Лекция №3. Объектно-ориентированное программирование в C++

СОДЕРЖАНИЕ

1 ПАРАДИГМА И ПРИНЦИПЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ И ИХ РЕАЛИЗАЦИЯ В C++

Каждая веха в развитии программирования характеризовалась созданием методов и средств, позволяющих разработчикам создавать все более сложные программы. Появление структурного программирования (выделение в программе блоков, условий и циклов) позволило программисту выражать логику решения задачи, не заботясь о том, как именно она представляется командами процессора. Процедурное программирование (с использованием подпрограмм) дало возможность делить программу на законченные фрагменты, выражающие некоторое действие, и повторно использовать их. Решение задачи удалось представить в виде последовательности операций, манипулирующих данными. Однако, реальный мир состоит не из операций, а значит, формулировать в их терминах задачи и решения неудобно. В жизни же мы сталкиваемся с предметами (объектами), которые могут сами совершать действия или манипулировать другими объектами. Желание решать задачи в естественном виде, оперируя моделями объектов и привело к изобретению *объектно-ориентированного программирования (ООП),* определив его идею: «программирование — это моделирование».

1.1 Основополагающие понятия

Ключевыми понятиями ООП являются класс и объект. *Класс* — это модель категории вещей, определяющая их общие черты; например, класс «Человек» может включать рост, вес, возраст и т. д, которые есть у каждого человека. *Объект* — это модель конкретной сущности, то есть отдельный человек — это объект (класса «Человек»), он имеет свой конкретный рост, вес и возраст. Класс и объект в программировании — это тип данных и значение такого типа.

Черты объекта, определяемые классом, включают свойства и действия над ними. Например, рост человека — это его свойство, а действием может быть «расти», причем оно изменяет свойство «рост» того человека, который его совершает. В программировании свойства объекта называются *данными-членами (data-members),* или *полями (fields),* и подобны полям структуры. Действия, которые может совершать объект, называются его *функциями-членами (member functions),* или *методами (methods),* и в первом приближении могут быть мыслимы как функции, которые оперируют данным объектом.

Модель всегда является упрощенным представлением реального объекта, так как учитывает только те его свойства, которые существенны для решения задачи. Это называется абстракцией, и ООП призвано обеспечить этот мощный мыслительный инструмент при помощи «трех китов»: инкапсуляции, наследования и полиморфизма.

Инкапсуляция — это сокрытие внутренней структуры объекта от внешней среды. Наследование отражает тот факт, что категории вещей включают друг друга: человек — это частный случай млекопитающего, млекопитающее — это категория животных и т. д. Полиморфизм связан с наследованием и выражает принцип, что оперировать объектами категории и её подкатегорий можно одинаково, хотя действия они при этом могут выполнять по-разному.

1.2 Инкапсуляция

Рассмотрим последовательно применение ООП на основе класса `String`, представляющего строку символов:

```
1       #include <cstring>
2       class String {
3       public:
4            char *data;
5            size_t length;
6            void setData(const char *data);
7       };
8       void String::setData(const char *data) {
9            length = strlen(data);
10           this->data = new char[length + 1];
11           strcpy(this->data, data);
12      }
13      …
14      String string;
15      string.setData("Некоторая строка");
16      string.setData("Тоже строка, но другая");
17      delete [] string.data;
```

Объявление класса начинается с ключевого слова **`class`**. Ключевое слово **`public`** указывает, что члены класса, описанные после него, доступны извне объекта (как в Delphi). Далее следуют объявления полей `data` и `length`, а также метода `setData()` для задания символов строки.

К полям данного объекта из методов возможно обращаться напрямую (строка 9), в теле методов может использоваться ключевое слово **`this`**, аналогичное `Self` в Delphi (строка 10). Здесь это необходимо, чтобы различать поле объекта `data` и локальную

переменную `data`. Объект **`this`** является неизменяемым указателем, в данном случае, `String*` **`const`**. Реализация метода `setData()` располагается вне класса и требует *полного (qualified) имени метода,* включающего имя класса через оператор разрешения области видимости (`::`).

1.2.1 Ограничение доступа к членам объекта

Данная программа не является объектно-ориентированной. Прежде всего, внешний код имеет полный доступ к данным-членам объекта `data` и `length` и способен изменить их значения, минуя `setData()`. При этом нарушится согласованность (consistency) данных объекта: можно присвоить `data` указатель на некоторую строку, а `length` — произвольное значение вместо длины этой строки. Решить проблему можно инкапсуляцией, скрыв структуру объекта от внешнего кода:

```
1       #include <cstring>
2       class String {
3       private:
4            char *data;
5            size_t length;
6       public:
7            void reset() {
8                 data = 0;
9                 length = 0;
10           }
11           void setData(const char *data) { /* Реализация та же. */ }
12           void clear() {
13                if (data)
14                     delete [] data;
15                reset();
16           }
17           const char *getData() { return data; }
18           unsigned int getLength() { return length; }
19      };
20      …
21      String string;
22      string.reset();
23      string.setData("Какая-то строка");
24      string.clear();
```

В новой версии кода прямой доступ к данным извне класса запрещен при помощи *спецификаторов доступа (access specifier):* **`public`** открывает доступ к членам класса извне, **`private`** — запрещает, и он действует по умолчанию [1, раздел 11]. Длина строки доступна только для чтения при помощи метода `getLength()`, реализованного *на месте (inline)*. Такой способ реализации применяется для очень простых методов, как в примере.

В примерах лекции мы будем часто реализовывать inline и более длинные функции — сугубо для экономии места. Поле `data` не может быть перезаписано напрямую, а только при помощи методов `setData()` и `reset()`, а его значение может быть получено методом `getData()`. Метод `clear()` предназначен для очистки памяти под символы строки.

Методы могут обращаться к членам *любого* объекта данного класса, минуя спецификаторы доступа.

1.2.2 Жизненный цикл объекта. Конструктор и деструктор

Недостатком очередной реализации класса `String` является то, что объект не может обеспечить согласованность своего состояния на протяжении всего времени жизни: до вызова метода `reset()` метод `clear()` не может корректно работать. Очевидно, необходим способ объекту реагировать на свое создание независимо от внешнего кода. Эта возможность реализуется в C++ специальным методом — конструктором (constructor). Он автоматически вызывается при создании объекта после выделения под него памяти, чтобы можно было выполнить произвольную инициализацию [1, подраздел 12.1]:

```
1       class String {
2       private:
3            void reset() {
4                 data = 0;
5                 length = 0;
6            }
7       public:
8            String() {
9                 reset();
10           }
11           String(const char *initial) {
12                reset();
13                setData(initial);
14           }
15           void setData(const char *data) {
16                clear();
17                // Здесь — старая реализация.
18           }
19
20           …
21      };
22      …
23      String *first = new String();
24      String second("Начальное значение");
25      first->setData("Другое значение");
26      first->clear();
```

```
27        second.setData("Очередное значение");
28        second.clear();
29        delete first;
```

Теперь при создании (строка 9) объекта `first` будет автоматически вызван конструктор `String::String()`, а при создании объекта `second` — перегрузка конструктора `String::String(const char*)`, поэтому оба вызова `clear()` корректны. Заметим, что память под сам объект `first`, выделенный в куче, необходимо освобождать явно при помощи оператора **`delete`**. Также стало возможным в методе `setData()` автоматически освобождать ранее выделенную память, и более не нужно вызывать `clear()` перед каждым обращением к `setData()` во избежание утечек памяти. Метод `reset()` сделан закрытым, так как внешний код в нем не нуждается.

Однако, поскольку памятью, выделенной под символы строки, может распоряжаться только сам объект (говорят: *владеет* этой областью памяти), без вызова `clear()` она окажется утеряна, как только объект выйдет из области видимости. Можно отследить уничтожение объекта при помощи другого специального метода — деструктора [1, подраздел 12.4]:

```
1         class String {
2         public:
3               ~String() {
4                     clear();
5               }
6               …
7         };
8         …
9         String *first = new String();
10        String second("Начальное значение");
11        first->setData("Другое значение");
12        second.setData("Очередное значение");
13        delete first;
```

Теперь метод `clear()` будет автоматически вызываться из деструктора при уничтожении объектов. Деструктор не может иметь параметров и быть перегружен, так как вызывается всегда неявно.

Конструкторы и деструкторы позволяют реализовать автоматическое управление ресурсами, причем эффективно и своевременно, поскольку правила и моменты их вызова точно определены стандартом.

Доступ к конструкторам и деструктору не обязательно быть открытыми (**`public`**). Часто применяют закрытые конструкторы, чтобы запретить создание объектов из внешнего

кода, и тогда вызывают их из статических методов класса (см. 2.5) или функций-друзей (см. 2.2). Закрытые деструкторы могут использоваться с той же целью. С появлением в C++11 удаленных определений функций (см. 2.10), такое использование спецификаторов доступа потеряло актуальность.

Жизненный цикл объекта может включать также копирование, например, при передаче в функцию по значению, и перемещение, например, при возврате объекта из функции. В этих случаях новый объект (в который делается копирование или перемещение) создается особыми конструкторами, которые также можно определить. Подробнее о копировании и перемещении объектов см. 2.1 и далее по ссылкам.

1.2.3 Интерфейс и состояние объекта. Контракт класса

Говоря о парадигме ООП: «программирование — это моделирование», уместно вспомнить, что всякая модель характеризуется входными воздействиями и выходными величинами. Их набор показывает, как модель связана с внешней средой, каким образом можно передавать и получать информацию от модели, задает способы взаимодействия. Это называется *интерфейсом класса (interface)* и выражается как набор доступных извне данных и действий модели. Можно видеть, что именно инкапсуляция позволяет определить интерфейс, ограничив доступ к части объекта из соображений поддержания целостности.

Интерфейс устанавливает не только синтаксис взаимодействия с объектами (имена членов, параметры методов), но и семантику — ожидаемое поведение объекта. Например, для объекта класса «Строка» методом «Вырезать подстроку» гарантируется изменение строки известным образом (исключение части символов, изменение длины) при определенных условиях (нельзя вырезать символы с 20-го по 30-й из строки «строка»). Сочетание семантики, *условий (requirements)* и *гарантий (guarantees)* составляет *контракт (contract)* метода, а те же компоненты для объекта — контракт класса.

Инкапсуляция скрывает детали реализации объекта от внешнего кода. Наблюдаемую извне часть объекта называют его *состоянием (state)*. В общем случае, внешний код полностью абстрагируется от того, какие данные, помимо состояния, содержит объект. Поэтому иногда говорят, что инкапсуляция — это возможность изменения состояния объекта исключительно посредством его методов.

1.3 Наследование и полиморфизм

1.3.1 Наследование

Разработанный класс `String` поддерживает свое состояние согласованным на протяжении всего жизненного цикла, но его функциональность скудна. Расширить её

можно было бы, например, добавлением метода проверки на равенство объектов-строк. Однако, у такого подхода есть принципиальные недостатки:

а) Требования к классу строки в разных задачах могут сильно различаться.
   — Если собирать в одном классе функциональность для разных задач, код усложняется, хотя в каждой отдельной задаче большая его часть не нужна.
   — Включение в класс всевозможных методов противоречит идее ООП: модель должна включать только то, что необходимо для решения задачи.

б) Требуется изменение кода класса, что бывает невозможно или нежелательно в случае, например, коммерческих библиотек.

Тем не менее, желательно повторно использовать (reuse) разработанный класс строки как базу для специфических реализаций. Можно было бы создать новый класс `EquatableString`, сделав одним из его полей объект `String`. Однако, это логически неправильно: сравнимая строка не содержит (has-a) просто строку, она ею является (is-a), представляя собой частный случай.

Решение в данной ситуации предоставляет механизм наследования классов [1, раздел 10]:

```
1   class EquatableString: public String {
2   public:
3       EquatableString(const char *data): String(data) { }
4       const bool equals(const String& other) {
5           if (this == &other) {
6               return true;
7           } else if (this->getLength() != other.getLength()) {
8               return false;
9           }
10          return !strcmp(this->getData(), other.getData());
11      }
12  };
13  …
14  EquatableString first("12345");
15  EquatableString second("54321");
16  first.equals(second); // Результат: false.
17  first.setData(second.getData());
18  first.equals(second); // Результат: true.
```

Отношение наследования определяется в строке 1: класс `String` указывается базовым, или родителем (родительским), что означает: все его члены будут присутствовать и в производном классе, или классе-наследнике, `EquatableString` (строки 3, 7 и 10). Присутствие не означает доступность: методы `EquatableString` не могут обращаться

к закрытым (**private**) полям базового класса (строки 7 и 10). Если бы это было необходимо, следовало бы объявить эти поля защищенными (**protected**) в базовом классе.

Открытый тип наследования **public** (строка 1) означает, что все члены базового класса в производном сохраняют свои спецификаторы доступа [1, подраздел 11.2]. Наследование других видов на практике применяется редко и, как правило, из технических соображений.

Среди открытых и защищенных методов наследуются конструкторы и деструкторы. Конструктор базового класса с параметрами можно вызвать явно, как это показано в строке 3. Отметим, что вызывать его как простой метод в теле конструктора наследника неправильно, так как выражение

```
String(data);
```

означает создание безымянного объекта класса `String`, который будет уничтожен по выходе из области видимости (из конструктора `EquatableString`). Если все базовые классы (наследование может быть многоуровневым) и члены-объекты можно сконструировать без параметров (с учетом доступа от наследника), у класса будет автоматически определен *конструктор по умолчанию (default constructor),* который вызывает конструктор базового класса без аргументов [1, подраздел 12.1:4]. Таким образом, у `EquatableString` два конструктора: приведенный в листинге и конструктор по умолчанию. Деструктор базового класса будет автоматически вызван после выполнения деструктора наследника, поэтому в данном случае определять деструктор в наследнике не требуется [1, подраздел 12.4:4].

Если нужно только сменить конструктору или деструктору спецификатор доступа, а реализация по умолчанию устраивает, в C++11 её можно задействовать явно (см. 2.9).

Можно запретить наследование от класса, см. 2.8.

1.3.2 Переопределение методов

В производных классах допускается не только дополнять набор методов, но и перегружать их (overload), а также *переопределять (override)* методы базового класса, дав им те же имена, но изменив поведение:

```
1       class DebugString : public String {
2       public:
3            void setData(const String& string) {
4                 setData(string.getData());
5            }
6            void setData(const char *data) {
7                 pritnf("[%p] data changed to [%s]", this, data);
8                 String::setData(data);
```

```
9               }
10              …
11      };
12      …
13      DebugString source("Источник"), destination("Приемник");
14      destination.setData(source);
```

Класс `DebugString` переопределяет в строках 6 — 9 метод `setData()` родителя, чтобы протоколировать изменения. В переопределенном методе вызывается и метод базового класса через оператор разрешения области видимости. Отметим, что в перегрузке метода `setData(const String&)` вызывается метод текущего класса `DebugString` (строка 4).

1.3.3 Полиморфизм

Важным является тот факт, что переопределенная версия метода определяется по типу переменной, к которой идет обращение: в строке 16 вызывается метод `setData()` класса `String`, поскольку тип `reference` — ссылка на объект `String`, хотя фактически она связана с объектом класса `DebugString`. Это называется *ранним связыванием (static/early binding),* потому что выбор метода производится еще на этапе компиляции («рано»). Однако, бывает желательно работать с объектами любых классов в иерархии наследования через интерфейс базового класса так, чтобы вызывались, при необходимости, переопределенные версии методов:

```
1       void fallbackToDefaultValue(
2            String& destination, String& defaultValue) {
3            if (!destination.getLength()) {
4                 destination.setData(defaultValue.getData());
5            }
6       }
7       …
8       DebugString address;
9       String localhost("127.0.0.1");
10      fallbackToDefaultValue(address, localhost);
```

В данном коде будет вызван метод `String::setData()` в строке 4, и присвоение значения по умолчанию «127.0.0.1» переменной `address` не будет запротоколировано, как ожидалось. Требуемое же поведение называется *поздним связыванием (late/dynamic binding),* частным случаем *полиморфизма (polymorphism),* и реализуется в C++ посредством *виртуальных методов (virtual functions)* [1, подраздел 10.3]:

```
class String {
public:
     virtual void setData(const char *data) {
          // Реализация прежняя.
     }
     …
};
class DebugString : public String {
public:
     virtual void setData(const char *data) {
          // Реализация прежняя.
     }
     …
};
```

При такой реализации функция `fallbackToDefaultValue()` будет вызывать метод `setData()` именно того класса, объект которого ей передан, и в случае `DebugString` — протоколировать смену строки.

Стандарт C++11 позволяет указывать виртуальным методам атрибуты, например, чтобы запретить дальнейшее переопределение или, напротив, потребовать виртуальности метода в базовом классе, см. 2.6.2.

1.3.4 Накладные расходы позднего связывания

Очевидно, что для вызова правильного варианта виртуальной функции во время выполнения компилятор должен предпринять некоторые дополнительные действия (установить тип объекта). На это тратится время, то есть возникают *накладные расходы на виртуальный вызов (virtual function call overhead).*

Способ реализации позднего связывания компилятором не стандартизирован. Однако, в настоящее время единственной практически применяемой реализацией является таблица виртуальных методов (virtual method table, VMT). Каждый объект класса хотя бы с одним виртуальным методом содержит дополнительный указатель на таблицу с адресами виртуальных методов *данного класса.* При вызове виртуального метода объекта компилятор вызывает метод, адрес которого записан в VMT этого объекта. При переопределении виртуального метода в производном классе указатель на новую реализацию у объектов класса-наследника записывается в ту же ячейку VMT, что и на метод базового класса (см. рис. 1).

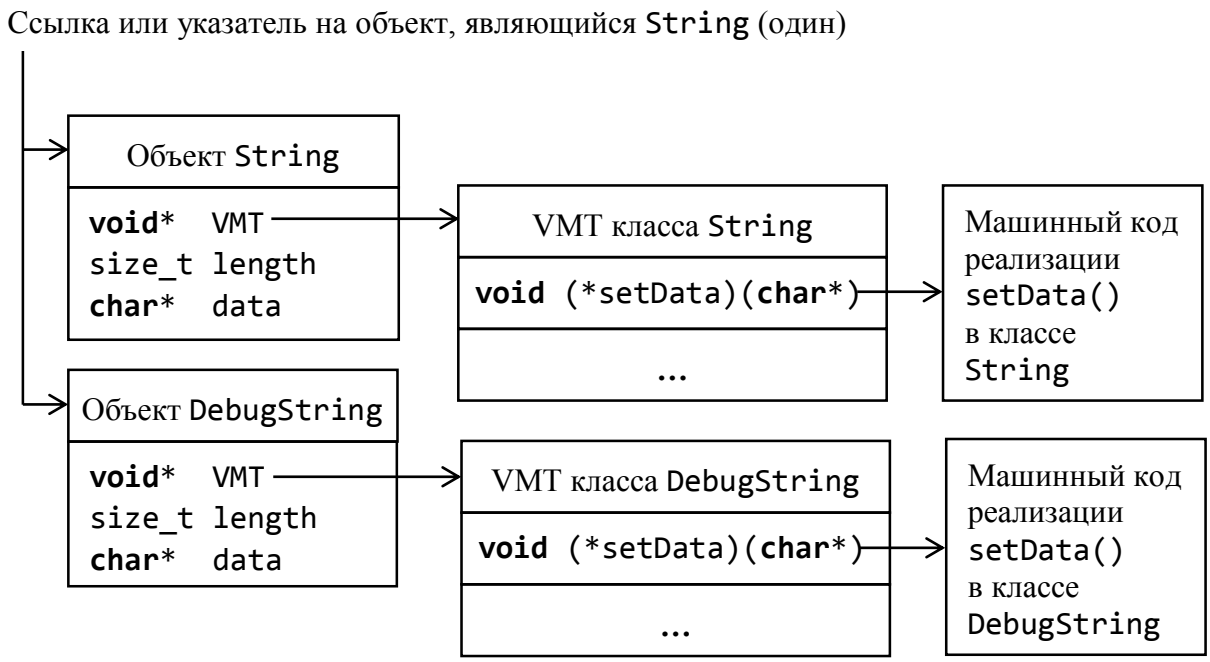

Рисунок 1 — Реализация позднего связывания таблицей виртуальных методов

Указатель на VMT недоступен программисту в штатном порядке, являясь деталью реализации. Тем не менее, его скрытое присутствие увеличивает размер объекта, что также относится к накладным расходам (overhead) на поддержку позднего связывания.

1.3.5 Виртуальность конструктора и деструктора

Важным частным случаем потребности в полиморфизме является деструктор, так как, во-первых, он часто вызывается неявно (труднее отследить обращения, читая код), во-вторых, вызов корректной версии деструктора принципиален — зачастую это вопрос освобождения или утечки ресурсов. По этой причине на практике придерживаются правила: деструктор всегда должен быть виртуальным, если тому нет технических препятствий. Де-факто, единственное разумное оправдание невиртуального деструктора — случай, когда в классе больше нет виртуальных методов, а производительность или размер объекта необычайно критичны.

Конструктор в C++ виртуальным быть не может из соображений, что при вызове конструктора объект еще не создан (incomplete), и говорить о его типе некорректно. Кроме того, в C++ конструктор можно вызвать только, зная тип создаваемого объекта, что делает виртуальность конструктора бессмысленной. Применительно же к ООП, виртуальный конструктор — это функция, которая получает тип (в каком-то виде) порождаемого объекта и создает — выделяет память и инициализирует — объект требуемого класса. Такой механизм в виде конструкторов реализован в Delphi, а практика подавляющего большинства ЯВУ показывает, что отсутствие виртуального конструктора не является проблемой.

1.3.6 Чисто виртуальные функции и абстрактные классы

Встречаются категории, у которых нет и быть не может конкретных воплощений. Например, не существует «просто млекопитающих», а только экземпляры подкатегорий: люди, бобры, манулы, — потому что «млекопитающее» — это обобщение. Тем не менее, такие категории (и соответствующие им *абстрактные классы* в программировании) бывают полезны, поскольку у них есть определенный интерфейс, и при помощи полиморфизма с ним удобно работать. С другой стороны, реализовать некоторые методы в абстрактных классах логически невозможно. Пусть класс `Printable` определяет интерфейс для объектов, которые можно напечатать, рассмотрим функцию для печати пар имя — значение, как в файлах конфигурации:

```
1       void writeConfigValue(Printable& name, Printable& value) {
2            name.print();
3            printf(" = ");
4            value.print();
5            putchar('\n');
6       }
```

Очевидно, у класса `Printable` должен быть метод с прототипом, например,

```
virtual void print();
```

Переопределение `print()` у конкретных наследников `Printable` может быть написано, однако, для самого `Printable` это сделать нельзя. Реализация в виде пустого метода — не выход: во-первых, это противоречит логике задачи (нельзя создать и напечатать «что-то печатаемое», но программа так делает), во-вторых, появляется опасность ошибок, так как наследники не обязаны предоставлять правильную реализацию. Проблема решается применением *чисто виртуальных методов (pure-virtual functions)*, которые делают класс абстрактным [1, подраздел 10.4]:

```
1       class Printable {
2       public:
3            virtual void print() = 0;        // Реалиации нет нигде.
4            virtual ~Printable() = 0;
5       };
6       Printable::~Printable() { }
```

Спецификатор чистой виртуальности метода (pure-virtual specifier) в виде «`= 0`» означает, что реализации данного метода в классе нет. Объект такого класса создать нельзя (в отличие от Delphi), поэтому все логические неувязки разрешаются. Наследники обязаны реализовать все виртуальные функции предков, иначе они тоже останутся абстрактными классами:

```
1       class String : public Printable {
2       public:
3            virtual void print() {
4                 if (data)
5                      printf("%s", data);
6            }
7            …
8       };
9       class Date : public Printable {
10      public:
11           virtual void print() {
12                pritnf("%u.%u.%u", day, month, year);
13           }
14           …
15      };
16      …
17      writeConfigValue(String("end_of_days"), Date(21, 12, 2012));
```

Строка 17 вызывает функцию `writeConfigValue()`, передав ей в качестве параметров экземпляры двух конкретных классов, унаследовавших от `Printable` интерфейс для печати. В результате будет напечатана строка с датой несостоявшегося Конца Света:

```
end_of_days = 21.12.2012
```

Отметим, что деструктор `Printable` не просто виртуальный, но чисто виртуальный, и при этом реализован. Это связано с тем, что деструктор базового класса обязан быть реализован, если в программе создаются объекты классов-наследников [1, абзац 12.4:9].

В случае, когда класс содержит только чисто абстрактные методы (и ничего больше), его часто называют (это не термин C++) чисто абстрактным классом или интерфейсом, поскольку, действительно, определяет только способы взаимодействия с наследниками. Видно, что именно таким и является `Printable`. Однако, часто абстрактные классы содержат реализованную функциональность, используемую наследниками, в том числе конструкторы и деструктор.

2 РАСШИРЕННЫЕ СРЕДСТВА ООП В C++

2.1 Копирование и перемещение объектов

В программах иногда возникает потребность копирования или перемещения объекта. Разница заключается в том, что после копирования оба объекта оказываются в одинаковом (логически) состоянии, а при перемещении состояние исходного объекта изменяется [3, подраздел 17.5], и он зачастую уже непригоден к использованию. Копирование или перемещение объектов происходит [1, подраздел 12.8:1]:

в) при инициализации (объявлении);

г) при передаче в функцию как аргумента;

д) при возврате объекта из функции;

е) во время присваивания.

Объект не может быть скопирован или перемещен, если не может быть скопировано или перемещено любое его поле, либо копирование или перемещение запрещено для базового класса [1, подраздел 12.8:11].

2.1.1 Конструктор копирования

В программах иногда возникает потребность создать объект-копию другого: при передаче параметра в функцию, при необходимости создать один объект точно таким же, как другой. При этом вызывается особый вид конструктора — *конструктор копирования (copy constructor)*. Конструктором копирования является любая перегрузка конструктора, принимающая первым аргументом ссылку на объект того же класса; при этом ссылка может быть как на изменяемый, так и на неизменяемый объект, а также могут присутствовать другие параметры, если все они имеют значения по умолчанию [1, подраздел 12.8:2]. Например, вот конструктор копирования класса `String`:

```
String(const String& copiedOne) {
     setData(copiedOne.data);
}
```

Примеры случаев, когда он используется:

```
void function(String string);
String first("Имя мое Легион, потому что нас много.");
String second(first);
function(first);
```

В этом коде объект `second` создается как явная копия объекта `first`, и вызывается функция `function()`, тело которой оперирует не объектом `first`, а его копией.

Конструктор копирования необходим классу `String` потому, что объекты содержат указатель на область памяти, и копия должна получить иной указатель, нежели оригинал (на данные, совпадающие с исходными). В противном случае при вызове деструктора копии память будет освобождена, а при вызове деструктора оригинала произойдет попытка освободить уже свободную память — ошибка.

Если конструктор копирования не определен, создается публичный конструктор копирования по умолчанию, выполняющий копирование почленно (см. также 2.9).

Конструктор копирования, как и любой другой, может не быть публичным. Если у объекта нет публичных конструкторов копирования, создать его копию из внешнего кода невозможно. Запрет копирования иногда применяется специально, например, если копирование слишком ресурсоемко, нежелательно или физически неосуществимо (например, объект ведет запись в единый для приложения файл журнала). См. также 2.10.

Копирование следует отличать от присвоения, см. 2.3.

2.1.2 Конструктор перемещения

Конструктор перемещения (move constructor) введен в C++11 для случаев, когда исходный объект является *rvalue,* например, это возвращаемое значение функции. Он объявляется подобно конструктору копирования, но первый параметр — ссылка на праводопустимое выражение [1, подраздел 12.8:3]:

```
String(String&& movedOne) {
      this->data = movedOne.data;
      this->length = movedOne.length;
      movedOne.data = 0;
      movedOne.length = 0;
}
```

О ссылках на праводопустимые выражения см. ниже.

Как видно из примера, в конструкторе перемещения класса `String` владение областью памяти старого объекта передается новому, а старый его лишается. При вызове деструктора перемещаемого объекта `movedOne` его поле `data` уже `0`, и попытки освободить область памяти с данными не будет.

Пример использования конструктора перемещения:

```
String function() {
      return String("We have to move mountains, "
                    "because our enemies move planets!");
}
String result = function();     // (1)
```

В данном коде функция `function()` возвращает созданный объект `String`, который разрушится после вычисления выражения (1). Объект `result` получит данные

возвращенного объекта, причем с использованием конструктора перемещения, то есть вместо того, чтобы выделять новую область памяти, копировать в нее содержимое старой, а затем сразу же освобождать старую, будет просто использована старая область памяти, что всегда эффективнее.

Если в классе отсутствует нетривиальный конструктор (простой, копирования и перемещения), деструктор и оператор присваивания (см. 2.3), считается, что у класса определен конструктор перемещения по умолчанию, который действует, как конструктор копирования [1, подраздел 12.8:9].

2.1.3 Виды выражений и ссылки на праводопустимые выражения

Стандарт C++11 вводит новый вид ссылок — *ссылки на rvalue (rvalue references)*.

Термин rvalue иногда переводится как «праводопустимое», а иногда не переводится, и означает следующее. Все выражения в C++ классифицируются, согласно схеме на рис. 2 [1, подраздел 3.10].

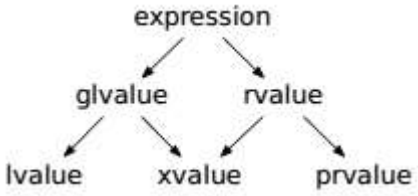

Рисунок 2 — Классификация выражений C++

Категории lvalue (леводопустимые) и rvalue являются основными. Они получили свои наименования от англ. left side value и right side value — «значение слева (справа) от оператора присвоения». Поэтому категория lvalue, говоря упрощенно, включает все, чему можно присвоить значение, в противовес rvalue. Пример lvalue — переменная, ссылка на переменную, указатель (естественно, не константный). Пример rvalue — результат вызова функции: где-то в памяти находится значение, оно не константа, но присвоить ему ничего нельзя.

Итак, в C++11 стало возможно получить ссылку на rvalue (rvalue-ссылку):

```
double &&rvalueReference = sqrt(100.0);
```

В большинстве ситуаций поведение rvalue-ссылок не отличается от обычных. Однако, rvalue-ссылка — это отдельный тип данных, различимый при перегрузке функций, поэтому и возможен конструктор перемещения. Однако, если нужно передать rvalue-ссылку аргументом, необходимо явно приводить тип, поскольку все аргументы вычисляются перед вызовом функции, а выражение в виде имени переменной уже не rvalue.

2.2 Друзья класса

Объявление класса может содержать объявления функций и классов, называемых *друзьями (friend) данного класса.* Друзья получают доступ к **`private`** и **`protected`** членам класса. Это не является нарушением принципа инкапсуляции, так как перечень друзей задается в классе, открывающем им доступ. Распространенным случаем использования друзе класса является перегрузка операторов (см. 2.3), где и приведен пример синтаксиса.

2.3 Перегрузка операторов

Перегрузка операторов (operator overloading) [1, подраздел 13.5] позволяет использовать операторы C++ для действий над пользовательскими классами. Например, можно определить оператор доступа к символам строки по индексу:

```
class String {
     …
public:
     char operator[](size_t index) const { return data[index]; }
```

Фактически, определяется метод со специальным именем **`operator[]`**, после чего становится возможным обращаться к символам объекта-строки индексацией:

```
String game("Mass Effect");
putchar(game[0]);
putchar(game.operator[](0));
```

Последняя строка демонстрирует полную форму вызова оператора как функции и полностью эквивалентна предпоследней. Выражение сложения `left + right` также можно представить себе как вызов специальной функции, которая принимает два аргумента (слагаемые) и возвращает результат:

```
int operator+(int left, int right);
```

Для встроенных типов такое представление чисто умозрительное — сложение целых чисел реализуется компилятором, вызвать этот оператор как функцию нельзя. Однако, для пользовательских типов можно действительно определить перегруженный вариант такой функции, который и будет вызываться при «сложении» объектов класса:

```
const String operator+(const String& left, const String& right)
{
     String result(left);
     result.concatenateWith(right.data, right.length);
     return result;
}
```

Как видно из реализации, функции требуется доступ к защищенным полям объекта. Но перегруженный оператор является простой внешней функций по отношению к классу строки. В данном случае удобно сделать перегрузку функции другом класса:

```
friend
const String operator+(const String& left, const String& right);
```

Это определение должно находиться внутри определения класса. Ограничение доступа той секции, где определение располагается, не действует на функцию-друга.

Теперь действия над строками можно выполнять естественным образом:

```
String name("Шепард");
String fullName = "Капитан " + name;
```

В приведенном фрагменте кода для литерала `"Капитан "` будет неявно вызван конструктор класса `String`, и созданный объект будет передан в качестве первого аргумента перегруженному оператору, результат которого (`"Капитан Шепард"`) присваивается переменной `fullName`.

Важно заметить, что при перегрузке бинарного оператора, такого как `+`, соответствующий оператор сокращенного присваивания (в данном случае — `+=`) не становится перегруженным автоматически и требует отдельной реализации вручную (если она нужна). Более того, учитывается порядок операндов: для реализации удобного сравнения объекта-строки с указателем на строку символов необходимо перегрузить оператор сравнения дважды:

```
friend bool operator==(const String& object, const char *ntcs);
friend bool operator==(const char *ntcs, const String& object);
```

Оператор проверки на неравенство (`!=`) при этом перегружен не будет.

Перегрузка постфиксной и префиксной форм инкремента и декремента выполняется раздельно — итого 4 оператора, различаемых согласно [1, п. 13.5.7 и п. п. 13.3.1.2].

Запрещена перегрузка операторов: `.` (доступ к члену объекта), `.*` (разыменование указателя на член объекта), `::` (разрешение области видимости) и `?:` (условное выражение). Заметим, что перегрузка оператора доступа к члену объекта по указателю (`->`) разрешена в C++11 [1, п. 13.5.6].

2.3.1 Перегрузка операторов приведения типов

Особым случаем перегрузки оператора является перегрузка приведения к типу [1, п. 12.3.2]. Например, можно перегрузить приведение строки к логическому типу, чтобы пустые и неинициализированные строки считались ложным выражением:

```
operator bool() const {
     return data && length;
```

```
}
```

Тип возвращаемого значения при перегрузке оператора приведения типа указывать запрещается, так как он уже однозначно задан именем оператора. Реализация использует тот факт, что указатели и целые приводятся к типу **bool** встроенным образом.

2.3.2 Перегрузка оператора присвоения

Перегруженные варианты оператора присваивания могут быть определены для присваивания объекту выражений других типов, причем эти перегруженные методы могут быть виртуальными (единственные из операторов). В C++11 выделяется копирующее и перемещающее присваивание (copy & move assignment). Присваивание часто перегружают подобно конструктору копирования (или перемещения, соответственно):

```
String& String::operator=(const String& rightSide) {
      setData(rightSide.data);
      return *this;
}
String& String::operator=(String&& rightSide) {
      this.data = rightSide.data;
      this.length = rightSide.length;
      rightSide.data = 0;
      rightSide.length = 0;
      return *this;
}
```

Выражение **return *this** является идиомой при перегрузке унарных операторов, таких как присваивание и сокращенное присваивание, для возврата данного объекта.

На практике для любых классов с нетривиальным копированием или перемещением необходимо перегружать (или запрещать) оператор присваивания. Элегантнее всего это можно делать при помощи т. н. идиомы copy & swap, ознакомление с развернутым описанием [4] которой настоятельно рекомендуется.

2.3.3 Функторы

Перегрузка оператора вызова позволяет использовать объект подобно функции:

```
class Functor {
public:
      void operator() (double, int) { }
};

…

Functor f;
f(3.14, 0);
```

Разумеется, объект с таким перегруженным оператором может иметь и другие методы, а также поля. Если продемонстрированное использование объекта является основным, принято называть такой объект *функтором (functor)*. Функторы активно используются в стандартной библиотеке языка C++.

2.3.4 Перегрузка операторов работы с динамической памятью

Перегрузка операторов **new** и **delete** для класса позволяет контролировать выделение и освобождение динамической памяти для его объектов [1, п. п. 5.3.4 — 5.3.5]:

```
1       class Object {
2       public:
3            static void* operator new (size_t) {
4                 Object *created = ::new Object;
5                 if (tail) {
6                      tail->next = created;
7                 } else {
8                      list = tail = created;
9                 }
10                created->next = 0;
11                return created;
12           }
13           static void operator delete (void* object) nothrow {
14           }
15           static void cleanUp() {
16                Object *current = list;
17                while (current) {
18                     Object *victim = current;
19                     current = current->next;
20                     ::delete victim;
21                }
22                list = tail = 0;
23           }
24      private:
25           Object* next;
26      private:
27           static Object *list;
28           static Object *tail;
29      };
30      Object* Object::list = 0;
31      Object* Object::tail = 0;
```

Класс `Object` отличается тем, что после выделения памяти под несколько объектов оператором **new** можно освободить всю занимаемую ими память вызовом статического метода `Object::cleanUp()`. Для этого в операторе **new** указатель на вновь созданный объект запоминается в связанном списке. Реальное выделение памяти при этом

выполняется оператором **new** по умолчанию, который вызывается как **::new**, то есть из глобальной области видимости. Оператор **delete** класса Object ничего не делает. Статический метод cleanUp() разрушает все элементы списка и освобождает память, выделенную под ним, глобальным оператором **::delete**. Использование класса таково:

```
Object *first = new Object;
Object *second = new Object;
…
Object::cleanUp();
```

Вызов Object::cleanUp() освободит память, занятую всеми объектами класса, созданными посредством **new**, и вызывать для них **delete** не нужно. Разумеется, после вызова Object::cleanUp() разыменовывать first и second нельзя. Заметим, что пара операторов **new**[] и **delete**[] не определена, поэтому они продолжат действовать, как обычно.

Допускается и перегрузка оператора взятия адреса (&) и разыменования (*) указателя на объект. Это может быть удобно для объектов, которым требуется имитировать поведение простых указателей.

2.4 Особенности приведения типов

2.4.1 Приведение dynamic_cast

С классами связан особый вид приведения типов — **dynamic_cast** [1, п. 5.2.7]. Данное приведение преобразует указатель p или ссылку r на объект к типу, который должен быть соответственно указателем или ссылкой на объект класса T, либо к **void*** (с сохранением модификатора константности). Если значение, на которое указывает p, является объектом самого класса T или его публичного наследника), преобразование проходит успешно. Иначе результат преобразования — нулевой указатель, а в случае использования ссылок возбуждается исключение std::bad_cast (об исключениях будет изложено далее в курсе). Это можно использовать для перемещения по иерархии наследования (inheritance hierarchy navigation), см. [3, п. 22.2.1], *и проверки типа объекта во время выполнения (run-time type checking):*

```
String *object = getString();
AdvancedString *searchable =
     dynamic_cast<AdvancedString*>(object);
if (searchable != 0) {
     // Переменная object указывает на объект, являющийся
     // AdvancedString, searchable можно использовать.
} else {
     // Переменная object указывает на объект, не являющийся
```

```
        // AdvancedString, searchable == 0.
}
```

Подобное поведение реализуется в Delphi и C# операторами **`is`** и **`as`**.

Приведение переменных базовых классов к типам наследников *(«вниз по иерархии», downcasting),* реализуемое **`dynamic_cast`**, используется на практике, но часто порицается и может свидетельствовать о несовершенстве архитектуры программы.

2.4.2 Явное и неявное приведение типов

Приведение типов в программе может быть *явным (explicit),* например, через синтаксис стиле C, или *неявным (implicit),* например, когда вещественной переменной присваивается целочисленное значение. Особый вид неявного приведения типов, *через конструктор (conversion by constructor),* возникает, когда на месте объекта используется выражение, пригодное для передачи в конструктор [1, п. 12.3.1]:

```
String sample = "Обычная строка C";
```

В приведенном фрагменте будет задействован конструктор с параметром-строкой C `String(const char *data)`. Эта возможность удобна тем, что функции, принимающей объект `String`, можно передать строку C, и объект-параметр будет сконструирован автоматически.

Однако, это бывает неудобно или недопустимо. В рассматриваемом примере создание объекта `String` — достаточно длительная операция, связанная с выделением памяти, поэтому может быть желательно контролировать все случаи приведения типов, используя его только при необходимости. Принудить компилятор задействовать оператор приведения типов или конструктор только при явном их вызове можно, указав перед объявлением метода ключевое слово **`explicit`**.

Считается хорошей практикой делать явными конструкторы с одним параметром, чтобы при чтении кода было очевидно создание объекта [3, п. 16.2.6].

2.5 Статические члены класса

Бывает удобно некоторые данные и функции связать не с конкретными объектами, а с классом в целом. Например, класс строки может реализовывать подсчет общего объема памяти, занимаемого всеми данными объектов-строк. Для этого в C++ предусмотрены статические члены класса, подобные полям и методам класса в Delphi. Объявление статических членов предваряется ключевым словом **`static`**:

```
class String {
    …
private:
```

```
        static size_t totalMemoryConsumed;
    public:
        static size_t getTotalConsumedMemory() {
            return totalMemoryConsmed;
        }
    };
```

Инициализация статических данных-членов должна производиться строго один раз в программе в любой единице трансляции; она выполняется перед началом работы программы:

```
size_t String::totalMemoryConsumed = 0;
```

При этом доступ к статическим членам разграничивается так же, как и к обычным, поэтому за пределами методов класса `String` (обычных и статических) поле `totalMemoryConsumed` недоступно, кроме как посредством статического метода:

```
size_t stringMemory = String::totalMemoryConsumed;
size_t stringMemory = String::getTotalMemoryConsumed();
```

Первая строка приведет к ошибке компиляции (так как поле объявлено как **`private`**), вторая же строка совершенно корректна (статический метод объявлен как **`public`**).

2.6 Атрибуты методов

Атрибуты методов служат для выражения некоторых ограничений или допущений, зачастую смысловых, на содержимое методов и их роль в иерархии классов. Атрибуты методов размещаются после списка аргументов. Атрибуты разных видов могут применяться совместно.

2.6.1 Неизменяющие методы

Метод называется *неизменяющим, или константным (constant member function),* если он не меняет состояние объекта. Очевидно, что у *неизменяемых (immutable) объектов* могут быть вызваны только неизменяющие методы. Пометить метод как неизменяющий можно атрибутом **`const`**. Компилятор отслеживает и запрещает изменение состояния класса в реализациях таких методов, в том числе вызовы изменяющих методов. В неизменяющем метода класса `T` тип **`this`** — неизменяемый указатель на неизменяемый объект [1, п. 9.3.1].

Метод может быть неизменяющим физически и логически [3, п. п. 16.2.9.2]. Разница заключается в том, с какой точки зрения может измениться состояние объекта: *физически* — поменяются значения полей, или *логически* — когда эти изменения, вдобавок, может наблюдать пользователь класса. Например, если метод `flip()` возвращает строку

задом наперед (не изменяя исходную), он логически неизменяющий. В то же время, этот метод может записывать в специальное поле нового объекта ссылку на исходный, чтобы ускорить работу (ни одну строку не придется переворачивать дважды), — физически метод изменяющий. Преодолеть противоречие можно, объявив поле как **`mutable`** (антоним **`const`**): изменение таких полей в неизменяющих методах допустимо. Важно понимать, что неизменяющие методы и **`mutable`** — это не противоречивые возможности языка, а средства разделения физически и логически неизменяющих методов.

2.6.2 Атрибуты виртуальных методов

Виртуальный метод может быть переопределен в классе-наследнике, а чисто виртуальный метод — может и должен быть переопределен. Запретить переопределение виртуального метода в наследниках можно при помощи атрибута **`final`**:

```
class String {
     virtual bool isUnicode() const = 0;
};
class UnicodeString : public String {
public:
     virtual bool isUnicode() const final { return true; };
};
```

Запрещать переопределение метода рекомендуется в случаях, когда это было бы бессмысленно или слишком сложно (чревато ошибками). В примере: строка (`String`) может быть в одной из кодировок Unicode, а может и не быть; любой наследник базового класса строк в одной из кодировок Unicode (`UnicodeString`) имеет одну и ту же реализацию метода проверки `isUnicode()`, и перекрывать ее не должен. Также, поскольку **`final`** делает метод невиртуальным, его использование может повысить производительность, однако, это зависит от компилятора и может быть значимым доводом только в конкретных случаях [3, п. п. 20.3.4.2].

Все рассмотренные атрибуты виртуальных методов (**`virtual`**, `= 0`, **`final`**) позволяют выразить ограничения или разрешения для наследников со стороны базового класса. Существует и средство выразить ограничение на базовый класс из класса-наследника — атрибут **`override`**. Данный атрибут означает требование виртуальности соответствующей (переопределяемой) функции в базовом классе, защищая от ошибок:

```
class FastSearchString : public AdvancedString {
     virtual int indexOf(const String& subsring) const override;
};
```

Класс строки с быстродействующей реализацией поиска подстроки полагается на то, что в базовом классе есть методы, использующие `indexOf()`, и при помощи **`override`**

обеспечивает уверенность в том, что эти методы в `FastSearchString` будут использовать полиморфную (лучшую) реализацию.

Атрибуты **`final`** и **`override`** введены в C++11 [1, подраздел 10.3].

2.7 Указатели на члены класса

Все объекты одного класса имеют идентичное расположение своих полей в памяти, то есть некоторое поле любого объекта располагается на одинаковом *смещении (offset)* от начала конкретного объекта в памяти. Становится очевидно, что при наличии адреса объекта и смещения поля технически возможно получить доступ к значению этого поля данного объекта. Кроме того, любой метод класса имеет адрес, не зависящий от объекта, а значит, вызов некоторого метода конкретного объекта технически реализуем при наличии объекта и адреса метода. Обе возможности реализуются *указателями на члена класса (pointers-to-members)* [1, п. 8.3.3, подразделы 5.3 и 5.5].

Рассмотрим пример. Окну необходимо реагировать на нажатие кнопки (окно может иметь множество кнопок). При этом кнопка не должна «знать» о классе окна, так как она может располагаться, например, на панели инструментов и т. п.

Решим задачу созданием следующей системы классов. Интерфейс `IClickDelegate` представляет некий обработчик события нажатия на кнопку, который может быть вызван методом `invoke()`:

```
class IClickDelegate {
public:
	virtual void invoke() = 0;
	virtual ~IClickDelegate() = 0;
};
IClickDelegate::~IClickDelegate() { }
```

Класс кнопки `Button` хранит указатель на такой обработчик `clickEvent` и имеет метод для его установки извне `setClickHandler()`. Метод `click()` имитирует нажатие на кнопку.

```
class Button {
public:
	Button() :clickEvent(0) { }
	virtual ~Button() {
		if (clickEvent) delete clickEvent;
	}
	void click() {
		if (clickEvent) clickEvent->invoke();
	}
	void setClickHandler(IClickDelegate *eventHandler) {
```

```
            clickEvent = eventHandler;
        }
    private:
        IClickDelegate *clickEvent;
    };
```

Реализуем интерфейс обработчика события нажатия на кнопку для случая, когда обработка выполняется методом объекта-окна (`Window`):

```
1   class Window;
2   class WindowClickDelegate : public IClickDelegate {
3   public:
4       typedef void (Window::*ButtonClickEventHandler)(void);
5       WindowClickDelegate(
6           Window& window, ButtonClickEventHandler handler)
7           : window(window), handler(handler) { }
8       virtual void invoke() {
9           (window.*handler)();
10      }
11  private:
12      Window& window;
13      ButtonClickEventHandler handler;
14  };
```

В первой строке предварительно объявляется класс `Window`, так как используется ссылка на его объект (поле `window` на строке 12). На строке 4 объявляется псевдоним `ButtonClickEventHandler` для следующего типа: `void Window::* (void)`. Этот тип — указатель на член класса `Window`, на метод, не имеющий параметров и возвращаемого значения. Конструктор сохраняет ссылку на объект окна, метод которого будет вызываться, и указатель на метод. Метод `invoke()` выполняет вызов метода с сохраненным адресом у объекта `window`. Для этого применяется оператор разыменования указателя на член класса — `.*`. Он имеет низкий приоритет, поэтому нужны скобки вокруг выражения разыменования.

Класс окна включает поле `button` кнопки «ОК» и поле `cancelButton` кнопки «Отмена», сделанные публичными для удобства демонстрации, а также методы для обработки нажатия на эти кнопки. В конструкторе кнопке устанавливается обработчик события[1, 2]:

```
class Window {
public:
    Window() : okButton(), cancelButton() {
```

[1] Перед выполнением тела конструктора явно вызываются конструкторы кнопок, чтобы в конструкторе окна они уже были созданы, и у них можно было вызывать методы.

[2] Создание объекта `WindowClickDelegate` в конструкторе окна, а его уничтожение в деструкторе кнопки — сомнительное решение, используемое здесь лишь для простоты.

```
            okButton.setClickEvent(new WindowClickDelegate(
                  *this, &Window::onOKButtonClicked));
            cancelButton.setClickEvent(new WindowClickDelegate(
                  *this, &Window::onCancelButtonClicked));
      }
      void onOKButtonClicked() {
            // Обработать нажатие на кнопку «OK».
      }
      void onCancelButtonClicked() {
            // Обработать нажатие на кнопку «Отмена».
      }
public:
      Button okButton, cancelButton;
};
```

В следующем коде вызов `window.okButton.click()` приводит к вызову обработчика `onOKButtonClicked()` именно объекта window:

```
Window window;
window.okButton.click();
```

Пример получился достаточно громоздким, и перспектива создавать однотипные реализации `IClickDelegate` для каждого класса, содержащего обработчики, неприглядна. Однако, в дальнейшем будут рассмотрены средства C++, позволяющие задействовать указатели на члены для решения таких задач гораздо более удобным способом.

2.8 Ненаследуемые классы

C++11 содержит способ запретить наследование от класса. Для этого служит ключевое слово **`final`**, размещаемое после имени класса [1, раздел 9, абзац 3]:

```
class UltimateString final;
```

Причины для запрета наследования зачастую архитектурные [3, п. п. 20.3.4.2].

2.9 Явное использование реализаций по умолчанию

Специальные методы: конструкторы простые, копирования и перемещения, деструкторы, — если не определены явно, имеют реализацию по умолчанию. Бывает удобно одновременно и определить такой метод явно, и задействовать реализацию по умолчанию. Пример: классу требуется и нетривиальный конструктор, и конструктор по умолчанию. C++11 вводит простой способ решить эту задачу [1, п. 8.4.2], [3, п. п. 17.6.1 — 17.6.3]:

```
String() = default;
String(const char* data);
```

Примечание. По состоянию на август 2013 г. данная возможность не поддерживается Visual Studio 2012 и Visual Studio 2013 Preview, но поддерживается GCC 4.8.1.

2.10 Удаленные определения функций

Бывает необходимо запретить вызов некоторых методов вообще, включая вызовы в других методах класса. C++11 вводит [1, п. 8.4.3] для этого *удаленные определения (deleted definitions):*

```
class LogFile {
private:
     LogFile();
public:
     LogFile(const UniqueLogFile& source) = delete;
};
```

Объект класса `LogFile` может быть создан только в другом методе класса, а копия такого объекта не может быть создана вообще. Такой метод запрета копирования является более строгим, чем предложенный в 2.1.2, и предпочтителен в C++11 [3, п. 17.6.4].

Примечание. По состоянию на август 2013 г. данная возможность не поддерживается Visual Studio 2012 и Visual Studio 2013 Preview, но поддерживается GCC 4.8.1.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. ISO/IEC 14882:2011 Programming Languages — C++. — Введен в 2011 г. — 1338 с.
2. Стивен Прата. Язык программирования C++. Лекции и упражнения, 6-е издание / пер. с англ. (C++ Primer Plus, Sixth Edition) — М.: Вильямс. — 2012 г. — 1248 с.
3. Stroustrup, Bjarne. The C++ Programming Language (Fourth Edition) / Bjarne Stroustrup. — Addison-Wesley. — 2013 г. — 1347 с.
4. GManNickG «What is the copy-and-swap idiom?» — Страница в интернете. — Режим доступа: http://stackoverflow.com/a/3279550, свободный.