешно, а все остальные — с ошибкой `ESRCH`. После

завершения `pthread_join()`, пространство стека связанное с потоком, может быть использовано приложением.

В каком-то смысле `pthread_join()` похожа на вызов `waitpid()`, ожидающую завершения исполнения процесса, но с некоторыми отличиями. Во-первых, все потоки одноранговые, среди них отсутствует иерархический порядок, в то время как процессы образуют дерево и подчинены иерархии родитель — потомок. Поэтому возможно ситуация, когда поток А, породил поток Б, тот в свою очередь заделал В, но затем после вызова функции `pthread_join()` А будет ожидать завершения В или же наоборот. Во-вторых, нельзя дать указание одному ожидать завершения любого потока, как это возможно с вызовом `waitpid(-1, &status, options)`. Также невозможно осуществить неблокирующий вызов `pthread_join()`.

Точно так же, как при управлении процессами, иногда необходимо досрочно завершить процесс, многопоточной программе может понадобиться досрочно завершить один из потоков. Для досрочного завершения потока можно воспользоваться функцией `pthread_cancel`.

При удачном завершении `pthread_cancel()` возвращает код 0, ненулевое значение сигнализирует об ошибке.

Важно понимать, что несмотря на то, что `pthread_cancel()` возвращается сразу и может завершить поток досрочно, ее нельзя назвать средством принудительного завершения потоков. Дело в том, что поток не только может самостоятельно выбрать момент завершения в ответ на вызов `pthread_cancel()`, но и вовсе его игнорировать. Вызов функции `pthread_cancel()` следует рассматривать как запрос на выполнение досрочного завершения потока. Поэтому, если важно, чтобы поток был удален, нужно дождаться его завершения функцией `pthread_join()`.

Небольшая иллюстрация создания и отмены потока:

```
/* создание потока */
pthread_create(&tid, 0, worker, NULL);
...
/* досрочное завершение потока */
```

Чтобы не создалось впечатление, что тут царит произвол и непредсказуемость результатов данного вызова, рассмотрим таблицу параметров, которые определяют поведение потока после получения вызова на досрочное завершение. Параметры потоков приведены в таблице 16.1.

Таблица 16.1. Параметры потоками

Mode	State	Type
Off	Disabled	-
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

Как видно из таблицы есть вовсе неотменяемые потоки, а поведением по умолчанию является отложенное завершение, которое происходит в момент завершения. Поэтому может узнать о необходимости завершения вызвав функцию `pthread_testcancel()`.

```
/* чего-то там делаем */  
/* пам-парам-пам-пам */  
/* не пора-ли сворачиваться? */
```

Мьютексы

Рассмотрим простой пример: несколько потоков обращаются к одной общей переменной. Часть потоков эту переменную увеличивают (`plus` потоки), а часть уменьшают на единицу (`minus` потоки). Число `plus` и `minus` потоков равно. Таким образом, мы ожидаем, что к концу работы программы значение исходной переменной будет прежним.

```
#include <stdio.h>  
#include <stdlib.h>  
#include <conio.h>  
#include <pthread.h>  
  
static int counter = 0;
```



```

void* minus(void *args) {
    int local;

    local = counter;
    printf("min %d\n", counter);
    local = local - 1;
    counter = local;
    return NULL;
}

void* plus(void *args) {
    int local;

    local = counter;
    printf("pls %d\n", counter);
    local = local + 1;
    counter = local;
    return NULL;
}

#define NUM_OF_THREADS 100

int main() {
    pthread_t threads[NUM_OF_THREADS];
    size_t i;

    printf("counter = %d\n", counter);
    for (i = 0; i < NUM_OF_THREADS/2; i++) {
        pthread_create(&threads[i], NULL, minus, NULL);
    }
    for (; i < NUM_OF_THREADS; i++) {
        pthread_create(&threads[i], NULL, plus, NULL);
    }
    for (i = 0; i < NUM_OF_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    printf("counter = %d", counter);
}

```

```
_getch());
```

Если выполнить код, то он будет возвращать различные значения. Чаще всего они не будут равны нулю. Разберёмся, почему так происходит. Рассмотрим код функций, которые выполняются в отдельных потоках:

```
void* minus(void *args) {  
    int local;  
  
    local = counter;  
    printf("min %d\n", counter);  
    local = local - 1;  
    counter = local;  
    return NULL;  
}
```

Во-первых, имеется локальная переменная `local`. Во-вторых, используется тяжёлая и медленная функция `printf`. В тот момент, когда происходит присваивание локальной переменной значение `counter`, другой поток может в то же самое время взять это значение и поменять.

Для простоты стоит рассмотреть 4 потока – два `plus` и два `minus` (таблица 16.2).

Таблица 16.2. Одно из возможных поведений программы

Действие					
plus 1 помещает в <code>local</code> значение <code>counter</code>					
plus 2 помещает в <code>local</code> значение <code>counter</code>					
plus 1 инкрементирует <code>local</code> и выводит значение на печать					
minus 1 помещает в <code>local</code> значение					
plus 1 помещает в переменную <code>counter</code> значение <code>local</code>					
minus 2 помещает в <code>local</code> значение					

plus 2 инкрементирует local и выводит значение на печать					
plus 2 помещает в переменную counter значение local					
minus 2 декрементирует local и выводит значение на печать					
minus 2 помещает в переменную counter значение local					
minus 1 декрементирует local и выводит значение на печать					
minus 1 помещает в переменную counter значение local					

Это один из возможных сценариев развития событий. Очевидно, что могут быть значения от минус 2 до плюс 2. Какой из них будет выполнен, в общем случае не известно.

Проблема заключается в том, что имеется несинхронизированный доступ к общему ресурсу. Необходимо сделать так, чтобы на время работы с ресурсом (всё тело функций `minus` и `plus`) к ним имел доступ только один поток, а остальные ждали, пока ресурс освободится. Это так называемое `mutual exclusion` – взаимное исключение, случай, когда необходимо удостовериться в том, что два (и более...) конкурирующих потока не находятся в критической секции кода одновременно.

В библиотеке `threads` один из методов разрешить эту ситуацию – это мьютексы. Мьютекс – это объект, который может находиться в двух состояниях. Он либо заблокирован (занят, залочен, захвачен) каким-то потоком, либо свободен. Поток, который захватил мьютекс, работает с участком кода. Остальные потоки, когда достигают мьютекса, ждут его разблокировки. Разблокировать мьютекс может только тот поток, который его захватил. Обычно освобождение занятого мьютекса происходит после исполнения критичного к совместному доступу участка кода.

Мьютекс – это экземпляр типа `pthread_mutex_t`. Перед использованием необходимо инициализировать мьютекс функцией

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                      const pthread_mutexattr_t *attr);
```

где первый аргумент – указатель на мьютекс, а второй – атрибуты мьютекса. Если указан NULL, то используются атрибуты по умолчанию. В случае удачной инициализации мьютекс переходит в состояние «инициализированный и свободный», а функция возвращает 0. Повторная инициализация инициализированного мьютекса приводит к неопределённому поведению.

Если мьютекс создан статически и не имеет дополнительных параметров, то он может быть инициализирован с помощью макроса

После использования мьютекса его необходимо уничтожить с помощью функции:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

В результате функция возвращает 0 в случае успеха или может вернуть код ошибки.

После создания мьютекса он может быть захвачен с помощью функции

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

После этого участок кода становится недоступным остальным потокам – их выполнение блокируется до тех пор, пока мьютекс не будет освобождён. Освобождение должен провести поток, заблокировавший мьютекс, вызовом

Пример реализации:

```
#include <stdlib.h>
#include <conio.h>
#include <pthread.h>

static int counter = 0;
pthread_mutex_t mutex;

void* minus(void *args) {
```

```
//Блокировка: теперь к ресурсам имеет доступ только один
//поток, который владеет мьютексом. Он же единственный,
//кто может его разблокировать
```

```
local = counter;
    printf("min %d\n", counter);
    local = local - 1;
    counter = local;
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

```
void* plus(void *args) {
    int local;
    pthread_mutex_lock(&mutex);
    local = counter;
    printf("pls %d\n", counter);
    local = local + 1;
    counter = local;
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

```
#define NUM_OF_THREADS 100
```

```
int main() {
    pthread_t threads[NUM_OF_THREADS];
    size_t i;

    printf("counter = %d\n", counter);
    //Инициализация мьютекса
    pthread_mutex_init(&mutex, NULL);
    for (i = 0; i < NUM_OF_THREADS/2; i++) {
        pthread_create(&threads[i], NULL, minus, NULL);
    }
    for (; i < NUM_OF_THREADS; i++) {
        pthread_create(&threads[i], NULL, plus, NULL);
    }
}
```

```

}
for (i = 0; i < NUM_OF_THREADS; i++) {
    pthread_join(threads[i], NULL);
}
//Уничтожение мьютекса
pthread_mutex_destroy(&mutex);
printf("counter = %d", counter);
_getch();

```

При использовании мьютекса исполнение защищённого участка кода происходит последовательно всеми потоками, а не параллельно. Порядок доступа отдельных потоков не определён.

Поддержка потоков в Qt

Для реализации поток в Qt используются те же самые механизмы, однако существуют оболочки для более удобной работы: классы QThread и

Класс QThread предназначен для создания и управления потоками. Каждый объект данного класса по сути является отдельным потоком. Существует два варианта его использования:

- Создание экземпляра данного класса и перенаправление главного цикла другого класса, основанного на QObject, в данный поток с помощью метода moveToThread;
- Создание собственного класса на основе QThread и переопределение метода run.

В разных источниках ведется спор какой из данных методов является верным, а какой нет. Однако все зависит от ситуации. Первый метод хорошо использовать для готовых классов, либо классов, которые работают на основе главного цикла. Второй метод по своей сути аналогичен созданию потоков pthread, когда в отдельный поток необходимо поместить отдельную функцию, которая будет выполнять определенную задачу. В таблицах 16.3, 16.4 и 16.5 приведены основные сведения необходимые для использования класса QThread.

Таблица 16.3. Сигналы

Сигнал	Описание
	Вызывается при завершении работы потока
	Вызывается при запуске потока

Таблица 16.4. Слоты

Слот	Описание
	Завершение работы потока
<code>void start(Priority priority = InheritPriority)</code>	Запуск потока, в качестве входного параметра можно установить приоритет потока
	Принудительно (аварийное) завершение работы потока

Таблица 16.5. Приоритеты

Приоритет	Описание
	Самый низкий приоритет. Выполняется когда другие потоки не занимают время процессора
<code>QThread::LowestPriority</code>	Низкий приоритет
<code>QThread::LowPriority</code>	Выполняется реже, чем обычный приоритет
<code>QThread::NormalPriority</code>	Приоритет по умолчанию в операционной системе
<code>QThread::HighPriority</code>	Высокий приоритет. Выполняется чуть выше чем обычный
<code>Q</code>	Выполняется чаще чем высокий приоритет
	Выполняется как можно чаще
	Использует приоритет создающего потока

Класс `QMutex` так же является программной оболочкой для мьютексов, описанных выше. И так же предназначен для синхронизации доступа к общим критическим областям данных.

Метод `lock ()` класса `QMutex` выполняет блокировку ресурса. Для обратной операции существует метод `unlock ()`, который открывает закрытый ресурс для других потоков.

Класс `QMutex` также содержит метод `tryLock()`. Его можно использовать для того, чтобы проверить, заблокирован ресурс или нет. Этот метод не приостанавливает исполнение потока и возвращается немедленно — со значением `false`, если ресурс уже захвачен другим потоком, не ожидая его освобождения. В случае успешного захвата ресурса этот метод вернет

значение true, и это значит, что ресурс принадлежит потоку, и он вправе распоряжаться им по своему усмотрению. Основные открытые методы класса QThread приведены в таблице 16.6.

Таблица 16.6. Открытые методы класса QThread

Метод	Описание
	Конструктор класса. В качестве входного параметра может передаваться режим работы. По умолчанию он равен
bool isRecursive() const	Возвращает true в случае если установлен рекурсивный режим работы
void lock()	Блокирует ресурс
bool tryLock(int timeout = 0)	Проверяет заблокирован ли ресурс. В случае блокировки возвращает значение false. В качестве входного параметра можно передавать время ожидания освобождения ресурса.
v	Освобождение ресурса

Рекурсивный режим работы позволяет осуществлять многократную блокировку ресурса одним и тем же потоком. Для разблокировки ресурса необходимо вызвать метод unlock столько же раз, сколько до этого вызывался метод lock. Данный режим предназначен для тех случаев, когда один и тот же поток осуществляет доступ к заблокированной секции многократно, не разблокировав ее перед этим.