А. В. Столяров

ВВЕДЕНИЕ В ЯЗЫК СИ++

тексты лекций

Москва – 2008

ПУБЛИЧНАЯ ЛИЦЕНЗИЯ

Учебное пособие Андрея Викторовича Столярова ≪Ведение в язык Си++≫, впервые опубликованное РИО МГТУ ГА в 2008 г. под заголовком ≪Методы и средства визуального проектирования. Раздел "введение в Си++"≫, называемое далее ≪Произведением≫, защищено действующим авторско-правовым законодательством. Все права на Произведение, предусмотренные действующим законодательством, как имущественные, так и неимущественные, принадлежат его автору.

Настоящая Лицензия устанавливает способы использования электронной версии Произведения, право на которые предоставлено автором и правообладателем неограниченному кругу лиц, при условии безоговорочного принятия этими лицами всех условий данной Лицензии. Любое использование Произведения, не соответствующее условиям данной Лицензии, а равно и использование Произведения лицами, не согласными с условиями Лицензии, возможно только при наличии письменного разрешения автора и правообладателя, а при отсутствии такого разрешения является нарушением действующего законодательства и преследуется в рамках гражданского, административного и уголовного права.

Автор и правообладатель настоящим **разрешает** следующие виды использования данного файла, являющегося электронным представлением Произведения, без уведомления правообладателя и без выплаты авторского вознаграждения:

1. Воспроизведение Произведения (полностью или частично) на бумаге путём распечатки с помощью принтера в одном экземпляре для удовлетворения личных бытовых или учебных потребностей, без права передачи воспроизведённого экземпляра другим лицам;

2. Копирование и распространение данного файла в электронном виде, в том числе путём записи на физические носители и путём передачи по компьютерным сетям, с соблюдением следующих условий: (1) **все воспроизведённые и передаваемые любым лицам экземпляры файла являются точными копиями исходного файла** в формате PDF, при копировании не производится никаких изъятий, сокращений, дополнений, искажений и любых других изменений, включая и изменение формата представления файла; (2) **распространение и передача копий другим лицам производится исключительно *бесплатно*, то есть при передаче не взимается никакое вознаграждение ни в какой форме**, в том числе в форме просмотра рекламы, в форме платы за носитель или за сам акт копирования и передачи, даже если такая плата оказывается значительно меньше фактической стоимости или себестоимости носителя, акта копирования и т. п.

Любые другие способы распространения данного файла при отсутствии письменного разрешения автора запрещены. В частности, **запрещается**: внесение каких-либо изменений в данный файл, создание и распространение искаженных экземпляров, в том числе экземпляров, содержащих какую-либо часть произведения; распространение данного файла в Сети Интернет через вебсайты, оказывающие платные услуги, через сайты коммерческих компаний, а также через сайты, содержащие рекламу любого рода; продажа и обмен физических носителей, содержащих данный файл, даже если вознаграждение значительно меньше себестоимости носителя; включение данного файла в состав каких-либо информационных и иных продуктов; распространение данного файла в составе какой-либо платной услуги или в дополнение к такой услуге. С другой стороны, **разрешается** дарение (бесплатная передача) носителей, содержащих данный файл, запись данного файла на носители, принадлежащие другим лицам, распространение данного файла через бесплатные файлообменные сети и т.п. Ссылки на экземпляр файла, расположенный на официальном сайте автора, разрешены без ограничений.

А. В. Столяров *запрещает* Российскому авторскому обществу и любым другим организациям производить любого рода лицензирование любых его произведений и осуществлять в интересах автора какую бы то ни было иную связанную с авторскими правами деятельность без его письменного разрешения.

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ВОЗДУШНОГО ТРАНСПОРТА
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«**МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ**
ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
ГРАЖДАНСКОЙ АВИАЦИИ»

Кафедра прикладной математики

А. В. Столяров

МЕТОДЫ И СРЕДСТВА ВИЗУАЛЬНОГО ПРОЕКТИРОВАНИЯ

раздел «**введение в язык Си++**»

тексты лекций

Москва – 2008

УДК 519.6
ББК
С81

Печатается по решению редакционно-издательского совета
Московского государственного технического университета ГА
Рецензенты: канд. физ.-мат. наук, доц. И. Г. Головин
д-р техн. наук, проф. А. А. Егорова

А. В. Столяров

С81 Методы и средства визуального проектирования. Раздел «введение в язык Си++»: тексты лекций. М.: МГТУ ГА, 2008.– 112 с., 4 рис.
ISBN 978-5-86311-660-0

В пособии представлено содержание лекций по языку программирования Си++, читаемых в рамках курса «Методы и средства визуального проектирования».

Целью лекций является краткое введение в основы объектно-ориентированного программирования и проектирования, необходимое для работы с существующими визуальными средствами проектирования программного обеспечения. Используемый в качестве иллюстративной основы язык Си++ является одним из наиболее популярных современных индустриальных языков программирования.

СТОЛЯРОВ Андрей Викторович
МЕТОДЫ И СРЕДСТВА ВИЗУАЛЬНОГО ПРОЕКТИРОВАНИЯ
Раздел «Введение в язык Си++»
тексты лекций

Лекция 1

§ 1.1. Введение

§ 1.1.1. Что такое ООП

Объектно-ориентированное программирование появилось в середине 70-х годов XX столетия. Проект Smalltalk известен как первый объектно-ориентированный проект и одновременно как проект, в котором впервые предложен оконный интерфейс пользователя.

Языки программирования, с которыми вам приходилось встречаться до сих пор, относятся к категории императивных языков программирования. Программа на императивных языках воспринимается как последовательность команд, изменяющих значения переменных и производящих другие действия (отсюда название парадигмы "императивное программирование", от слова "императив" — приказ, команда).

Помимо императивного программирования, существуют такие парадигмы, как логическое программирование, где программа воспринимается как набор логических высказываний, а выполнение — как доказательство или опровержение некоторого высказывания; функциональное программирование, где программа представляется как набор математических функций, а исполнение программы представляет собой вычисление некоторой главной функции.

Объектно-ориентированное программирование представляет собой еще одну парадигму программирования. Представляя себе программу, мы прежде всего представляем данные в виде некоторых ***объектов*** — «черных ящиков». Внутреннее устройство объекта извне недоступно (и в ряде случаев может быть просто неизвестно — например, если данный объект реализован другим программистом). Все, что можно сделать с объектом — это послать ему сообщение и получить ответ. Так, операция $2 + 3$ в терминах объектно-ориентированного программирования

выглядит как «мы посылаем двойке сообщение *прибавь к себе тройку*, а она отвечает нам, что получилось 5». Вполне возможно, что, получив сообщение, объект произведет еще и какие-то действия, сменит свое внутреннее состояние или пошлет сообщение другому объекту.

§ 1.1.2. Язык Си++ и его совместимость с Си

Язык Си++ был предложен Бьёрном Страуструпом в начале 80х годов прошедшего столетия в качестве ответа на назревшую в индустрии потребность в индустриальном объектно-ориентированном языке.

Си++ является расширением языка Си с объектно-ориентированными возможностями. Таким образом, на языке Си++ можно использовать как традиционное императивное, так и объектно-ориентированное программирование.

Большинство программ, написанных на языке Си, будет корректно с точки зрения языка Си++.

Некоторые программы на языке Си не могут быть откомпилированы транслятором Си++. Так, в Си++ присутствует некоторое количество ключевых слов, которых не было в языке Си. Поэтому, например, программа, содержащая такое описание:

```
int try;
```

не является корректной с точки зрения языка Си++, поскольку слово `try` — ключевое в Си++.

Кроме того, в языке Си++ нет отдельного пространства имен для тэгов структур; так, описание

```
struct mystruct {
   int a, b;
};
```

в программе на Си++ является описанием полноценного *типа* `mystruct`, тогда как на Си такое же описание означало бы лишь введение *тэга структуры*. В результате, например, часто встречающиеся в старых программах на Си описания вида

```
typedef struct mystruct {
   int a, b;
} mystruct;
```

являются с точки зрения Си++ некорректным из-за возникающего конфликта имен (в языке Си такого конфликта не возникало, т.к. тэги структур и имена типов относились к различным пространствам имен).

Заметим, что в Си++ при описании структуры нет необходимости использовать `typedef`, поскольку идентификатор, стоящий после слова `struct`, уже сам по себе представляет собой имя нового типа (в отличие от Си, где этот идентификатор считался тэгом структуры и превращался в имя типа только в сочетании со словом `struct`). Так, после объявления

```
struct s1 {
    int x, y;
};
```

в языке Си++ мы можем сразу описывать переменные типа `s1`:

```
s1 a;
```

тогда как в языке Си мы вынуждены были использовать слово `struct`:

```
struct s1 a;
```

§ 1.2. Методы, объекты и защита

В этой части курса мы будем отталкиваться от уже известных нам средств, пришедших в Си++ из языка Си, постепенно вводя новые (специфические для Си++) средства.

§ 1.2.1. Функции-члены (методы)

Для поддержки объектно-ориентированного программирования в Си++ вводятся понятия ***функций-членов*** и ***классов***.

Приведем пример функции-члена. Опишем структуру для представления комплексного числа через действительную и мнимую части:

```
struct str_complex {
    double re, im;
};
```

Введем теперь операцию вычисления модуля комплексного числа. На языке Си нам пришлось бы описать примерно такую функцию:

```
double modulo(struct str_complex *c)
{
    return sqrt(c->re*c->re + c->im*c->im);
}
```

Язык Си++ позволяет сделать то же самое несколько более элегантно, подчеркнув непосредственное отношение функции `modulo()` к сущности комплексного числа:

```
struct str_complex {
    double re, im;
    double modulo() { return sqrt(re*re + im*im); }
};
```

Теперь мы можем написать, например, такой код:

```
str_complex z;
double mod;
z.re = 2.7;
z.im = 3.8;
mod = z.modulo();
```

Функция `modulo()` называется *функцией-членом* или ***методом*** структуры `str_complex`. Иногда речь идет о *методе объекта* (в данном случае — объекта `z`), при этом имеется в виду метод того типа, к которому принадлежит объект.

Вызов функции-члена — это именно то, что в теории объектно-ориентированного программирования понимается под *отправкой сообщения объекту*. Иначе говоря, *термины "вызов метода" и "передача сообщения" являются синонимами.*

§ 1.2.2. Указатель `this`

На самом деле, вызов функции-члена (метода) объекта с точки зрения реализации представляет собой абсолютно то же самое, что и вызов обычной функции, первым параметром которой является адрес объекта; таким образом, когда в предыдущем параграфе мы заменили внешнюю функцию `modulo`, получавшую адрес структуры, на метод `modulo`, описанный внутри структуры, а вызов с явной передачей адреса — на вызов метода для объекта, на уровне машинного кода никаких изменений не произошло.

Говорят, что функциям-членам класса при вызове их для объекта класса передаётся ***неявный параметр*** — адрес объекта, для которого функция вызывается.

К этому параметру при необходимости можно обратиться; его именем является ключевое слово `this`. Можно воспринимать `this` как локальную константу, имеющую тип `A*`, где `A` — имя описываемого класса.

Например, описанную в предыдущем параграфе версию структуры можно было бы переписать и так:

```
struct str_complex {
    double re, im;
    double modulo() {
        return sqrt(this->re*this->re +
                    this->im*this->im);
    }
};
```

В данном конкретном случае это не имеет особого смысла, однако бывают и такие ситуации, в которых использование `this` оказывается необходимым; достаточно представить себе, что из метода нужно вызвать какую-либо функцию (обычную или метод другого объекта), аргументом которой должен стать как раз объект, для которого нас вызвали.

§ 1.2.3. Защита. Понятие конструктора

Чтобы стать полноценным объектом, нашей структуре не хватает свойства закрытости. Действительно, поля `re` и `im` доступны из любого места в программе, где доступна сама структура `str_complex`. Для того, чтобы скрыть детали реализации объекта, в языке Си++ введен механизм ***защиты***.

Для поддержания этого механизма имеются ключевые слова `public:` и `private:`, которыми в описании структуры могут быть помечены поля, доступные извне структуры (`public:`) и, наоборот, доступные только из тел функций-методов (`private:`).[1] Попытаемся переписать нашу структуру с использованием этих ключевых слов.

```
struct str_complex {
private:
    double re, im;
public:
```

[1] Несколько позже мы введем еще и ключевое слово `protected:`.

```
    double modulo() { return sqrt(re*re + im*im); }
};
```

Теперь поля `re` и `im` доступны только из тела метода `modulo()`.

Легко заметить, однако, что пользоваться такой структурой мы не сможем, т.к. в ней не предусмотрено никаких средств для задания значений полей `re` и `im`. Так, фрагмент кода, приведенный на стр. 6, попросту будет отвергнут компилятором.

Решить проблему можно, введя соответствующий метод для задания значений полей. Описание нашей структуры могло бы выглядеть в этом случае, например, так:

```
struct str_complex {
private:
    double re, im;
public:
    void set(double a_re, double a_im)
        { re = a_re; im = a_im; }
    double modulo()
        { return sqrt(re*re + im*im); }
};
```

а соответствующий код, вычисляющий модуль заданного числа, перепишем так:

```
str_complex z;
double mod;
z.set(2.7, 3.8);
mod = z.modulo();
```

Данное решение имеет очень серьёзный недостаток.[2] Дело в том, что с момента объявления переменной `z` до вызова функции `set()` наш объект (переменная `z`) оказывается в *неопределенном* состоянии, т.е. попытки его использовать будут заведомо ошибочны. Очевидно, было бы лучше произвести инициализацию объекта непосредственно в момент его создания.

Для этого вводится еще одно важное понятие — ***конструктор объекта***.

Конструктор — это функция-метод специального вида, задающая инструкции по инициализации объекта. Конструктор может, как и

[2]Более того, большинство компиляторов выдаст предупреждение при попытке компиляции такого объявления структуры

обычная функция, иметь параметры; описав конструктор, мы тем самым «объясняем» компилятору, как (в соответствии с какой инструкцией) создавать новый объект данного типа, какие параметры должны быть для этого заданы и как воспользоваться значениями этих параметров.

Компилятор отличает конструкторы от обычных методов по имени, которое совпадает с именем описываемого типа (в данном случае структуры). Поскольку конструктор играет специальную роль и в явном виде не вызывается, тип возвращаемого значения для конструктора указывать нельзя (он не возвращает никаких значений; в определенном смысле результатом работы конструктора является сам объект, для которого его вызвали).

Проиллюстрируем сказанное, переписав структуру `str_complex`:

```
struct str_complex {
private:
    double re, im;
public:
    str_complex(double a_re, double a_im)
        { re = a_re; im = a_im; }
    double modulo()
        { return sqrt(re*re + im*im); }
};
```

Введя конструктор, мы сообщили компилятору, что для создания объекта типа `str_complex` необходимо знать два числа типа `double` и задали набор действий, подлежащих выполнению при создании такого объекта. В этом наборе действий говорится в том числе и о том, как воспользоваться заданными числами типа double: первое из них использовать в качестве действительной части, второе — в качестве мнимой части создаваемого комплексного числа).

Вообще, в семантике Си++ любая переменная создается с помощью конструктора. Во многих случаях компилятор считает конструктор существующим («неявно»), несмотря на то, что в программе конструктор не описан.

Код вычисления модуля теперь можно переписать вот так:

```
str_complex z(2.7, 3.8);
double mod;
mod = z.modulo();
```

Более того, для такой операции нам не обязательно описывать переменную, имеющую имя. Мы могли бы написать и так:

```
double mod = str_complex(2.7, 3.8).modulo();
```

В последнем случае мы создали временную ***анонимную переменную*** типа `str_complex` и для этой переменной вызвали метод `modulo()`.

Здесь необходимо сделать очень важное замечание. После введения конструктора, имеющего параметры, компилятор **откажется** создавать объект типа `str_complex` без указания требуемых конструктором двух значений, то есть предыдущая версия кода, содержавшая описание

```
str_complex z;
```

компилироваться больше не будет. Если это создаёт неудобства, то при необходимости можно заставить компилятор снова считать такие объявления корректными; о том, как это делается, мы узнаем из § 2.3.

Следует отметить, что синтаксис создания переменной по заданному параметру допустим в Си++ и для переменных встроенных типов. Так, например, оператор

```
int v(7);
```

означает абсолютно то же самое, что и привычный по языку Си оператор

```
int v = 7;
```

В заключение параграфа, посвященного защите, отметим ещё один очень важный момент. **Единицей защиты в Си++ является не объект, а тип (в данном случае структура) целиком**. Это означает, что из тел методов мы можем обращаться к закрытым полям не только ≪своего≫ объекта (того, для которого вызван метод), но и вообще любого объекта того же типа.

§ 1.2.4. Зачем нужна защита

Смысл механизма защиты часто оказывается непонятен программистам, начинающим осваивать объектно-ориентированное программирование. Попробуем пояснить его, основываясь на нашем примере.

Представим себе, что наша структура `str_complex` используется в большой программе, активно работающей с комплексными числами. Может случиться так, что в программе будет очень часто требоваться вычисление модулей комплексных чисел. Более того, может оказаться

и так, что именно модуль комплексного числа требуется нам даже чаще, чем его действительная и мнимая части. В такой ситуации, скорее всего, наша программа будет проводить значительную часть времени своего выполнения в вычислениях модулей. Обнаружив это, мы можем в какой-то момент понять, что в данной конкретной задаче удобнее хранить комплексные числа в полярных координатах, а не в декартовых, то есть в виде модуля и аргумента, а не в виде действительной и мнимой частей.

Если мы не применяли защиту, то, скорее всего, все модули нашей программы, в которых используются комплексные числа, содержат обращения к полям `re` и `im`. Если в такой программе изменить способ хранения комплексного числа, убрав поля `re` и `im` и введя вместо них, скажем, поля `mod` и `arg` для хранения модуля и аргумента (т.е. полярных координат), то все части программы, использовавшие нашу структуру, перестанут компилироваться и нам придётся их исправлять. Если программа достаточно большая (а современные программные проекты состоят из многих сотен и даже тысяч модулей), такое редактирование может потребовать существенных трудозатрат, приведёт к внесению в программу новых ошибок и т.п., так что, вполне возможно, нам придётся отказаться от изменений, несмотря на всю их полезность.

Допустим теперь, что мы использовали защиту и сделали все поля недоступными откуда бы то ни было, кроме методов нашей структуры. Конечно, при использовании комплексных чисел часто бывает нужно узнать отдельно действительную и мнимую части числа, но для этого можно ввести специальные методы, например:[3]

```
struct str_complex {
private:
    double re, im;
public:
    str_complex(double a_re, double a_im)
        { re = a_re; im = a_im; }
    double get_re() { return re; }
    double get_im() { return im; }
    double modulo() { return sqrt(re*re + im*im); }
    double argument() { return atan2(im, re); }
};
```

[3]Здесь мы введём заодно функцию вычисления аргумента, которая будет использовать стандартную функцию `atan2()` для определения соответствующего арктангенса

Поскольку к полям `re` и `im` теперь не могут обращаться никакие части программы, кроме наших методов, мы можем быть уверены, что переписать нам придётся только наши методы, а весь остальной текст программы, из скольки бы модулей он ни состоял, сохранится без изменений и будет работать, как и раньше. Конечно, методы `get_re()` и `get_im()` теперь станут гораздо сложнее, чем были, зато упростятся методы `modulo()` и `argument()`:

```
struct str_complex {
private:
    double mod, arg;
public:
    str_complex(double re, double im) {
        mod = sqrt(re*re + im*im);
        arg = atan2(im, re);
    }
    double get_re() { return mod * cos(arg); }
    double get_im() { return mod * sin(arg); }
    double modulo() { return mod; }
    double argument() { return arg; }
};
```

В такой ситуации мы даже можем себе позволить иметь две реализации структуры `str_complex`, между которыми выбор осуществляется директивами условной компиляции. Таким образом, не меняя текста программы, мы сможем откомпилировать её с использованием одной реализации, измерить быстродействие, откомпилировать программу с использованием другой реализации, снова измерить быстродействие, сравнить результаты измерений и принять решение, какую реализацию использовать. В случае, если используемые в программе алгоритмы изменятся и нам снова станет выгодно хранить комплексные числа в декартовых координатах, то для возврата к старой модели вообще не придётся ничего редактировать.

Разумеется, всё это было бы совершенно невозможно, если бы мы не использовали защиту.

Очень важно понять, что защита в языке Си++ предназначена не для того, чтобы защищать нас от врагов, но исключительно для защиты нас от самих себя, от собственных ошибок. Если задаться целью обойти механизм защиты, это можно сделать без особых проблем путём преобразования типов указателей или другими способами. Защита работает только в случае, если программисты не

предпринимают целенаправленных действий по её обходу. Но обычно программисты таких действий не предпринимают, поскольку понимают полезность механизма защиты и выгоды от его использования, а также и то, что попытки его обойти, скорее всего, приведут к внесению в программу ошибок.

§ 1.2.5. Классы

Поскольку большинство внутренних полей объектов обычно закрыты, для описания объектов в Си++ используют специально введенный для этой цели тип составных переменных, называемый ***классом***.

Класс — это тип переменной, напоминающий запись (структуру), но отличающийся тем, что к полям (членам) класса доступ по умолчанию есть только из методов самого этого класса.

Перепишем нашу реализацию комплексного числа в виде класса:

```
class Complex {
    double re, im;
public:
    Complex(double a_re, double a_im)
        { re = a_re; im = a_im; }
    double get_re() { return re; }
    double get_im() { return im; }
    double modulo() { return sqrt(re*re + im*im); }
    double argument() { return atan2(im, re); }
};
```

Все, что описано в классе до слова `public:`, компилятор рассматривает как детали реализации класса и запрещает доступ к ним отовсюду, кроме тел функций-членов данного класса. Слово `public:` меняет режим защиты с закрытого на открытый, т.е. все, что описано после этого слова (в данном случае это функции-члены `Complex(...)`, `modulo()` и др.) будет доступно во всем тексте программы.

Важно помнить, что модель защиты, включаемая по умолчанию — это **единственное** отличие классов от структур. Больше они ничем друг от друга не отличаются, по крайней мере, с точки зрения компилятора Си++. Тем не менее, обычно программисты используют структуры для случаев, когда по смыслу поля должны быть открыты и доступны (например, при организации списков), а классы — для случаев, когда поля представляют собой детали реализации объекта, которые лучше всего скрыть.

Лекция 2

§ 2.1. Переопределение символов стандартных операций

В реальной задаче мы вряд ли захотим ограничиться единственным действием, осуществляемым с комплексными числами. Скорее всего, нам потребуются также сложение, вычитание, умножение и деление.

Язык Си++ предоставляет возможности записи таких действий с помощью привычных символов арифметических операций. Кроме того, логично предусмотреть возможность узнать действительную и мнимую части комплексного числа, если вдруг нам это понадобится. Для этого дополним наш класс еще несколькими функциями-членами:[1]

```
class Complex {
    double re, im;
public:
    Complex(double a_re, double a_im)
        { re = a_re; im = a_im; }
    double modulo() { return sqrt(re*re + im*im); }
    double argument() { return atan2(im, re); }
    double get_re() { return re; }
    double get_im() { return im; }
    Complex operator+(Complex op2)
    {
        Complex res(re+op2.re, im+op2.im);
        return res;
```

[1] Методы в этом примере обращаются к закрытым полям не только ≪своего≫ объекта, но и объекта, переданного как параметр. Это не нарушает защиту: как уже говорилось на стр. 10, единицей защиты является не объект, а класс или структура как таковые.

```
    }
    Complex operator-(Complex op2)
    {
        Complex res(re-op2.re, im-op2.im);
        return res;
    }
    Complex operator*(Complex op2)
    {
        Complex res(re*op2.re-im*op2.im,
                    re*op2.im+im*op2.re);
        return res;
    }
    Complex operator/(Complex op2)
    {
         double dvs = op2.re*op2.re+op2.im*op2.im;
         Complex res((re*op2.re+im*op2.im)/dvs,
                     (im*op2.re-re*op2.im)/dvs);
         return res;
    }
};
```

Необходимо пояснить, что слово `operator` является зарезервированным (ключевым) словом языка Си++. Стоящие подряд две лексемы `operator` и `+` образуют *имя функции*, которую можно вызвать и как обычную функцию:

```
a = b.operator+(c);
```

однако, в отличие от обычной функции, появление функции-операции позволяет записать то же самое в более привычном для математика виде:

```
a = b + c;
```

Теперь мы можем, к примеру, узнать, каков будет модуль суммы двух комплексных чисел:

```
Complex c1(2.7, 3.8);
Complex c2(1.15, -7.1);
double mod = (c1+c2).modulo();
```

Как мы увидим позже, аналогичным образом в языке Си++ можно переопределить символы любых операций, включая присваивания, индексацию, вызов функции и прочую «экзотику». Существуют только

две операции, которые нельзя переопределить: это тернарная ***условная операция*** (`a ? b : c`) и операция выборки поля из структуры или класса (точка). Необходимо сразу же заметить, что операция выборки поля по указателю (стрелка) переопределяема, хотя и несколько экзотическим способом. Ко всем этим вопросам мы вернёмся позже.

§ 2.2. Перегрузка имён функций

Прежде чем перейти к дальнейшему описанию возможностей, предоставляемых понятием конструктора в Си++, необходимо обратить внимание на свойство языка Си++, называемое ***перегрузкой имен функций***.

Язык Си++ позволяет ввести в рамках одной области видимости *несколько различных функций, имеющих одно имя и различающихся количеством и/или типами параметров.* Например, будет вполне корректен следующий код:

```
void print(int n) { printf("%d\n", n); }
void print(const char *s) { printf("%s\n", s); }
void print() { printf("Hello world\n"); }
```

При обработке вызова функции компилятор определяет, какую функцию из имеющих одно имя необходимо вызвать в данном конкретном случае, используя количество и типы фактических параметров. Например:

```
print(50);                 // вызывается print(int)
print("Have a nice day"); // вызывается print(const char*)
print();                   // версия без параметров
```

Таким же образом можно перегружать и методы в классах и структурах. В особенности это свойство оказывается полезным при описании конструкторов.

Отметим, что введение перегруженных функций может привести к совершенно неожиданным последствиям. Так, следующий код сам по себе корректен:

```
void f(const char *str)
{
    printf("Это строка: %s\n", str);
}
```

```
void f(float f)
{
    printf("Это число с плавающей точкой: %f\n", f);
}
```

Причем компилятор вполне справится с вызовами `f("string");` и `f(2.5);`, поскольку никаких разночтений тут не возникает. Однако вызов `f(0)` будет расценен компилятором как ошибочный, т.к. компилятор с совершенно одинаковым успехом может преобразовать `0` как к типу `const char*`, так и к типу `float`. Между тем, если бы в программе присутствовала только одна из двух вышеописанных функций `f` (любая!), вызов `f(0)` был бы заведомо корректен.

§ 2.3. Конструктор умолчания. Массивы объектов

Возвращаясь к нашему классу `Complex`, напомним (см. стр. 10), что описать переменную типа `Complex` привычным нам образом без всяких параметров нельзя, т.к. для единственного конструктора класса `Complex` требуются два параметра. Таким образом, следующий код будет некорректен:

```
Complex sum; // ошибка - нет параметров для конструктора
sum = c1+c2;
```

Кроме того, **мы не можем описать массив** комплексных чисел, поскольку синтаксис языка не позволяет задать параметры для конструкторов каждого элемента массива.

Решить проблему позволяет введение еще одного конструктора. Напомним, что компилятор считает конструктором функцию-член класса (или структуры), имя которой совпадает с именем класса (или структуры). Таким образом, нам необходимо ввести еще одну функцию-член с именем `Complex`. В языке Си++, в отличие от языка Си, это можно сделать благодаря перегрузке функций. Главное, чтобы функции, имеющие одинаковые имена, различались количеством и/или типом параметров. Итак, добавим в класс `Complex` еще один конструктор:

```
class Complex {
    double re, im;
public:
    Complex(double a_re, double a_im)
```

```
        { re = a_re; im = a_im; }
    Complex() { re = 0; im = 0; }

//....
```

Теперь мы можем описывать переменные типа `Complex`, не задавая никаких параметров. Поэтому конструктор, имеющий пустой список параметров, называют ***конструктором по умолчанию***. Его наличие делает возможными, в частности, следующие описания:

```
Complex z3; // используем конструктор по умолчанию
Complex v[50]; // конструктор будет вызван 50 раз
               // т.е. для каждого элемента
```

§ 2.4. Конструкторы преобразования

В языке Си присутствует так называемое *неявное преобразование типов*. Так, если в программе описана функция `void f(float)`, мы можем записать вызов `f(25)`, и компилятор сочтет такой код корректным, т.к. ≪знает≫, каким образом из целого числа сделать число с плавающей точкой; иначе говоря, в языке Си присутствует правило преобразования значений типа `int` в значения типа `float`.

Язык Си++ имеет средства указания правил преобразования для *типов данных, введенных пользователем*. Одним из таких средств являются ***конструкторы преобразования***.

Отвлекшись на время от конкретики языка Си++, заметим, что задать правило преобразования значений типа `A` в значения типа `B` — это то же самое, что задать инструкцию по созданию объекта типа `B` по имеющемуся значению типа `A`.

Как отмечалось в § 1.2.3, инструкции компилятору Си++ по созданию объектов класса на основе заданных параметров даются путем описания конструкторов с соответствующими параметрами. Таким образом, если ввести в классе `B` конструктор, получающий параметр типа `A`, это как раз и будет инструкция по созданию объекта типа `B` по имеющемуся значению типа `A`. Представляется поэтому вполне логичным использовать такие инструкции для *выполнения неявного преобразования типов*.

Итак, конструктор, который получает на вход ровно один параметр, имеющий тип, отличный от описываемого, называется ***конструкто-***

ром преобразования[2] и используется компилятором не только в случае явного создания объекта, но и для проведения неявного преобразования типов.

Проиллюстрируем сказанное на примере введенного ранее класса `Complex`. Ясно, что по смыслу комплексных чисел любое действительное число может быть преобразовано к комплексному добавлением нулевой мнимой части точно так же, как целое число преобразуется к числу с плавающей точкой добавлением нулевой дробной части. Чтобы выразить это соотношение между действительными и комплексными числами, дополним класс следующим конструктором:

```
Complex(double a) { re = a; im = 0; }
```

С одной стороны, этот конструктор позволяет нам описывать комплексные числа с указанием одного действительного параметра, например:

```
Complex c(9.7);
```

С другой стороны, если в программе имеется функция

```
void f(Complex a);
```

мы можем вызвать ее для действительного параметра:

```
f(2.7);
```

Благодаря наличию в классе `Complex` конструктора преобразования компилятор сочтет такой вызов корректным; для его обработки с помощью конструктора преобразования будет создан временный объект типа `Complex`, который и будет подан на вход функции `f`

§ 2.5. Ссылки

Понятие ***ссылки*** является ключевым для понимания дальнейшего материала. Отметим, что в языке Си ничего подобного нет, так что ссылки вызывают определенные трудности у многих студентов. Постарайтесь поэтому подойти к изучению данного параграфа особенно внимательно, а при возникновении вопросов обязательно задайте их вашему лектору или преподавателю.

[2] если только специальное значение такого конструктора не отменить директивой `explicit`

Ссылка в Си++ — это особый вид объектов данных, реализуемый путем хранения адреса объекта, но семантически эквивалентный самому объекту, на который ссылается. Иначе говоря, любые операции над ссылкой будут на самом деле производится над той переменной, на которую эта ссылка установлена.

Синтаксически тип данных «ссылка» описывается аналогично указателю, только вместо символа `*` используется символ `&`.[3]

Проиллюстрируем сказанное простым примером:

```
int i;       // целочисленная переменная
int *p = &i; // указатель на переменную i
int &r = i;  // ссылка на переменную i

i++;      // увеличить i на 1
(*p)++;   // то же самое через указатель
r++;      // то же самое по ссылке
```

Строго говоря, ссылки нельзя называть переменными, т.к. изменить значение *самой ссылки* нельзя: если ссылка встречается в левой части операции присваивания — это означает, что присваивание будет выполнено не для ссылки как таковой, а для той переменной, на которую она ссылается. Таким образом, описав ссылку на переменную `i`, мы на самом деле ввели *синоним* имени `i` — имя `r`, обозначающее тот же самый объект данных.

Такое использование ссылок может показаться бессмысленным (хотя в некоторых случаях оказывается полезно). Однако существенно более интересные возможности ссылочный тип в Си++ раскрывает при передаче его как параметра в функции и возврате из функций в качестве значения.

Так, благодаря ссылочному типу в нашем распоряжении оказывается отсутствовавший в языке Си механизм передачи параметров по ссылке.[4]

Допустим, нам нужно написать функцию, находящую максимальный и минимальный элементы заданного массива чисел типа float. На языке Си такая функция выглядела бы так:

```
void max_min(float *arr, int len, float *min, float *max)
```

[3] Важно понимать, что в данном случае символ & не имеет ничего общего с операцией взятия адреса! Почему автор языка Си++ Б. Страуструп выбрал именно такое обозначение — вопрос открытый.

[4] Читатель, скорее всего, уже знаком с передачей параметров по ссылке: в языке Pascal такие параметры называются параметрами-переменными (var-parameters).

```
{
    int i;
    *min = arr[0];
    *max = arr[0];
    for(i=1; i<len; i++) {
        if(*min>arr[i]) *min = arr[i];
        if(*max<arr[i]) *max = arr[i];
    }
}
```

а ее вызов — например, так:

```
float a[500];
float min, max;
// ...
max_min(a, 500, &min, &max);
```

Поскольку из функции необходимо вернуть больше одного значения, приходится использовать возврат через параметры. Между тем, в языке Си все параметры передаются *по значению*, поэтому нам приходится вручную *имитировать* передачу по ссылке, передавая значение адреса переменной, подлежащей модификации, и вместо имени переменной использовать леводопустимое выражение разыменования (т.е. ставить перед идентификаторами `min` и `max` символ операции разыменования `*`, чтобы преобразовать адрес переменной в саму переменную).

С использованием ссылок и код функции, и ее вызов обретают более ясный вид:

```
void max_min(float *arr, int len, float &min, float &max)
{
    int i;
    min = arr[0];
    max = arr[0];
    for(i=1; i<len; i++) {
        if(min>arr[i]) min = arr[i];
        if(max<arr[i]) max = arr[i];
    }
}
// ...
float min, max;
max_min(a, 500, min, max);
```

Довольно интересные возможности открывает использование ссылочного типа как типа возвращаемого значения функции. Допустим, необходимо произвести поиск целочисленной переменной в составе сложной структуры данных (например, искомая переменная является полем структуры, которая, в свою очередь, является элементом массива и т. п.), а затем либо использовать значение найденной переменной, либо выполнить над ней то или иное присваивание. В такой ситуации поиск переменной можно выделить в отдельную функцию, возвращающую ссылку на найденную переменную. Если такая функция имеет профиль

```
int &find_var(/*params*/);
```

то допустимы будут, например, такие операторы:

```
int x = find_var(/*...*/) + 5;
find_var(/*...*/) = 3;
find_var(/*...*/) *= 10;
find_var(/*...*/)++;
int y = ++find_var(/*...*/);
```

Лекция 3

§ 3.1. Модификатор const

Читателю, возможно, уже знакомо ключевое слово `const`, которое изначально появилось в Си++, но с недавних пор вошло и в стандарт языка Си. Оно показывает, что мы имеем дело с областью памяти, не подлежащей изменению.

Наиболее простое и очевидное применение слова `const` — это введение в программе ***именованных констант***.

Допустим, наша программа имеет некое свойство, задаваемое в виде числа (примером может служить ограничение на длину строки, читаемой со стандартного ввода). Обычно конкретное число снабжают именем и всю программу выстраивают таким образом, чтобы использовалось это имя, а не числа. В результате мы можем в любой момент изменить глобальное свойство программы, внеся лишь одно изменение в ее текст.

В классических версиях языка Си для введения имени константы использовался препроцессор. Например, ограничение на максимальную длину входного слова можно было задать следующим образом:

```
#define MAX_LINE_LENGTH 130
```

Появление модификатора `const` позволило в подобных ситуациях обходиться без препроцессора, вводя ***константу*** средствами самого языка:

```
const int max_line_length = 130;
```

Чтобы понять, почему этот вариант решения лучше предыдущего, рассмотрим следующую гипотетическую ситуацию. Пусть в нашей программе необходима константа с именем `c`, равная 15. Введем ее средствами препроцессора:

```
#define c 15
```

Представим себе теперь, что при работе над текстом программы мы в некий момент забыли о существовании макроса `c` и ввели, скажем, локальную переменную с таким именем:

```
void f(int i)
{
    int c;
    // ...
}
```

Вместо объявления `int c` компилятор благодаря работе макропроцессора "увидит" строку `int 15` и, разумеется, выдаст сообщение об ошибке. Однако понять, в чем ошибка заключается, будет весьма и весьма нелегко, особенно если мы *действительно* прочно забыли о существовании макроса `c`.

Если же константа `c` будет введена без применения препроцессора:

```
const int c = 15;
```

то описанная проблема не возникнет, т.к. имя `c`, встреченное в функции `f()`, будет воспринято компилятором как локальное и никакой ошибки не произойдёт.

Вообще, применения макропроцессора рекомендуется по мере возможности избегать, поскольку макроимена не подчиняются обычным для языка Си/Си++ соглашениям об областях видимости и другим правилам.

При описании адресных и ссылочных типов модификатор `const` позволяет указать, что область памяти, на которую указывает/ссылается описываемый объект, не подлежит изменению. Например:

```
const char *p;
    // p указывает на неизменяемую область памяти
p = "A string";
    // все в порядке, переменная p может изменяться
*p = 'a';
    // ОШИБКА! Нельзя менять область памяти,
    // на которую указывает p
p[5] = 'b';     // Это также ОШИБКА
```

Отметим еще раз, что `const char *p` — это именно *указатель на константу*, а не *константный указатель*. Чтобы описать константу адресного типа, необходимо расположить ключевые слова в другом порядке:

```
char buf[20];
char * const p = buf+5;
    // p - константа-указатель, указывающая на
    // шестой элемент массива buf (buf[5])
p = "A string";
    // ОШИБКА, значение p не может быть изменено
*p = 'a';
    // Все в порядке, изменяем значение buf[5]
p[5] = 'b';
    // Все в порядке, изменяем значение buf[5+5]
```

Аналогичным образом дело обстоит со ссылочными типами:

```
int i;
const int &r = i; // константная ссылка
int x = r+5; // все в порядке
i = 7;       // тоже все в порядке
r = 12;  // ОШИБКА, значение по ссылке r нельзя менять

const int j = 5;
int &jr = j;  //  ОШИБКА, нельзя ссылаться
              // обычной ссылкой на константу
const int &jcr = j; // все в порядке
```

Отметим, что константные ссылки позволяют передавать в функции в качестве параметра адрес переменной вместо копирования значения, при этом не позволяя эту переменную менять, как и при обычной передаче по значению. Это может быть полезно, если параметр представляет собой класс или структуру, размер которой существенно превышает размер адреса. Так, если в ранее описанном классе `Complex` вместо

```
Complex operator+(Complex op2)
    { return Complex(re+op2.re, im+op2.im); }
```

написать

```
Complex operator+(const Complex &op2)
    { return Complex(re+op2.re, im+op2.im); }
```

семантика кода не изменится, но физически (на уровне машинного кода) вместо копирования двух полей типа `double` будет происходить передача адреса существующей переменной и обращение по этому адресу.

§ 3.2. Константные методы

Использование модификатора `const` делает возможной ситуацию, в которой нам доступен некоторый `константный` объект, то есть переменная типа `class` или `struct`, которую нельзя изменять. При этом вполне возможно, что все поля объекта скрыты, то есть взаимодействие с объектом возможно только через методы. В то же время метод может, вообще говоря, изменить внутреннее состояние (то есть значения скрытых полей) объекта, нарушив требование о его неизменности. Поэтому для обеспечения неизменности объекта компилятор вынужден запрещать вызовы методов. Получается, что с таким объектом мы вообще никак не сможем работать, если не предпримем специальных мер.

Для работы с константными объектами в языке Си++ предусмотрены ***константные методы***. Константным считается метод, после заголовка которого (но перед телом) поставлен модификатор `const`:

```
class C1 {
    // ...
    void method(int a, int b) const
        { /* .... */ }
    // ...
 };
```

В теле такого метода поля объекта доступны, но запрещены действия, изменяющие их или способные привести к их изменению, в том числе присваивания полям новых значений, присваивания адресов этих полей неконстантным указателям, передача их в функции по неконстантным ссылкам и т. п.

Таким образом, относительно константного метода известно, что он заведомо не может изменить состояние объекта (значения его полей). Поэтому константные методы можно без опаски вызывать для объектов, которые нельзя изменять.

Например, такой код:

```
void f(const Myclass *p) {
    p->my_method();
}
```

будет корректным только в случае, если в классе `Myclass` метод `my_method()` объявлен как константный.

При написании программ **рекомендуется все методы, которые по своему смыслу не должны изменять состояние объекта, обязательно помечать как константные, разрешая, таким образом,**

вызывать их для константных объектов. В частности, в описанном ранее классе `Complex` все методы, кроме конструкторов, никаких изменений в поля класса не вносят. Чтобы с объектами типа `Complex` было удобнее работать, следует пометить все методы класса модификатором `const`:

```
class Complex {
    double re, im;
public:
    Complex(double a_re, double a_im)
        { re = a_re; im = a_im; }
    Complex(double a_re)
        { re = a_re; im = 0; }
    Complex() { re = 0; im = 0; }
    double modulo() const
        { return sqrt(re*re + im*im); }
    double argument() const
        { return atan2(im, re); }
    double get_re() const { return re; }
    double get_im() const { return im; }
    Complex operator+(const Complex &op2) const
        { return Complex(re+op2.re, im+op2.im); }
    // ...
};
```

§ 3.3. Деструкторы

Наряду с конструкторами, контролирующими процесс *создания* (инициализации) объектов, в языке Си++ предусмотрены также ***деструкторы***, предназначенные для контроля над процессом *уничтожения* объекта.

Представим себе, что объект в процессе своей деятельности захватывает некий ресурс (например, открывает файл), причем об этом известно только самому объекту, т.е. на захваченный ресурс ссылаются только закрытые поля объекта (в примере с файлом это может означать, что дескриптор открытого файла хранится в закрытом поле объекта). В случае, если объект по тем или иным причинам прекратит существование, ресурс так и останется захваченным.

Решить проблему позволяет описание в классе объекта функции-деструктора. Деструктор — это функция-член класса, вызов которой

автоматически вставляется компилятором в код для любой ситуации, в которой объект прекращает существование. Функция-деструктор имеет имя, представляющее собой имя описываемого типа (класса или структуры), к которому спереди добавлен знак ~ (тильда). Список параметров функции-деструктора всегда пуст, т.к. в языке отсутствуют средства передачи параметров деструктору. Поскольку деструктор играет специальную роль и в явном виде не вызывается, тип возвращаемого значения для деструктора также не указывается (деструктор никогда не возвращает никаких значений).

Проиллюстрируем понятие деструктора на примере класса `File`, инкапсулирующего дескриптор файла.

```
class File {
    int fd; // Дескриптор. -1 означает, что файла нет
public:
    File() { fd = -1; }
        // Конструктор устанавливает отсутствие файла

    bool OpenRO(const char *name) {
        fd = open(name, O_RDONLY);
        return (fd != -1);
    }   // метод пытается открыть файл на чтение,
        // возвращает true в случае успеха,
        // false в случае неудачи

    // ...
    // ... методы работы с файлом ...
    // ...

    ~File() { if(fd!=-1) close(fd); }
         // Деструктор закрывает файл, если он открыт
};
```

Деструктор будет вызван в любой ситуации, когда объект типа `File` прекращает существование. Например, если объект был описан как локальный в функции, то при возврате из функции (в том числе и досрочном вызове оператора `return`) для этого объекта отработает деструктор.

Вообще, **при создании объекта ровно один раз отрабатывает конструктор, при уничтожении объекта ровно один раз отрабатывает деструктор**.

§ 3.4. Операции работы с динамической памятью

Известно, что язык Си сам по себе не включает средств работы с динамической памятью; создание и уничтожение динамических структур данных вынесено в библиотеку и производится обычно с помощью функций `malloc()`, `realloc()` и `free()`.

В языке Си++ одновременно с выделением и освобождением памяти в некоторых случаях, а именно, при создании и удалении объекта типа структуры или класса, имеющего конструкторы и/или деструкторы, необходимо выполнять дополнительные действия, заданные этими конструкторами и деструкторами. Кроме того, при создании динамических объектов с помощью конструктора иного, нежели конструктор по умолчанию, необходима возможность указания параметров конструктора.

Функции `malloc()` и `free()` ничего не знают о конструкторах, деструкторах и параметрах, поэтому для создания и удаления объектов они непригодны.

Более того, информацией о конструкторах и деструкторах обладает только компилятор, поэтому в языке Си++ вообще невозможно вынести работу с динамической памятью из языка в библиотеку без введения дополнительных средств. Автор языка Си++ Бьерн Страуструп решил пойти более простым путем и внес в язык соответствующие синтаксические конструкции для создания и удаления объектов.

Для создания в динамической памяти одиночного объекта (переменной произвольного типа) в языке Си++ используется операция `new`, аргументом которой является имя типа создаваемого объекта. Например:

```
int *p;
p = new int;
```

Если создаваемый объект принадлежит типу, имеющему конструктор, и возникает необходимость передать конструктору параметры, то эти параметры указываются в скобках после имени типа. Так, создание объекта описанного ранее типа `Complex` (см. § 1.2.5) с указанием действительной и мнимой частей может выглядеть так:

```
Complex *p;
p = new Complex(2.4, 7.12);
```

Для удаления используется операция `delete`:

```
delete p;
```

Для создания и удаления динамических массивов используются специальные ***векторные формы*** операций `new` и `delete`, синтаксически отличающиеся наличием квадратных скобок:

```
int *p = new int[200]; // массив из 200 целых чисел
delete [] p;           // удаление массива
```

Эти формы отличаются тем, что соответствующий конструктор и деструктор вызываются для *каждого элемента массива*.

Необходимо отметить, что векторная форма операции `new` не имеет синтаксических средств для передачи параметров конструкторам, поэтому для создания массива элементов типа класс или структура *необходимо наличие у этого типа конструктора по умолчанию*, как и для обычных массивов (см.§ 2.3).

Важно знать, что объекты в динамической памяти, созданные с помощью векторной формы операции `new`, нельзя удалять с помощью обычной формы операции `delete` и наоборот. Дело в том, что реализация менеджера динамической памяти вправе выделять память под обычные переменные и под массивы из разных областей динамической памяти, имеющих, возможно, различную организацию служебных структур данных.

Также не следует удалять с помощью `delete` объекты, созданные функцией `malloc()`, и наоборот, не следует удалять обекты, созданные операциями `new`, с помощью `free()`. Все это может привести к непредсказуемым последствиям.

§ 3.5. Конструктор копирования

Рассмотрим следующую ситуацию. В реализации некоторого класса (назовем его `Cls1`) нам потребовался динамический массив, который мы создаем в теле конструктора класса; естественно, в деструктор следует поместить оператор для уничтожения этого массива.

```
class Cls1 {
    int *p;
public:
    Cls1() { p = new int[20]; }
    ~Cls1() { delete [] p; }
    // ...
};
```

Теперь предположим, что кто-то создает в программе копию объекта класса `Cls1`. Такое может произойти, например, если объект окажется передан *по значению* в качестве параметра функции. Например:

```
void f(Cls1 x) {
   // ...
}
int main() {
   // ...
   Cls1 c;
   f(c);
   //...
}
```

Проанализируем происходящее со структурами данных при вызове функции `f()`. Локальная переменная `x` является копией объекта `c`. Копия любого объекта данных создается путем обычного побитового копирования, если не указать иного. Следовательно, при копировании объекта класса `Cls1` скопирован окажется *указатель* на динамический массив; иначе говоря, у нас появятся два объекта, использующие один и тот же экземпляр динамического массива: оригинал объекта `c` и его локальная копия `x`. Возникшая ситуация показана на рис. 3.1.

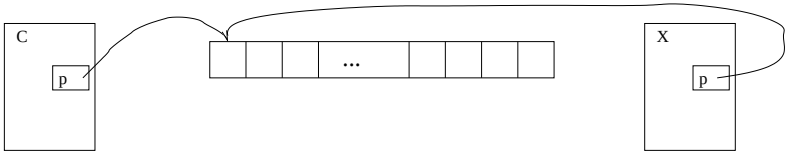

Рис. 3.1. Схема структуры данных после побитового копирования объекта

Даже если такое разделение не приведет к немедленным ошибкам (например, функция `f()` может изменить объект `x`, что отразится на внутреннем состоянии объекта `c`, чего мы могли и не ожидать), в любом случае при выходе из функции `f()` отработает деструктор (для объекта `x`), который уничтожит динамический массив, в результате чего объект `c` окажется в заведомо ошибочном состоянии, поскольку будет содержать указатель на уничтоженный массив. К возникновению ошибки теперь приведет любое действие с объектом `c`, если же никаких действий не предпринимать, то ошибка возникнет при уничтожении

объекта (когда деструктор попытается вновь уничтожить уже уничтоженный массив).

Очевидно, что побитовое копирование нас не устраивает и необходимо проинструктировать компилятор о том, каким именно способом можно корректно скопировать объект класса `Cls1`. В языке Си++ для таких случаев предусмотрен специальный случай конструктора, называемый ***конструктором копирования***.

Конструктор копирования — это конструктор, имеющий ровно один параметр, причем тип этого параметра представляет собой ссылку на объект данного (описываемого) класса. Обычно ссылка берется константная. Конструктор копирования представляет собой инструкцию компилятору относительно того, как *скопировать* объект данного типа, или, иначе говоря, как создать объект данного типа, уже имея один такой объект.

Снабдим конструктором копирования наш класс `Cls1`:

```
class Cls1 {
    int *p;
public:
    Cls1() { p = new int[20]; }
    Cls1(const Cls1& a) {
        p = new int[20];
        for(int i=0; i<20; i++) p[i] = a.p[i];
    }
    ~Cls1() { delete [] p; }
    // ...
};
```

Теперь ситуация при вызове функции `f()` будет выглядеть так, как показано на рис. 3.2.

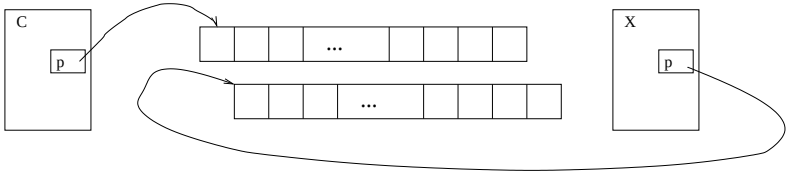

Рис. 3.2. Схема структуры данных после применения конструктора копирования

Лекция 4

§ 4.1. Значения параметров по умолчанию

§ 4.1.1. Параметры функций со значениями по умолчанию

Язык Си++ позволяет при объявлении функции (т.е. в том месте единицы трансляции, где впервые фигурирует прототип функции) указать для некоторых или всех ее параметров ***значения по умолчанию***, в результате чего при вызове функции можно будет указать меньшее количество параметров. Недостающие параметры компилятор подставит сам.

Например, опишем следующую функцию:

```
void f(int a = 3, const char *b = "string", int c = 5);
```

Теперь возможны такие вызовы:

```
f(5, "name", 10);
f(5, "name"); // то же, что f(5, "name", 5);
f(5);         // то же, что f(5, "string", 5);
f();          // то же, что f(3, "string", 5);
```

На значения по умолчанию накладываются определенные ограничения. Во-первых, выражение, задающее значение по умолчанию, должно быть константным, т.е. таким, которое может быть вычислено на этапе компиляции. Так, следующие описания ошибочны:

```
int a;
//...
void f(int b = a); // a - переменная
void f(int b = a+5); // значение a+5 зависит от переменной
```

```
void f(int b = strlen(s));
             // выражение содержит вызов функции strlen()
```

В то же время, следующие описания полностью корректны:

```
const int a = 15;
//...
void f(int b = a); // a - константа
void f(int b = a+5);
void f(int b = 37*(1024+2));
void f(int b = sizeof(int));
```

Во-вторых, для функции может быть задано значение по умолчанию любого количества ее параметров, но при этом *все параметры функции, следующие в списке параметров за первым, имеющим значение по умолчанию, должны также иметь значение по умолчанию.* Например, следующие описания корректны:

```
void f(int a = 0, int b = 10, int c = 20);
void f(int a, int b = 10, int c = 20);
void f(int a, int b, int c = 20);
```

а следующие — некорректны:

```
void f(int a = 0, int b, int c = 20);
void f(int a = 0, int b = 0, int c);
void f(int a = 0, int b, int c);
     // b и c должны иметь умолчание, т.к. его имеет a

void f(int a, int b = 10, int c);
     // c должен иметь умолчание, т.к. его имеет b
```

Это ограничение введено в язык, чтобы упростить компилятору поиск подходящей функции. Если при вызове указано n фактических параметров, компилятор сопоставляет их с n *первыми* формальными параметрами функции. Так, если задана функция

```
int f(int a, const char *str = "name", int *p = 0);
```

то следующий фрагмент будет ошибочен:

```
int x;
f(5, &x);
```

поскольку фактический параметр `&x` имеет тип `int *`, а сопоставлен он будет строго со вторым параметром функции `f`, который имеет тип `const char *`.

§ 4.1.2. Еще раз о видах конструкторов

В §§ 2.3, 2.4 и 3.5 мы рассматривали конструкторы специального вида, говоря, что

- конструктор без параметров воспринимается компилятором как *конструктор по умолчанию*
- конструктор с одним параметром, имеющим тип, отличный от описываемого, воспринимается компилятором как *конструктор преобразования*
- конструктор с одним параметром, имеющим тип «ссылка на описываемый класс или структуру», воспринимается компилятором как *конструктор копирования*

Учитывая возможность задания значений параметров по умолчанию, следует говорить, что компилятор воспримет как конструктор специального вида такой конструктор, который *допускает его вызов* с соответствующим количеством и типом параметров. Так, например, мы могли бы в классе `Complex` описать всего один конструктор, который бы служил и конструктором по умолчанию, и конструктором преобразования, и обычным конструктором от двух аргументов:

```
Complex(double a_re = 0, double a_im = 0)
   { re = a_re; im = a_im; }
```

В самом деле, такой конструктор может быть вызван и без параметров (то есть как конструктор по умолчанию), и с одним параметром типа `float` (то есть как конструктор преобразования из типа `float`).

§ 4.2. Неявные конструкторы

Если описать в программе на Си++ структуру или даже класс, не содержащий (по крайней мере на первый взгляд) никаких конструкторов, то переменную такого типа всё же окажется возможным создать. Это вполне понятно с позиций здравого смысла: ведь структуры в Си++ часто используются в их исходной роли, пришедшей из языка Си, то есть в качестве обычной структуры данных, безо всяких методов.

Между тем, семантика языка Си++ подразумевает, что каждый экземпляр структуры или класса является объектом, а для каждого объекта при его создании вызывается конструктор. Возникающее противоречие решается введением понятия ***неявного конструктора***.

Неявный конструктор — это конструктор, который генерируется компилятором автоматически, несмотря на отсутствие соответствующего конструктора в коде, описывающем класс или структуру.

Компилятор Си++ неявно генерирует только два вида конструкторов: конструкторы по умолчанию (то есть конструкторы без параметров) и конструкторы копирования. При этом конструктор копирования неявно генерируется для **любого класса или структуры, в которых программист не описал конструктор копирования явно**, то есть получается, что конструктор копирования на самом деле присутствует **вообще в любом классе или структуре**. Неявный конструктор копирования производит копирование наиболее очевидным способом: поля, которые сами имеют конструкторы копирования, копируются с помощью этих конструкторов, прочие поля — побитовым копированием.

С конструктором по умолчанию ситуация чуть сложнее: он генерируется неявно, **если программист не описал в структуре или классе вообще ни одного конструктора**. Если в классе или структуре явно описать хотя бы один конструктор (любой), компилятор не будет генерировать неявный конструктор по умолчанию, поскольку сочтёт, что программист взял заботу о конструировании в свои руки.

Именно поэтому в примерах, которые рассматривались в § 1.2.3, мы могли описывать переменные типа `str_complex` без всяких параметров, пока не ввели первый конструктор, после чего возможность описания переменных этого типа без указания параметров нами была утрачена до тех пор, пока мы (уже сами, явно) не определили конструктор по умолчанию.

Сделаем ещё одно важное замечание. Поскольку конструктор копирования может быть сгенерирован неявно, отсутствие явно описанного конструктора копирования не означает невозможности создания копии объекта. Существует, однако, способ **запретить копирование объектов некоторого класса**: для этого достаточно **описать конструктор копирования** явно, но сделать это **в приватной части класса**. Такой приём часто применяют для классов, объекты которых по смыслу копироваться не должны, чтобы исключить случайные ошибки, связанные, например, с передачей их по значению. Ясно, что объект, для которого копирование запрещено, не может быть ни передан по значению в функцию, ни возвращён из функции; это не исключает, разумеется, передачи по ссылке.

Забегая вперёд, отметим, что аналогичный приём можно применить и для запрещения присваивания объектов (именно, убрать в приватную

часть соответствующую форму оператора присваивания).

§ 4.3. Описание тела метода вне класса. Раскрытие области видимости

До сих пор все описываемые нами методы состояли из одной-двух строк кода. Конечно, так получается далеко не всегда, а размер тела метода, вообще говоря, не ограничен, как не ограничен и размер тела обычной функции.[1]

Вместе с тем, описание класса при наличии в нём нескольких методов значительного объёма может стать (в силу своих размеров) совершенно недоступным для понимания. Вообще говоря, чаще всего программисты читают описания классов, чтобы узнать, как с соответствующим классом работать. Ясно, что для этого необходим список (публичных) методов. Пока описание класса умещается на одной странице, список его методов можно охватить одним взглядом; если же в классе описаны громоздкие методы, для ознакомления с заголовками методов может потребоваться долгое перелистывание кода туда и обратно, причём в процессе поиска заголовка одного метода программист может успеть забыть про другие, так что изучение класса становится занятием долгим и утомительным.

Есть и ещё одна проблема с телами методов, которая возникает при написании многомодульных программ. В языке Си мы помещали описания структур, необходимые более чем в одном модуле, в заголовочные файлы. В Си++ с классами и структурами ситуация совершенно такая же: если мы хотим использовать некоторый класс или структуру в нескольких модулях, следует поместить описание этого класса или структуры в заголовочный файл и включить этот файл директивой `#include "..."` во все нужные модули.

Если при этом класс или структура содержит тела методов, это будет означать, что код, составляющий их тела, окажется откомпилирован в **каждом** из модулей. Если таких модулей много, в итоговом испол-

[1] Это, впрочем, не означает, что написание громоздких функций в чём-то правильно. Чем функция длиннее, тем труднее стороннему программисту понять, что она делает. Многие программисты придерживаются точки зрения, что всё описание любой функции (тело вместе с заголовком) должно укладываться в 25 строк или в крайнем случае быть чуть-чуть больше. Такую функцию можно охватить одним взглядом. Если код функции разбух и стал гораздо больше, из неё следует выделить логические части и оформить их в виде отдельных функций, то есть разбить одну функцию на несколько. Всё это справедливо и для методов.

няемом файле окажется значительное количество дублируемого кода: методы нашего класса (один и тот же код!) будут присутствовать в таком количестве копий, сколько модулей используют наш класс.

Обе проблемы (громоздкость описания класса и дублирование объектного кода методов) решаются вынесением тел методов за пределы описания класса (называемого также ***заголовком класса***). При этом в заголовке класса оставляется только прототип (заголовок) функции-метода, то есть тип возвращаемого значения, имя метода, список формальных параметров и (возможно) модификатор `const`, после чего вместо тела метода ставится точка с запятой, как и после обычного прототипа функции.

Тело функции-метода при этом описывается в другом месте, за пределами заголовка класса, возможно даже, что в другом файле (чаще всего это происходит, если заголовок класса вынесен в заголовочный файл; тела методов при этом описываются в файле реализации соответствующего модуля).

При описании функции-метода за пределами заголовка класса необходимо указать компилятору, что речь идёт именно о методе определённого класса, а не о простой функции. Это делается с помощью ***символа раскрытия области видимости***, в роли которого в Си++ выступает два двоеточия `::` (иногда программисты называют этот символ «четвероточием»).

Например, если мы описываем некий класс `C1` и в нём есть конструктор по умолчанию и некоторые методы `f()` и `g()`, вынесение описания методов за пределы класса может выглядеть так:

```
class C1 {
    // ...
public:
    C1();
    void f(int a, int b);
    int g(const char *str) const;
};

// ... //

C1::C1()
{
   // тело конструктора
}
```

```
void C1::f(int a, int b)
{
   // тело метода f
}

int C1::g(const char *str) const
{
   // тело метода g
}
```

Смысл «раскрытия области видимости» можно пояснить следующим образом. Имена полей и методов локализованы в классе в том смысле, что, если в классе есть метод с именем `f`, то вне класса мы можем использовать идентификатор `f` для других целей, либо вовсе не использовать его; появление идентификатора `f` вне класса не имеет никакого отношения к методу `f()`, описанному в классе. Вместе с тем, в некоторых случаях имя метода `f` всё же появляется вне класса именно как имя этого метода — например, при вызове его для объекта класса. Таким образом, в отличие от, например, локальных переменных в функциях, имена членов класса **доступны** вне класса (конечно, если они не закрыты механизмом защиты), но только если имеются указания на то, что требуется имя из класса; при вызове метода для объекта таким указанием считается тип объекта.

Говорят поэтому, что класс (или структура) представляет собой ***область видимости***. В области видимости имеются свои имена, не конфликтующие с именами вне её даже при совпадении идентификаторов. Вместе с тем, имена, локализованные в области видимости, от этого не становятся, вообще говоря, недоступны; они доступны, но только при наличии указаний на область видимости. Символ раскрытия области видимости как раз и является средством для таких указаний.

Можно сказать и так, что один и тот же метод имеет внутри класса `C1` (то есть в телах его методов) имя `f`, а вне класса — имя `C1::f`.

Отметим, что символ раскрытия области видимости применяется не только для описания методов вне заголовка класса. С другими случаями его применения мы столкнёмся позже.

§ 4.4. Инициализация членов класса в конструкторе

До сих пор мы задавали значения полей объекта с помощью обычного оператора присваивания в теле конструктора. Такой способ может оказаться неприемлемым; действительно, никто не мешает объявить в классе поле, имеющее, в свою очередь, тип класс, причём не имеющий конструктора по умолчанию.

Рассмотрим эту ситуацию подробнее. Пусть `A` — класс, единственный конструктор которого требует на вход два параметра типа `int`:

```
class A {
   // ...
public:
   A(int x, int y) { /*...*/ }
   // ...
};
```

Опишем теперь класс `B`, в котором есть поле типа `A`. Для простоты картины будем считать, что в классе `B` нам нужен только конструктор по умолчанию (для других конструкторов ситуация будет абсолютно аналогична):

```
class B {
   A a;
public:
   B();
   // ...
};
```

Рассмотрим теперь тело конструктора `B()`. В нём (то есть во время его работы) **уже** доступно поле `a`, а для этого, как мы знаем, должен был отработать конструктор класса `A`. Но ведь единственный конструктор класса `A` требует параметров! Как же их ему передать?

Чтобы решить возникшую проблему, в Си++ введён специальный синтаксис для ***инициализации полей объекта***. При описании конструктора между его заголовком и началом тела (открывающей фигурной скобкой) можно поставить двоеточие, после которого через запятую перечислить вызовы конструкторов для некоторых или всех полей класса. В рассматриваемом примере выглядеть это будет так:

```
B::B() : a(2, 3) { /*...*/ }
```

Таким образом мы указываем компилятору, что поле `a` следует инициализировать (сконструировать) с помощью конструктора от двух параметров (`2` и `3`). Обнаружив такой код, компилятор вставит вызов соответствующего конструктора класса `A` непосредственно в начале тела конструктора `B`.

Отметим, что так можно инициализировать любые поля, а не только поля типа класс. В частности, для нашего типа `Complex` конструкторы могли бы выглядеть и так:

```
class Complex {
    double re, im;
public:
    Complex(double a_re, double a_im)
        : re(a_re), im(a_im) {}
    Complex(double a_re)
        : re(a_re), im(0) {}
    Complex() : re(0), im(0) {}
    // ...
```

Сделаем ещё одно важное замечание. Инициализаторы полей должны следовать в списке (после двоеточия) в том же порядке, в котором соответствующие поля описаны в классе. Иное, формально говоря, не является ошибкой, но хороший компилятор обязательно выдаст предупреждение.

§ 4.5. Описание символов операций вне класса

В классе `Complex` мы перегружали символы арифметических операций в виде функций-методов. Благодаря наличию в классе конструктора преобразования возможно, например, прибавить к комплексному числу действительное:

```
Complex z, t;
// ...
z = t + 0.5;
```

(при этом константа `0.5` будет преобразована к объекту `Complex` с помощью конструктора преобразования). В то же время, следующая операция окажется ошибочной:

```
z = 0.5 + t; // ошибка!
```

Дело тут в том, что метод `operator+` может быть вызван только для объекта класса `Complex`, а константа `0.5` таковым не является. В отличие от аргументов функций, объекты, для которых вызываются методы, компилятор не преобразует.

Эту проблему также можно решить. Дело в том, что операцию сложения двух комплексных чисел можно представить не только как метод самого комплексного числа (с одним параметром, по принципу «прибавь к себе этот параметр и скажи, что получится»), но и как стороннюю функцию, которая по двум заданным комплексным числам выдаёт другое комплексное число. Итак, уберём `operator+` из класса `Complex`, а вне класса напишем следующее:

```
Complex operator+(const Complex& a, const Complex& b)
{
    return Complex(a.get_re() + b.get_re(),
                   a.get_im() + b.get_im());
}
```

Теперь мы можем написать:

```
Complex z, t;
// ...
z = t + 0.5;
z = 0.5 + t;
```

и никаких ошибок это не вызовет.

Перегрузка символов стандартных операций в виде отдельных функций (а не методов в классах) может оказаться полезной также и в том случае, если нам необходимо ввести операцию для объектов некоторого класса, который мы по тем или иным причинам не можем изменить; например, этот класс может быть частью библиотеки, созданной другим программистским коллективом, от которой у нас нет исходных текстов (к сожалению, несмотря на успехи OpenSource, такие ситуации пока что не редкость).

Лекция 5

§ 5.1. Дружественные функции и классы

В некоторых случаях бывает полезно сделать исключения из запретов, налагаемых механизмом защиты.

В языке Си++ класс или структура, имеющие защищенную часть, могут объявить ту или иную функцию своим «другом» (friend); в этом случае из тела такой ***дружественной функции*** все детали реализации класса или структуры будут доступны.

Можно объявить «дружественным» также целый класс или структуру; эффект от этого будет такой же, как если бы мы объявили дружественными все методы дружественного класса. Иначе говоря, если в классе `A` будет заявлено, что класс `B` является для него дружественным, то во всех методах класса `B` будут доступны все детали реализации класса `A`.

Применять механизм дружественных функций и классов следует с осторожностью. Как мы видели в § 1.2.4, защита деталей реализации класса — механизм очень полезный; как можно догадаться, необдуманные исключения из него могут нанести существенный вред.

Одна из ситуаций, в которых применение механизма дружественных функций можно считать практически безопасным — это вынос перекрытых символов стандартных операций за пределы класса, как это обсуждалось в предыдущем параграфе. Как можно заметить, в теле функции `operator+` мы были вынуждены пользоваться методами `Complex::get_re()` и `Complex::get_im()`, поскольку из функции, не являющейся формально методом класса `Complex`, прямого доступа к полям `re` и `im` нет. Однако использование методов доступа (так называемых аксессоров) не всегда удобно, к тому же их может попросту не быть; часто бывает так, что некоторые детали реализации не следует делать доступными даже через метод доступа.

Ситуацию можно исправить с помощью механизма дружественных функций. Для начала в заголовок класса `Complex` вставим директиву `friend`, объявив функцию `operator+` дружественной. Для этого нужно записать прототип дружественной функции, предварив его директивой `friend`:

```
class Complex {
    friend Complex operator+(const Complex& a,
                             const Complex& b);
    //...
```

Теперь мы можем переписать функцию `operator+`, используя напрямую поля складываемых объектов (то есть без обращения к методам `get_re()` и `get_im()`):

```
Complex operator+(const Complex& a, const Complex& b)
{
    return Complex(a.re + b.re, a.im + b.im);
}
```

Сравните это описание с тем, которое мы приводили на стр. 42.

Конечно, дружественной может быть и обычная функция:

```
class Cls1 {
    friend void f(int, const char *);

    //...
};

void f(int, const char *)
{
    // здесь можно использовать закрытые поля Cls1
}
```

Чтобы сделать дружественным сразу целый класс, следует после слова `friend` написать «`class` *имя*»:

```
class A {
    friend class B;
    //...
};
```

Использовать такой вариант «дружбы» следует с особенной осторожностью, только в ситуации, когда понятия, описываемые обоими классами, тесно связаны между собой; желательно сначала задать себе вопрос, нельзя ли обойтись без `friend`. Бóльшим злом, чем использование «дружественных» отношений, является разве что введение публичных методов доступа к деталям, которые по смыслу должны быть скрыты, либо снятие защиты с внутренних полей: ясно, что «дружественность» всё-таки ограничивает объём кода, который придётся просмотреть при изменении реализации класса, так что использование `friend` заведомо предпочтительнее полного отказа от защиты.

С примерами ситуаций, когда применение «дружественных» классов оказывается оправданно, мы столкнёмся в нашем курсе позже.

§ 5.2. Особенности переопределения некоторых операций

§ 5.2.1. Переопределение операций присваивания

Напомним, что присваивание является в языках Си и Си++ *операцией*, а не оператором, как в Паскале и большинстве других языков. Это выражается в том, что запись вида «`a =` *выражение*» сама по себе является выражением, имеющим значение, и, таким образом, может входить в состав других выражений.

То же самое можно сказать и про операции присваивания, совмещённые с арифметическими действиями, такие как `+=`, `-=`, `<<=`, `|=` и др.

Как мы уже отмечали, все эти операции могут быть переопределены для классов и структур, введённых пользователем. Здесь действует, однако, некоторое ограничение: все операции присваивания могут переопределяться **только как методы класса или структуры**, а определять их в виде обычных функций нельзя.

Как правило, при переопределении операций присваивания учитывают, что обычное присваивание возвращает значение, которое только что было присвоено. Из этих соображений из операции присваивания возвращают либо копию объекта, либо (что предпочтительнее) константную ссылку на объект, для которого операция была вызвана.

Это, однако, не обязательно: язык Си++ не требует именно такого оформления операций присваивания. Можно, в частности, сделать операцию присваивания функцией типа `void`, то есть не возвращающей

никакого значения.

Например, для нашего класса `Complex` мы могли бы написать такие операции:

```
class Complex {
    // ...
    const Complex& operator=(const Complex& c)
        { re = c.re; im = c.im; return *this; }
    const Complex& operator+=(const Complex& c)
        { re += c.re; im += c.im; return *this; }
    // ...
};
```

или, если нас не слишком волнуют семантические традиции, можно было бы написать и так:

```
class Complex {
    // ...
    void operator=(const Complex& c)
        { re = c.re; im = c.im; }
    void operator+=(const Complex& c)
        { re += c.re; im += c.im; }
    // ...
};
```

Здесь стоит ответить на один вопрос, часто возникающий у студентов. Операции `+=`, `-=` и другие подобные им являются с точки зрения языка Си++ полностью самостоятельными, то есть, например, операция `+=` никак не связана ни с операцией `=`, ни с операцией `+`. Если мы опишем в некотором классе операции `=` и `+`, это само по себе не даст нам возможности применять к объектам этого класса операцию `+=`, она должна быть описана отдельно.

Аргумент операции присваивания не обязан иметь тот же тип, что и описываемый класс. Так, для комплексных чисел мы могли бы определить и присваивание комплексной переменной действительного числа, например, так:

```
class Complex {
    // ...
    void operator=(double x)
        { re = x; im = 0; }
    // ...
};
```

Однако операция присваивания, аргумент которой представляет объект того же класса (или ссылку на такой объект, как в предыдущем примере), имеет одну важную особенность: как и некоторые конструкторы, такая операция *генерируется неявно*, если её не описать. Это вполне понятно, ведь в языке Си можно было присваивать между собой переменные, имеющие один структурный тип, и эта возможность была унаследована в Си++.

В случае, если по каким-либо причинам присваивание объектов описываемого класса желательно запретить, достаточно описать операцию присваивания с аргументом «константная ссылка на объект описываемого класса» **в приватной части класса**, например:

```
class A {
    // ...
public:
    // ...
private:
    void operator=(const A& ref) {}
};
```

§ 5.2.2. Переопределение операции индексирования

Операция извлечения элемента из массива, обозначаемая квадратными скобками, как известно, является арифметической операцией над указателем и целым числом; в языке Си выражение «`a[b]`» полностью эквивалентно выражению «`*(a+b)`».

В языке Си++ эту операцию можно переопределить для объектов класса или структуры, заставив, таким образом, объект в некоторых случаях выглядеть синтаксически похожим на обычный массив или даже просто выполнять роль массива.

Для примера опишем класс, объект которого представляет собой массив целых чисел с заранее неизвестным размером, который при обращении к несуществующим (пока) элементам автоматически увеличивается в размерах. Хранить элементы массива будем в динамически создаваемом массиве, скрытом в приватной части класса. Исходно создадим массив размером 16 элементов, а при возникновении такой необходимости будем удваивать его размеры до тех пор, пока нужный нам индекс не станет допустимым.

Заметим, что при попытке копирования объекта такого класса возникнет проблема с разделением одного и того же массива в динамической памяти между двумя объектами класса (подробно проблема опи-

сана в § 3.5). В нашей упрощенной версии мы просто запретим присваивание и копирование, описав фиктивные конструктор копирования и операцию присваивания в приватной части. Описание полноценного конструктора копирования и полноценной операции присваивания оставим читателю в качестве упражнения.

Для начала напишем заголовок класса:

```
class IntArray {
    int *p;   // указатель на хранилище
    int size; // текущий размер хранилища
public:
    IntArray() {
        size = 16;
        p = new int[size];
    }
    ~IntArray() { delete[] p; }
    int& operator[](unsigned int idx);
private:
    void Resize(int required_index);
        // запретим копирование и присваивание
    void operator=(const IntArray& ref) {}
    IntArray(const IntArray& ref) {}
};
```

Тела конструктора и деструктора имеют сравнительно небольшой размер, поэтому мы совершенно спокойно можем оставить их внутри заголовка класса. С другой стороны, тело операции индексирования будет состоять из нескольких строк, поэтому мы опишем его отдельно. Для лучшей ясности его реализации мы предусмотрели также вспомогательную функцию `Resize()`, которая будет осуществлять изменение размера массива. Поскольку эта функция, очевидно, является деталью реализации и не предназначена для пользователя (с точки зрения пользователя наш массив представляется бесконечным), мы скрыли эту функцию в приватной части класса.

Читатель, возможно, обратил внимание на тип возвращаемого значения операции индексации. Функция `operator[]()` возвращает **ссылку** на соответствующий элемент массива, чтобы сделать возможным как выборку значения из массива, так и присваивание его элементам новых значений.

Опишем теперь тело операции индексации:

```
int& IntArray::operator[](unsigned int idx) {
```

```
    if(idx >= size)
        Resize(idx);
    return p[idx];
}
```

Нам осталось описать функцию `Resize()`:

```
void Resize(int required_index) {
    int new_size = size;
    while(new_size <= required_index)
        new_size *= 2;
    int *new_array = new int[new_size];
    for(int i = 0; i < size; i++)
        new_array[i] = p[i];
    delete[] p;
    p = new_array;
    size = new_size;
}
```

Теперь мы можем в программе использовать, например, такие операторы:

```
IntArray arr;
arr[500] = 15;
arr[1000] = 30;
arr[10] = arr[500] + 1;
arr[10]++;
```

Операция индексации должна иметь строго один параметр, но этот параметр, вообще говоря, может быть любого типа. Это позволяет создавать «массивы», использующие в качестве индекса текстовые строки или даже объекты других классов. Хранить элементы массива мы также можем любым удобным нам способом, например, в виде списка (в некоторых случаях это может быть оправдано). Более того, возможно организовать объект, выглядящий как массив, но хранящий свои элементы в файле на диске, так что операция индексирования реально будет представлять собой операцию чтения или записи в файл.

§ 5.2.3. Переопределение операций ++ и --

Как и операции присваивания, операции инкремента и декремента (++ и --) могут переопределяться исключительно методами класса или

структуры, то есть их нельзя переопределять отдельной функцией вне класса.

У этих операций есть ещё одна особенность. Можно было бы переопределять их как обычные унарные операции (то есть в виде метода без параметров), если бы не то обстоятельство, что каждая из этих операций имеет две формы: префиксную (`++i`) и постфиксную (`i++`). Вообще говоря, эти две формы представляют собой *различные* операции.

Чтобы различать эти формы между собой, принято следующее соглашение. Поскольку префиксная форма более «традиционна» для унарных операций, метод с именем `operator++()` или `operator--()` без параметров определяет *префиксную* форму соответствующей операции (например, `++i`).

Что касается *постфиксной* формы, то она переопределяется функцией с тем же именем `operator++`, но имеющей один (фиктивный) параметр типа `int`. Наличие параметра благодаря перегрузке имён функций позволяет иметь две функции с одним и тем же именем; параметр, введённый исключительно ради этого различия, реально никогда не используется.

Приведём пример. Пусть класс `A` описан следующим образом:

```
class A {
public:
    void operator++() { printf("first\n"); }
    void operator--() { printf("second\n"); }
    void operator++(int) { printf("third\n"); }
    void operator--(int) { printf("fourth\n"); }
};
```

Рассмотрим теперь фрагмент кода:

```
A a;
++a;   // first
a++;   // third
--a;   // second
a--;   // fourth
```

В результате выполнения этого фрагмента будут напечатаны (в столбик) слова `first`, `third`, `second` и `fourth`, именно в таком порядке.

§ 5.2.4. Переопределение операции ->

Пожалуй, наиболее экзотическим образом обстоят дела с операцией `->`. Напомним, что эта операция в языке Си означает выборку поля из

структуры, на которую указывает заданный адрес. В Си++ она означает практически то же самое; естественно, с её помощью можно также обращаться и к методам, а работать она может как со структурами, так и с классами.

Перегрузка этой операции обычно требуется нам, если мы хотим создать объект, ведущий себя подобно указателю, но при этом выполняющий некоторые дополнительные действия.

Отметим для начала один неочевидный факт. Операция `->`, несмотря на её внешний вид, является *унарной*, то есть имеет всего один аргумент (указатель). Действительно, пусть у нас описана структура

```
struct s1 {
    int a, b;
};
```

и имеется указатель на неё:

```
s1 *p = new s1;
```

Теперь обращение к полю `a` будет выглядеть так: `p->a`. Здесь как будто бы два операнда, `p` и `a`; но ведь `a` — это *имя поля*, которое никоим образом не является самостоятельным выражением, не имеет ни типа, ни значения! Иначе говоря, на месте `a` не может стоять ничего, кроме имени поля. Язык не содержит никаких иных выражений, *вычисляющихся* в значение, способное заменить имя поля.

Таким образом, единственным полноценным операндом в выражении `p->a` является адрес, представленный указателем `p`. Заметим, в отличие от `a`, на месте `p` может стоять выражение произвольной сложности, вычисляющее значение типа `s1*` (адрес структуры `s1`).

Итак, мы имеем дело с *унарной* операцией, полным именем которой можно считать `->a` (операция выборки поля `a`). Таким образом, можно при желании считать, что символ `->` задаёт целое семейство операций. Впрочем, будучи семантически безупречным, на практике такое рассмотрение нам не понадобится; здесь мы приводим его только для иллюстрации всего вышесказанного.

Вернёмся к вопросу о переопределении операции `->`. Наряду с многими другими операциями, операция `->` переопределяется исключительно методами класса или структуры, то есть не может быть переопределена отдельной функцией. Чтобы избежать рутинной работы по переопределению отдельных операций для выборки каждого поля, в Си++ принято неочевидное, но вполне работающее соглашение: операция `->` определяется методом `operator->()`, не имеющим параметров

(как и обычные унарные операции), но при этом метод обязан возвращать либо *указатель на некую другую структуру или класс*, либо объект (или ссылку на объект), для которого, в свою очередь, операция `->` переопределена как метод

Таким образом, компилятор вставляет в код последовательно вызовы методов `operator->()`, пока очередной метод не окажется возвращающим обычный указатель на структуру или класс; именно по этому указателю и производится в итоге выборка заданного поля.

Попробуем проиллюстрировать сказанное на примере. Пусть у нас есть структура `s1` и нам необходим класс, объекты которого будут вести себя как указатели на `s1`, но при уничтожении такого «указателя» (например, при завершении функции, в которой он описан как локальная переменная) объект, на который он указывает, будет уничтожаться. Начнём описание класса:

```
class Pointer_s1 {
    s1 *p;
public:
```

Будем предполагать, что хранящийся внутри объекта простой указатель на `s1` может иметь и нулевое значение, что будем расценивать обычным образом как отсутствие объекта; естественно, в этом случае удалять ничего не будем. Опишем теперь конструктор и деструктор:

```
    Pointer_s1(s1 *ptr = 0) { p = ptr; }
    ~Pointer_s1() { if(p) delete p; }
```

Для удобства работы введём операцию присваивания обычного адреса:

```
    s1* operator=(s1 *ptr) {
        if(p) delete p;
        p = ptr;
    }
```

Заметим, что копирование и присваивание объектов класса `Pointer_s1` заведомо приведёт к ошибкам, так как один и тот же объект типа `s1` в этих случаях будет удаляться дважды. В связи с этим запретим присваивание и копирование объектов класса `Pointer_s1`, убрав конструктор копирования и соответствующий оператор присваивания в приватную часть:

```
private:
    Pointer_s1(const Pointer_s1&) {}
```

```
        // copying prohibited
    void operator=(const Pointer_s1&) {}
        // assignments prohibited
```

Наконец, опишем операции, которые превратят объекты нашего класса в подобие указателя на `s1`, а именно, операции разыменования (унарную `*`) и выборки поля (`->`), и завершим описание класса:

```
public:
    s1& operator*() { return *p; }
    s1* operator->() { return p; }
};
```

Такой класс удобно использовать, например, внутри функций. Так, если в начале некоторой функции описать объект класса `Pointer_s1`, то ему можно будет присваивать адреса новых экземпляров `s1`, причём он будет каждый раз автоматически удалять старый экземпляр, а когда работа функции завершится, автоматически удалит последний из экземпляров `s1`:

```
int f() {
    Pointer_s1 p;
    p = new s1;
    p->a = 25;
    p->b = p->a + 36;
    // ...
    // при завершении f
    // память будет освобождена
}
```

§ 5.2.5. Переопределение операции вызова функции

Операция вызова функции, синтаксически представляющая собой постфиксную операцию, символом которой служат круглые скобки (возможно, содержащие список параметров), может быть, как и другие операции, переопределена. Поскольку вызов функции обозначается круглыми скобками, имя функции, которая переопределяет эту операцию, будет состоять из слова `operator` и круглых скобок; как обычно, после имени функции записывается список формальных параметров. Например, операция вызова функции без параметров записывается так:

```
void operator()() { /* тело */ }
```

Наряду с многими другими операциями, вызов функции переопределяется только методом класса.

Приведём пример. Опишем класс `Fun`, для которого переопределены операции вызова функции без параметров, а также с одним и двумя целочисленными параметрами:

```
class Fun {
public:
    void operator()()
        { printf("fun0\n"); }
    void operator()(int a)
        { printf("fun1: %d\n", a); }
    void operator()(int a, int b)
        { printf("fun2: %d %d\n", a, b); }
};
```

Теперь мы можем написать следующий фрагмент кода:

```
Fun f;
f();
f(100);
f(25, 36);
```

В результате выполнения этого фрагмента будет напечатано:

```
fun0
fun1: 100
fun2: 25 36
```

§ 5.2.6. Переопределение операции преобразования типа

Рассмотрим следующий фрагмент кода:

```
int i;
double d;
// ...
i = d;
```

В строчке, содержащей присваивание, задействована (неявно) ***операция преобразования типа выражения***, позволяющая в данном случае построить значение типа `int` на основе имеющегося значения типа `double`.

Аналогичную возможность неявного преобразования можно предусмотреть и для классов, введённых программистом. Один из способов этого мы уже знаем — это ***конструктор преобразования*** (см. § 3.5).

В некоторых случаях применить конструктор преобразования не удаётся. В частности, конструктором можно задать преобразование из базового типа в тип, описанный пользователем, но не наоборот. Кроме того, иногда бывает по каким-то причинам невозможно изменить описание некоторого класса, но в программе нужно преобразование в объекты этого класса. Бывают и другие (довольно экзотические) ситуации.

Операция неявного преобразования типов определяется методом, имя которого состоит из слова `operator` и имени типа, к которому необходимо преобразовывать тип выражения. Тип возвращаемого значения для такой функции не указывается, так как он определяется именем. Например:

```
class A {
    //...
public:
    //...
    operator int() { /* ... */ }
};
```

Наличие в классе A операции преобразования к `int` делает возможным, например, такой код:

```
A a;
int x;
// ...
x = a;
```

Учтите, что чрезмерное увлечение переопределениями операции преобразования приводит, как правило, к тому, что компилятор находит больше одного способа преобразования из одного типа к другому и в результате не применяет ни одного из них, выдавая ошибку. Поэтому на практике переопределение операции преобразования типа используется крайне редко.

Лекция 6

§ 6.1. Пример: разреженный массив

Под ***разреженным массивом*** понимается массив относительно большого объема (например, состоящий из нескольких миллионов элементов), большинство элементов которого равны одному и тому же значению (чаще всего нулю) и лишь некоторые элементы, количество которых ничтожно в сравнении с размером массива, от этого значения отличаются.

Естественно, в такой ситуации целесообразно хранить в памяти лишь те элементы массива, значения которых отличны от нуля (или другого значения, которому равно большинство элементов).

Попробуем написать на Си++ такой класс, который ведёт себя как целочисленный массив потенциально бесконечного размера[1] в том смысле, что к объекту этого класса применима операция индексирования (квадратные скобки), однако объект реально хранит только те элементы, значения которых отличаются от нуля. Назовём этот класс `SparseArrayInt`.

Для простоты картины будем считать, что количество ненулевых элементов массива настолько незначительно, что даже линейный поиск среди них может нас устроить по быстродействию. Это позволит хранить ненулевые элементы массива в виде списка пар «индекс/значение».

Все детали реализации, включая и структуру, представляющую звено списка, скроем в приватной части класса. Это позволит при необходимости сменить реализацию на более сложную.

Основная проблема при реализации разреженного массива возни-

[1] На самом деле мы будем использовать в качестве индекса тип `unsigned long`; значения этого типа могут достигать $2^{32} - 1$, т.е. чуть больше четырёх миллиардов

кает в связи с поведением операции индексирования. Дело в том, что выражение «элемент массива» может встречаться как в правой, так и в левой части присваиваний, и в обоих случаях используется одна и та же функция-метод класса `SparseArrayInt`, называемая `operator[]`. Если бы все элементы массива хранились в памяти (как это было в примере из § 5.2.2), можно было бы просто возвратить ссылку на соответствующее место в памяти, где хранится элемент, соответствующий заданному индексу. С разреженным массивом так действовать не получится, поскольку в большинстве случаев элемент в памяти не хранится. Однако же просто возвратить нулевое значение (без всякой ссылки) нельзя, ведь это означало бы, что выражение, содержащее нашу операцию индексирования, нельзя применять в левой части присваивания.

Итак, что же делать в случае, если операция индексирования вызвана для индекса, для которого соответствующий элемент в памяти не хранится, поскольку равен нулю? Одно из возможных решений — завести в памяти соответствующий элемент, пусть даже и с нулевым значением, и вернуть соответствующую ссылку. Это позволит, безусловно, выполнять присваивания, однако приведёт к тому, что при каждом обращении к нулевым элементам (в том числе на чтение, а не на запись) соответствующие элементы будут заводиться в памяти и, таким образом, окажется, что изрядное количество памяти занято теми самыми нулевыми значениями, хранения которых можно избежать.

Схему можно несколько усовершенствовать, если каждый раз при вызове операции индексирования сохранять в приватной части объекта значение индекса или даже адрес заводимого звена списка. Это позволит при следующем вызове проверить, хранит ли запись, заведённая на предыдущем вызове, число, отличное от нуля, и если нет, то удалить её, освободив память. У такого варианта реализации есть, однако, свой недостаток: на заведение и удаление элементов будет тратиться изрядное количество времени.

Очевидно, было бы великолепно иметь возможность вызывать разные функции-методы для случаев, когда происходит обращение к элементу массива на чтение и когда индексация используется как присваивание. В первом случае можно было бы тогда возвращать просто значение элемента или ноль, если соответствущего элемента в памяти нет; во втором случае можно было бы без особых проблем создавать новый элемент и возвращать ссылку на поле, хранящее значение; это, впрочем, не исключает необходимости проверки на следующем вызове, не оказался ли в соответствующем элементе ноль. К сожалению, такой возможности в Си++ нет, как нет и возможности определить из тела

операции индексирования, вызвана она из левой части присваивания или нет.[2]

Соответствующую возможность, однако, можно *смоделировать*. Никто не мешает нам описать небольшой вспомогательный класс, объекты которого как раз и будет возвращать наша операция индексирования. Такой вспомогательный объект будет просто хранить всю имеющую отношение к делу информацию, а в нашем случае такой информации немного: адрес главного объекта (то есть самого разреженного массива), с которым нужно работать, и значение индекса, переданное в качестве параметра операции индексирования.

Имея эту информацию и соответствующий доступ к объекту массива, наш вспомогательный объект в зависимости от ситуации может произвести любые действия, которые в этой ситуации могут потребоваться. Что же касается определения, какая из возможных ситуаций имеет место, то у вспомогательного объекта для этого есть существенно больше возможностей, чем было в теле операции индексирования: ведь все операции, применяемые к результату операции индексирования, как раз и будут на самом деле применены к этому объекту.

Таким образом, достаточно, например, определить для нашего вспомогательного класса операции преобразования к типу `int` и присваивания, чтобы сделать возможным и корректным следующий код:

```
SparseArrayInt arr;
int x;
//...
x = arr[500];  // чтение
arr[300] = 50; // запись (возможно создание элемента)
arr[200] = 0;  // удаление элемента, если таковой есть
```

К сожалению, у такой реализации есть и недостаток. Дело в том, что обычное присваивание — это далеко не единственная операция, способная изменить значение целочисленной переменной. Пользователь вполне может попытаться использовать, например, такие выражения:

```
arr[15] += 100;
x = arr[200]++;
y = --arr[200];
```

Чтобы все подобные операции работали, нам придётся для нашего вспомогательного объекта определить их все явным образом, что потребует

[2]Это вполне можно считать недостатком Си++, однако критика имеющегося языка выходит за рамки данного пособия.

изрядного объёма работы: языки Си и Си++ поддерживают десять операций присваивания, совмещённых с арифметическими действиями (+=, -=, <<= и т. п.), плюс к тому четыре операции инкремента/декремента (++ и -- в префиксной и постфиксной форме).

В нашем примере мы ограничимся описанием операции += и двух форм операции ++ Остальные подобные операции читатель без труда опишет самостоятельно по аналогии.

Прежде чем продемонстрировать заголовок класса `SparseArrayInt`, отметим, что для удобства реализации всех модифицирующих операций целесообразно в классе вспомогательного объекта ввести две внутренние функции. Функция `Provide` будет отыскивать в списке элемент, соответствующий заданному индексу, и возвращать ссылку на поле, содержащее значение; если соответствующего элемента в списке нет, функция будет его создавать. Вторая функция, `Remove`, будет удалять из списка элемент с заданным индексом, если таковой в списке присутствует.

Вспомогательный класс назовём `Interm` от английского intermediate. С учётом этого заголовок класса будет выглядеть так:

```
class SparseArrayInt {
    struct Item {
        int index;
        int value;
        Item *next;
    };
    Item *first;
public:
    SparseArrayInt() : first(0) {}
    ~SparseArrayInt();
    class Interm {
        friend class SparseArrayInt;
        SparseArrayInt *master;
        int index;
        Interm(SparseArrayInt *a_master, int ind)
            : master(a_master), index(ind) {}
        int& Provide(int idx);
        void Remove(int idx);
    public:
        operator int();
        int operator=(int x);
        int operator+=(int x);
```

```
        int operator++();
        int operator++(int);
    };
    friend class Interm;

    Interm operator[](int idx)
        { return Interm(this, idx); }
private:
    SparseArrayInt(const SparseArrayInt&) {}
    void operator=(const SparseArrayInt&) {}
};
```

В самом классе `SparseArrayInt`, как можно заметить, оказывается не так много функций: основная функциональность оказывается вытеснена в методы вспомогательного класса. Единственным методом основного класса, тело которого в силу размеров не включено в заголовок класса, является его деструктор. Его задача — освободить память от элементов списка. Деструктор описывается так:

```
SparseArrayInt::~SparseArrayInt() {
    while(first) {
        Item *tmp = first;
        first = first->next;
        delete tmp;
    }
}
```

Опишем теперь методы класса `SparseArray::Interm`, в которых и заключается основная функциональность нашего разреженного массива:

```
int SparseArrayInt::Interm::operator=(int x)
{
    if(x == 0) {
        Remove(index);
    } else {
        Provide(index) = x;
    }
    return x;
}
int SparseArrayInt::Interm::operator+=(int x)
{
    int& location = Provide(index);
```

```
    location += x;
    int res = location;
    if(res == 0) Remove(index);
    return res;
}
int SparseArrayInt::Interm::operator++()
{
    int& location = Provide(index);
    int res = ++location;
    if(location == 0) Remove(index);
    return res;
}
int SparseArrayInt::Interm::operator++(int)
{
    int& location = Provide(index);
    int res = location++;
    if(location == 0) Remove(index);
    return res;
}
SparseArrayInt::Interm::operator int()
{
    Item* tmp;
    for(tmp = master->first; tmp; tmp = tmp->next) {
        if(tmp->index == index) {
            return tmp->value;
        }
    }
    return 0;
}
```

Наконец, опишем вспомогательные функции `Provide` и `Remove`:

```
int& SparseArrayInt::Interm::Provide(int idx)
{
    Item* tmp;
    for(tmp = master->first; tmp; tmp = tmp->next) {
        if(tmp->index == index)
            return tmp->value;
    }
    tmp = new Item;
    tmp->index = index;
```

```
    tmp->next = master->first;
    master->first = tmp;
    return tmp->value;
}
void SparseArrayInt::Interm::Remove(int idx)
{
    Item** tmp;
    for(tmp = &(master->first); *tmp;
        tmp = &(*tmp)->next)
    {
        if((*tmp)->index == index) {
            Item *to_delete = *tmp;
            *tmp = (*tmp)->next;
            delete to_delete;
            return;
        }
    }
}
```

§ 6.2. Статические поля и методы

Иногда бывает полезно иметь переменные и методы, которые, с одной стороны, доступны только из класса и/или воспринимаются как его часть, но, с другой стороны, не привязаны ни к какому из объектов и могут использоваться даже тогда, когда ни одного объекта данного класса не существует.

В языке Си++ такие члены класса называются ***статическими*** и обозначаются ключевым словом `static`.[3]

§ 6.2.1. Статические поля и особенности их определения

Статическое поле описывается в классе точно так же, как и обычное поле класса, только с добавлением спереди слова `static`:

[3]Не путайте их с локальными переменными и функциями модулей, а также с локальными переменными функций, которые сохраняют своё значение при повторном входе в функцию; как и в языке Си, в Си++ эти сущности тоже обозначаются словом `static`, но практически ничего общего не имеют со статическими членами класса.

```
class Cls {
    //...
    static int the_static_field;
    //...
};
```

Имеется, однако, очень важное отличие: если описание обычного поля само по себе уже означает, что в объекте под это поле будет выделено место, то описание статического поля означает лишь, что *место под это поле будет выделено где-то в одном из модулей программы.*

Читатель, хорошо знакомый с языком Си, может заметить здесь прямую аналогию с объявлениями глобальных переменных с директивой `extern`. Чтобы понять, почему описание статического поля в классе считается именно декларацией, а не определением, подумайте, что будет, если заголовок класса будет вынесен в заголовочный файл, а сам этот файл — включён из нескольких модулей.

Итак, чтобы завершить создание статического поля, необходимо **вне класса** поместить его определение:[4]

```
int A::the_static_field = 0;
```

Обычно определения статических полей снабжают инициализацией, как это сделано и в нашем примере, хотя это и не обязательно.

Обратите внимание, что, если ваше описание класса вынесено в заголовочный файл, то определения статических полей следует **обязательно поместить в файл реализации одного из модулей**, ни в коем случае не в заголовочный файл!

Введённое таким образом статическое поле будет существовать в программе **в единственном экземпляре и в течение всего времени выполнения программы** вне всякой зависимости от того, будут ли в вашей программе заводиться объекты класса `A` и в каких количествах.

Если поместить объявление статического поля в приватной части класса, то соответствующее имя будет доступно только в методах класса и в «дружественных» функциях. Объявление можно поместить и в открытую часть класса. В этом случае оно будет доступно отовсюду, а обратиться к нему можно будет как через существующий объект, так и без всякого объекта с помощью явного раскрытия области видимости:

```
A a;
```

[4] Символ `::` обозначает *раскрытие области видимости*. Мы уже встречались с ним в § 4.3 на стр. 38.

```
a.the_static_field = 15; // правильно
x = A::the_static_field; // тоже правильно
```

Следует отметить, что статические поля представляют своего рода глобальные переменные как минимум в том смысле, что в них программа может *накапливать глобальное состояние*, в результате чего для стороннего наблюдателя поведение одних и тех же функций окажется изменяющимся по неочевидным законам. Кроме того, статические поля могут существенно затруднить масштабирование программы. Так, если вам зачем-то понадобился список объектов определённого класса и вы ввели для этой цели статический указатель на первый элемент такого списка, то это означает, что в программе такой список может быть только один, а если когда-нибудь понадобится два таких списка (из объектов одного и того же типа), программу придётся очень серьёзно переделать.

Поэтому использование статических полей рекомендуется в одной и только одной ситуации: когда такое поле **представляет собой константу, то есть его значение никогда не изменяется во время работы программы**. Например, статическим можно объявить какой-нибудь крупный массив, содержащий данные, необходимые для работы объектов данного класса, но не требующиеся нигде за его пределами. Примером такого массива могут служить, например, таблица переходов между состояниями конечного автомата в классе, реализующем этот конечный автомат; массив строк с возможными диагностическими сообщениями; таблица соответствия пользовательских команд (строк) вызываемым функциям, и т. п.

§ 6.2.2. Статические методы

Статический метод — это особый вид функции-метода, которая, являясь методом класса и имея доступ к его закрытым деталям реализации, при этом **вызывается независимо от объектов класса**. Первоначально статические методы предназначались для работы со статическими полями, но получившийся механизм нашел в итоге существенно более широкий спектр применений.

Описание статического метода аналогично описанию обычного метода, но перед таким описанием ставится ключевое слово `static`, как в следующем примере:

```
class Cls {
    //...
    static int TheStaticMethod(int a, int b);
    //...
};
```

Обращение к статическому методу, как и к статическому полю, возможно как через объект класса, так и без такового, с помощью символа

раскрытия области видимости:

```
Cls::TheStaticMethod(5, 15); // всё правильно
Cls c;
c.TheStaticMethod(5, 15); // так тоже можно
```

Поскольку статическая функция-метод может быть вызвана без объекта, у неё, как следствие, отсутствует неявный параметр `this` (см. § 1.2.2). Кроме всего прочего, это означает, что такая функция не может просто так обращаться, как другие методы, к полям объекта, ведь объекта у неё нет. Вызывать нестатические функции-методы статическая функция тоже просто так не может, поскольку нестатические методы должны вызываться для конкретного объекта.

Ситуация кардинально меняется, если статическая функция тем или иным способом всё-таки получает доступ к объекту своего класса. Такое вполне может произойти и никоим образом не противоречит определению статического метода: действительно, никто не мешает передать объект через один из явно обозначенных параметров; вполне возможно получить доступ к объекту своего класса через глобальные переменные или с помощью глобальных функций; наконец, **статическая функция вполне может создать объект сама**.

Итак, несмотря на отсутствие указаний на конкретный объект при вызове статической функции, она в некоторых случаях может получить доступ к объекту своего класса. И тогда **статическая функция, как и любой метод класса, может обращаться к закрытым (приватным) полям и методам объекта** (действительно, как уже говорилось ранее, единицей защиты в Си++ является не объект, а класс или структура целиком).

Наличие в Си++ статических методов позволяет в ряде случаев применять весьма изящные приёмы программирования. Например, мы вполне можем убрать в закрытую часть класса все имеющиеся конструкторы, запретив, таким образом, создание объектов данного класса извне его самого, и поручить создание объектов статическому методу, который можно вызвать, не имея ни одного объекта.

Лекция 7

§ 7.1. Обработка исключительных ситуаций

§ 7.1.1. Ошибочные ситуации и проблемы их обработки

Любая сколь бы то ни было нетривиальная программа содержит фрагменты, которые в некоторых обстоятельствах не могут отработать корректно. Например, программа, анализирующая содержимое заданного файла, не сможет работать, не сумев открыть файл на чтение; программа, работающая по компьютерной сети, не сможет работать, если не работает сеть или если недоступен нужный сервер; функция, производящая сложные вычисления, не может корректно их завершить, если в ходе вычислений потребовалось деление на ноль или, например, вычисление логарифма по единичному основанию, и т. п.

Подобные ситуации называют ошибочными, однако же это совершенно не обязательно означает, что ошибся программист, писавший программу. В случае с файлом «виноват» в возникновении ошибочной ситуации, скорее всего, пользователь, в случае с сетью — обслуживающий персонал сети или сервера; пример с делением на ноль указывает на ошибку программиста, вызвавшего функцию для некорректных исходных параметров, но это может быть не тот программист, который написал саму функцию.

Таким образом, мы при написании программ часто сталкиваемся со случаями, когда успешная работа нашей программы зависит от внешних условий, которые мы сами гарантированно обеспечить не можем. В таких случаях приходится предусматривать в программе ***обработку ошибок***.

Проверить все нужные условия обычно несложно. Существенно сложнее может оказаться следующий вопрос: а что же делать, если

условия оказались неудовлетворительны — не открылся файл, не установилось соединение, в делителе оказался ноль —, то есть возникла та самая ошибочная ситуация?

Многие студенты в такой ситуации поступают просто, дёшево и сердито: печатают какое-нибудь сообщение (очень часто просто `"ERROR"`) и завершают выполнение программы, например, вызовом `exit()`. В реальной жизни такой вариант, как правило, категорически недопустим. Чтобы понять причины этой недопустимости, представьте себе, что вы долго набирали текст в каком-нибудь текстовом редакторе, потом при сохранении *случайно* ввели неправильное имя директории или, например, попытались осуществить запись на защищенный носитель. Если бы автор текстового редактора обрабатывал ошибки «по-студенчески», программа редактора бы немедленно завершилась, уничтожив все результаты вашей работы. Маловероятно, что такое могло бы вам понравиться.

Более того, не всегда и не везде можно так вот просто «напечатать сообщение». Например, при программировании оконного приложения под MS Windows никакого потока стандартного вывода в распоряжении программиста нет, так что вместо печати необходимо создавать модальный диалог с соответствующим текстом и кнопками. Под ОС Unix всё не так плохо, поток стандартного вывода есть всегда, есть даже специальный поток для вывода сообщений об ошибках; проблема только в том, что далеко не во всех случаях результаты вывода в эти потоки кто-то читает, и в некоторых случаях следует вместо них пользоваться системой журнализации.

Кроме того, не все ошибочные ситуации фатальны. В вышеупомянутом примере с редактором текстов в ошибке, допущенной пользователем, нет вовсе ничего страшного: нужно просто констатировать факт ошибки и попросить пользователя ввести другое имя файла. Точно так же может не быть фатальной ошибка, связанная с установлением сетевого соединения, ведь во многих случаях можно обратиться к серверу-дублёру.[1] Даже ситуация с делением на ноль в некоторых случаях может оказаться нефатальной, как, например, при исследовании некоторых некорректно поставленных задач (обычно в этих случаях головная программа «знает», как можно скорректировать исходные данные, чтобы избежать деления на ноль, и нужную комбинацию ищет методом проб и ошибок).

Наконец, заметим, что один и тот же код может использоваться

[1] Такое возможно при обращениях к системе доменных имён (DNS), а в некоторых случаях — и при отправке электронной почты, и в других ситуациях тоже.

в совершенно разных программах, причём для одних ошибка может оказаться фатальной, для других — не стоящей даже упоминания. Например, многие программы выполняют поиск нужного файла в разных директориях, просто пытаясь его открыть и продолжая поиск дальше после каждой неудачи; в этом случае ошибка вообще перестаёт быть ошибкой.

Из всего вышесказанного можно сделать один очень серьёзный и важный вывод: **принимать решения о том, что делать при возникновении ошибочной ситуации — это прерогатива головной программы**. Завершать программу, а также и выполнять тем или иным способом выдачу сообщений может только функция `main()` и те из вызываемых ею функций, которые для этого специально предназначены. Что же касается кода, не попавшего в эту категорию, то его дело в случае возникновения ошибки — оповестить об этом головную программу, и не более того.

Звучит это достаточно просто, однако все, кто имеет опыт программирования даже не очень сложных программ, знают, с каким трудом это воплощается на практике. В простых языках программирования, таких как Си или Pascal, приходится из функций в случае возникновения ошибок возвращать специальные значения, а в точке вызова функции, соответственно, проверять, не вернула ли вызванная функция значение, соответствующее ошибке. В большинстве случаев при этом вызвавшая функция, в свою очередь, вынуждена возвращать признак ошибки и т.д. Представьте себе теперь, что функция `f1()` вызвала функцию `f2()`, та, в свою очередь, функцию `f3()` и т.д., а в функции, скажем, `f10()` возможно возникновение ошибочной ситуации. Тогда нужно предусмотреть значение, возвращаемое в случае этой ошибки, в функции `f9()` сделать проверку и в случае ошибки, в свою очередь, возвратить нечто ошибочное, и так во всех функциях.

Теоретически именно так всё и должно делаться, все возможные ошибочные ситуации должны проверяться и т. п., но на практике аккуратное соблюдение требований по обработке ошибок может оказаться настолько трудоёмким, что программисты прибегают к известному «алгоритму решения всех проблем», а именно — (1) поднять вверх правую руку и (2) резко махнуть ею, одновременно произнося что-то вроде «а, ладно». Есть даже шуточная фраза на эту тему: «никогда не проверяйте на ошибки, которые вы не знаете, как обработать».

Разумеется, такое обычно даром не проходит. Ошибка, которую не стали обрабатывать в надежде, что она никогда не произойдёт, проявится в самый неподходящий момент, причём чем дальше находится

заказчик и чем больше денег он заплатил, тем выше вероятность, что наша программа сломается именно у него.

Именно поэтому во многих языках предусмотрены специальные возможности для обработки ошибочных ситуаций. В языке Си++ соответствующий механизм называется ***обработкой исключений*** (англ. ***exception handling***).

§ 7.1.2. Общая идея механизма исключений

Отметим ещё один недостаток использования специальных возвращаемых значений для индикации ошибки. В некоторых случаях среди всех значений возвращаемого функцией типа просто нет ни одного неиспользуемого; отличный пример такого рода функции — `atoi()`, переводящая строковое представление целого числа в значение типа `int`. Возвращает она, естественно, число типа `int`, а поскольку любое такое число имеет строковое представление, то и вернуть (при отсутствии каких-либо ошибок) она может, вообще говоря, любое значение типа `int`. В то же время возможно, что в строке, поданной `atoi` на вход, содержится текст, не являющийся представлением какого-либо числа (например, состоящий из букв, а не из цифр). В этой ситуации разработчики функции решили, за неимением лучшего, возвращать ноль, но это не решает проблему, ведь получается, что функция возвращает одно и то же и для строки, содержащей белиберду, и для строки `"0"`, которая является вполне корректным представлением числа.

Это приводит нас к идее *исключения*, которое представляет собой *особый способ завершения функции*. Таким образом, если в языках, не поддерживающих механизм исключений, функция могла только вернуть одно из возможных значений, то в языках, поддерживающих исключения, функция может либо вернуть значение, либо ***возбудить исключение***.

Отметим в этой связи, что исключения вовсе не являются составной частью парадигмы объектно-ориентированного программирования, как это почему-то часто утверждается. Напротив, они скорее относятся к *функциональному программированию*; небезызвестная функция `callcc`, поддерживаемая во многих функциональных языках, представляет собой ничто иное как обобщение механизма исключений; в языке Common Lisp, не имеющем `callcc`, при этом имеются, тем не менее, конструкции, совершенно аналогичные конструкциям обработки исключений Си++.

Возбуждение исключения означает, на самом деле, переход программы в особое состояние, в котором все активные на настоящий момент функции досрочно завершаются одна за другой, пока не найдётся такая,

которая может справиться с данным типом исключительных ситуаций. Чтобы понять, о чём идёт речь, вспомним ситуацию, рассматривавшуюся в предыдущем параграфе: функция `f1()` вызвала функцию `f2()`, та, в свою очередь, функцию `f3()` и т.д. вплоть до `f10()`, где и возникает ошибка. При этом нам совершенно не обязательно делать какие-либо проверки в функциях `f9`, `f8` и т.д., вполне достаточно пометить, например, функцию `f1` как умеющую справляться с данным типом ошибок. Тогда при возникновении ошибки во время исполнения `f10` будет завершена досрочно и она сама, и вызвавшая её функция `f9`, и `f8`, и так вплоть до `f1`, которая и обработает ошибку. Иначе говоря, если некая функция `g()` вызывает другую функцию `h()`, а эта последняя возбуждает исключение, для которого в `g()` нет обработчика, то это эквивалентно тому, как если бы функция `g()` сама возбудила то же самое исключение, не вызывая `h()`.

Иногда всю обработку ошибок делают в функции `main()`. Она для этого удобна, поскольку остаётся активной всё время исполнения программы[2] и, таким образом, может обработать исключение, возбуждённое в практически любой части программы.

§ 7.1.3. Возбуждение исключений

При возникновении ситуации, трактуемой как ошибочная и требующая обработки в качестве исключительной, следует *возбудить исключение* с помощью оператора `throw` (англ. *бросить*). У этого оператора имеется один параметр, в качестве которого может выступать выражение, вообще говоря, произвольного типа, в том числе как любого базового (`int`, `float`, ...), так и производного (указатель и т. п.), а равно и определённого пользователем (в том числе, что немаловажно, типа класс или структура). Тип выражения в операторе `throw` будем называть ***типом исключения***, а значение выражения — ***значением исключения***.

Для примера рассмотрим функцию, которая принимает на вход имя (текстового) файла, а возвращает количество строк (т.е. символов перевода строки) в этом файле. Ситуация, когда файл не удалось открыть, для такой функции оказывается заведомо исключительной; в этом случае в языке Си нам пришлось бы возвращать специальное значение (например, `-1`), а при вызове проверять, не его ли вернула функция. В

[2]Строго говоря, это не совсем так, поскольку есть ещё, например, конструкторы и деструкторы глобальных объектов, которые отрабатывают, соответственно, до и после функции `main()`, но в некотором приближении можно этим пренебречь.

Си++ это можно сделать проще:

```
unsigned int line_count_in_file(const char *file_name)
{
    FILE *f = fopen(file_name, "r");
    if(!f)
        throw "couldn't open the file";
    int n = 0;
    int c = 0;
    while((c = fgetc(f)) != EOF)
        if(c == '\n') n++;
    fclose(f);
    return n;
}
```

В этом примере мы в качестве исключения «бросили» строку (точнее, адрес её начала, т.е. указатель типа `const char*`). Мы могли бы «бросить» выражение другого типа; например, следующий оператор «бросает» исключение типа `int`:

```
throw 27;
```

Чаще, однако, в роли исключений выступают объекты специально введённых для этой цели классов; речь об этом пойдёт впереди.

§ 7.1.4. Обработка исключений

Обработку исключительных ситуаций следует предусматривать, естественно, только в тех местах, в которых мы можем с соответствующей ситуацией справиться. Так, если дальнейшая работа после возникновения исключительной ситуации требует обращения к пользователю, то обработку такой ситуации следует вставить в одну из функций, наделённых правом вести диалог с пользователем (возможно, в функции `main`).

Общий принцип организации обработки исключений таков. Из программы выделяется некоторый блок кода, исполнение которого, предположительно, может привести к возникновению исключительных ситуаций. Этот блок явно выделяется и снабжается одной или несколькими инструкциями о том, как необходимо действовать при возникновении исключения того или иного типа. При этом такие инструкции действуют в отношении исключений соответствующих типов, **возникших в программе с момента входа в помеченный блок и до момента**

выхода из него, в том числе в функциях, вызванных из блока прямо или косвенно. Глубина вызовов в данном случае роли не играет.

Для пометки блока с целью обработки исключительных ситуаций в Си++ используется ключевое слово `try` (англ. *попытаться*), за которым следует собственно блок, то есть последовательность операторов, заключенная в фигурные скобки. Сразу после `try`-блока необходимо поместить один или больше ***обработчиков исключений***. Обработчик исключений синтаксически несколько напоминает функцию с одним параметром, хотя это только внешнее сходство; на самом деле, обработчик начинается с ключевого слова `catch` (англ. *поймать*), сразу после него ставятся круглые скобки, внутри которых — описание формального параметра (точно так, как это делается при описании функции с одним параметром). Тип параметра задаёт тип обрабатываемого исключения, а по имени параметра можно получить доступ к значению исключения.

Для примера припомним функцию из предыдущего параграфа, подсчитывающую количество строк в файле, и на основе этой функции напишем программу целиком. Будем считать, что функция `line_count_in_file` находится в отдельном модуле, так что в программе нам потребуется только её прототип.

```
#include <stdio.h>

unsigned int line_count_in_file(const char *file_name);

int main(int argc, char **argv) {
    if(argc<2) {
        fprintf(stderr, "No file name\n");
        return 1;
    }
    try {
        int res = line_count_in_file(argv[1]);
        printf("The file %s contains %d lines\n",
               argv[1], res);
    }
    catch(const char *exception) {
        fprintf(stderr, "Exception (string): %s\n",
                exception);
        return 1;
    }
    return 0;
}
```

Эта программа, если всё будет в порядке, напечатает имя файла и количество строк в нём; если же открыть файл не удастся, функция `line_count_in_file` переведёт программу в состояние исключительной ситуации, а соответствующий обработчик будет найден в функции `main`. Ни остаток функции `line_count_in_file`, ни остаток `try`-блока в этом случае выполняться не будут, управление окажется сразу передано в обработчик (`catch`-блок).

В этом примере исключение может возникнуть в функции, непосредственно вызванной из `try`-блока, но на самом деле это не важно: с таким же успехом оно могло возникнуть в функции, вызванной из `line_count_in_file`, и т.д. на любую глубину.

Обратите внимание, что мы в данном случае ≪ловим≫ исключение `const` char *., а не просто `char *`, поскольку именно такой тип (указатель на константу типа `char`) имеет в Си и Си++ строковый литерал — строка, заключённая в двойные кавычки. Если убрать модификатор `const`, исключение поймано не будет.

В рассмотренном примере мы предусмотрели всего один обработчик исключения, хотя язык Си++ допускает произвольное их количество (не менее одного). Если бы из нашего `try`-блока вызывалось больше функций и потенциально они могли бы привести к исключительным ситуациям других типов, мы могли бы написать что-то вроде следующего:

```
try {
    // ...
    // --//--
    // ...
}
catch(const char *x) {
    // ...
}
catch(int x) {
    // ...
}
```

Здесь важно понимать несколько простых правил:

- **Обработчики (`catch`-блоки) рассматриваются по порядку, один за другим, причём сработать может только один из них, или ни одного**. Это значит, в частности, что писать два

обработчика одного типа или типов, сводимых один к другому,[3] нет никакого смысла.

- **Обработчик может поймать только исключение, возникшее во время работы соответствующего `try`-блока**. В частности, обработчик не может поймать исключение, которое выбросил один из предыдущих обработчиков.

- **Если исключение не поймано ни одним из обработчиков, оно «выбрасывается» дальше, как если бы фрагмент кода, в котором исключение возникло, вовсе не был обрамлён никаким `try`-блоком**. Таким образом, на пути одного исключения может оказаться несколько `try`-блоков, как бы вложенных друг в друга (во всяком случае, в смысле временны́х периодов исполнения), и в некоторых из них может не оказаться подходящего обработчика. В таком случае досрочное завершение активных функций и уничтожение стековых фреймов продолжается дальше, пока не будет найден `try`-блок с подходящим обработчиком.

§ 7.1.5. Обработчики с многоточием и `throw` без параметров

В некоторых случаях оказывается осмысленным обработчик, способный поймать *произвольное исключение независимо от его типа*. Такой обработчик записывается так же, как и обычный, только вместо типа исключения и имени параметра в скобках указывается многоточие. Конечно, значение исключения в этом случае использовано быть не может, но в некоторых случаях это и не важно. Например, если мы подключаем к нашей программе модули, написанные другими программистами, и не уверены в том, что нам известны все типы исключений, генерируемые такими модулями, может быть неплохой идея расположить в функции `main` примерно такой `try`-блок:

```
int main() {
    try {
        // здесь пишем весь код главной функции
        return 0;
    }
```

[3] Правила преобразования типов при обработке исключений несколько отличаются от правил, применяемых при вызове функций; мы рассмотрим эти правила в одном из следующих параграфов

```
    catch(const char *x) {
        fprintf(stderr, "Exception (string): %s\n", x);
    }
    catch(int x) {
        fprintf(stderr, "Exception (int): %d\n", x);
    }
    catch(...) {
        fprintf(stderr, "Something strange caught\n");
    }
    return 1;
}
```

В обработчиках исключений можно использовать специальную форму оператора `throw`, а именно — написать этот оператор без параметров. Это означает «бросить то исключение, которое только что было поймано». Такой вариант особенно актуален в обработчиках с многоточием, поскольку в таких обработчиках у нас нет доступа к самому исключению.

Использование специальной формы `throw` позволяет, кроме прочего, поймать исключение произвольного типа, выполнить какие-то действия, после чего бросить исключение дальше. Это может оказаться полезным, если в нашем коде локально захватываются те или иные ресурсы (выделяется память, открываются файлы и т. п.) и их следует освободить, прежде чем функция окажется завершена. Рассмотрим для примера функцию, в которой выделяется динамический массив целых чисел:

```
void f(int n) {
    int *p = new int[n];
    // весь остальной код функции
    delete[] p;
}
```

Если во время выполнения кода, находящегося между `new` и `delete`, возникнет исключение, то `delete` не будет выполнен и массив, на который указывал `p`, будет продолжать занимать память (то есть станет мусором). Проблема снимается, если весь этот код заключить в `try`-блок, после которого есть обработчик для исключений произвольного типа, который, прежде чем бросить исключение дальше, освобождает память:

```
void f(int n) {
```

```
    int *p = new int[n];
    try {
        // весь остальной код функции
    }
    catch(...) {
        delete[] p;
        throw;
    }
    delete[] p;
}
```

§ 7.1.6. Объект класса в роли исключения

Обычно при возникновении исключительной ситуации возникает желание передать обработчику максимум информации о возникших трудностях. Встроенные типы данных плохо пригодны для этого. Действительно, в рассмотренном выше примере было бы неплохо передать обработчику не только текстовое сообщение «couldn't open the file», но и имя файла, и значение переменной `errno` сразу после выполнения `fopen`, которое может помочь в анализе проблемы. Мы, однако, ограничились в примере одной строковой константой, чтобы не формировать строку, содержащую всю перечисленную информацию — ведь это потребовало бы значительного увеличения кода и создало бы определённые трудности с освобождением памяти от такой строки, поскольку саму строку пришлось бы создавать динамически.

Кроме того, есть и ещё одна проблема. Программа может использовать несколько библиотек, причём изготовленных разными производителями; также программа может быть разделена на несколько подсистем, создаваемых более или менее независимо. В такой ситуации весьма желательно иметь некий универсальный способ разделения исключений по признаку их возникновения в той или иной подсистеме программы или библиотеке. Встроенные типы для этого не подходят совсем, поскольку идея использования тех же текстовых строк в качестве исключений может прийти в голову одновременно авторам сразу всех используемых нами библиотек и половины наших подсистем.

Именно поэтому чаще всего в качестве типа исключения выступает специально для этой цели описанный класс, а значением исключения — соответственно, объект такого класса.

С одной стороны, в объект класса можно поместить всю информацию, какая нам только может показаться полезной для обработчика;

действительно, никто ведь не мешает описать в классе столько полей, сколько нам захочется.

С другой стороны, каждая подсистема и каждая библиотека в этой ситуации, скорее всего, введёт свои собственные классы для представления исключений. В головной программе мы сможем обрабатывать такие классы отдельными обработчиками, что позволит разделить обработку исключений, сгенерированных в разных подсистемах программы.

Отметим один очень важный момент. Как правило, выбрасываемый в качестве исключения объект создаётся локально. Локальные объекты, естественно, исчезают вместе со стековым фреймом создавшей их функции; таким образом, тот объект, который «поймается» в обработчике исключения, заведомо не может быть тем же экземпляром объекта, который фигурировал в операторе `throw`, и является, как нетрудно догадаться, его копией. Поэтому **практически всегда в классе, используемом для исключений, обязан присутствовать конструктор копирования**.

Для примера опишем класс, объект которого было бы удобно использовать в качестве исключения в нашей функции `line_count_in_file()`. Объект класса будет хранить имя файла, текстовый комментарий (этот комментарий поможет понять, в каком месте программы произошла ошибка) и будет запоминать значение глобальной переменной `errno` на момент создания объекта. Для копирования строк мы опишем в приватной части класса вспомогательную функцию `strdup`[4]; поскольку доступ к объекту ей не нужен, опишем её с квалификатором `static`.

```
class FileException {
    char *filename;
    char *comment;
    int err_code;
public:
    FileException(const char *fn, const char *cmt);
    FileException(const FileException& other);
    ~FileException();
    const char *GetName() const { return filename; }
    const char *GetComment() const { return comment; }
    int GetErrno() const { return err_code; }
private:
    static char *strdup(const char *str);
```

[4] Эта функция будет полностью аналогична одноимённой функции из стандартной библиотеки Си, но будет использовать для выделения памяти `new[]`, а не `malloc()`

```
};
```

Описания конструкторов, деструктора и функции `strdup` вынесем за пределы заголовка класса:

```
FileException::FileException(const char *fn,
                             const char *cmt)
{
    filename = strdup(fn);
    comment = strdup(cmt);
    err_code = errno;
}
FileException::FileException(const FileException& other) {
    filename = strdup(other.filename);
    comment = strdup(other.comment);
    err_code = other.err_code;
}
FileException::~FileException() {
    delete[] filename;
    delete[] comment;
}
char* FileException::strdup(const char *str) {
    char *res = new[strlen(str)+1];
    strcpy(res, str);
    return res;
}
```

Теперь мы можем изменить начало функции `line_count_in_file` (см. стр. 71) на следующее:

```
unsigned int line_count_in_file(const char *file_name)
{
    FILE *f = fopen(file_name, "r");
    if(!f)
        throw FileException(file_name,
           "couldn't open for line counting");
  // ...
```

Что касается обработки такого исключения, то обработчик (`catch`-блок) теперь может выглядеть, например, так:

```
    catch(const FileException &ex) {
        fprintf(stderr, "File exception: %s (%s): %s\n",
```

```
                ex.GetName(), ex.GetComment(),
                strerror(ex.GetErrno()));
        return 1;
    }
```

§ 7.1.7. Автоматическая очистка

В примере, рассмотренном на стр. 75, мы были вынуждены перехватывать и потом заново «выбрасывать» исключения произвольного вида, чтобы обеспечить корректное удаление локально выделенной динамической памяти.

К счастью, так приходится действовать не всегда. Работа с исключениями изрядно облегчается тем, что **компилятор гарантирует, что прежде, чем функция завершится (по исключению ли или обычным путём), для всех локальных объектов, имеющих деструкторы, эти деструкторы будут корректно вызваны**.

Пусть, например, имеется класс `A`, снабженный нетривиальным деструктором, и описана функция

```
void f() {
    A a;
    //..
    g();
    //..
}
```

В момент завершения работы функции `f()` для объекта `a` будет вызван деструктор, причём произойдёт это даже в том случае, если функция `g()` «выбросит» исключение и именно оно станет причиной завершения `f()`.

Это свойство языка Си++ называется ***автоматической очисткой***.

§ 7.1.8. Преобразования типов в обработчиках исключений

Тип исключения, указанный в обработчике (`catch`-блоке), не всегда полностью совпадает с типом выражения, использованного в операторе `throw`, однако правила преобразования типов в данном случае отличаются от правил, действующих при вызове функций.

Одно из основных отличий тут в том, что целочисленные типы при обработке исключений не преобразуются друг к другу, то есть, например, обработчик с указанным типом исключения `int` не поймает исключение типа `char` или `long`.

Допустимые преобразования проще будет явно перечислить. Во-первых, любой тип может быть преобразован к ссылке на этот тип, то есть если оператор `throw` использует выражение типа `A`, то такое исключение может быть обработано `catch`-блоком вида `catch(A &ref)`.

Во-вторых, неконстантные указатели и ссылки могут быть преобразованы к константным, то есть, например, если мы бросим исключение типа `char*`, то оно может быть поймано не только как `char*`, но и как `const char*`. Отметим, что в обратную сторону преобразование запрещено, так что обработчик типа `char *` (без модификатора `const`) не может поймать выражение типа `const char*`, и, в частности, не может обработать исключение, «выброшенное» с использованием строкового литерала, например такое:

```
throw "I can't work";
```

Последний (и, возможно, самый важный) вид преобразований имеет отношение к наследованию и полиморфизму. К обсуждению этой темы мы вернёмся после рассказа о наследовании.

Лекция 8

§ 8.1. Наследование

§ 8.1.1. Иерархические предметные области

Очень часто, и особенно в более-менее крупных компьютерных программах, возникает ситуация, при которой объекты предметной области естественным образом объединяются в некие категории, причём объекты каждой категории могут подразделяться на подкатегории и т.д., образуя, таким образом, ***иерархию типов объектов***. Например, в программе, осуществляющей имитационное моделирование дорожного движения, могут быть объекты категории «участник дорожного движения», причём эта категория подразделяется на подкатегории «пешеходы» и «транспортные средства», транспортные средства, в свою очередь, делятся на «велосипеды», «гужевые повозки» и «механические транспортные средства», эти последние делятся на «трамваи» и «автомобили», автомобили — на грузовые и легковые, и т.д.

Легко заметить, что в такой иерархии каждая категория объектов (в том числе и такая, которая сама подразделяется на категории) обладает некоторыми свойствами, специфичными для данной категории, причём этими же свойствами обладают и все объекты из подкатегорий данной категории. Так, все автомобили имеют, по-видимому, характеристики «объем двигателя», «тип топлива», «объем топливного бака» и «количество топлива в баке», у них может заканчиваться топливо, над ними должна быть определена операция «заправка топлива». При этом все эти характеристики не имеют никакого смысла для объектов, не входящих в категорию «автомобили» — ни для трамваев, ни для велосипедов, ни тем более для пешеходов. С другой стороны, у грузовых автомобилей есть характеристика «объем кузова» и операции «погрузка» и «разгрузка», которых нет у автомобилей, не являющих-

ся грузовыми. Наконец, такие характеристики, как текущая скорость и направление движения, очевидно, присущи каждому объекту рассматриваемой иерархии, вплоть до пешеходов.

Таким образом, у нас возникают структуры данных, имеющие, с одной стороны, различный набор полей и обрабатывающих функций, но, с другой стороны, имеющие и некоторые общие поля, и определённые общие операции.

Существуют весьма различные подходы к решению возникшего противоречия. При работе на языке Си чаще всего выходят из положения описанием структур данных, у которых первые несколько полей совпадают; это сопровождается частым использованием преобразования указателей. Ясно, что такой подход чреват множеством труднообнаружимых ошибок.

Объектно-ориентированные языки программирования предлагают более корректный подход, называемый обычно ***наследованием***.

§ 8.1.2. Наследование структур данных и полиморфизм адресов

С точки зрения структур данных, расположенных в памяти, наследование представляет собой добавление новых полей данных к ранее описанной структуре. С теоретической точки зрения такое добавление представляет собой *уточнение* сведений об описываемом объекте, и, таким образом, **наследование рассматривается как переход от общего к частному**.

Пусть, например, у нас имеется структура данных, описывающая персону (человека) и содержащая поля для имени, пола и года рождения:

```
struct person {
    char name[64];
    char sex; // 'm' or 'f'
    int year_of_birth;
};
```

Пусть теперь нам необходима структура данных, описывающая *студента* (частный случай персоны). Относительно студента нас может интересовать код специальности, номер курса и показатель успеваемости (средний балл). Вместе с тем, несомненно, студент — это тоже человек, так что ему присущи и свойства, общие для всех персон: имя, пол

и год рождения. Таким образом, мы можем перейти от общего (персона) к частному (студент), добавив уточняющие сведения, и сделать это можно с помощью механизма наследования. На языке Си++ это записывается так:

```
struct student : person {
    int code;
    int year; // курс
    float average;
};
```

Если теперь описать переменную типа `student`, она будет иметь все свойства структуры `person` плюс дополнительные поля:

```
student s1;
strcpy(s1.name, "John Doe");
s1.sex = 'm';
s1.year_of_birth = 1989;
s1.code = 51311;
s1.year = 2;
s1.average = 4.75;
```

Структура `person` в данном случае называется ***базовой*** или ***родительской***, а структура `student` — ***унаследованной*** или ***порождённой***, иногда ***дочерней***. Также употребляются термины «предок» и «потомок» (в данном случае, соответственно, для `person` и `student`).

Расположение полей переменной `s1` в памяти показано на рис. 8.1. Здесь следует заметить, что поля, относящиеся к части структуры `student`, унаследованной от `person`, располагаются на тех же местах (по тем же смещениям относительно начала), где они располагались бы в структуре `person`. Получается, что мы можем при необходимости работать с переменной `s1` точно так, как если бы она была переменной типа `person`, а не типа `student` (важно понимать, что обратное неверно: в структуре `person` отсутствуют поля, которые ожидались бы от структуры `student`).

В связи с этим язык Си++ **разрешает неявное преобразование** адресов структур типа `student` в адреса структур типа `person`, или, в общем случае, **адресов структур-потомков к адресам структур-предков**. Соответствующие преобразования разрешены как для указателей, так и для ссылок. Таким образом, например, возможны такие фрагменты кода:

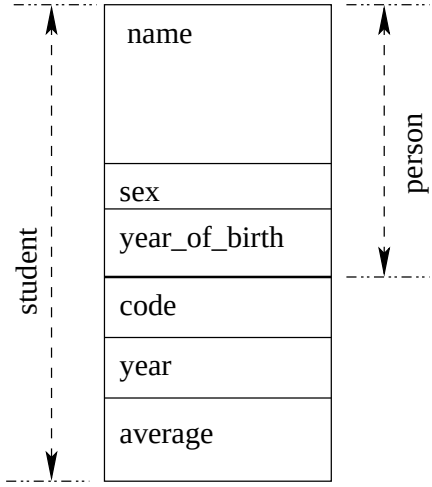

Рис. 8.1. Унаследованная структура данных

```
student s1;
person *p;
p = &s1;
person &ref = s1;
```

Если в программе описана функция, принимающая на вход указатель или ссылку на объект типа `person`, ей без каких-либо сложностей можно передать, соответственно, указатель или ссылку на объект типа `student`:

```
void f(person &pers) { ... }
// ...
student s1;
// ...
f(s1); // корректно!
```

Это свойство предков, потомков и их адресов называется ***полиморфизм***.

Если говорить точнее, полиморфизм не сводится только к преобразованию указателей и ссылок. Вообще говоря, полиморфизм в зависимости от контекста понимается либо как способность некоторых объектов выступать в качестве объектов разных типов (в данном случае `s1` выступает и как `student`, и как

person), либо как способность некоторых функций или других операций работать с объектами разных типов (в данном случае f() работает как с person, так и с любыми её потомками, в том числе student).

§ 8.1.3. Наследование и методы; конструкторы и деструкторы

Чаще всего наследование применяется для структур и классов, имеющих функции-члены (методы). В дальнейшем изложении мы будем опускать слово «структура» и говорить только о классах, хотя всё, что будет сказано, может быть применено также и к структурам.

Методы, имеющиеся в базовом классе, доступны и для порождённого класса. Здесь необходимо отметить, что, если не предпринять специальных мер, то методы базового класса не будут ничего знать о том, что их вызывают для объекта порождённого класса, то есть будут работать так же, как и для объектов базового класса; здесь мы имеем описанный в предыдущем параграфе эффект полиморфизма в применении к параметру `this` (см. § 1.2.2). Например, если мы сделаем класс «автомобиль» и предусмотрим для него операцию «заправка бензином», а потом породим от него класс «грузовик» с дополнительными свойствами, то операция «заправка бензином» будет доступна и для грузовика.

Таким образом, можно сказать, что *объект порождённого класса является в полном смысле слова также и объектом базового класса*, в том числе для него доступны и все операции, которые доступны для базового класса.

С другой стороны, как видно из рис. 8.1, объект порождённого класса **содержит** в себе объект базового класса в качестве своей части. Никакого противоречия тут нет, это просто две разные модели восприятия или, если угодно, точки зрения: одна — точка зрения реализации (реализаторская семантика), вторая — точка зрения логики проектирования (пользовательская семантика).

Особого упоминания в связи с введением наследования заслуживают конструкторы и деструкторы. С какой бы точки зрения мы ни рассматривали порождённый объект, очевидно, что в момент его создания также создаётся и объект базового класса, причём неважно, считаем мы его «частью» порождённого объекта или же его «другой ипостасью». Таким же точно образом при уничтожении порождённого объекта исчезает и базовый объект.

Следовательно, при создании объекта порождённого класса должен отработать и конструктор базового класса, а при уничтожении такого

объекта — деструктор базового класса. При этом, очевидно, в телах конструктора и деструктора порождённого класса должны быть доступны все части объекта, для которого они вызываются. Базовый объект, таким образом, должен быть инициализирован раньше, а уничтожен — позже объекта порождённого. Получается, что время существования объекта класса-наследника в его «базовой» ипостаси как бы чуть-чуть больше, чем время существования его же в качестве объекта своего собственного типа.

С деструктором дела обстоят достаточно просто: компилятор сначала вызывает тело деструктора порождённого класса, а затем — тело деструктора базового класса.

С конструктором всё тоже могло бы быть просто (сначала вызвать тело конструктора базового класса, потом тело конструктора класса порождённого), если бы не то обстоятельство, что конструкторы (в общем случае) могут требовать входных параметров.

С аналогичной проблемой мы уже сталкивались при рассмотрении классов, имеющих поля типа класс (см. § 4.4). Решается проблема в данном случае совершенно аналогично: в описании конструктора порождённого класса между заголовком и телом вставляется список инициализаторов, начинающийся с инициализатора базового класса (обозначаемого в данном случае именем класса) с указанием всех нужных параметров конструктора. Так, если `A` — базовый класс, конструктору которого требуются два параметра типа `int`, `B` — класс, унаследованный от `A`, снабженный конструктором по умолчанию и имеющий поле `int i`, то описание его конструктора может выглядеть так: .

```
B::B() : A(2, 3), i(4) { /* ... */ }
```

где 2 и 3 — параметры для конструктора базового класса, 4 — начальное значение поля `i`.

§ 8.1.4. Наследование и защита

Взаимодействие механизма наследования с механизмом защиты деталей реализации заслуживает отдельного разговора. Прежде всего отметим, что сам факт наследования одного класса от другого может в некоторых случаях рассматриваться как деталь реализации этого класса и, соответственно, нуждаться в сокрытии от остального кода программы.

В связи с этим в Си++ различают наследование открытое (public) и закрытое (private). Тип наследования указывается в заголовке класса непосредственно перед названием порождаемого класса, например:

```
class B : public A { /*...*/ };
```

или

```
class C : private A { /*...*/ };
```

Если в первом случае все свойства класса `A` (его открытые методы и поля, если такие в нём есть) будут доступны для объектов класса `B` отовсюду, то во втором случае — только из методов класса `C`, а вся остальная программа вообще не будет знать, что класс `C` унаследован от `A`.

Тип наследования можно не указывать, как мы это делали в примере на стр. 83. В этом случае будут действовать умолчания: для структуры наследование по умолчанию открытое (public), для класса — закрытое (private).

В реальной жизни потребность в закрытом наследовании возникает редко, поэтому при описании наследуемых классов обычно указывают слово `public`, а при описании наследуемых структур не указывают ничего, полагаясь на умолчания.

Заметим теперь, что для базового класса порождённый класс ничуть не «лучше» всей остальной программы и, как и любой недружественный фрагмент кода, не должен иметь доступа к деталям реализации класса. Поэтому, очевидно, закрытые поля и методы базового класса не будут доступны порождённым классам.

Вместе с тем в реальных задачах часто возникает потребность в таких деталях интерфейса класса, которые предназначены только, и исключительно, для его потомков. Для таких случаев вводится, наряду с режимами `public` и `private`, ещё и третий режим защиты, `protected` (защищённый). Поля и методы, помеченные словом `protected`, будут доступны только в самом классе (то есть в его методах), в дружественных функциях и в методах непосредственных потомков данного класса, а во всей остальной программе — недоступны.

Важно понимать, что поля и методы, имеющие режим защиты `protected`, не могут считаться в полном смысле слова деталями реализации, которые не касаются никого, кроме данного класса. Разница здесь достаточно принципиальна. Детали, помещённые в закрытую (private) часть класса, можно заведомо безболезненно изменять, исправляя при этом только методы самого класса и дружественные функции, которые опять-таки перечислены в явном виде в заголовке класса, то есть мы всегда знаем, где проходит граница кода, который нам, возможно, придётся исправлять. В противоположность этому, особенности

реализации класса, помеченные как `protected`, могут использоваться в самых неожиданных частях программы: достаточно кому-то где-то описать ещё одного потомка нашего класса.

Поэтому, в отличие от приватных полей и методов, поля и методы с режимом `protected` должны обязательно документироваться, а к их изменению следует подходить столь же осторожно, как и к изменению открытой (публичной) части класса.

Лекция 9

§ 9.1. Динамический полиморфизм

§ 9.1.1. Виртуальные функции

Механизм виртуальных функций позволяет в классе-потомке частично модифицировать поведение некоторых методов, определённых в предке. Чтобы понять, как это происходит и зачем это нужно, мы для начала рассмотрим пример, а затем поясним, как соответствующий механизм устроен.

Итак, допустим, что перед нами стоит задача, связанная с компьютерной графикой, и необходимо описать сцену (то есть набор графических элементов) в виде некоего списка или массива объектов, задающих графические объекты различного типа, например, отдельные пиксели, линии, окружности и т. п.

Для начала опишем класс, объекты которого будут представлять в нашей сцене простейшие графические примитивы — отдельные пиксели. Ясно, что у пикселя имеются две координаты (их обычно задают числами с плавающей точкой) и цвет (целое число). Будем считать, что основные действия с графическим объектом, в том числе и с пикселями — это показать объект на экране (мы обозначим это действие функцией `Show()`), убрать его с экрана (`Hide()`) и переместить его в новую позицию (`Move()`).

```
class Pixel {
    double x, y;
    int color;
public:
    Pixel(double ax, double ay, int acolor)
        : x(ax), y(ay), color(acolor) {}
    void Show();
```

```
    void Hide();
    void Move(double nx, double ny);
};
```

Конкретика описания тел функций `Show()` и `Hide()` зависит от используемой графической библиотеки, правил пересчёта координат и т. п.; мы на этом останавливаться не будем, просто предположим, что эти функции где-то описаны. С другой стороны, мы очень легко можем описать функцию `Move()`. Действительно, перемещение состоит в том, что объект сначала убирают с экрана, потом меняют его координаты и, наконец, снова показывают:

```
void Pixel::Move(double nx, double ny)
{
    Hide();
    x = nx;
    y = ny;
    Show();
}
```

Предположим теперь, что нам нужен также объект «окружность». Понятно, что такой объект обладает теми же свойствами (координаты и цвет) и, в дополнение к ним, характеризуется ещё радиусом. Поэтому мы воспользуемся уже имеющимся классом `Pixel` в качестве базового,[1] а класс `Circle` от него *унаследуем*. Конечно, методам класса `Circle` потребуется знать координаты и цвет, хотя бы для того, чтобы уметь рисовать заданную окружность на экране. В нашем примере мы решим вопрос доступа самым простым (хотя и не самым лучшим) способом: сделаем поля базового класса защищенными (`protected`), а не закрытыми; для этого вставим в самое начало описания класса `Pixel` директиву `protected`:

```
class Pixel {
protected:
    double x;
    //...
```

Теперь мы можем перейти к описанию класса `Circle` и для начала опишем поле, которого нам не хватает, чтобы превратить точку в окружность:

[1] Здесь мы несколько нарушаем принципы объектно-ориентированного проектирования, поскольку окружность, очевидно, не является частным случаем точки. Мы исправим эту оплошность в одном из следующих параграфов.

```
class Circle : public Pixel {
    double radius;
public:
```

Опишем теперь конструктор для класса `Circle`. При этом нам придётся принять на один параметр больше, чем в конструкторе класса `Pixel`. Кроме того, поскольку единственный конструктор базового класса требует указания трёх параметров, нам придётся задать параметры для вызова конструктора базового класса, как это обсуждалось на стр. 86. Окончательно описание выйдет таким:

```
Circle(double x, double y, double rad, int color)
    : Pixel(x, y, color), radius(rad) {}
```

Разумеется, в объекте необходимо описать функции `Show()` и `Hide()`, предназначенные для рисования и стирания с экрана окружности. Естественно, они будут отличаться от функций, предназначенных для одиночного пиксела; мы, как и при рассмотрении класса `Pixel`, воздержимся от описания конкретных тел этих функций, ограничившись их заголовками:

```
void Show();
void Hide();
```

Нетерпеливый читатель может предположить, что сейчас мы будем описывать функцию `Move()`, аналогичную той, что описана выше для класса `Pixel`. Вместо этого мы предложим немного подумать. Легко заметить, что **функция `Move()` для нового класса окажется абсолютно такой же, как и для класса базового**, то есть нам потребуется написать не только точно такой же заголовок, но и точно такое же, с точностью до последней буквы, тело функции! Это и не удивительно, ведь вне всякой зависимости от формы графической фигуры её перемещение по экрану производится абсолютно одинаково, в три шага: стереть, изменить координаты, нарисовать в новом месте.

Возникает естественный вопрос, нельзя ли использовать для класса `Circle` ту же самую функцию `Move()`, которая уже описана для класса `Pixel`, не описывая новой версии этой функции для `Circle`.

Легко заметить, что (если не предпринять специальных мер) простой вызов функции `Move()` для объекта класса `Circle`, конечно, возможен (ведь это же публичная функция базового класса, а наследование произведено по публичной схеме), но **не приведёт к нужным**

нам результатам, и вот почему. Во время трансляции тела функции `Pixel::Move()` компилятор может ничего не знать о существовании потомков у класса `Pixel`, которые к тому же вводят свои версии функций `Show()` и `Hide()`. Поэтому компилятор, естественно, вставляет в машинный код функции `Pixel::Move()` обычные вызовы функций `Pixel::Show()` и `Pixel::Hide()`, то есть инструкции `CALL` с жестко заданными исполнительными адресами, не подразумевающие никаких изменений.

Если теперь вызвать функцию `Move()` для объекта класса `Circle`, она для стирания с экрана объекта вызовет функцию `Hide()` в той её версии, в которой она описана для класса `Pixel`. Эта функция выполнит действия по «стиранию» с экрана объекта-точки, тогда как стереть на самом деле нужно окружность. То же самое произойдёт с функцией `Show()`: вместо окружности в новой позиции на экране будет отрисована точка. Ясно, что это совсем не то, что нам нужно.

Тем не менее, кажется странным и неправильным описывать для класса `Circle` функцию, слово в слово повторяющую другую, которая уже имеется в базовом классе. И здесь нам на помощь приходит механизм ***виртуальных функций***.

Кратко говоря, виртуальная функция (или виртуальный метод) — это функция, описывающая такое действие над объектом, относительно которого предполагается, что аналогичное действие будет определено и для объектов классов-потомков, но, возможно, для потомков оно будет выполняться иначе, чем для базового класса.

В терминах теории ООП можно сказать, что виртуальным методом задаётся реакция объекта класса на некоторый тип сообщений в случае, если:

- предполагается, что у данного класса будут классы-потомки;
- объекты классов-потомков будут способны получать сообщение того же типа;
- объекты некоторых или всех потомков будут реагировать на эти сообщения иначе, чем это делает объект класса-предка.

Язык Си++ позволяет объявить в качестве виртуальной любую функцию-метод (кроме конструкторов и статических методов). Для этого перед заголовком функции ставится ключевое слово `virtual`. В отличие от вызовов обычных функций, вызовы функций виртуальных обрабатываются компилятором в предположении, что тип объекта, для которого вызывается функция, может отличаться от базового класса, и

что для этого объекта может потребоваться вызов совсем другой функции, нежели для базового класса.

В частности, если в классе `Pixel` поставить слово `virtual` перед каждой из функций `Show()` и `Hide()`, то при компиляции тела функции `Move()` компилятор будет знать, что объект, на который указывает параметр `this`, может быть как объектом самого класса `Pixel`, так и объектом какого-нибудь его класса-потомка, причём для такого объекта может потребоваться вызов *других версий* функций `Show()` и `Hide()`, введённых в соответствующем потомке. Код, сгенерированный компилятором в такой ситуации, будет несколько сложнее, чем для обычного вызова функции, но зато он будет вызывать именно функцию, соответствующую типу объекта.

Для любознательных читателей расскажем, как это достигается. Если в классе описана хотя бы одна виртуальная функция, то компилятор вставляет во все объекты этого класса невидимое поле, называемое ***указателем на таблицу виртуальных методов*** (англ. *virtual method table pointer, vmtp*). Для всего класса создаётся (в одном экземпляре) неизменяемая ***таблица виртуальных методов***, содержащая указатели на каждую из описанных в классе виртуальных функций.

Когда компилятор встречает вызов виртуальной функции, он генерирует объектный код, содержащий инструкции извлечь из объекта значение поля `vmtp`, затем обратиться по этому адресу к таблице виртуальных методов и из неё извлечь адрес нужной функции, и, наконец, обратиться к этой функции по полученному адресу.

Объект класса-потомка содержит, как мы знаем, все поля, присущие классу-предку. Это касается и невидимого поля `vmtp`, причём объекты класса-потомка имеют в этом поле иное значение, нежели объекты класса-предка. Для каждого класса-потомка компилятор создаёт (опять-таки в единственном экземпляре) свою собственную таблицу виртуальных методов, содержащую адреса соответствующих версий виртуальных функций; именно адрес этой таблицы заносится в поле `vmtp`.

Итак, объявим функции `Show()` и `Hide()` виртуальными в базовом классе. Это позволит использовать функцию `Move()` для любого из классов, унаследованных от `Pixel`, если только для этих классов описаны собственные (правильно работающие) функции `Show()` и `Hide()`.

Прежде чем завершить описание классов `Pixel` и `Circle`, отметим, что в классе, в котором имеется хотя бы одна виртуальная функция, рекомендуется всегда явно описывать деструктор и объявлять его виртуальным. Зачем это нужно, станет ясно из дальнейшего материала. Сейчас просто сделаем это, чтобы компилятор не выдавал предупреждения.

Окончательно заголовки наших классов примут следующий вид:

```
class Pixel {
protected:
    double x, y;
    int color;
public:
    Pixel(double ax, double ay, int acolor)
        : x(ax), y(ay), color(acolor) {}
    virtual ~Pixel() {}
    virtual void Show();
    virtual void Hide();
    void Move(double nx, double ny);
};

class Circle : public Pixel {
    double radius;
public:
    Circle(double x, double y, double rad, int color)
        : Pixel(x, y, color), radius(rad) {}
    virtual ~Circle() {}
    virtual void Show();
    virtual void Hide();
};
```

Отметим, что в описании класса `Circle` слово `virtual` можно, в принципе, опустить; функции, совпадающие по профилю с виртуальными функциями базового класса, объявляются виртуальными автоматически.

§ 9.1.2. Чисто виртуальные функции. Абстрактные классы

Как отмечалось в сноске на стр. 90, наследование класса «окружность» от класса «точка» нарушает принципы объектно-ориентированного проектирования, поскольку окружность не является частным случаем точки. Попробуем исправить ситуацию. Для этого заметим, что и точка, и окружность — частные случаи *геометрических фигур*, причём можно считать, что каждая геометрическая фигура обладает цветом и имеет координаты *точки привязки*. Для обычной точки в качестве точки привязки выступает она сама, для окружности точкой привязки будет её центр. Для других фигур точку привязки можно выбрать разными способами; так, для какого-нибудь прямоугольника это

может быть либо центр пересечения диагоналей, либо одна из вершин, и т. п.

Отметим, что такое представление об абстрактном понятии геометрической фигуры позволяет нам указать единый для всех фигур алгоритм передвижения фигуры по экрану: стереть фигуру с экрана, изменить координаты точки привязки, отрисовать фигуру в новом месте. С этим алгоритмом мы уже знакомы, он был реализован в функции `Move()` на стр. 90.

Теперь уже ясно, как нужно изменить архитектуру нашей библиотеки классов, чтобы привести её в соответствие с основными принципами объектно-ориентированного программирования. Понятия «точка» и «окружность» не являются частными случаями друг друга, но оба они являются частным случаем понятия «геометрическая фигура». Поэтому, если мы опишем класс для представления абстрактной геометрической фигуры и от него унаследуем оба класса `Pixel` и `Circle`, такая архитектура будет полностью удовлетворять требованиям теории объектно-ориентированного проектирования.

Прежде чем приступать к описанию класса, представляющего геометрическую фигуру, отметим ещё один важный момент. Описывая в предыдущем параграфе классы для точек и окружностей, мы не писали конкретных тел для методов `Show()` и `Hide()`, но предполагали при этом, что в рабочей программе тела этих методов будут описаны (с учетом конкретной платформы разработки, используемой графической библиотеки и т. п.). В противоположность этому, тела методов `Show()` и `Hide()` для абстрактной геометрической фигуры описать *невозможно*; действительно, как можно нарисовать на экране фигуру, относительно которой неизвестно, как она выглядит?!

Несмотря на это, мы точно знаем, как будет выглядеть функция `Move()` в предположении, что для классов-потомков будут правильно описаны методы `Show()` и `Hide()`.

Иначе говоря, мы знаем, что все объекты классов-потомков данного класса должны уметь получать сообщение определённого типа, но мы не можем при описании базового класса описать какую бы то ни было реакцию на такие события, поскольку таковая реакция полностью зависит от типа нашего потомка. При этом для описания некоторых других (более общих) методов базового класса нам необходимо знание о том, что реакция на соответствующие события будет предусмотрена во всех наших потомках.

Язык Си++ имеет специальный механизм, отражающий подобные случаи. Этот механизм называется ***чисто виртуальные функции***

(англ. *pure virtual functions*). Описывая в классе чисто виртуальную функцию, программист информирует компилятор о том, что функция с таким профилем будет существовать во всех классах-потомках, что под неё необходимо зарезервировать позицию в таблице виртуальных функций, но при этом сама функция (её тело) для базового класса описываться не будет, так что значение адреса этой функции в таблице виртуальных функций следует оставить нулевым. Синтаксис описания чисто виртуальной функции таков:

```
class A {
    // ...
    virtual void f() = 0;
    // ...
 };
```

Видя на месте тела функции специальную лексическую последовательность «`= 0;`», компилятор воспринимает функцию как чисто виртуальную.[2]

Если в порождённом классе не описать одну из функций, объявленных в базовом классе как чисто виртуальные, компилятор считает, что функция осталась чисто виртуальной и предполагает, что от такого класса, в свою очередь, будет унаследован потомок, который и определит, наконец, конкретное тело для виртуальной функции соответствующего профиля.

Класс, в котором есть хотя бы одна чисто виртуальная функция, называется ***абстрактным классом***. Полезно будет запомнить, что **компилятор не позволяет создавать объекты абстрактных классов**. Единственное назначение абстрактного класса — служить базисом для порождения других классов, в которых все чисто виртуальные функции будут конкретизированы.

Подытожим наши рассуждения. Чтобы соответствовать принципам объектно-ориентированного программирования, нам следует описать класс, представляющий абстрактную геометрическую фигуру, обладающую цветом и координатами точки привязки, но не обладающую конкретной формой; назовём этот класс `GraphObject`. Классы `Pixel` и `Circle` нужно переписать, унаследовав от `GraphObject`. В классе `GraphObject` методы `Show()` и `Hide()` будут объявлены как чисто виртуальные, что сделает сам этот класс абстрактным классом.

[2]Такой синтаксис трудно назвать удачным, особенно если учесть, что никакое число, кроме нуля, использоваться здесь не может; тем не менее, именно таков синтаксис в языке Си++.

Опишем теперь класс `GraphObject`:

```
class GraphObject {
protected:
    double x, y;
    int color;
public:
    GraphObject(double ax, double ay, int acolor)
        : x(ax), y(ay), color(acolor) {}
    virtual ~GraphObject() {}
    virtual void Show() = 0;
    virtual void Hide() = 0;
    void Move(double nx, double ny);
};
```

Описание функции `Move()` мы не приводим, поскольку оно слово в слово повторяет описание метода `Pixel::Move()` на стр. 90. Отметим, что в теле функции `Move()` вызываются функции `Show()` и `Hide()`. То, что тела для этих функций нами не заданы, не создаёт никаких проблем, поскольку для любого класса-потомка, объекты которого можно будет создавать, таблица виртуальных методов будет содержать адреса конкретных функций, описанных в этом классе-потомке.

Заголовки классов `Pixel` и `Circle` будут теперь выглядеть так:

```
class Pixel : public GraphObject {
public:
    Pixel(double x, double y, int col)
        : GraphObject(x, y, col) {}
    virtual ~Pixel() {}
    virtual void Show();
    virtual void Hide();
};
class Circle : public GraphObject {
    double radius;
public:
    Circle(double x, double y, double rad, int color)
        : GraphObject(x, y, color), radius(rad) {}
    virtual ~Circle() {}
    virtual void Show();
    virtual void Hide();
};
```

Для этих классов, в отличие от класса `GraphObject`, необходимо описать конкретные тела методов `Show()` и `Hide()`.

§ 9.1.3. Наследование ради частного случая конструирования

В практическом программировании часто применяют один упрощённый случай наследования, при котором класс-потомок отличается от предка только набором конструкторов, то есть он не вводит ни новых методов, ни новых полей. Объекты такого класса создаются из соображений экономии объёма кода, чтобы не повторять одни и те же действия при конструировании однотипных объектов.

Чтобы проиллюстрировать сказанное на примере, для начала мы введём ещё один класс-потомок класса `GraphObject`, представляющий ломаную линию, а затем опишем графический объект «квадрат» как частный случай ломаной линии.

Напомним, что каждая геометрическая фигура в нашей системе имеет точку привязки; ломаную при этом проще всего хранить в виде списка координатных пар, задающих смещение каждой вершины ломаной относительно точки привязки. Для организации такого списка мы в закрытой части класса опишем структуру, задающую элемент списка. Исходно будем создавать ломаную, не имеющую ни одной вершины, а для добавления новых вершин будем использовать метод, который назовём `AddVertex()`.[3] Напишем заголовок класса:

```
class PolygonalChain : public GraphObject {
    struct Vertex {
        double dx, dy;
        Vertex *next;
    };
    Vertex *first;
public:
    PolygonalChain(double x, double y, int color)
        : GraphObject(x, y, color), first(0) {}
    virtual ~PolygonalChain();
    void AddVertex(double adx, double ady);
```

[3] Это лучше, чем пытаться тем или иным способом передать координаты вершин в конструктор, поскольку мы не знаем заранее количество вершин, так что для передачи их в качестве параметра конструктора пришлось бы в том месте, где создаётся объект, формировать некую динамическую структуру данных (массив либо список), что потребовало бы дополнительных усилий.

```
    virtual void Show();
    virtual void Hide();
};
```

Функция `AddVertex()` будет (для экономии усилий) добавлять новую вершину в начало списка, а не в конец; линия при этом будет изображаться на экране в обратном порядке, но это, естественно, ничего не изменит:

```
void PolygonalChain::AddVertex(double ax, double ay) {
    Vertex *tmp = new Vertex;
    tmp->dx = ax;
    tmp->dy = ay;
    tmp->next = first;
    first = tmp;
}
```

Ясно, что для объекта, использующего динамическую память, необходимо предусмотреть деструктор, освобождающий память. Напишем его:

```
PolygonalChain::~PolygonalChain() {
    while(first) {
        Vertex *tmp = first;
        first = first->next;
        delete tmp;
    }
}
```

Как обычно, мы воздержимся от описания тел функций `Show()` и `Hide()`, но будем предполагать, что это сделано.

Пусть теперь нам нужен класс для представления квадрата, стороны которого параллельны осям координат, а длина стороны задаётся параметром конструктора. Ясно, что такой квадрат представляет собой частный случай ломаной, описываемой классом `PolygonalChain`. Если в качестве точки привязки выбрать левую нижнюю вершину квадрата, а длину стороны квадрата обозначить буквой a, то ломаная должна начаться в точке привязки (что соответствует вектору $(0,0)$), пройти через точки $(a,0)$, (a,a), $(0,a)$ и вернуться в точку $(0,0)$; всего, таким образом, ломаная будет содержать пять вершин, причём первая и последняя будут совпадать, чтобы сделать ломаную замкнутой.

Чтобы не приходилось каждый раз для представления квадрата писать шесть строк кода (одну для описания объекта `PolygonalChain`,

остальные для добавления вершин), можно описать класс (мы назовём его `Square`), который будет унаследован от `PolygonalChain`, причём отличаться будет только конструктором:

```
class Square : public PolygonalChain {
public:
    Square(double x, double y, double a, int color)
        : PolygonalChain(x, y, color)
    {
        AddVertex(0, 0);
        AddVertex(0, a);
        AddVertex(a, a);
        AddVertex(a, 0);
        AddVertex(0, 0);
    }
};
```

Подчеркнём ещё раз, что больше ничего описывать для квадрата не нужно, всё остальное сделают методы базового класса.

§ 9.1.4. Виртуальный деструктор

Ранее при обсуждении виртуальных функций на стр. 93 мы отметили, что в классе, имеющем хотя бы одну виртуальную функцию, деструктор тоже следует сделать (объявить) виртуальным, но не объяснили причины этого. Попробуем сделать это сейчас.

При активном использовании полиморфизма нередко возникает ситуация, когда нужно применить оператор `delete` к указателю, имеющему тип ≪указатель на базовый класс≫, притом что указывать он может и на объект потомка. В этой ситуации необходимо, естественно, вызвать деструктор, соответствующий типу уничтожаемого объекта, а не указателя. Так, допустим, мы описали указатель на класс `GraphObject`:

```
GraphObject *ptr;
```

Теперь вполне корректным будет такое присваивание:

```
ptr = new Square(27.3, 37.7, 0xff0000, 10.0);
```

В результате этого наш указатель имеет тип ≪указатель на `GraphObject`≫, но реально указывает на объект класса `Square`. Если теперь потребуется уничтожить этот объект, мы можем без всяких опасений выполнить

```
delete ptr;
```

Поскольку указатель имеет тип `GraphObject*`, а деструктор класса `GraphObject` виртуальный, компилятор произведёт вызов деструктора через таблицу виртуальных методов уничтожаемого объекта. В результате этого будет вызван неявный деструктор для класса `Square`, который вызовет деструктор класса `PolygonalChain`, а тот, в свою очередь, деструктор класса `GraphObject`. Если бы мы не объявили деструктор класса `GraphObject` как виртуальный, компилятор произвёл бы жесткий вызов деструктора по типу указателя, то есть был бы вызван только деструктор класса `GraphObject`. Между тем, уничтожаемый объект класса `Square` порождает и использует список в динамической памяти (из пяти элементов), и если деструктор класса `PolygonalChain` не будет вызван, то эти элементы превратятся в «мусор», то есть будут по-прежнему занимать память, не выполняя никакой полезной работы.

Объявление деструктора как виртуального практически не приводит к расходу памяти: таблица виртуальных методов увеличивается на один слот, что составляет несколько лишний байтов на каждый новый *класс* (не объект!). С другой стороны, сам факт наличия в классе виртуальных функций указывает на то, что при работе с объектами класса будет активно использоваться полиморфизм. Отметим, что при описании класса в большинстве случаев трудно предсказать, будут ли объекты потомков такого класса уничтожаться оператором `delete`, применяемым к указателю, имеющему тип «указатель на предка». Решив, что такое удаление нам не понадобится, мы рискуем в дальнейшем забыть об этом и получить трудную для обнаружения ошибку. В связи с этим считается, что деструктор любого класса, имеющего хотя бы одну виртуальную функцию, следует объявлять как виртуальный, не задумываясь о том, понадобится это в программе или нет; многие компиляторы выдают предупреждение, если этого не сделать.

§ 9.1.5. Ещё о полиморфизме

Приведём ещё один пример использования полиморфизма. Пусть мы создаём графическую сцену,[4] состоящую из разных графических объектов — точек, окружностей, многоугольников и, возможно, каких-то других элементов, представляемых объектами классов, унаследованных от `GraphObject`. При этом на момент написания программы мы не знаем,

[4]Напомним, что под сценой в компьютерной графике обычно понимается весь набор графических объектов, видимых одновременно.

сколько и каких именно объектов будет в сцене; так бывает, если описание сцены нужно получить из внешнего источника (например, прочитать из файла), либо если сцена генерируется во время исполнения программы (например, случайным образом, что часто используется во всевозможных программах-«скринсейверах»).

Допустим, в некий момент в программе нам всё же становится известно, сколько объектов будет содержать сцена. Это позволит использовать для хранения всей сцены динамически создаваемый массив указателей на объекты потомков `GraphObject`. Пусть, например, количество объектов сцены будет храниться в переменной `scene_length`, а указатель на сам массив мы назовём просто `scene`:

```
int scene_length;
GraphObject *scene;
```

Когда переменная `scene_lenght` тем или иным способом получит значение (например, оно будет прочитано из файла), можно будет завести массив:

```
scene = new GraphObject*[scene_lenght];
```

Теперь благодаря полиморфизму оказываются корректны, например, следующие присваивания (при условии, конечно, что `i` не превышает `scene_length`):

```
scene[i] = new Pixel(1.25, 15.75, 0xff0000);
// ...
scene[i] = new Circle(20.9, 7.25, 0x005500, 3.5);
// ...
scene[i] = new Square(55.0, 30.5, 0x008080, 10.0);
// ...
```

и тому подобные. Таким образом, мы получаем массив указателей типа `GraphObject*`, каждый из которых на самом деле указывает на некоторый объект *класса-потомка*.

Обратим внимание на то, что нам может вовсе никогда больше не потребоваться знать, на объекты какого типа указывают конкретные указатели в нашем массиве. Вне зависимости от конкретных типов, мы вполне можем перемещать объекты по экрану, гасить их и снова отрисовывать, ведь методы `Show()`, `Hide()` и `Move()` доступны для класса `GraphObject`, а значит, могут быть вызваны по указателю типа `GraphObject*` без уточнения типа объекта.

Точно таким же образом благодаря наличию виртуального деструктора можно уничтожить все объекты сцены, а потом и саму сцену:

```
for(int i=0; i<scene_length; i++)
    delete scene[i];
delete [] scene;
```

Подобные ситуации часто возникают в более-менее сложных программах. Поскольку конкретные методы, которые нужно вызывать, становятся известны только во время исполнения программы, такой вид полиморфизма называется ***динамическим полиморфизмом***; он становится возможным благодаря механизму виртуальных функций и является, в конечном итоге, их предназначением.

§ 9.2. Теоретико-множественное описание наследования

Всякая селёдка — рыба, но не всякая рыба — селёдка.

Как уже говорилось, при описании объектно-ориентированного программирования можно использовать различные варианты терминологии. Так, термин «вызов метода объекта» эквивалентен термину «отправка сообщения объекту», просто эти термины относятся к разным терминологическим системам: о вызовах методов мы говорим при изучении практического применения ООП, тогда как передача сообщения — термин, относящийся к теории ООП и к тем языкам программирования, которые полностью соответствуют этой теории (к таким языкам относится, например, Smalltalk).

Класс с теоретической точки зрения представляет собой *множество* объектов, удовлетворяющих определённым условиям. В этом смысле порождённый (наследуемый, или дочерний) класс представляет собой *подмножество*. Действительно, согласно закону полиморфизма объект порождённого класса может быть использован в качестве объекта базового класса, то есть, попросту говоря, является одновременно и объектом дочернего, и объектом базового класса; в то же время объект базового класса вовсе не обязан быть объектом класса порождённого.

С другой стороны, как мы уже говорили, наследование представляет собой *уточнение* свойств объекта, или, иначе говоря, переход от общего случая к частному. Ясно, что множество частных случаев представляет собой подмножество множества случаев общих.

Итак, класс есть описание множества объектов, а порождённый класс представляет собой описание подмножества такого множества.

Естественным следствием такого рассмотрения является термин ***подкласс*** (англ. ***subclass***) для обозначения порождённого класса и термин ***надкласс*** или ***суперкласс*** (англ. ***superclass***) — для обозначения базового.

Такая терминология часто порождает определённую путаницу. Дело тут в том, что объект порождённого класса (т.е. подкласса) мало того, что памяти занимает заведомо не меньше (а при добавлении новых полей — и больше), нежели объект класса базового (т.е. суперкласса), но ещё и *содержит в себе объект базового класса*, т.е. объект суперкласса оказывается *подобъектом* объекта подкласса.

Такая путаница обусловлена исключительно применением двух разных терминологических систем в одном месте. Когда речь идёт о суперклассах и подклассах — это значит, что используется теоретико-множественная терминология. Когда же речь заходит об используемой памяти и подобъектах — очевидно, что разговор идёт в терминах реализаторской (прагматичной) точки зрения.

Так или иначе, обе терминологические системы используются и имеют право на существование; вы на практике можете столкнуться как с одним вариантом терминологии, так и с другим, а в некоторых случаях — и с их смешением, как в вышеприведённом примере. Поэтому желательно понимать, что означают термины обеих систем.

§ 9.3. Операции приведения типа

В процессе программирования часто приходится изменять тип выражения. Иногда это делается *неявно*, как, например, в случае сложения целочисленного значения со значением дробным (с плавающей точкой). В иных случаях (как, например, при изменении типа указателя) приходится явно указывать компилятору новый тип выражения.

В языке Си это делалось с помощью ***операции преобразования типа***, записываемой как унарная операция, символ которой есть имя типа, заключенное в круглые скобки, как, например, в следующем выражении:

```
char *p = (char*)malloc(100);
```

Здесь значение выражения `malloc(100)`, имеющее тип `void*`, приводится к типу `char*`.

Операция приведения типа *опасна* в том смысле, что её применение позволяет при желании обойти любые ограничения, вводимые системой типизации, включая, например, запреты на запись в константные

области памяти и даже защиту данных в классах. Необдуманное применение преобразования типов приводит к запутыванию программы и, в конечном счёте, к трудно выявляемым ошибкам.

Для снижения негативного эффекта операции приведения типов, а также для поддержки полиморфного программирования в языке Си++ вводятся четыре дополнительные операции, предназначенные для преобразования типа выражения. Эти операции имеют достаточно нетривиальный синтаксис: сначала записывается ключевое слово, задающее операцию (`static_cast`, `dynamic_cast`, `const_cast` или `reinterpret_cast`), затем в угловых скобках ставится имя нового типа и, наконец, в круглых скобках записывается само выражение, тип которого необходимо изменить, например:

```
Square *sp = static_cast<Square*>(scene[i]);
```

В отличие от операции приведения типов в языке Си, которая применялась для всех случаев смены типа, каждая из операций Си++ предназначена для своего случая. Так, операция `const_cast` позволяет снять или, наоборот, установить сколько угодно модификаторов `const`;[5] попытка сделать с её помощью любое другое изменение типа вызовет ошибку при компиляции:

```
int *p;
const int *q;
const char *s;
// ...
q = p;  // можно без преобразования
p = q;  // ошибка! снятие const
p = const_cast<int*>(q); // правильно
p = const_cast<int*>(s); // ошибка!
```

Отметим, что наличие в языке операции `const_cast` не отменяет опасности такого преобразования. Реальная потребность в обходе константной защиты возникает крайне редко; прежде чем применять преобразование, подумайте, всё ли вы правильно делаете, не забыли ли вы, например, пометить словом `const` функцию-метод, не изменяющую состояние объекта, и т.п. Для применения операции `const_cast` необходимы очень веские причины, и сакраментальное «без неё не работает» такой причиной не является. В некоторых программистских коллективах на

[5] Также эта операция работает с модификатором `volatile`, который мы в нашем курсе не рассматриваем.

каждое применение `const_cast` необходимо личное разрешение руководителя разработки.

Операция `static_cast` предназначена для работы с наследуемыми объектами и позволяет преобразовать указатель или ссылку в направлении, противоположном закону полиморфизма, т.е. от базового класса к порождённому. Попытка произвести любое другое преобразование вызовет ошибку. Приведём примеры:

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C { /* ... */ };
A *ap;
B *bp;
C *cp;
// ...
ap = bp; // можно без преобразования
bp = ap; // ошибка!
bp = static_cast<B*>(ap); // допустимо
cp = static_cast<C*>(ap); // ошибка!
```

Отметим, что делать это следует только в том случае, если мы **действительно** уверены, что по данному адресу расположен объект именно того типа, к которому мы намерены преобразовывать; иное приведёт к аварии.

Операция `reinterpret_cast` позволяет произвести любое преобразование (чего угодно во что угодно), если только компилятор понимает, как это сделать (в частности, преобразовать объекты структур разных типов друг к другу не получится, поскольку непонятно, как такое преобразование производить). Фактически эта операция эквивалентна операции языка Си. Рекомендуется, однако, применять именно `reinterpret_cast`, а не операцию Си, поскольку такие преобразования требуют особого внимания, а выражение с использованием `reinterpret_cast` лучше заметно в тексте программы, чем имя типа в скобках.

Несколько особое место занимает операция `dynamic_cast`. Три операции, которые мы рассмотрели выше, служат для управления системой контроля типов, т.е. для управления компилятором, и не порождают действий, осуществляемых во время исполнения программы.[6] Опе-

[6] Кроме преобразования между целыми числами и числами с плавающей точкой, что требует неких действий; иногда бывает необходимо изменить и численное значение указателя, но в нашем курсе не рассматриваются механизмы, порождающие

рация `dynamic_cast`, в отличие от остальных, предполагает проведение нетривиальной проверки во время исполнения программы.

Подобно операции `static_cast`, операция `dynamic_cast` предназначена для преобразования адресов объектов в направлении, противоположном закону полиморфизма, т.е. от адреса предка к адресу потомка. В случае со `static_cast` ответственность за корректность такой операции возлагается на программиста: именно программист тем или иным способом должен проверить, что преобразуемый адрес (будь то указатель или ссылка) указывает на объект нужного типа. Если же применить `dynamic_cast`, то она сама проведёт необходимую проверку и в случае, если преобразование некорректно (то есть по заданному адресу в памяти не находится объект нужного типа), вернёт нулевой указатель. Проверка, таким образом, осуществляется во время исполнения, т.е. *динамически*, отсюда название операции.

Проверка типа производится на основании значения указателя на таблицу виртуальных функций; дело в том, что такая таблица уникальна для каждого класса, имеющего виртуальные методы, т.е. её адрес однозначно идентифицирует класс объекта. Таким образом, `dynamic_cast` может работать только с классами (или структурами), имеющими виртуальные функции. В некоторых источниках такие классы называют ***полиморфными***, что не совсем корректно: как мы видели, полиморфизм в определённом смысле работает и для классов, не имеющих виртуальных функций.

Отметим ещё один немаловажный момент. Обычно реализации `dynamic_cast` весьма неэффективны по времени исполнения, т.е. работают очень медленно. Поэтому злоупотреблять ими не следует.

§ 9.4. Ещё о преобразовании типов в обработчиках исключений

При обсуждении преобразований типов выражений в обрабочиках исключительных ситуаций (см. стр. 80) мы отметили, что одним из наиболее важных видов преобразования является преобразование по закону полиморфизма, однако подробное обсуждение этого отложили, поскольку на тот момент ещё не было введено наследование.

Возвращаясь к этому вопросу, заметим, что третий и последний вид допустимых преобразований от типа выражения в операторе `throw` к типу, указанному в заголовке `catch` — это преобразование адреса (т.е.

такую необходимость.

указателя или ссылки) объекта-потомка к соответствующему адресному типу объекта-предка. Таким образом, если мы опишем два класса, причём один унаследуем от другого:

```
class A { /* ... */ };
class B : public A { /* ... */ };
```

то обработчик вида

```
catch(const A& ex) { /* ... */ }
```

сможет обрабатывать исключения *обоих* типов, т.е. результат как оператора `throw A(...);`, так и `throw B(...);`.

Это свойство используется для создания *иерархий исключительных ситуаций*. Например, мы можем поделить все ошибки, возникающие в какой-либо программе или библиотеке, на следующие категории:

- ошибки, возникающие по вине пользователя:
  - синтаксические ошибки при вводе (например, буквы там, где ожидается число);
  - неправильно указано имя файла;
  - неправильно введённый пароль;
  - недопустимая комбинация требований (например, одновременное требование упорядочивания по возрастанию и по убыванию);
- ошибки, обусловленные внешними факторами (средой выполнения):
  - переполнение диска;
  - отсутствие файлов, необходимых для работы;
  - прочие ошибки операций ввода-вывода;
  - недостаток оперативной памяти;
  - ошибки при работе с сетью;
  - и т.п.
- ситуации, свидетельствующие об ошибке в самой программе и требующие её исправления (например, переменная принимает значение, которое по тем или иным причинам принимать не должна).

Опишем теперь класс `Error`, соответствующий понятию «любая ошибка». В полном соответствии с принципами объектно-ориентированного программирования перейдём от общего к частному, унаследовав от класса `Error` подклассы `UserError`, `ExternalError` и `Bug` для обозначения, соответственно, пользовательских ошибок, внешних ошибок и ошибок в программе. От класса `UserError`, в свою очередь, унаследуем классы `IncorrectInput`, `WrongFileName`, `IncorrectPassword` и т.д.

После этого обработчик вида

```
catch(const IncorrectPassword& ex) { /* ... */ }
```

будет обрабатывать только исключения, связанные с неправильным паролем, тогда как обработчик вида

```
catch(const UserError& ex) { /* ... */ }
```

будет реагировать на любые ошибки пользователя, что же касается обработчика

```
catch(const Error& ex) { /* ... */ }
```

то он сможет «поймать» вообще любое исключительное событие из нашей иерархии.

Содержание