Объектно-ориентированное программирование. Семинары.

Бабичев С.Л.

Данный документ содержит материалы семинаров по предмету «Объектно-ориентированное программирование» в 4-м семестре на факультетах ФОПФ и ФМБФ с 2006 по 2015 годы.

1 Основные понятия.

Цели курса:

- познакомить с одним из способов создавать действительно большие программы — объектно ориентированным программированием;
- изучить основные понятия объектно ориентированного программирования в целом;
- ознакомиться с языком программирования `C++`
- ознакомиться с отображением конструкций объектно-ориентированного программирования в конструкции языка `C++`;
- научиться создавать модели методом декомпозиции (разбивая проблему на объекты и проектируя взаимодействие между объектами);
- научиться реализовывать простые модели на языке `C++`.

1.1 Что будем делать на семинарах

На семинарах мы будем:

- изучать основы объектно ориентированного программирования;
- изучать базовые конструкции языка `C++`;
- писать небольшие программы, иллюстрирующие ООП и `C++`.

1.2 А может ну его, это ООП? А что другое можно предложить?

Главная мысль: на написание программы затрачивается ценное время программиста T_{pr}. На исполнение программы требуется ценное время конечного пользователя T_{us}. Общие затраты есть $\sum\limits_{i=1}^{N_{pr}} T_{pr_i} \times C_{pr_i} + \sum\limits_{j=1}^{N_{us}} T_{us_j} \times C_{us_j}$, где N_{pr} есть число программистов, T_{pr_i} — время, затраченное $i-$м программистом, C_{pr_i} — стоимость единицы времени $i-$го программиста, а N_{us} — число пользовательских запусков программы, T_{pr_i} — время, затраченное на $j-$й запуск программы, C_{us_j} — стоимость единицы времени пользователя на $j-$й запуск программы. Увеличивая затраты на первую компоненту мы иногда можем уменьшить затраты на вторую. Объектно ориентированное программирование предназначено для того, чтобы достаточно резко уменьшить затраты на первую компоненту, возможно, увеличивая затраты на вторую.

1.3 Технологии создания программ. Исторический экскурс.

1.3.1 Понятие сложности программы

Что есть сложная программа? Если имеются две программы, `A` и `B`, как определить, какая из них более сложная?

```
// Программа A
int foo(int x, double y, char z) {
  int ret = x + y*2 + (z - '0');
  return ret;
}

int main() {
  int x = 10;
  double y = 20.0;
  char z = '7';
  int k = foo(x,y,z);
  printf("k=%d\n", k);
}
```

и

```
// Программа B
int x;
double y;
char z;
int ret;

void foo() {
  ret = x + y*2 + (z-'0');
}

int main() {
  x = 10;
  y = 20.0;
  z = 'z';
  foo();
  printf("ret=%d\n", ret);
}
```

Попробуем нарисовать картинку, на которой овалами мы будем рисовать строительные блоки, из которых мы и создаём программу, а линиями со стрелками — связи между этими блоками.

Вот как будет выглядеть первая программа:

Мы видим два блока — `main` и `foo`. Между ними имеется две группы связей — три связи от `main` к `foo` — они помечены как `x`, `y` и `z`, и одна связь от `foo` к `main`, помеченная `ret`.

Вот как будет выглядеть вторая программа (рис. 2):

Упс, а почему так сложно???? Впрочем, если подумать, то всё становится понятно. Блоков у нас стало не два, как раньше, а целых шесть — переменные `x`, `y`, `z` и `ret` теперь глобальные и поэтому они тоже стали строительными блоками. Если блоки `foo` и `main` есть блоки кода, то `x`, `y`, `z` и `ret` есть блоки данных. Если блок данных доступен блоку кода на чтение, то стрелка направлена от блока данных к блоку кода, если на запись, то от блока кода к блоку данных. Все наши блоки данных доступны всем нашим блокам кода как на чтение, так и на запись (пусть потенциально, но доступны), отсюда такое множество пар стрелок. Блоки кода тоже могут быть связаны между собой отношениями передачи управления. После того, как `main` вызывает `foo`, происходит передача управления (стрелка, помеченная `CI`), после завершения `foo` управление возвращается `main` (стрелка `CO`).

Рис. 1: Структурная схема программы `A`

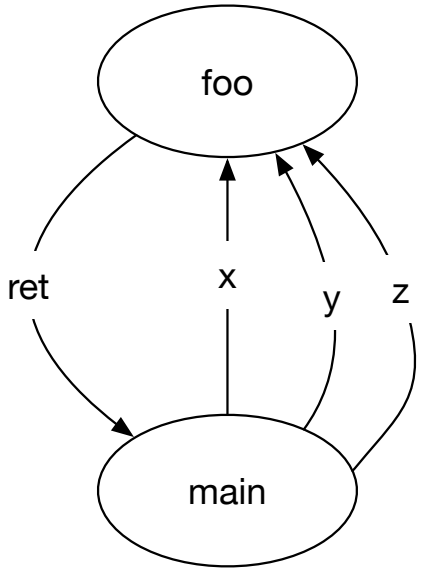
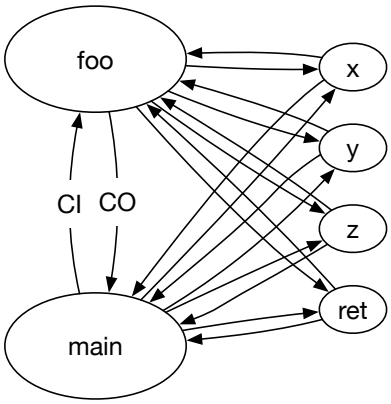


Рис. 2: Структурная схема программы B



Сложность программы можно определить как комбинаторную функцию общего количества различных комбинаций всевозможных связей. Например, во второй программе добавление переменной `t` добавит связи от неё до всех блоков кода, а добавление функции `bar` добавит связи от её до всех глобальных переменных. Если определять сложность как количество возможных комбинаций, то сложность будет пропорциональна N^2, где N — количество блоков в программе. С целью уменьшения сложности следует уменьшить либо само количество блоков, либо количество связей между ними, путём запрета определённых связей. Давайте посмотрим, как исторически происходила эта борьба.

1.3.2 Программирование в машинных кодах

Программирование в машинных кодах вынуждает программиста использовать адреса переменных в качестве строительных блоков данных и машинные команды в качестве строительных блоков кода. Машинные команды передачи управления могут ссылаться на любой машинный адрес (в том числе и невозможный). Для программирования требуется скрупулёзно вести таблицу соответствия желаемых переменных с их адресами. Метрика сложности программы получается очень высокой — каждый блок данных требуется выбирать из большого количества возможных адресов (высока энтропия системы) и в каждой команде перехода

адрес назначения также можно выбирать из такого же множества возможных машинных адресов, сами машинные команды тоже выбираются из пространства чисел.

1.3.3 Программирование на языке ассемблера

Сложность программы немного уменьшается за счёт то, что введение языка ассемблера позволяет немного уменьшить энтропию системы:

1. коды команд теперь выбираются из осмысленного множества сокращений, например, операция сложения, которая в машинных кодах кодировалась числом `107` (например), теперь записывается как `ADD`.
2. блоки данных выбираются из множества идентификаторов, придуманных программистом.
3. переходы на блоки кода теперь возможны не в любую точку программы, а только на метки, назначенные программистом.

1.3.4 Первые языки высокого уровня

Первые языки программирования служили целью упростить, в первую очередь, написание кода программы и они были во многом, по сути, тем же языком ассемблера. Типы данных и операции позволяли их точное отображение на машинный язык. Полезным стало введение библиотек — процедур и функций, которые уже кто-то реализовал до нас и включил в поставку. Программист, однако, мог допустить ошибку в имени переменной, которая осталась бы незамеченной.

1.3.5 Структурное программирование

В структурном программировании весь поток управления программы делится на так называемые управляющие структуры — *если*, *альтернативное если*, несколько видов *циклов*.

Основной целью структурного программирования была попытка уменьшить сложность написания, и, что главное — верификации программного кода.

Все управляющие структуры имели ровно один вход и ровно один выход. Гарантировалось, что если управление попадает на вход управляющей структуры, то оно (при правильном использовании управляющих данных) обязательно дойдёт до выхода их управляющий структуры. В классическом варианте были категорически запрещены операторы перехода изнутри управляющей структуры наружу и снаружи внутрь. Это реально уменьшило эффективную сложность программ, и, как ни странно, вначале усложнило жизнь программистам, приучившихся к анархическому программированию по принципу «всё дозволено».

Впрочем, структурное программирование почти не повлияло на сложность программ, связанную с блоками данных — оно затронуло только блоки кода и переходы между ними.

1.3.6 Процедурное программирование

Процедурное программирование расширяет структурное. Задача разбивается на процедуры, связь между которыми осуществляется либо через параметры процедур (что предпочтительнее), либо через общие переменные (что мы, как уже видели, приводит к более сложным программам). Процедурное программирование позволило немного уменьшить сложность, связанную с блоками данных — благодаря введению понятия *локальные для процедуры переменные*. Это позволило уменьшить количество общих переменных, что, как мы уже видели, благотворно влияет на уменьшение сложности программы.

Процедурное программирование привело к созданию понятия *библиотека функций* — например, графическая библиотека или библиотека математических функций или библиотека линейной алгебры. Это, наконец-то, позволило набрать критическую массу удобных и полезных алгоритмов. Это позволило выбирать, каким образом проектировать большую программу — методом сверху вниз или методом снизу вверх.

Метод сверху вниз разбивает большую проблему на ряд более мелких, которые

- можно реализовывать разным программистам;

- проще в реализации.

Каждая из подпроблем в свою очередь разбивается на подподпроблемы и так далее. Все под(под)проблемы делятся на уже реализованные и ещё не реализованные. Для тестирования нереализованные подпроблемы заменяют *заглушками*, которые просто возвращают управление назад (возможно, что-то передавая для корректной работы вызывающей части). По мере готовности заглушки заменяют на реально решённые подпроблемы.

Метод снизу вверх сначала опирается на создание необходимых библиотек. Эти библиотеки отлаживаются на тестовых программах, затем, после их готовности, на них реализуется более крупная подзадача.

Практика показала, что оба способа годятся, хотя каждый из них имеет и свои недостатки. Например, при проектировании снизу вверх часто оказывается, что большая часть процедур и функций, которые имеются в написанных для задачи библиотеках, никогда не используются — они просто не потребовались (при проектировании сверху вниз о них бы и не вспомнили). Зато следующий проект на существующих библиотеках (отлаженных библиотеках!) уже создавать гораздо легче.

Здесь у нас впервые появляется понимание того, что большую программу пишет не один человек, и что, например, для библиотеки её автором может быть один, а использовать эту библиотеку может совершенно другой человек. Важным становится их взаимодействие, прямое — через личное общение, или косвенное — через документацию к библиотеке (кстати, такую документацию может писать ещё кто-то другой — библиотекарь, технический писатель). Мы в дальнейшем будем разделять тех, кто пишет функцию (библиотеку, модуль, класс, ...) — *авторов* и тех, кто ими пользуется — *пользователей*.

1.3.7 Модульное программирование

Развитие процедурного программирования, в котором вводится понятие *модуль*. Так же, как в процедурном программировании масса переменных покинула глобальную область видимости и осела внутри процедур (и функций), в модульном программировании стало возможным скрыть наличие ряда функций (и, наконец-то, данных) и собрать их использование внутри модуля. Появилось разделение на *интерфейс* (*interface*) модуля и его *реализацию* (*implementation*). В интерфейсе модуля описывалось, какие процедуры, функции и переменные может использовать субъект, которого мы назвали *пользователем*. Ему не разрешено менять детали реализации (возможно, ему даже не предоставят возможность узнать алгоритмы реализации), но тем, что описано в интерфейсе модуля, он пользоваться может.

По сути дела, все современные, не объектно-ориентированные языки давно предоставляют своим пользователям именно модульный подход. Если мы вспомним `Pascal` и `Delphi`, то конструкция

```
uses crt;
```

служит именно для подключения модуля с названием `crt` к нашему коду. После этой строки программе становятся доступны новые имена процедур, например `clrscr`.

В классическом Си включение

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

тоже можно рассматривать (с некоторым, но достаточно хорошим приближением) как включение модуля стандартного ввода-вывода. В самом деле, в *пространство имён* добавляются новые типы данных (такие, как `FILE`), константы (`NULL`, `EOF`), функции (`printf`, `scanf`, `fopen`) и т. д.

Вообще-то надо отметить, что управление *пространством имён* становится очень важным именно для больших программ и именно для программ, которые разрабатываются не одним программистом, а группой. Это позволяет каждому из участников проекта не заботиться о точном знании всех имён, которые могут использоваться во всём проекте остальными участниками. В процедурном программировании это не так. Если один из участников проекта, использующего процедурное программирование, написал функцию с именем `max`, принимающую массив целых чисел, то любой другой участник этого же проекта не может создать функцию с этим же именем, принимающую другие аргументы. В модульном программировании это оказывается возможным, если эти функции принадлежат разным модулям.

1.3.8 Объектно ориентированное программирование

ООП есть дальнейшее развитие модульного программирования и в него вводится понятие *объект*. Если модуль, как мы уже знаем, есть совокупность процедур и данных, выполняющих части одной задачи, то что есть объект? Может ли объект представиться модулем? Да, может, модульное программирование есть чистое подмножество объектно-ориентированного. А может ли модуль представиться объектом? В общем случае — нет, не может. Объектно ориентированное программирование — очередная итерация, попытка отобразить предметный мир на язык математических моделей. Это не панацея, существует множество проблем, для решения которых можно и нужно применять другой подход, например, функциональное программирование — не путать с процедурным программированием (это всё равно, что перепутать программирование как прикладную науку и раздел математики, называющийся *математическое программирование*).

2 Что есть ООП и почему мы о нём говорим

Итак, основным понятием ООП является *объект*, а не *модуль* или *процедура*. Основная программа в этом случае решает проблему — как разделить модель на объекты, что будет представлять из себя каждый объект и какую часть общей задачи он готов на себя принять.

Для примера давайте возьмём игру в шахматы. Какие объекты мы можем увидеть при попытке смоделировать шахматную игру?

2.0.1 Объекты. Методы и свойства

- *Шахматная доска*. Она имеет ряд *свойств*, таких, как количество горизонталей `lines` и количество вертикалей `columns`. В классических шахматах и то, и другое числа равны 8. Однако, в игре в международные шашки, проводящейся на похожей доске, количество клеток уже 100. Не исключено, что разработчик программы для игры в шахматы захочет в будущем попробовать силы в программировании международных шашек, поэтому в разных проектах эти свойства доски могут быть разными.

- Шахматная доска состоит из *полей*, каждое из которых имеет свой *цвет* и на которых могут размещаться

- шахматные фигуры. Их несколько видов и каждый из этих видов характеризуется набором возможных на пустой доске ходов. Более того, набор возможных ходов изменяется в зависимости от цвета фигуры — например, белые пешки могут двигаться только вверх, чёрные — только вниз. Расположение фигур на шахматной доске называется

- *позицией*. Являются ли одинаковыми позиции с одинаковым расположением на ней фигур? Как ни странно, могут не являться. Например, право на рокировку теряет та фигура, которая уже делала ход. Если в одной позиции белый король стоит на поле `e1`, ладья на `h1` и эти фигуры до этого ещё не ходили, то при отсутствии других ограничений (фигур между ними, король находится под шахом или проходит битое поле, совершая рокировку) среди возможных ходов в этой позиции будет присутствовать короткая рокировка. Если же в той же визуально позиции либо король, либо ладья уже совершали ход и вернулись на место, рокировки среди возможных ходов не будет. Можно сказать, что *позиция* имеет следующие *свойства* — «белый король совершал ход», «белая ладья h1 совершала ход» и так далее и что эти свойства, в отличие от количества горизонталей и вертикалей могут изменяться в зависимости от того, как эта позиция получилась из игры.

  Позиция имеет множество свойств — таких, как *очередь хода* (ход белых или ход чёрных), *номер хода в партии*. Каждое из свойств меняется не само по себе — нельзя, например, назначить позиции из играющейся партии ход белых или принудительно назначить следующий ход сороковым — эти свойства меняются только после совершения каких-либо *действий*. Только *действия* могут изменить состояние шахматной позиции и действие это (в шахматах) называется совершением хода. Однако, совершаемый ход должен удовлетворять правилам шахматной игры, такое действие, как совершение хода,

зависит от текущей позиции, в которой ход совершается. После совершения хода состояние *объекта* шахматная доска меняется — меняется ряд *свойств*, таких, как наличие права рокировки, очередь хода, расположение фигур. Такие действия, направленные на изменение состояния объекта мы будем называть *методами*.

2.1 Три кита ООП: инкапсуляция

Продолжая рассмотрение нашей шахматной программы, представим, что мы — разработчики *ядра* программы, то есть, той компоненты, которая отвечает за саму игру. Другая группа отвечает за внешний вид программы — как именно выглядит доска и фигуры, как программа взаимодействует с пользователем. Наша, разработчиков ядра, цель — обеспечить *визуальщикам* (тем, кто отвечает за программирование взаимодействия с пользователем, `GUI` — `Graphical User Interface` все возможности для работы. Например, мы должны быть способны ответить на вопросы:

- корректна ли расставленная пользователем позиция?
- корректен ли сделанный пользователем ход?
- какой ход программа считает наилучшим? И. т. п.

Чтобы ответить на эти вопросы наше ядро должно предоставить визуальщикам набор чего-то, только чего? Функций? Данных? Давайте предоставим им набор *объектов*, каждый из которых будет обладать своими *свойствами* и *методами*. Давайте учтём, что визуальщики правил игры не знают (точнее сказать, они имеют право их не знать), да и квалификация у них, как у программистов, скажем, так, не очень. Не забываем законы Мёрфи — если систему можно использовать неверным образом, именно так она и будет использована. Поэтому нашими целями будут

1. предоставить набор, обладающий всеми необходимыми функциями
2. не разрешать пользоваться этими функциями неправильно

.

Для примера возьмём поле доски. Мы, скорее всего, должны предоставить визуальщикам возможность работы с полями (например, если пользователь взялся за коня, то они могут подсветить доступные этому коню ходы). Мы, ядерщики, проектируем работу с объектами-полями. У нас возникает вопрос: каким образом представлять координаты поля? В виде строки (`"D5"`)? Это будет неудобно при вычислении возможных ходов фигуры — придётся каждый раз преобразовывать строку в координаты и наоборот. В виде пар координат? Это будет неудобно при печати хода и, возможно, неудобно при вычислении ходов. Просто числом, номером поля, чтобы было удобно составлять таблицы перемещений? Это удобно для вычисления, но неудобно ни для печати хода, ни для его визуализации.

Так что же делать?

Выход здесь прост: давайте скроем от пользователя детали реализации кода, но предоставим ему различные способы получения. Что там будет внутри пусть пользователя нашего объекта никак не касается. Мы скрываем представление, *инкапсулируем* его. Для получения информации мы будем использовать соответствующие *методы*, например, представить поле в текстовом виде или представить поле в виде пары координат. Если пользователю объектов не предоставили знания о внутреннем содержимом объекта, то единственный способ работать с этим объектом — использовать *методы*. Если разработчик объекта решит изменить внутреннее представление, не затронув методов работы с ним, пользователь объекта даже знать не будет о произошедших изменениях. Хорошо это? Отлично! Разработчик теперь не привязан к первоначальному проекту и может изменять реализацию, как удобно ЕМУ, а не пользователю, пользователя это не затронет совсем.

Резюмируя — инкапсуляция нужна для сокрытия необязательных деталей реализации объекта. Меньше знаешь — крепче спишь.

2.2 Три кита ООП: полиморфизм

Итак, у нас имеется несколько типов фигур. Вероятно, понадобится метод (да, мы теперь так будем говорить) узнать, какие ходы возможны конкретной фигурой с конкретного поля. Предположим, что этот метод мы захотели назвать `get_available_moves`. В классическом языке программирования это может быть функцией или процедурой. Однако возникает вопрос? Какие аргументы передавать в эту процедуру/функцию? Цвет фигуры, это понятно. А что насчёт фигуры? Если мы передадим туда код фигуры (например, 1 — король, 2 — ферзь и так далее), то мы должны создать функцию, которая разбирается в ходах всех фигур — и короля, и ферзя... Хорошо бы было, если бы можно было передать объект с самой фигурой, но тогда потребуется 6 функций с одним именем и с разными аргументами, но и Си, и Паскаль и тем более Фортран запрещают нам создавать несколько функций с одним именем! Как выход из положения в этих языках к имени функции добавляют тип аргументов (например, `knight_get_available_moves`), но это уже довольно громоздко и немного неуклюже. Ну что же, *полиморфизм*, второй из китов ООП, позволяет нам создавать различные функции с одним и тем же именем.

Вообще-то говоря, в любом из языков программирования имеются зачатки полиморфизма. Например, в чистом, Виртовском Паскале, языке сугубо процедурного программирования:

```
var
  a,b,c: integer;
  x,y,z: real;
begin
  c := a + b;
  z := x + y;
end.
```

разве символ `+` означает одно и то же в обоих исполнимых строках? Ничего подобного. В первой строке он означает операцию целочисленного сложения, во второй — операцию вещественного сложения. Так почему бы не разрешить использовать одни и те же символы (знаки операций или имена функций) для исполнения семантически одинаковых действий? Почему в книге, про язык Паскаль, Никлаус Вирт хвалит язык за то, что он может вводить новые типы данных, например, комплексные числа:

```
type complex = record
     re,im: real;
  end;
var
  a,b,c,d: complex;
```

и тут же забывает добавить, что свободы математической записи типа

$$d = a + b * c$$

язык не предоставляет. Вместо этого требуется следующая уродливая конструкция:

```
procedure add(x,y: complex; var res: complex);
begin
  res.re := x.re + y.re;
  res.im := x.im + y.im;
end;

var
  a,b,c,d: complex;
begin
  add(b,c,t);
  add(a,t,d);
end.
```

Более того, имя `add` теперь нельзя использовать ни для именования любой другой функции, процедуры или переменной!

Забегая вперёд скажем, что во всех ООП языках можно применить математическую форму записи:

$$d = a + b * c,$$

доопределяя операции сложения, умножения и присваивания для любых объектов, в нашем случае комплексного типа.

2.3 Три кита ООП: наследование

Последний кит — *наследование*. В нашей шахматной программе имеется несколько различных видов фигур и было бы неплохо иметь объекты, которые моделируют каждый тип фигуры — например, модель ферзя могла бы определять возможные ходы с нужного поля. Если мы заведём каждой фигуре объект своего типа возникнет вопрос: как же теперь представлять позицию без излишнего усложнения логики программы? На помощь приходит *наследование*. Позиция может представляться, например, массивом *фигур*, причём король, ферзь, слон, конь, ладья и пешка будут именно *фигурами*, они *унаследуют* от *базового объекта* такие свойства, как цвет, местоположение, а часть *методов* для них будет разной по действиям, но одинаковой по имени. Например, если все фигуры будут иметь метод `getAvailablaMoves`, то для того, чтобы получить полный список всех возможных в позиции ходов будет достаточно вызвать этот метод для всех фигур с нужным цветом!

Впрочем, пора уже приступать к изучению конкретных примеров на языке C++.

3 ООП и основы C++

3.1 Классы: базовое ООП понятие C++

Понятие *объект* в C++ реализуется с помощью механизма *классов*. Ключевое слово `class` позволяет это сделать:

```
class field {

};
```

Обратите внимание на точку с запятой за закрывающей фигурной скобкой! Это — единственный случай когда точку запятой за закрывающей фигурной скобкой ставить нужно.

Пока то, что мы написали — пустышка, заготовка, — мы дали классу имя — `field`, но у него нет содержимого.

// a01.cc

```
class field {

};

int main() {
  field f, g;
  return 0;
}
```

Этот файл можно даже скомпилировать:

```
$c++ a01.cc
$./a.out
$
```

Видите, у нас появилось новое ключевое слово `field`, которое обозначает новый тип данных (пока пустой). Конструкция

```
class field {
};
```

в данном случае является *определением definition* класса.

Прежде, чем двигаться дальше, давайте различать понятия *класс* и *экземпляр класса*. Под термином *класс* мы будем понимать тип данных, обладающий определёнными свойствами (и методами!), как слово *собака* не определяет какую-то конкретную собаку, а вид собак как целое — нечто, имеющее такие свойства, как хвостатость, умение лаять и умение приносить тапки и такие методы, как лаяние и принесение тапок. *Экземпляр* класса — конкретный Тузик, хотя собакой и является, лаять умеет, приносить тапки не умеет.

3.2 Классы: поля и функции

Свойства можно реализовать добавлением *полей* в *определение* класса, а методы (это новое) — добавлением функций!

```
// a02.cc
#include <stdio.h>

class Dog {
public:
   bool hasTail;
   bool canBark;
   bool canBringSlippers;
   void bark() {
     if (canBark) {
       printf("BARK!!!\n");
     }
   }
   void bringSlippers() {
     if (canBringSlippers) {
       printf("Here your slippers, master!\n");
     } else {
       printf("RRRRRRRRR!!!!!\n");
     }
   }
};

int main() {
  Dog Tuzik;
  Tuzik.hasTail = true;
  Tuzik.canBark = true;
  Tuzik.canBringSlippers = false;
  Tuzik.bark();
  Tuzik.bringSlippers();
}


$c++ a02.cc
$./a.out
BARK!!!
RRRRRRRRR!!!!!
$
```

Разберём пример по косточкам.

Ключевое слово `public:` (с двоеточием!) пока воспримем, как необходимость. Далее идут три объявления полей класса. С точки зрения ООП они могут быть, например, свойствами объекта (почему они могут не быть ими мы обсудим позже).

Далее, мы объявляем метод класса, под названием `bark`. Обратите внимание на то, что внутри этого метода (а он очень уж похож на функцию) мы используем переменную `canBark`, которая внутри этой функции не объявлена. Однако, она объявлена внутри класса и уже поэтому доступна всем методам класса. Аналогично с объявлением другого метода.

После завершающей фигурной скобки в классе (с неизбежной точкой с запятой) класс `Dog` описан как целое, но никаких собак ещё не существует. В `main` мы создаём первую собаку — Тузика. Следующие три оператора присваивания награждают данного Тузика свойствами хвостатости, умения лаять и неумения приносить тапки. Следующие строки — попытки заставить Тузика лаять (получилось) и принести тапки (облом). Обратите внимание на синтаксис: Тузик есть *экземпляр класса* `Dog` или, другими словами, *объект типа Dog*. `Dog Tuzik;` создаёт новый объект (экземпляр) класса `Dog`.

К сожалению, данный пример не особо точно моделирует объект «собака». Любой пользователь объекта может волюнтаристски использовать свойство `canBringSlippers`.

3.3 Field: Пример простого класса. Инкапсуляция, модификаторы защиты `public` и `private` как проявление инкапсуляции

В реальной жизни собаки не могут изменять явным образом свою способность приносить тапки, для этого их тренируют. Не всякую собаку можно выучить это делать сразу. Предположим, что каждая тренировка увеличивает случайным образом некий уровень натренированности. После того, как очередная тренировка его повысит до необходимого значения, собака научится приносить тапки. Чтобы предотвратить изменение значения этого уровня (и способности приносить тапки), мы поместим их в защищённую секцию, начинающуюся со слова `private:` (опять обратите внимание на двоеточие за словом).

```
// a03.cc
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

class Dog {
private:
   bool canBringSlippers;
   int trainLevel;
public:
   void train() {
      trainLevel += rand() % 10;
      printf("new trainLevel=%d\n", trainLevel);
      if (trainLevel > 25) {
         canBringSlippers = true;
      }
   }

   void bringSlippers() {
     if (canBringSlippers) {
       printf("Here your slippers, master!\n");
     } else {
       printf("RRRRRRRRR!!!!!\n");
     }
   }
};

int main() {
  srand(time(NULL));
  Dog Tuzik;
  for (int i = 0; i < 5; i++) {
     Tuzik.train();
```

```
    Tuzik.bringSlippers();
  }
}

$c++ a03.cc
$./a.out
new trainLevel=32774
Here your slippers, master!
new trainLevel=32783
Here your slippers, master!
new trainLevel=32786
Here your slippers, master!
new trainLevel=32794
Here your slippers, master!
new trainLevel=32794
Here your slippers, master!
$
```

Немного комментариев о том, что мы добавили.

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

означает (в терминах модульного программирования!) включение описания модуля `stdlib`, который содержит нужные нам функции: `rand`, выдающую случайное число в диапазоне в диапазоне `[0..RAND_MAX)` (границу которого нам тоже предоставляет этот модуль — как константу) и `srand`, запускающую процесс образования псевдослучайных чисел с новой затравки, текущего времени в секундах от 1 января 1970 года. Функция `time` предоставляется нам модулем `time`.

Вернёмся к результатам. Что-то не так. Попробуем ещё раз?

```
$./a.out
new trainLevel=32769
Here your slippers, master!
new trainLevel=32773
Here your slippers, master!
new trainLevel=32776
Here your slippers, master!
new trainLevel=32776
Here your slippers, master!
new trainLevel=32781
Here your slippers, master!
$
```

Что-то действительно неверно. Мы долго говорили о том, что мы вводим инкапсуляцию в том числе и для того, чтобы поддерживать объекты в *консистентном*, непротиворечивом состоянии. Что же мы видим в нашем случае? Переменная `trainLevel`, которая отвечает за уровень натренированности и которая должна бы быть равной нулю при порождении объекта `Dog` равна чему угодно, только не нулю. Написать в `main`

```
  Dog Tuzik;
  Tuzik.trainLevel = 0;
```

уже не получится (попробуйте это сделать) — всё, что находится вне области деятельности класса, не имеет доступа к переменным и функциям, помеченным ключевым словом `private:`.

Это не так страшно, как кажется и означает лишь то, что для этого и подавляющего большинства других классов требуется метод, который создаёт объекты сразу в непротиворечивом состоянии. Наша попытка

```
Dog Tuzik;
Tuzik.trainLevel = 0;
```

(в предположении, что `trainLevel` нам доступен из `main`) и была попыткой создания объекта класса `Dog` под именем `Tuzik` в непротиворечивом состоянии, однако она разбивалась на две фазы:

1. создания объекта в неизвестном состоянии;
2. приведения состояния объекта в консистентное.

Это две различные операции и состояние объекта между ними остаётся неопределённым. Использование его чревато ошибками, часто непонятными и во всяком случае непредсказуемыми, объект напоминает необезвреженную мину. Про вторую фазу мы можем вообще забыть навсегда, кстати. Для того, чтобы в жизни объекта не было такого неопределённого интервала, необходимо обе фазы свести в одну, или говоря более формальным языком, совершить пару операций *атомарно*. Эту проблему, как и проблему сокрытия доступа к внутренним переменным, решает понятие *конструктор* класса.

3.4 Конструкторы

Поместим строки

```
Dog() {
   trainLevel = 0;
   canBringSlippers = false;
}
```

где-нибудь в описании класса `Dog` в секции `public`.

Синтаксически конструктор класса выглядит как метод, однако имеются две особенности:

1. имя метода совпадает с именем класса, в нашем случае имя метода `Dog` и имя класса `Dog`;
2. отсутствует спецификация типа возвращаемого значения, обязательного для других методов. Например, метод `train`, который не возвращает никаких значений, имеет тип `void` — пустой. Попытка убрать слово `void` приведёт к ошибкам компиляции, синтаксически тип возвращаемого значения обязательно требуется перед любым методом (и любой функцией языка Си, кстати говоря).

```
// a03a.cc
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

class Dog {
private:
   bool canBringSlippers;
   int  trainLevel;
public:
   Dog() {
      trainLevel = 0;
      canBringSlippers = false;
   }
   void train() {
      trainLevel += rand() % 10;
      printf("new trainLevel=%d\n", trainLevel);
      if (trainLevel > 25) {
         canBringSlippers = true;
      }
```

```
   }

   void bringSlippers() {
     if (canBringSlippers) {
       printf("Here your slippers, master!\n");
     } else {
       printf("RRRRRRRRR!!!!!\n");
     }
   }
};

int main() {
  srand(time(NULL));
  Dog Tuzik;
  for (int i = 0; i < 5; i++) {
     Tuzik.train();
     Tuzik.bringSlippers();
   }
}

$ ./a.out
new trainLevel=1
RRRRRRRRR!!!!!
new trainLevel=3
RRRRRRRRR!!!!!
new trainLevel=12
RRRRRRRRR!!!!!
new trainLevel=21
RRRRRRRRR!!!!!
new trainLevel=23
RRRRRRRRR!!!!!
$ ./a.out
new trainLevel=8
RRRRRRRRR!!!!!
new trainLevel=9
RRRRRRRRR!!!!!
new trainLevel=14
RRRRRRRRR!!!!!
new trainLevel=21
RRRRRRRRR!!!!!
new trainLevel=26
Here your slippers, master!
$
```

Ну вот, и овцы сыты (мы спрятали переменную `trainLevel` в приватные переменные), и волки целы (она принимает правильное значение при создании объекта класса).

Ну а то, что различные запуски программы приводят к различным результатам, заслуга функций `rand` и `srand`. К тому же это нам показывает, что различные запуски программы приводят у созданию различных *экземпляров* объектов класса.

3.5 Классы как целое; экземпляры объектов классов

Описав класс `Dog` мы создали модель поведения однотипных объектов, собак. Каждое *объявление*

```
  Dog Tuzik;
```

приводит к созданию *экземпляра* объекта класса `Dog`. У этого экземпляра имеется имя

переменной, Tuzik, и с помощью этого имени мы можем управлять этим экземпляром — тренировать его, приказывать принести тапки.

3.6 Деструкторы

Итак, конструктор создаёт объект, по возможности, в консистентном состоянии. А что происходит при уничтожении объекта? Ясно, что при уничтожении объекта что-то происходит. Ведь при создании объекта кто-то должен выделять память для его переменных, значит, при его уничтожении, эта память должна освобождаться. Можно ли контролировать этот процесс? Конечно, можно.

Если мы не создаём ни одного собственного конструктора, память под переменные, тем не менее, выделяется. Компилятор в любом случае создаёт конструктор, так называемый *конструктор по умолчанию*, который кроме выделения памяти под подобъекты ничего не делает. Если мы хотим присвоить подобъектам какие-либо конкретные значения, мы должны создать свой собственный конструктор. Конструктор, созданный нами, пользуется фактом, что память под подобъекты выделяется автоматически и просто присваивает им нужные значения (для поддержания объекта в состоянии консистентности).

При уничтожении любого объекта вызывается специальный метод, называемый *деструктором*. Если мы не создаём свой деструктор, система создаст *деструктор по умолчанию*, который просто освободит память, занятую нашими подобъектами.

В программе деструктор записывается своеобразно, он по синтаксису напоминает конструктор, только перед именем класса располагается знак *тильды* ~:

```
  ~Dog() {
    printf("Destructor ~Dog()\n");
  }
```

Давайте посмотрим, в какой момент времени вызывается деструктор:

```
// a03a.cc
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

class Dog {
private:
   bool canBringSlippers;
   int  trainLevel;
public:
   Dog() {
      printf("constructor Dog()\n");
      trainLevel = 0;
      canBringSlippers = false;
   }
   ~Dog() {
      printf("destructor ~Dog()\n");
   }
   void train() {
      trainLevel += rand() % 10;
      printf("new trainLevel=%d\n", trainLevel);
      if (trainLevel > 25) {
         canBringSlippers = true;
      }
   }
   void bringSlippers() {
     if (canBringSlippers) {
       printf("Here your slippers, master!\n");
     } else {
```

```
      printf("RRRRRRRRR!!!!!\n");
    }
  }
};

int main() {
  srand(time(NULL));
  Dog Tuzik;
  for (int i = 0; i < 5; i++) {
     Tuzik.train();
     Tuzik.bringSlippers();
   }
   printf("Bye\n");
}
```

```
$ ./a.out
constructor Dog()
new trainLevel=4
RRRRRRRRR!!!!!
new trainLevel=11
RRRRRRRRR!!!!!
new trainLevel=19
RRRRRRRRR!!!!!
new trainLevel=23
RRRRRRRRR!!!!!
new trainLevel=25
RRRRRRRRR!!!!!
Bye
destructor ~Dog()
$
```

Интересно, но деструктор вызвали *после* того, как программа, по нашему мнению, завершилась, то есть после прощания `Bye`! О том, когда именно вызываются конструкторы и деструкторы следующий раздел.

3.7 Управление размещением объектов в стеке.

Не трогая класс `Dog`, попробуем изменить функцию `main()`.

```
int main() {
  srand(time(NULL));
  {
     Dog Tuzik;
     for (int i = 0; i < 5; i++) {
        Tuzik.train();
        Tuzik.bringSlippers();
      }
   }
   printf("Bye\n");
}
```

Поместим код, содержащий работу с объектом, в фигурные скобки — *блок*. Вот что получается после запуска программы:

```
$ ./a.out
constructor Dog()
new trainLevel=3
```

```
RRRRRRRRR!!!!!
new trainLevel=8
RRRRRRRRR!!!!!
new trainLevel=15
RRRRRRRRR!!!!!
new trainLevel=21
RRRRRRRRR!!!!!
new trainLevel=23
RRRRRRRRR!!!!!
destructor ~Dog()
Bye
$
```

Мы видим, что деструктор на этот раз был вызван *до* строки `Bye`. Это наталкивает нас на верную мысль:

> **Все созданные в блоке объекты уничтожаются при выходе из блока, то есть, по достижению закрывающей блок фигурной скобки**.

Все объекты, которые мы создавали до этого, создавались в стеке, или, как говорят, *на стеке*. Напомним, что стек — структура данных, которая работает по принципу — первый пришёл, последний ушёл. Собственно говоря, так и происходит. Для того, чтобы подробнее исследовать это, нам нужно познакомиться с ещё одним видом конструкторов — *конструкторами с параметрами*.

Положим, что каждой собаке мы можем давать какой-то номер, целое число, которое не будет изменяться во время жизни объекта. Это число мы будем распечатывать при каждой операции с собакой, чтобы мы смогли различать объекты, над которыми производятся операции.

```
// a03d.cc
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

class Dog {
private:
   bool canBringSlippers;
   int  trainLevel;
   int  number;
public:
   Dog(int num) {
      printf("constructor Dog(%d)\n", num);
      number = num;
      trainLevel = 0;
      canBringSlippers = false;
   }
   ~Dog() {
      printf("destructor ~Dog(%d)\n", number);
   }
   void train() {
      trainLevel += rand() % 10;
      printf("%d: new trainLevel=%d\n", number, trainLevel);
      if (trainLevel > 25) {
         canBringSlippers = true;
      }
   }

   void bringSlippers() {
```

```
    if (canBringSlippers) {
      printf("%d: Here your slippers, master!\n", number);
    } else {
      printf("%d: RRRRRRRRR!!!!!\n", number);
    }
  }
};

int main() {
  srand(time(NULL));
  Dog Tuzik(1);
  for (int i = 0; i < 5; i++) {
     Tuzik.train();
     Tuzik.bringSlippers();
   }
   printf("Bye\n");
}


$./a.out
constructor Dog(1)
1: new trainLevel=0
1: RRRRRRRRR!!!!!
1: new trainLevel=0
1: RRRRRRRRR!!!!!
1: new trainLevel=7
1: RRRRRRRRR!!!!!
1: new trainLevel=10
1: RRRRRRRRR!!!!!
1: new trainLevel=19
1: RRRRRRRRR!!!!!
Bye
destructor ~Dog(1)
$
```

Теперь в `main` заведём ещё одну собаку, но не будем с ней производить никаких операций:

```
int main() {
  srand(time(NULL));
  Dog Tuzik(1);
  Dog Sharik(2);
  for (int i = 0; i < 5; i++) {
     Tuzik.train();
     Tuzik.bringSlippers();
   }
   printf("Bye\n");
}

$./a.out
constructor Dog(1)
constructor Dog(2)
1: new trainLevel=0
1: RRRRRRRRR!!!!!
1: new trainLevel=0
1: RRRRRRRRR!!!!!
1: new trainLevel=7
1: RRRRRRRRR!!!!!
1: new trainLevel=10
```

```
1: RRRRRRRRR!!!!!
1: new trainLevel=19
1: RRRRRRRRR!!!!!
Bye
destructor ~Dog(2)
destructor ~Dog(1)
```

Заметили, что деструкторы вызываются в обратном порядке? Вот это и есть стековый принцип размещения объектов — первым создался, последним уничтожился.

Конструкторов у класса может быть любое количество. Если мы не определили ни одного конструктора, то компилятор создаёт свой собственный минимальный конструктор *по умолчанию*, достаточный для того, чтобы выделить память под размещение подобъектов. Если мы определили *хотя бы один* из конструкторов, то системного конструктора по умолчанию не создаётся.

Деструктор у класса может быть только один. Если мы его не определили, то компилятор создаст минимальный деструктор, который освобождает память, занятую подобъектами, а если тип подобъекта имеет собственные деструкторы, то они вызываются и так рекурсивно.

Когда мы должны создавать свой деструктор? Собственный деструктор обязателен, если в объекте присутствуют подобъекты, полученные заказом памяти по `new` или средствами Си (`calloc`, `malloc` и т. д.). Деструктор в этом случае должен содержать код, возвращающий ресурсы в систему (`delete` или `free`). Общее правило такое:

Если объект содержит ресурсы, заказанные у системы, то деструктор объекта должен возвращать эти ресурсы в систему.

Если в методах объекта, например, в конструкторе, открываются файлы, то они должны где-то закрываться, возможно, в деструкторе.

3.8 Операторы `new` и `delete`

Другой способ создать новый объект — использовать оператор `new`. Попробуем заменить создание Тузика и Шарика в стеке на создание их же с использованием оператора `new`.

Перед этим вспомним, что такое указатель. Указатель на объект — переменная, которая содержит адрес какого-то объекта. Тип указателя определяется типом объекта, на который тот указывает. Например:

```
Dog Tuzik;
Tuzik.train();
Dog *Sharik; // Мы просто выделили память, достаточную для хранения адреса.
      // Пока она указывает в случайное место, находящееся неизвестно где.
```

Присвоив указателю адрес существующего объекта типа `Dog`

```
Sharik = &Tuzik;
```

мы получим следующую схему:

Если мы добавим ещё Жучку и присвоим её адрес Шарику, то получим следующее:

```
Dog Tuzik(1);
Tuzik.train();
Dog *Sharik;
Sharik = &Tuzik;
Dog Zhuchka(2);
Zhuchka.train();
Sharik = &Zhuchka;
```

Использовать методы и свойства для переменной, представленной указателем просто: достаточно вместо точки, разделяющей имя объекта и его метод или свойства применять стрелку `->`, составленную из знаков «минус» и «больше».

Итак, программа с хранением объектов в куче:

Рис. 3: Объекты и указатели 1

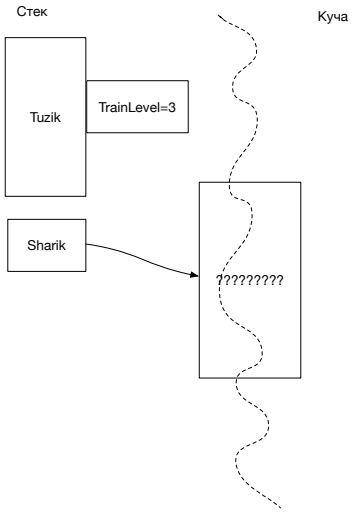
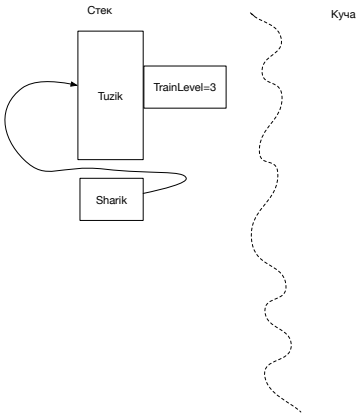


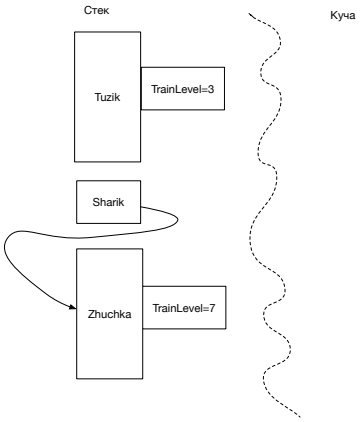
Рис. 4: Объекты и указатели 2




```
int main() {
  srand(time(NULL));
  Dog *Tuzik = new Dog(1);
  Dog *Sharik = new Dog(2);
  for (int i = 0; i < 5; i++) {
     Tuzik->train();
     Tuzik->bringSlippers();
   }
   printf("Bye\n");
}

$./a.out
constructor Dog(1)
constructor Dog(2)
1: new trainLevel=0
1: RRRRRRRRR!!!!!
1: new trainLevel=0
1: RRRRRRRRR!!!!!
1: new trainLevel=7
1: RRRRRRRRR!!!!!
1: new trainLevel=10
```

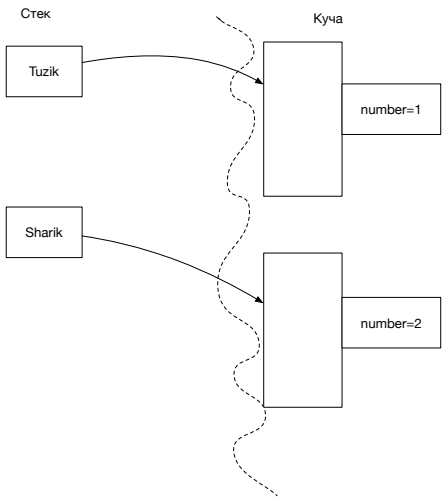
Рис. 5: Объекты и указатели 3




```
1: RRRRRRRRR!!!!!
1: new trainLevel=19
1: RRRRRRRRR!!!!!
Bye
```

Объекты теперь создаются не в стеке, а в *куче* (`heap`).

Рис. 6: Объекты и указатели 4



Соответственно, доступ к ним теперь возможен только через указатели — увы, без них в этом случае никуда. Оператор `new Dog(1)` создаёт объект типа `Dog` в куче и возвращает указатель на него, который мы можем использовать почти так же, как и обычный объект. В глаза бросаются две разницы:

1. доступ к методам и свойствам, который для обычных объектов осуществлялся через точку (`.`) теперь осуществляется через стрелку (`->`);

2. куда-то исчез вызов деструктора! Сообщений о том, что вызывается деструктор теперь нет! Означает ли это, что собаки теперь живут вечно? Нет, конечно. Операционная система уничтожает всю память процесса, в которой находились эти объекты, заодно уничтожает и собак.

Удалить созданный по `new` объект можно оператором `delete`:

```
delete Tuzik;
```

```
  delete Sharik;
  printf("Bye\n");
```

Заметим, что если мы вызываем оператор `new` в цикле, то мы можем с лёгкостью исчерпать адресное пространство процесса собаками-зомби.

Задание: попробуйте запустить две разных программы:

```
int main() {
  for (int i = 0; i < 100; i++) {
    Dog d(i);
    d.train();
  }
  printf("Bye\n");
}
```

и

```
int main() {
  for (int i = 0; i < 100; i++) {
    Dog *d = new Dog(i);
    d.train();
  }
  printf("Bye\n");
}
```

и попробуйте ответить себе на вопрос: куда деваются собаки во второй программе.

Ответ на вопрос: собаки никуда не деваются. На время работы программы они превращаются в *зомби* — объекты, которые существуют в памяти процесса, но к которым нельзя получить доступ. Другое название — *мусор*. Языки программирования `Java` и `C#`, в которых тоже имеются оператор `new`, тоже создают объекты в *куче*, но в них **не имеется** явного оператора `delete`. Что же там происходит? Каждому процессу в этом языке выделяется определённое количество памяти под *кучу*. Каждый вызов оператора `new` выделяет из *кучи* память под объект. В какой-то момент времени памяти в *куче* не хватает на очередной объект и исполняющая система языка начинает *сборку мусора*.

Для того, чтобы *собрать мусор* требуется знать, используется ли данный объект где-либо, поэтому в каждом объектом (или рядом с каждым объектом) обязательно присутствует *счётчик использования* (там могут присутствовать и более сложные структуры данных, но концептуально мы можем считать, что достаточно одного счётчика).

Сборщик мусора пробегает по всем объектам, полученным по оператору `new` и возвращает в *кучу* те объекты, к которым невозможен доступ, например, счётчик использования у которых стал равен нулю.

Это обязательно займёт какое-то время, может быть, значительное.

Если обычный вызов оператора `new`, скажем, требует 100 наносекунд, то какой-то из очередных вызовов оператора `new` может потребовать, скажем, 100 миллисекунд.

Для систем, в которых требуется строгий отклик на событие это часто неприемлемо.

Возможен другой вариант сборщика мусора, когда отдельный, невидимый пользователю, вычислительный поток сборки мусора обнаруживает неиспользуемые объекты и возвращает их в *кучу*. Тогда возникают те самые проблемы синхронизации при одновременном доступе к одним и тем же переменным, о котором мы говорили в третьем семестре и которые программу также не ускоряют. Итак, плюсы и минусы подхода к объектному программированию с применением *сборки мусора*:

1. Плюс: не требуется заботиться о явном удалении объекта. Объекты, которые мы не можем использовать когда-либо будут удалены исполнительной системой. Утечка памяти (*memory leak*)- одна из самых главных, трудно обнаруживаемых проблем при программировании на `C++`. Для того, чтобы писать программы на `C++`, избавленные от утечки памяти, требуется определённая квалификация.

2. Минус: время, затрачиваемое на сборку мусора отнюдь не нулевое, возможны либо непредсказуемые задержки при очередном вызове оператора `new` либо общее замедление исполнения программы, в зависимости от стратегии сборки мусора.

3. Минус: сборщик мусора включается при каком-то пороге использования памяти. Это означает, что в памяти присутствуют как активные, используемые объекты, так и неактивные и они в памяти перемешаны. Это уменьшает *локальность* использования памяти, уменьшает процент попадания в процессорный кэш, увеличивает количество частично заполненных страниц виртуальной памяти.

4. Минус: для сборки мусора исполнительная системы должна иметь список всех объектов (это — добавочная память) и каждый из объектов должен иметь индикаторы используемости (это тоже добавочная память).

3.9 Передача объектов в функции

Предположим, что мы хотим написать функцию, которая тренирует нашу собаку ровно 50 раз. Назовём её, например, train50. Аргументом функции должен быть объект собака. После окончания работы функции собака должна быть более-менее натренирована. Попробуем наивный способ передачи аргумента в функцию:

```
// a03e.cc
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

class Dog {
private:
   bool canBringSlippers;
   int  trainLevel;
   int  number;
public:
   Dog(int num) {
      printf("constructor Dog(%d)\n", num);
      number = num;
      trainLevel = 0;
      canBringSlippers = false;
   }
   ~Dog() {
      printf("destructor ~Dog(%d)\n", number);
   }
   int getTrainLevel() {
      return trainLevel;
   }
   void train() {
      trainLevel += rand() % 10;
      printf("%d: new trainLevel=%d\n", number, trainLevel);
      if (trainLevel > 25) {
         canBringSlippers = true;
      }
   }

   void bringSlippers() {
     if (canBringSlippers) {
       printf("%d: Here your slippers, master!\n", number);
     } else {
       printf("%d: RRRRRRRRR!!!!!\n", number);
     }
   }
};

void train50(Dog dog) {
for (int i = 0; i < 50; i++) {
dog.train();
}
printf("Done! Dog has been trained and reached level %d\n", dog.getTrainLevel());
}

int main() {
  srand(time(NULL));
  Dog Tuzik(1);
  train50(Tuzik);
```

```
  Tuzik.bringSlippers();
  printf("Bye\n");
}

$ ./a.out
constructor Dog(1)
1: new trainLevel=1
...
1: new trainLevel=216
Done! Dog has been trained and reached level 216
destructor ~Dog(1)
1: RRRRRRRRR!!!!!
Bye
destructor ~Dog(1)
$
```

Странно, Тузика мы тренировали, он достиг уровня `216`, но тапки приносить не стал! Проверим в `main`, какого уровня он достиг:

```
...
int main() {
  srand(time(NULL));
  Dog Tuzik(1);
  train50(Tuzik);
  Tuzik.bringSlippers();
  printf("Bye\n");
}

$ ./a.out
constructor Dog(1)
1: new trainLevel=1
...
1: new trainLevel=216
Done! Dog has been trained and reached level 216
destructor ~Dog(1)
Tuzik has reached level 0
1: RRRRRRRRR!!!!!
Bye
destructor ~Dog(1)
$
```

Эй, мы вызвали функцию, но Тузика не обучили? Почему?

Посмотрим на вывод программы поподробнее. Не обратили ли вы внимание на то, что было вызвано **два** деструктора собаки под номером 1? Что же произошло? А произошло вот что: в функцию `train50` мы передавали объект `Tuzik`, а передали *копию* этого объекта. В функции был создан новый объект типа `Dog`. Где? В стеке, конечно. Этот объект мы благополучно тренировали, но пришло время покинуть функцию и этот объект исчез (деструктор перед `Done!`). Наш же Тузик остался нетренированным...

Итак, важно:

В функции всегда передаются только значения аргументов, их копии.

Вот когда нам потребуются указатели! Если в функцию передать *указатель* на объект класса `Dog`, то мы его сможем изменять в функции:

```
void train50(Dog *dog) {
  for (int i = 0; i < 50; i++) {
    dog->train();
```

```
  }
  printf("Done! Dog has been trained and reached level %d\n", dog->getTrainLevel());
}

int main() {
  srand(time(NULL));
  Dog Tuzik(1);
  train50(&Tuzik);
  Tuzik.bringSlippers();
  printf("Bye\n");
}

$./a.out
constructor Dog(1)
1: new trainLevel=1
...
1: new trainLevel=216
Done! Dog has been trained and reached level 222
Tuzik has reached level 222
1: Here your slippers, master!
Bye
destructor ~Dog(1)
$
```

Отлично! Мы своего добились! Правда, произошло это за счёт усложнения кода: при вызове функции нам пришлось вызвать операцию получения адреса объекта `&`, а внутри функции пришлось заменить все точки на стрелки. Второй положительный момент, важность которого нам пока трудно оценить — внутри функции не происходило создания новых объектов и при выходе из функции они не уничтожались. Создание и уничтожение объектов требует драгоценного времени и мы его во втором варианте не потратили. Одним из недостатков ООП является уменьшение скорости работы программ при неверном использовании средств ООП. Мы будем стараться здесь и в дальнейшем применять такие средства и таким образом, чтобы это минимально влияло на эффективность программ.

3.10 Переменная `this`

Когда мы вызываем метод объекта, в него невидимо для нам передаётся переменная `this`, указатель на текущий объект. Слово `this` является ключевым, нельзя создавать переменные с таким именем и этим именем называть функции. Если мы пишем функцию `train` класса `Dog`и в ней имеется поле `trainLevel`, то мы уже привыкли к тому, что если мы напишем

```
trainLevel += rand() % 10;
```

то обращение произойдёт к полю, как мы говорили, *текущего объекта*. Понятие *текущий объект* можно немного конкретизировать: это нечто, что имеет адрес, по которому содержатся поля объекта. Так вот этот адрес и имеет имя `this`. Мы его можем передавать в функции, которые требуют указатели на объекты класса. Конечно, работа с адресами требует крайней осторожности (вам ещё не надоело слушать про осторожность, как только речь заходит об адресах и указателях? Не должно, так как это та область языка, в которой делается наибольшее количество ошибок и мы должны их избегать).

Обратиться к полю класса `trainLevel` можно и так: `this->trainLevel++` Зачем так сделано? Одна из причин — нужно допустить доступ к полю класса даже в том случае, если имя поля срыто другой декларацией. Если мы захотим реализовать отдельную функцию инициализации уровня тренировки в классе `Dog void initTrainLevel(int trainLevel)`, то имя аргумента *скроет* имя поля класса. Если мы напишем

```
class Dog {
...
```

```
  void initTrainLevel(int trainLevel) {
    trainLevel = 5;
  }
}
```

то мы присвоим значение 5 не полю класса, а аргументу, который, к тому же, является копией переданного значения.

Следующий код показывает, как всё же можно использовать скрытый аргументом идентификатор:

```
class Dog {
...
  void initTrainLevel(int trainLevel) {
    this->trainLevel = trainLevel;
  }
}
```

Другие причины использования идентификатора `this` мы ещё увидим в дальнейшем.

3.11 Ссылки. Что это такое и зачем они появились в языке

Плата за передачу аргументов в функции по указателям вроде бы как невелика (ну что там, заменить точки на стрелки внутри функции, тоже мне работа ...), но иногда бывает удобно полностью скрыть даже такой, вроде бы незначительный факт. Для этого мы можем использовать *ссылки* (*references*).

Ссылка в чём-то похожа на указатель. Если указатель мы объявляем, указывая звёздочку перед именем переменной, то ссылку — указывая амперсенд (`&`).

```
Dog Tuzik(3);
Dog Sharik(4);
Dog *pdog; // Указатель можно создавать без инициализации
pdog = &Tuzik; // Указатель содержит адрес Tuzik
pdog->train(); // Тренируем Тузика
pdog = &Sharik; // Указатель содержит теперь адрес Sharik
pdog->train(); // Тренируем Шарика
Dog &rdog = Tuzik; // Ссылку можно создавать только с инициализацией.
rdog.train(); // Тренируем Тузика
rdog = Sharik; // Тузик становится Шариком!!!
```

Основные отличия ссылки от указателя:

- Указатель может содержать любой адрес, в том числе невозможный, в том числе и объекта другого типа. Ссылка после объявления всегда ссылается на один и тот же объект.
- Указатель можно не инициализировать (хотя лучше так не делать). Создать ссылку и не инициализировать её невозможно.
- Указателю нужно присваивать либо адрес объекта либо другой указатель на тот же тип. Присваивание по ссылке присваивает новое значение ссылочному объекту.
- Для получения значения переменной по указателю надо использовать звёздочку перед указателем. Ссылка смотрится как обычный объект.

Проще всего считать, что ссылка создаёт постоянный псевдоним на **уже существующий** объект.

Объявление ссылки не создаёт новых объектов.

Для чего же создали столь странный механизм — ссылки? Для нескольких целей, в том числе и для нашего удобства. Основная цель — аккуратная передача аргументов в функции и методы. Попробуем переписать нашу программу с использованием ссылок:

```
void train50(Dog &dog) {
for (int i = 0; i < 50; i++) {
dog.train();
}
printf("Done! Dog has been trained and reached level %d\n", dog.getTrainLevel());
}

int main() {
  srand(time(NULL));
  Dog Tuzik(1);
  train50(Tuzik);
  Tuzik.bringSlippers();
  printf("Bye\n");
}
```

Самое главное изменение, которое мы видим в пользовательской части (функции `main`) — исчезновение операции адреса при вызове функции train50. Как мы увидим в дальнейшем, это не просто косметическое изменение, это — часть решения, позволяющего прозрачно использовать средства ООП.

3.12 Модификатор `const`

Иногда нам может потребоваться, что созданный объект нельзя изменять. Это означает, что мы больше не сможем пользоваться методами и свойствами, которые изменяют состояние объекта. Например, метод `train` изменяет состояние объекта, потому, он присваивает новое значение переменной объекта `trainLevel`, метод `bringSlippers` объекта не изменяет.

Объявить постоянный, *константный* объект можно, добавив в его объявление ключевое слово `const`. Например:

```
  const Dog Sharik;
```

Понятно, что мы его обучить (`train`) уже не сможем. Попытка вызвать метод `train` для объекта `Sharik` будет обнаружена и заблокирована компилятором:

```
  Sharik.train();
  ...

$c++ a03b.cc
a03b.cc:36:16: error: passing ‘const Dog’ as ‘this’ argument of ‘void Dog::train()’ discards
   Sharik.train();
```

Но можем ли мы сделать попытку попросить Шарика принести тапки?

```
  Sharik.bringSlippers();
```

Увы, опять этот злобный компилятор не разрешает нам это сделать, выводя ту же самую диагностику. Он, видите ли, не понимает, что мы отнюдь не собираемся менять объект, когда вызываем метод `bringSlippers()`. Мы можем ему в этом помочь, поставив *квалификатор* `const` **после** описания метода, непосредственно перед открывающей фигурной скобкой:

```
   void bringSlippers() const {
     if (canBringSlippers) {
       printf("Here your slippers, master!\n");
     } else {
       printf("RRRRRRRRR!!!!!\n");
     }
   }
```

Обратите внимание, что смысл квалификатора `const` меняется, если его помещать в разные места объявления:

```
class bar {
...
  const int foo1() {
    ..
  }

  int foo2() const {
    ..
  }
};
```

Метод `foo1` возвращает константу целого типа и он может изменять объект класса `bar`, для которого он вызван.

Метод `foo2` возвращает переменную целого типа и он не может изменять объект, для которого вызван.

Второе важное применение этот модификатор имеет при передаче аргументов. Мы знаем, что передача аргумента по ссылке в функцию или метод позволяет сделать программу более эффективной — не вызываются конструкторы и деструкторы для объектов. Однако, такая передача может быть чревата — чем? Тем, что вызванная функция может совершать с объектом различные действия, например, изменять его. Если мы вызываем функцию именно для того, чтобы изменить объект — это нормально и предсказуемо. Однако странно было бы, если, скажем, функция, выводящая информацию об объекте на экран, захотела бы его изменить. С точки зрения сложности программ, передавая аргумент по ссылке в функцию, мы формируем двухстороннюю связь, усложняя программу, хотя для наших целей было бы достаточно односторонней связи. Здесь опять на помощь приходит квалификатор `const`.

```
void printDogInfo(Dog const &dog) {
  printf("Dog %d has train level %d\n", dog.getNumber(), dog.getTrainLevel());
}
```

Теперь мы видим, что функция `printDogInfo` не изменит посланный в неё объект. Конечно, методы `getNumber` и `getTrainLevel` тоже должны быть объявлены константными в классе.

3.13 Конструктор копирования

Запустив программу `a03e` (ту, в которой мы в функцию `train50` передавали объект `Dog` целиком и потом удивлялись, почему собака не тренируется), мы обнаружили, что вызывается *два* деструктора класса `Dog`, хотя конструктор, вроде бы, один. На самом деле, конечно, и для аргумента функции вызывается конструктор. Этот конструктор должен создать объект класса `Dog` по образу и подобию другого. Такой конструктор называется *конструктором копирования* и он автоматически создаётся компилятором, если мы его не создали сами. Сам компилятор, ничего на зная о семантике объекта, просто выделяет память под подобъекты и вызывает для них рекурсивно конструкторы копирования. Для подобъектов элементарных типов, таких, как `int`, `double` и прочих подобных типов, копируется содержание подобъектов. Например, для класса

```
  class Kennel {
private:
     Dog habitant;
     double volume;
     ...
};

int main() {
    Kennel original; ...
    Kennel  copy(original);
```

конструктор копирования по умолчанию сделает следующее:

1. Выделит память в `copy`, достаточную для хранения всех подобъектов.
2. Для подобъекта `copy.habitant` вызовет конструктор копирования из такого же подобъекта оригинала.
3. Скопирует значения: `copy.volume = original.volume`

Ровно те же действия производятся при передаче объекта целиком, не по ссылке и не по указателю в функцию — объект аргумент функции создаётся конструктором копирования.

Мы пока ничего не говорили про тип аргумента конструктора копирования, но его можно угадать.

- Аргумент должен передаваться быстро — поэтому он должен быть ссылкой.
- Аргумент нельзя изменять — он является просто образцом, калькой, для создания нового объекта.

Итак, допишем к многострадальному классу `Dog` последний штрих — конструктор копирования.

```
  Dog(Dog const &oth) {
   printf("constructor copy Dog(%d)\n", oth.number);
   number = oth.number;
   trainLevel = oth.trainLevel;
   canBringSlippers = oth.canBringSlippers;
  }
```

Добавив этот конструктор в программу `a03e.cc`, увидим, что появился вывод из недостающего конструктора.

Наш конструктор, по сути, не лучше того, что создал компилятор. Единственное отличие в том, что мы добавили диагностический вывод из конструктора.

Когда же возникает настоятельное **требование** создания конструктора копирования? В случаях, когда объект содержит динамические ресурсы, то есть, ресурсы, располагаемые в куче. Вот небольшой пример:

```
// a04a.cc
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

class mystr {
private:
   char *ptr;
public:
   mystr(const char *arg = NULL) {
      ptr = strdup(arg == NULL ? "" : arg);
   }
   ~mystr() {
      free(ptr);
   }
   const char * str() const {
      return ptr;
   }
};

int main() {
   mystr s1("foo");
   printf("s1='%s'\n", s1.str());
   {
     mystr s2(s1);
```

```
      printf("s2='%s'\n", s2.str());
   }
   printf("s1='%s'\n", s1.str());
}
```

Небольшие комментарии: `string.h` содержит интерфейс модуля языка Си для работы со строками, который в Си представляются массивом символов, завершающимся символом с нулевым кодом. Функция `strdup` создаёт копию своего аргумента — она заказывает память в куче и копирует туда аргумент, возвращая указатель на заказанную память.

Не запуская программу скажите, всё ли в порядке?

Хорошо, компилируем и запускаем.

```
$ ./a.out
s1='foo'
s2='foo'
s1='foo'
a.out(32393,0x7fff73013300) malloc: *** error for object 0x100304a90: pointer being freed wa
*** set a breakpoint in malloc_error_break to debug
Abort trap: 6
$
```

На разных операционных системах текст сообщения будет разным но итог почти всегда печальный — программа аварийно завершилась. Почему? Судя по диагностике, она пыталась освободить память, которая уже была освобождена. Кем? Деструктором, очевидно. Почему? Потому, что конструктор копирования по умолчанию, сделал копию указателя `ptr`. Поэтому `free` будет дважды вызван с одним и тем же аргументом, что совсем не есть хорошо. Если вы думаете, что замена `free` на `delete` хоть как-то поможет, то вы сильно заблуждаетесь.

Программу можно исправить, создав конструктор копирования.

```
class mystr {
private:
   char *ptr;
public:
   mystr(const char *arg = NULL) {
      ptr = strdup(arg == NULL ? "" : arg);
   }
   ~mystr() {
      free(ptr);
   }
   mystr(mystr const &oth) {
      ptr = strdup(oth.ptr);
   }
   const char * str() const {
      return ptr;
   }
};
```

```
$ ./a.out
s1='foo'
s2='foo'
s1='foo'
$
```

Всё отлично работает. Давайте экспериментировать дальше. Чуть-чуть изменим `main`.

```
int main() {
   mystr s1("foo");
```

```
   printf("s1='%s'\n", s1.str());
   {
     mystr s2 = s1;
     printf("s2='%s'\n", s2.str());
   }
   printf("s1='%s'\n", s1.str());
}

$ ./a.out
s1='foo'
s2='foo'
s1='foo'
$
```

Мы создали новый объект и прямо в объявлении переменной объекта присвоили ему начальное значение — инициализировали объект. Ничего не изменилось.

При инициализации объекта в объявлении вызывается конструктор копирования соответствующего типа.

```
int main() {
   mystr s1("foo");
   printf("s1='%s'\n", s1.str());
   {
     mystr s2;
     s2 = s1;
     printf("s2='%s'\n", s2.str());
   }
   printf("s1='%s'\n", s1.str());
}

$ ./a.out
s1='foo'
s2='foo'
s1='foo'
a.out(89537,0x7fff73013300) malloc: *** error for object 0x7fe608c04a90: pointer being freed
*** set a breakpoint in malloc_error_break to debug
Abort trap: 6
$
```

Ошибка вернулась! Мы просто переместили присваивание одного объекта другому на строчку ниже. В первоначальном варианте создавался новый объект и сразу инициализировался образцом, в новом варианте создаётся объект с пустым конструктором (по умолчанию) и следующим шагом этому объекту присваивается новый. Последовательность действий здесь другая и результат другой. Мы подозреваем, что при копировании объекта происходит примерно то же самое, что и при создаваемом компилятором конструкторе копирования. На самом деле так и есть. Компилятор при копировании объектов копирует все подобъекты, указатель `ptr` объектом класса не является, поэтому он копируется побитово. К счастью, можно этого избежать, используя второго кита ООП — *полиморфизм*.

3.14 Полиморфизм: `operator =`

Как мы уже говорили, полиморфизм позволяет использовать одно имя для реализации разных сущностей. Мы знаем, что операция `=` уже имеет несколько разных сущностей — очевидно, что копирование целых и вещественных чисел, скорее всего, производится различными машинными командами. Сейчас мы должны реализовать эту операцию для копирования объектов типа `mystr`.

```
class mystr {
private:
   char *ptr;
public:
   mystr(const char *arg = NULL) {
      ptr = strdup(arg == NULL ? "" : arg);
   }
   ~mystr() {
      free(ptr);
   }
   mystr(mystr const &oth) {
      ptr = strdup(oth.ptr);
   }
   const char * str() const {
      return ptr;
   }
   mystr& operator=(mystr const &oth) {
      free(ptr);
      ptr = strdup(oth.ptr);
      return *this;
   }
};

int main() {
   mystr s1("foo");
   printf("s1='%s'\n", s1.str());
   {
     mystr s2;
     s2 = s1;
     printf("s2='%s'\n", s2.str());
   }
   printf("s1='%s'\n", s1.str());
}

$ ./a.out
s1='foo'
s2='foo'
s1='foo'
$
```

Всё заработало. Давайте поясним, что же происходит в операторе копирования.

```
01   mystr& operator=(mystr const &oth) {
02     free(ptr);
03     ptr = strdup(oth.ptr);
04     return *this;
05   }
```

Первая строка. Мы объявляем новый метод. Возвращаемое значение — ссылка на объект типа `mystr`. Аргументы — как у конструктора копирования. Почему такой странный тип возвращаемого значения? Давайте сделаем небольшое отступление.

Вспомним семантику операций и операторов языка Си (и C++, кстати). Что значит запись:

```
a = 5
```

Это операция (*operator*) присваивания. Если мы поставим точку с запятой после неё, мы получится оператор присваивания (*statement*)[1]

[1] Видите, русский и английский язык использует слегка различную терминологию, которая в нашем случае

Пока операция не стала оператором, её результат (а всякая операция имеет результат) можно использовать в другой операции (операции присваивания выполняются **справа налево**, хотя проще всего подстраховаться и поставить скобки):

```
b = a = 5
```

Первая операция — присваивание числа 5 переменной `a`. Её результат — 5. Вторая операция — присваивание переменной `b` результата предыдущей операции. Если мы поставим после пятёрки точку с запятой, то этот набор операций опять превратится в оператор присваивания.

Мы использовали термин *результат операции*. Он обычно равен последнему присвоенному значению. А оно сохраняется в переменной (сначала `a`, затем `b`). Поэтому каждая из операций присваивания возвращает результат операции — себя. Ссылка, как обычно, здесь применяется по соображениям эффективности.

Вторая строка. Зачем мы здесь освобождаем память? Дело в том, что мы можем присвоить новое значение переменной, которой ранее присваивали другое значение. Старое значение хранится в куче (результате работы функции `strdup`). Если мы указателю `ptr` присвоим новое значение из аргумента, то память, на которую указывает старое значение `ptr` будет недоступна. Впрочем, об этом мы уже говорили, когда обсуждали сборку мусора.

Четвёртая строка. Мы уже знаем, что `this` — указатель на текущий объект. Следовательно, `*this` есть сам текущий объект, который мы и возвращаем в виде ссылки.

Мы можем сформулировать строгий принцип:

Если объект содержит динамические объекты, надо обязательно реализовать в нём конструктор копирования, оператор присваивание и деструктор.

3.15 Полиморфизм: доопределение операций

Для примера определения операций над классами посмотрим на класс, реализующий понятия комплексного числа, как пары вещественных и нескольких операций:

```
#include <stdio.h>

class complex {
public:
   double re,im;
   complex(double newRe = 0, double newIm = 0) {
      printf("1 complex(%g,%g)\n", newRe, newIm);
      re = newRe; im = newIm;
   }
   void print() {
     printf("(%g,%g) ", re, im);
   }
   complex(complex const &oth) {
     printf("2 complex(%g,%g)\n", oth.re, oth.im);
     re = oth.re;
     im = oth.im;
   }

   complex & operator+=(complex const &oth) {
      printf("operator (%g,%g) += (%g,%g)\n", re, im, oth.re, oth.im);
      re += oth.re;
      im += oth.im;
      return *this;
   }
```

может привести к конфузу — слово *операция* переводится как *operator*. В реальном общении программистов между собой, однако, гораздо чаще звучит «определение операторов», как калька слова *operator*, чем более правильное «переопределение операций».

```
   complex operator+(complex const &oth) {
      printf("operator (%g,%g) + (%g,%g)\n", re, im, oth.re, oth.im);
      complex ret(*this);
      ret += oth;
      return ret;
   }

   complex & operator=(complex const &oth) {
     printf("operator=(%g,%g)\n", oth.re, oth.im);
     re = oth.re;
     im = oth.im;
     return *this;
   }
};

int main() {
   complex a(5, 10);
   complex c(11, 22);
   complex d;
   d = a + c;

   a.print(); c.print(); d.print(); printf("\n");
}
```

3.15.1 Два способа определения операций — внутри и вне класса. Достоинства и недостатки.

@@@

3.16 Исключения

Предположим, мы хотим расширять класс комплексных чисел дальше, для практического применения. Что произойдёт, если мы попытаемся разделить какое-либо комплексное число на ноль?

```
   complex operator/(complex const &oth) {
      printf("operator (%g,%g) + (%g,%g)\n", re, im, oth.re, oth.im);
      double d2 = oth.re*oth.re + oth.im*oth.im;
      complex ret;
      ret.re = (re*oth.re+im*oth.im) / d2;
      ret.im = (im*oth.re-re*oth.im) / d2;
      return ret;
}
```

Мы знаем, что делить на ноль нельзя. Поэтому, как только мы выясним, что `d2` равно нулю... А что мы должны сделать, если это выясним?

- Изменить метод?

  Например, добавить ещё один логический аргумент. Увы, в данном случае это невозможно. Доопределение операций требует строгого набора аргументов.

- Завершить программу?

  При делении на ноль целых чисел ведь программа завершается аварийно. Но в случае целых чисел мы можем сравнить делитель с нулём, это достаточно дешёвая операция — и делить только в случае неравенства делителя нулю. В нашем же случае проверка на равенство нулю не столь тривиальная операция. Можно, конечно, сделать метод `isZero()` в классе `complex` и вызывать его перед каждой операцией деления для сравнения делителя с нулём, но это точно замедлит программу и, вероятно, усложнит код.

- Вернуть особое значение в качестве результата?

  Наверное, в данном случае худшее из решений. Нам придётся изменять **все** методы класса `complex` и добавлять проверки на это самое особое значение.

Лучший способ совместить эффективность и удобство заключается в использовании *исключений* языка C++. Для этого нам понадобится три новых ключевых слова:

- `throw` — швырнуть, бросить, *возбудить* исключение.
- `try` — сообщить, что следующий участок кода может возбудить исключения (а может и не возбудить).
- `catch` — поймать, *перехватить* исключение и обработать его.

Для успешной обработки исключительных ситуаций потребуется обязательно все три слова.

Общий синтаксис таков:

```
void foo(x) {
  if (x < 0) throw "OOPS";
  ...
}

  try {
     // какой-то код, который может быть, возбудит исключение
     foo(-1);
  } catch (const char *s) {
    printf("I've caught exception %s\n", s);
  } catch (...) {
    printf("I've caught unknown exception\n");
    // перехват всех возможных исключений
  }
```

Оператор `throw` возбуждает исключение. Каждое исключение имеет свой тип и за оператором `throw` обычно следует выражение, по типу которого создаётся объект, который и будет перехвачен обработчиком.

Зона деятельности ключевого слова `try` распространяется только на следующий за ним блок — содержимое, заключённое в фигурные скобки. `catch(...)` перехватывает абсолютно все возможные исключения. `catch(const char *s)` перехватит исключения, тип которых `const char *` — как в нашем примере. Существует специальное указание — `catch(...)` (*elipsis* для перехвата тех исключений, которые до этого не были перехвачены.

Исключения исключительно полезны для обработки исключительных ситуаций. Исключительная ситуация — нештатное событие в программе, которое редко происходит. Например, если процесс читает из открытого файла, то нормальной ситуацией является получение очередной порции данных, а исключительной — ошибка чтения. При заказе памяти у операционной системы (по `calloc`, например) нормальная ситуация возникает, если память нам предоставлена, исключительной — если не предоставлена. Если действия, приводящие к исключительной ситуации, находятся где-то в глубине метода класса, то возникновение исключительной ситуации должно быть как то обнаружено и обычно требуется известить пользователя об этом.

Вот типичный пример использования исключений в простой функции копирования файлов. Функция для быстрого копирования выделит буфер из кучи и откроет файлы средствами операционной системы (захватит ресурсы). Эти ресурсы должны быть обязательно освобождены перед выходом из функции. Пусть функция возвратит признак успеха.

```
// Копировать файл с именем in в файл с именем out. Вариант 1 - без исключений.
int copy(const char *in, const char *out) {
   int fdin, fdout;
```

```
   char *buf;
   fdin = open(in, O_RDONLY);
   if (fdin < 0) return 1;
   fdout = open(out, O_WRONLY | O_CREAT, 0666);
   if (fdout < 0) {
      close(fdin);
      return 2;
  }
  char *buf = malloc(4096);
  if (buf == NULL) {
    close(fdin);
    close(fdout);
    return 3;
  }
  int rd;
  while ( (rd = read(fdin, buf, 4096)) > 0) {
    int wr = write(fdout, buf, rd);
    if (wr != rd) {
       close(fdin);
       close(fdout);
       free(buf);
       return 4;
    }
  }
  free(buf);
  close(fdin);
  close(fdout);
  return 0;
}
```

Функция обрабатывает типичные ошибки и возвращает код ошибки. Вызывающая её функция должна изучить код ошибки и выдать соответствующее сообщение. Попробуем переписать функцию с использованием исключений.

```
// Копировать файл с именем in в файл с именем out. Вариант 2 - с исключениями.
void copy(const char *in, const char *out) {
   int fdin = -1, fdout = -1;
   char *buf = NULL;
   const char *ex = NULL;
   try {
      fdin = open(in, O_RDONLY);
      if (fdin < 0) throw "Can't open Input file";
      fdout = open(out, O_WRONLY | O_CREAT, 0666);
      if (fdout < 0) throw "Can't open output file";
      char *buf = malloc(4096);
      if (buf == NULL) throw "Can't allocate buffer";
      int rd;
      while ( (rd = read(fdin, buf, 4096)) > 0) {
          int wr = write(fdout, buf, rd);
          if (wr != rd) throw "Error writing file";
      }
  } catch (const char *s) {
    ex = s;
  }
  if (fdin >= 0)  close(fdin);
  if (fdout >= 0) close(fdout);
```

```
  if (buf != NULL) free(buf);
  if (ex != NULL) throw ex;
}

int foobarcopy() {
  try {
    copy("foo", "bar");
  } catch (const char *s) {
    printf("Copy foo to bar failed: %s\n", s);
 }
 // продолжаем работу
}
```

Как мы видим, логика значительно упростилась. Мы надеемся, что штатная работа функции будет происходить без особых эксцессов, но подозреваем, что они могут произойти (помещаем вызов `copy` в блок `try`). Если ничего не исключительного не произошло, то мы продолжаем работу. Если функция `copy` возбудила исключение, то мы попадаем в блок обработки исключения, где и печатаем сообщение о проблеме (это — нештатная ситуация).

Что же происходит внутри самой функции `copy`? Там мы тоже используем механизм исключений для предотвращения утечки ресурсов. Мы пользуемся тем свойством исключений, что при его возбуждении и дальнейшем перехвате происходит переход непосредственно в блок перехвата. Инициализируя ресурсы (`fdin`, `fdout` и `buf`) невозможными значениями, мы можем обнаружить, что эти ресурсы заказаны и освободить их.

В C++ без исключений, собственно говоря, можно и обойтись, хотя они часто бывают удобны. В некоторых языках, ярким представителем которых является язык `Java`, без обработки исключений реальную программу написать невозможно. Например, в языке `Java` даже операция открытия файла возбуждает исключение при ошибке. Современные проектировщики языков считают, что создатели языка `Java` уж явно переборщили с исключениями. Например, достаточно современный язык `Go` никаких механизмов обработки исключений не содержит и это принципиальная позиция создателей языка.

3.16.1 Исключения и объекты

А что происходит с объектами, которые были созданы в функции до возбуждения исключения?

```
#include <stdio.h>

class foo {
public:
  foo() { printf("foo\n"); }
  ~foo() { printf("~foo\n"); }
};

void bar() {
  foo f;
  printf("bar()\n");
  throw "!";
}

int main() {
  try {
    bar();
  } catch (const char *s) {
    printf("caught %s\n", s);
  }
}
```

```
$./a.out
foo
bar()
~foo
caught !
$
```

Очень хорошо! Перед передачей управления в блок обработки исключений (`catch`) был вызван деструктор созданного в функции объекта.

При возникновении исключения вызываются деструкторы всех объектов, покидающих данную область видимости

3.17 Модификатор `static`

Модификатор `static`, который мы ранее использовали для обычных функций и глобальных переменных (они становились видимыми только внутри своего файла), для локальных переменных функции (они сохраняли своё значение между вызовами), может применяться также для методов класса и для полей класса. Рассмотрим пример:

```
class Dog {
public:
  static int counter;
  int trainLevel;
  Dog() { counter++;  trainLevel = 0;}
  ~Dog() { counter--; }
};
```

Класс `Dog` сейчас содержит статическое поле `counter` и обычное поле `trainLevel`. Каждая вновь созданная собака будет иметь индивидуальное поле `trainLevel` и все созданные собаки будут иметь одно и то же общее поле `counter`. Каждый вызванный конструктор класса `Dog` (каждый созданный объект этого класса) будет увеличивать это поле на единицу, каждый деструктор — уменьшать на единицу. Таким образом мы можем сосчитать популяцию всех собак.

Поле класса, объявленное статическим, существует в единственном экземпляре для всех объектов класса и разделяет одно и то же значение между всеми объектами класса.

Можно постараться обобщить этот же принцип единственности и на статические методы класса. В обычный, нестатический метод, как мы знаем, передаётся указатель `this`, с помощью которого мы имеем доступ к полям текущего объекта. Напоминаем, что внутри методов класса `Dog` идентификатор `trainLevel` означает именно `this->trainLevel`.

В статические методы класса переменная `this` не передаётся, идентификатор `this` недоступен и к нестатическим полям класса доступ существует только через соответствующие объекты.

```
class Dog {
public:
  static int counter;
  int trainLevel;
  Dog() { counter++;  trainLevel = 0;}
  ~Dog() { counter--; }
  static void f() {
      printf("counter=%d\n", counter); // OK, counter доступен
      printf("trainLevel=%d\n", trainLevel); // Ошибка компиляции, trainLevel недоступен
};
```

Статические методы классов играют роль глобальных функций внутри класса. С помощью статических функций и методов класса легко имитировать понятие *модуля* в модульном программировании, о котором мы уже упоминали. Именно с помощью статических методов класса можно передать объект класса внутрь функции потока. Как известно, во всех операционных системах функция потока (`thread`) должна @@@@@@@@@@@

3.18 Наследование

Наследование — последний, третий кит, на котором стоит ООП. Если мы захотим наряду с классом `Dog` ввести новый класс — `Cat`, то мы обнаружим, что между ними очень много общего — хвостатость, способность издавать звуки, четырёхлапость. Мы можем захотеть собрать всё общее в один класс, назовём его `Pet` — домашний питомец.

```
class Pet {
public:
  Pet() { printf("Pet()\n"); }
  ~Pet() { printf("~Pet()\n"); }
  void sound() { printf("Beeeeee\n"); }
  int legs;
  int trainLevel;
};
```

Тогда классы `Dog` и `Cat` смогут унаследовать от класса `Pet`.

```
class Cat: public Pet {
public:
  Cat() { printf("Cat()\n");  legs = 4; trainLevel = 0;}
  ~Cat() { printf("~Cat()\n"); }
};
```

3.19 Наследование. Базовые и порождённые классы

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

class Animal {
//   virtual ~Animal() = 0;
public:
   virtual ~Animal() {}
   virtual void sound() const = 0;
   virtual const char *getID() const = 0;
};

class Dog : public Animal {
public:
   Dog() { printf("Dog()\n"); }
   ~Dog() { printf("~Dog()\n"); }
   void sound() const { printf("Bark!\n"); }
   const char *getID() const { return "Dog"; }
};

class Cat : public Animal {
public:
   Cat() { printf("Cat()\n"); }
   ~Cat() { printf("~Cat()\n"); }
   void sound() const { printf("Meow\n"); }
```

```
   const char *getID() const { return "Cat"; }
};

int main() {
  const int PACK_SIZE = 10;
  Animal *pack[10];
  for (int i = 0; i < PACK_SIZE; i++) {
    if (rand() % 2 == 1) {
      pack[i] = new Dog();
    } else {
      pack[i] = new Cat();
    }
  }
  for (int i = 0; i < PACK_SIZE; i++) {
    pack[i]->sound();
  }
  for (int i = 0; i < PACK_SIZE; i++) {
    delete pack[i];
  }
}
```

3.20 Наследование. Пример фабрики классов

Сложный пример.

```
#include <stdio.h>
#include <vector>
using namespace std;

class T {
public:
  virtual ~T() {}
  virtual char getID() const = 0;
  virtual T* clone() const = 0;
  virtual void init(char t) = 0;
  virtual void print() const = 0;
};

struct T1 : public T {
  char getID() const { return '1'; }
  void print() const { printf("T1=%c\n", body); }
  T * clone() const { return new T1(); }
  void init(char b) { body = b; }
  char body;
};

struct T2: public T {
  char getID() const { return '2'; }
  void print() const { printf("T2=%c\n", body); }
  T * clone() const { return new T2(); }
  void init(char b) { body = b; }
  char body;
};

struct T3: public T {
  char getID() const { return '3'; }
  void print() const { printf("T3=%c\n", body); }
```

```
  T * clone() const { return new T3(); }
  void init(char b) { body = b; }
  char body;
};

int main() {
   const char *inp = "1A2B1C3D";
   T *patterns[] = {
      new T1(),
      new T2(),
      new T3()
   };

   for (int i = 0; ;) {
     char type = inp[i++];
     if (type == 0) break;
     char arg =  inp[i++];

     for (int p = 0; p < sizeof patterns / sizeof patterns[0]; p++) {
        if (patterns[p]->getID() == type) {
   T *n = patterns[p]->clone();
          n->init(arg);
          n->print();
   break;
        }
     }
  }
}
```

4 Немного о STL

Мы уже умеем писать полезные классы. Но не для всех задач требуется изобретать велосипед. Например, для комплексных чисел, которые мы попытались реализовать, такой велосипед уже изобретён. Очень полезным инструментом являются *контейнеры* — классы, которые заключают в себе другие элементы — объекты классов или обыкновенных типов.

4.1 Контейнеры

В этом разделе мы рассмотрим несколько различных полезных контейнеров а затем посмотрим, в каких случаях следует применять тот или другой контейнер. Давайте начнём с самого простого и популярного.

4.2 Контейнер `vector`

Для его подключения к нашему коду потребуется include-файл `<vector>`. Обратим внимание на то, что каждый из предлагаемых включаемых файлов **не содержит расширения .h**, к которому мы уже привыкли, программируя на языке `C`.

Представим, что нам нужно решить достаточно следующую задачу: со стандартного ввода нужно считывать натуральные числа, признаком конца пусть служит число 0. Сколько будет таких чисел заранее неизвестно, может быть, 10, а может быть 1000000. Первая подзадача — все эти числа нужно поместить в единый массив для дальнейшей обработки.

Поскольку количество чисел заранее неизвестно, мы не можем использовать массив фиксированного размера — какой бы мы размер не назначили массиву, найдутся такие исходные

данные, которые в этот массив не поместятся. Следовательно, требуется использовать динамический массив — тот, который располагается в *куче*, для доступа к которому потребуется указатель, полученный вызовом функций `calloc` или `realloc` или вызовом оператора `new`. Мы изберём следующую стратегию — мы будем заказывать память не единичными объектами, а группами объектов, поскольку операция заказа памяти достаточно трудоёмкая. Итак, вот пример программы:

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

int main() {
   const int CHUNK_SIZE = 16; // Заказываем память фрагментами по 16 объектов
   int *array = (int *)calloc(CHUNK_SIZE, sizeof(int));
   int  allocated_elems = CHUNK_SIZE;
   int  used_elements = 0;
   int x;
   scanf("%d", &x);
   while (x != 0) {
      array[used_elems] = x;
      used_elems++;
      if (used_elems >= allocated_elems} {
        array = realloc(array, (allocated_elems + CHUNK_SIZE) * sizeof(int));
        allocated_elems += CHUNK_SIZE;
     }
      scanf("%d", &x);
   }
   for (int i = 0; i < used_elems; i++) {
     printf("array[%d]=%d\n", i, array[i]);
   }
   ...
   delete array;
}
```

Код получился достаточно громоздким. Если в программе часто используется подобный алгоритм, можно попробовать его реализовать в виде функции. Хуже будет, если нужно будет использовать этот алгоритм для переменных разных типов — то есть, для массивов целых, вещественных и т. д. — тогда придётся писать несколько функций (или использовать *механизм шаблонов*, мы про него ещё не говорили). Вместе с каждым из таких массивов мы обязаны держать пару переменных — реальный размер массива, сколько элементов в нём содержится (`used_elements`) и количество заказанных элементов (чтобы расширить массив, когда это будет нужно). Полученный указатель нужно будет освободить в подходящий момент времени, а, если мы захотим его использовать повторно, то нужно будет опять обнулить сопутствующие переменные. Всё это, как мы понимаем, алгоритм не упрощает — мы получили несколько троек связанных элементов, а лишние связи усложняют программу.

Вот как выглядит этот алгоритм, реализованный с помощью контейнера `vector`:

```
#include <stdio.h>
#include <vector>
using namespace std;

int main() {
   vector<int> array;
   int x;
   scanf("%d", &x);
   while (x != 0) {
      array.push_back(x);
      scanf("%d", &x);
   }
```

```
for (int i = 0; i < array.size(); i++) {
  printf("array[%d]=%d\n", i, array[i]);
}
...
```

Мало того, что программа резко сократилась, она упростилась. Вместо группы разнородных переменных и констант теперь имеется один объект контейнерного класса `vector<int>`, который в данном случае решает все наши проблемы. Давайте подробнее рассмотрим, что происходит в этой программе.

Во-первых, этот контейнерный тип (впрочем, как и все остальные) использует механизм *шаблонов* для создания разных версий векторов. Контейнер `vector` создавали для удобной и эффективной замены основного типа любого из языков — массива. `vector<int> array;` сообщает нам, что мы создаём новый объект под именем `array`, этот объект будет содержать целые числа (`int`) в контейнере. Пока этот контейнер пуст, но мы в любой момент можем изменить его размер, например, добавив один элемент к концу (`array.push_back(x)`). Контейнерный объект сам определит, для скольких элементов уже выделена память (объект содержит в себе скрытую переменную, аналог переменной `allocated_elems`) и сколько элементов используется (аналог `used_elems`) и либо быстро положит новый элемент в конец массива, скорректировав его используемый размер, либо постарается увеличить выделенную память и затем добавит новый элемент. Эта операция достаточно эффективна и не вызывает изменения выделения памяти в каждом вызове метода (мы примерно так и поступали, когда писали первый вариант задачи).

Если к размеру выделенной памяти мы доступа не имеем (впрочем, зачем он нам?), то количество реальных элементов мы можем узнать методом `size()`. Что самое главное, контейнерный класс `vector` определяет операцию индексирования (`array[i]`) и поэтому после создания вектора работа с ним напоминает работу с обычным массивом. Перечислим основные методы класса `vector`.

- `vector<double> vd;` // пустой вектор вещественных
- `vector<int> vi(125);` // вектор целых, содержащий сразу `125` элементов, равных нулю.
- `vector<int> avi = vi;` // вектор целых, копия вектора `vi`, то есть тоже `125` элементов
- `vd.resize(333);` // `vd` теперь содержит `333` элемента.
- `vd[123] = 777;` // `123`-й по индексу элемент изменяет значение
- `size_t svd = vd.size();` // `svd` будет равно `333`
- `vd.push_back(555);` // в `vd` добавлен элемент `555` к концу. Теперь там `334` элемента
- `vd.pop_back();` // в `vd` уничтожен последний элемент. Теперь там `333` элемента.

	Массив `a` статический динамический	**Вектор** `v`
Объявление	`int a[10];` `int *a;`	`vector<int> v(10);`
Выделение памяти	`a = new int[10];`	`v.resize(10);`
Освобождение памяти	Достижение `}` `delete a;`	Достижение `}`
Определение размера	`sizeof a/sizeof a[0]` добавочная переменная	`v.size();`
Чтение элемента	`x = a[i];`	`x = v[i];`
Запись элемента	`a[i] = x;`	`v[i] = x;`
Копирование массивов	`memcpy(b, a, sizeof(a));` `memcpy(b, a, sizeof(a[0])*elems);`	`vector<int> v2 = v;`
Изменение размера	невозможно `a = realloc(a,sizeof(int)*new_size);`	`v.resize()`
Получение адреса элемента	`&a[i]`	`&v[i]`
Расположение элементов	последовательное	последовательное

В таблице приведено небольшое сравнение привычных операций с массивами и подобных операций с векторами. К явным плюсам векторов мы можем отнести:

- лёгкость создания
- не требуется заботиться об уничтожении. Как только вектор покидает область видимости, память, им занятая освобождается.
- лёгкость копирования. Просто используется оператор присваивания.
- лёгкость изменения размера в любой момент времени
- возможность контроля за индексами. Например, в `a[20]` в массиве из `10` элементов, скорее всего, приведёт к необнаруживаемой ошибке времени исполнения. `v[20]` в ряде реализаций вызовет исключительную ситуацию.
- наличие динамического массива (адресуемого через указатель) в объекте обязательным образом требует реализации собственного оператора копирования и конструктора инициализации (об этом говорилось в соответствующих разделах). Наличие вектора в объекте такого не требует (мы не говорим о случаях, когда вектор содержит указатели на динамические объекты — тогда и копирование векторов опасно по своей сути).

Удобно? Да, конечно. Есть ли минусы? Конечно есть, но, к счастью немного. Во-первых, вектор — объект языка `C++` и его нельзя передавать как единое целое в функции языка `C` и в системные вызовы. Однако, гарантируется, что все элементы вектора расположены последовательно и при наличии операции взятия адреса элемента эта проблема решается. Вот, например, как можно записать системным вызовом `write` содержимое вектора:

```
vector<int> v;
...
write(fd, &v[0], v.size() * sizeof(int));
```

Во-вторых, существует вероятность того, что в критических по времени исполнения фрагментах вектор будет менее эффективен, чем массив. К счастью, такие компиляторы, как `Borland C++` и подобные остались в прошлом, и все компиляторы, по крайней мере начиная с 2005 года, реализуют обращение к элементам вектора так же эффективно, как и к элементам массива. К проверенным компиляторам относятся

- `Microsoft Visual Studio 2005,2008,2010,2012...` В режиме компиляции `Debug` этот компилятор вставляет проверку на попадание индекса в границы вектора. В режиме `Release` компилятор генерирует сравнимый с массивами код.
- `g++ 3,4` ...
- `clang++` всех версий
- `Intel C++` всех версий

Совет: используйте вектор вместо динамических массива везде, где только можно. Пусть в вашей программе останутся только небольшие статические массивы заранее известной длины.

4.3 итераторы

@@

4.4 string

@@

4.5 deque

@@

4.6 set

@@

```
#include <vector>
#include <stdio.h>
#include <stdlib.h>
#include <algorithm>
#include <set>
using namespace std;

int main() {
  set<int> vi;
  for (int i = 0; i < 5; i++) {
     vi.insert(rand() % 10);
  }
  printf("size=%d\n", vi.size());
  for (set<int>::const_iterator it = vi.begin(); it != vi.end(); it++) {
    printf("%d ", *it);
  }
  printf("\n");
  set<int> s2;
  for (int i = 0; i < 10; i++) {
    s2.insert(rand() % 10);
  }
  printf("s2 size=%d\n", s2.size());
  vi.insert(s2.begin(), s2.end());
  for (set<int>::const_iterator it = vi.begin(); it != vi.end(); it++) {
    printf("%d ", *it);
  }
  printf("\n");
  set<int>::const_iterator f = vi.find(4);
  if (f != vi.end()) {
    printf("found %d\n", *f);
  } else {
    printf("not found %d\n", *f);
  }
}
```

```
#include <vector>
#include <stdio.h>
#include <stdlib.h>
#include <algorithm>
#include <set>
#include <string>
using namespace std;

struct person {
  string name;
  int    salary;
};

bool operator<(const person &l, const person &r) {
  if (l.name < r.name) return true;
  if (l.name > r.name) return false;
  return l.salary < r.salary;
}
```

```
int main() {
  set<person> vi;
  for (int i = 0; i < 5; i++) {
     char buf[10]; sprintf(buf,"p%d", i);
     person p; p.name = buf; p.salary = rand() % 100;
     vi.insert(p);
  }
  for (set<person>::const_iterator it = vi.begin(); it != vi.end(); it++) {
    printf("(%s,%d) ", it->name.c_str(), it->salary);
  }
  printf("\n");
}
```

4.7 map

```
#include <map>
#include <stdio.h>
#include <string>
using namespace std;

int main() {
  map<string,int> m;

  m.insert(make_pair("foo", 123));
  m["bar"] = 444;
  m["bar"] = 555;

  for (map<string,int>::const_iterator it = m.begin(); it != m.end(); it++) {
    printf("(%s,%d)\n", it->first.c_str(), it->second);
  }

  map<string,int>::iterator it = m.find("foo");
  if (it != m.end()) {
    it->second = 666;
    *(string *)(&it->first)  = "OOPS";
  }
  m["zzz"] = 999;

  for (map<string,int>::const_iterator it = m.begin(); it != m.end(); it++) {
    printf("(%s,%d)\n", it->first.c_str(), it->second);
  }
}
```

4.8 алгоритмы

```
#include <vector>
#include <stdio.h>
#include <stdlib.h>
#include <algorithm>
using namespace std;

struct point {
  int x,y;
  point(int x, int y) {
    this->x = x; this->y = y;
  }
  //point &operator=(point const &oth) {
```

```
  //  printf("="); this->x = oth.x; this->y = oth.y; return *this;
  //}
};

bool operator<(const point &l, const point &r) {
  return l.x < r.x;
}

int main() {
  vector<point> vi;
  for (int i = 0; i < 190; i++) {
    point p(rand() % 3, rand() % 3);
    vi.push_back(p);
  }
  for (int i = 0; i < vi.size(); i++) {
    printf("(%d,%d) ", vi[i].x, vi[i].y);
  }
  printf("\n");

  sort(vi.begin(), vi.end());
  for (vector<point>::const_iterator it = vi.begin(); it != vi.end(); it++) {
    printf("(%d,%d) ", it->x, it->y);
  }
  printf("\n");

}
```

4.9 C++11

5 Применение C++ для решения практических задач

5.1 Немного о понятии проекта

В реальных проектах очень часто удобно разбивать исходный код классов на две части — часть, описывающую интерфейс класса (обычно она хранится в файле с расширением `*.h`) и часть, содержащую реализацию класса (обычно с расширениями `*.cpp` или `*.cc`. Любой метод класса можно разделить таким образом — в `*.h` отправить только описание метода, без реализации, а в `*.cc` — только реализацию, без описания.

До сего момента мы обычно описывали методы внутри объявления класса и они становились определениями.

Напомним, что имеются два термина:

1. определение, декларация, интерфейс, interface, — это синонимы одного понятия, показывающего, что в данном месте имеется только описание метода, его *прототип* и что сам метод будет присутствовать в каком-то другом месте, не здесь.

2. объявление, дефиниция, имплементация, implementation — это синонимы другого понятия, показывающего, что в данном месте имеется реализация метода, то есть исполнимый код метода.

Уфф.

Объявления возможны только внутри описания класса — после ключевого слова `class` последующим именем класса. Если после объявления метода мы поставим точку с запятой — мы объявим метод и это значит, что мы обязаны определить его в каком-то другом месте. Если после объявления метода мы поставим открывающую фигурную скобку — мы поместим определение метода сразу же после его объявления.

Пример:

```
// foo.h
class foo {
public:
   int func1(int x);   // объявление метода func1()
   int func2(int x) { // определение метода func2()
      return x*x;       // код, реализующий метод
   }
   ...
};

// foo.cc
#include "foo.h"
int foo::func1(int x) { // определение метода func1  класса foo
  return x * x * x;
}

//main.cc
#include "foo.h"
...
int main() {
   foo a;
   int b = a.func1();
   ...
}
```

Определения методов можно, и часто удобнее всего, помещать в отдельный файл. Теперь для компиляции проекта в командной строке требуется перечислить все файлы, содержащие определения функций (в том числе функции `main` и методов классов.

```
$g++ -o main main.cc foo.cc
```

Можно, конечно, писать подобным образом каждый раз, даже в тех случаях, когда в проект будут добавляться новые файлы. Но ведь это неудобно! Если мы изменили файл `foo.h`, то стоит ли нам перекомпилировать файлы `main.cc` и `foo.cc`? Да, стоит, эти файлы зависят от файла `foo.h`. Если мы изменили файл `foo.cc`, следует ли нам перекомпилировать файл `main.cc`? Если мы его уже компилировали, то уже не обязательно. Это у нас такие маленькие файлы, в реальных проектах исходные файлы состоят из тысяч строк (и включают в себя десятки тысяч строк из системных включаемых файлов, типа `<vector>` или `<stdio.h>`.

Во-первых, нам нужно уметь сохранять результаты компиляции одиночного файла:

```
$g++ -c main.cc
```

сохранит нам результаты компиляции в файл `main.o`, так называемый *объектный* файл.

```
$g++ -c foo.cc
$g++ -o main main.o foo.o
```

Последняя строчка создаст из объектных файлов проекта исполнимый файл `main`, подключив на этом этапе (и только на этом этапе!) библиотеки (например, `libc`).

Однако теперь на предстоит небольшой ад. Мы должны всё время помнить, какой файл мы меняли, и какую последовательность команд нужно исполнить, чтобы создать исполнимый файл. Это несерьёзно. Нужно освоить понятие `make`.

5.2 `Make` — один из способов собирать проекты

Создадим файл с именем `Makefile`. Именно с таким именем. Оно не то, чтобы было фиксированным, но именно такое имя удобно.

Давайте посмотрим на пример такого файла для нашего суперпроекта:

```
all: main

main: main.o foo.o
g++ -o main main.o foo.o

main.o: main.cc foo.h

foo.o: foo.cc foo.h
```

Это всё! Программа `make` — достаточно сложная программа с большими возможностями. Пока нам достаточно небольшого подмножества. Главное понятие — *цель*. Самая первая цель — главная. Её `make` и пытается построить. Этот файл можно прочитать так:

1. главная цель — `all`, она зависит от `main`
2. цель `main` зависит от подцелей `main.o` и `foo.o`. Как только подцели будут созданы, цель `main` будет получена следующей командой:

   ```
   $g++ -o main main.o foo.o
   ```

   > **Обратите внимание! Обязательно!**
   >
   > Команды для построения цели обязательно начинаются с табуляции, не с пробелов!

3. цель `main.o` зависит от подцелей `main.cc` и `foo.h`. Если команда построения цели не указана, программа `make` достаточно умна для того, чтобы понять, что надо просто вызвать компилятор с ключом `-c`, примерно как мы компилировали до этого.

Построить проект можно командой

```
$make
```

Попробуйте создать небольшой проект и поэкспериментировать с изменением файлов и запуском построения.

5.3 Небольшой проект: объекты в многопоточном программировании

5.3.1 Makefile

```
all: project

OBJS = main.o Mutex.o Thread.o
project: $(OBJS)
        g++ -o project $(OBJS) -lpthread

Mutex.o main.o: Mutex.h

AutoUnlocker.h: Mutex.h

main.o: AutoUnlocker.h

Thread.o main.o: Thread.h
```

5.3.2 main.cc

```
#include <stdio.h>
#include "Mutex.h"
#include "Thread.h"
```

```
#include "AutoUnlocker.h"

class point {
public:
  point(double _x = 0, double _y = 0) :x(_x), y(_y) {}
  double getx() const { return x; }
  double gety() const { return y; }

private:
  double x,y;
public:
static void *func(void *c) {
  point *p = reinterpret_cast<point *>(c);
  printf("func()\n");
  printf("point = (%g,%g)\n", p->x, p->y);
  return NULL;
}

};

int main() {
  point a(10, 20), b(30, 40);
  Thread t1;
  t1.run(point::func, &a);
  Thread t2;
  t2.run(point::func, &b);
  t1.join();
  t2.join();
  printf("All threads gone\n");
}
```

5.3.3 Mutex.h

```
#pragma once

class Mutex {
public:
  Mutex();
  ~Mutex();
  int lock();
  int unlock();
private:
  struct Context;
  Context *context;
  Mutex(Mutex const &);
  Mutex & operator=(Mutex const &);
};
```

5.3.4 Thread.h

```
#pragma once

class Thread {
public:
  Thread();
  ~Thread();
  typedef void *(*routine)(void *);
```

```
  int run(routine, void *);
  int join();
  int kill();
private:
  struct Context;
  Context *context;
};
```

5.3.5 Thread.cc

```
#include "Thread.h"
#include <pthread.h>
#include <stdio.h>
#include <signal.h>

struct Thread::Context {
  pthread_t thr;
};

Thread::Thread() {
  context = new Context;
  context->thr = 0;
  printf("Thread()\n");
}

Thread::~Thread() {
  printf("~Thread()\n");
  if (context->thr != 0) {
    kill();
  }
  delete context;
}


int Thread::run(routine r, void *arg) {
  printf("Thread::run(%p,%p)\n", r, arg);
  if (context->thr != 0) {
    kill();
  }
  int code = pthread_create(&context->thr, NULL, r, arg);
  return code != 0;
}

int Thread::join() {
  void *arg;
  printf("Thread::join()\n");
  int code = 0;
  if (context->thr != 0) {
    code = pthread_join(context->thr, &arg);
  }
  context->thr = 0;
  return code;
}

int Thread::kill() {
  if (context->thr != 0) {
    pthread_kill(context->thr,9);
```

```
  }
  printf("Thread::kill()\n");
  context->thr = 0;
  return 0;
}
```

5.3.6 Mutex.cc

```
#include "Mutex.h"
#include <pthread.h>
#include <stdio.h>

struct Mutex::Context {
    pthread_mutex_t mutex;
};


Mutex::Mutex()
{
   context = new Context;
   pthread_mutex_init(&context->mutex, NULL);
}

Mutex::~Mutex()
{
   pthread_mutex_destroy(&context->mutex);
   delete context;
}

int Mutex::lock() {
   int code = pthread_mutex_lock(&context->mutex);
   if (code != 0) {
      printf("Lock returns %d\n", code);
   }
   return code != 0;
}

int Mutex::unlock() {
  return pthread_mutex_unlock(&context->mutex) != 0;
}
```