Написание кода на C++

Дима Трушин

2024 — 2025

Содержание

Аннотация

Сразу дисклеймер, я не являюсь профессиональным программистом, я – математик, который что-то понимает в программировании и Computer Science. Я постарался собрать в этом тексте какие-то наиболее полезные вещи, которые мне встретились за время работы над программными проектами на ПМИ ФКН. Я так же постарался добавлять ссылки на более подробное обсуждение затронутых вопросов или полезные материалы.

1 Несколько общих слов

Пару слов о себе Как я уже заявил выше, я не программист, я – математик. У этого есть несколько важных импликаций, которые стоит понимать:

1. У меня другой взгляд на программирование. Так как я все же рос и развивался в среде математиков и в программирование пришел сам и по своей воле, то я уже обладал достаточной профдеформацией, что скорее всего моя точка зрения будет сильно отличаться от того, что вы видите и слышите вокруг. Это наверное самое ценное, чем я могу поделиться с вами в идейном плане, ибо новые идеи обычно брать неоткуда. Надеюсь, что то что вижу и замечаю я окажется для вас полезным.
2. Отсутствие опыта в индустрии означает сразу две вещи:
   (a) Отсутствие опыта и знакомства со стандартными инструментами разработки. Это моя самая слабая часть. Но тут я все же наверстываю.
   (b) Отсутствие вредных привычек. Когда вы пишите код на работе у вас обычно куча дедлайнов и самая главная цель – получить результат в установленный срок. При этом у вас обычно есть какая-то команда с разным бэкграундом, есть какое-то legacy и проблемы с документацией к коду. В итоге приходится писать так как получается, лишь бы работало. Не всегда есть время подумать над тем как лучше. В моем случае я всегда неограничен во времени над тем, чтобы подумать как решать ту или иную проблему при написании кода.

Как ориентироваться? Надо понимать, что какой бы признак вы ни выделили среди людей, реально крутых людей с данным признаком будет мало. Навык программирования (в широком смысле) не исключение. Потому большинство того, что вы видите, слышите и читаете – это мусор. Со временем вы конечно разберетесь с этим вопросом, но пока нет достаточного опыта, то тут важно смотреть на ключевых игроков индустрии и понять, как на индустрию смотрят они, что они предлагают, какие проблемы видят и в целом перенимать опыт у ключевых фигур – очень важно. Вот для ориентира некоторый список людей, на которых я бы обратил внимание

1. Andrei Alexandrescu
2. Sean Parent
3. Scott Mayers

Так же следите за разными конферейнциями по C++, из которых самая известная пожалуй cppcon. Правда и на ней выступают убогие с мусорными докладами или что хуже с вредящими докладами. Я бы мог добавить к списку выше еще с десяток приличных людей, чьи взгляды и мнение повлияли на меня, но я думаю, что этих я бы выделил отдельно. Буду ли я расширять этот список или добавлю ли еще кого-то, посмотрим.

Про что этот текст? Умение проектировать программы и писать хороший код – это умение, которому тоже надо учиться. Написать так, чтобы просто работало – этого мало. Потому я хочу посвятить этот текст следующим вопросам:

1. Как писать хороший код. Что значит хороший код. Какие есть метрики или способы сравнивать подходы для написания кода.
2. Архитектура и дизайн. Какие приемы, шаблоны и техники стоит использовать и почему, а какие лучше не вспоминать.
3. Языковые особенности. Какие тонкости языка вам надо знать, чтобы не огрести от него. Язык C++ очень плохой и очень тяжелый, схлопотать undefined behavior можно запросто независимо от вашего опыта, а компилятор никогда ничего вам не должен и не поможет ни в чем. Потому важно обращать внимание на важные детали языка и быть готовыми к возможным проблемам.

Чтобы почувствовать стиль изложения Я хочу начать с двух аксиом:

1. C++ – говно.

2. Для меня это самый комфортный для использования язык.

Надо понимать, что язык действительно очень плохой и архаичный. С плохой экосистемой, плохими решениями на очень разных уровнях. Стихийно развивавшийся и обросший кучей противоречивых костылей. А самое главное – никакой помощи со стороны компилятора. А чем хуже язык – тем больше вам понадобится помощь со стороны инструментов разработки. Так же важно понимать, что чем больше в языке ограничений – тем лучше. Чем меньше у программиста свободы, тем сложнее написать код плохо. В C++ к сожалению свобода программиста безгранична. Это компилятор Rust-а умрет прежде чем пропустит вашу ошибку, потом прочитает вам лекцию почему так делать не надо, как это исправить и будет долго и упорно вести вас к верному решению. В случае C++ мы можем лишь верить, что наше приключение не закончится так и не начавшись.

2 Полезные ссылки

В этом разделе я собрал какие-то самые базовые ссылки на полезные материалы. Возможно в будущем я все же дойду до ссылок на список литературы, посмотрим. А пока перечень:

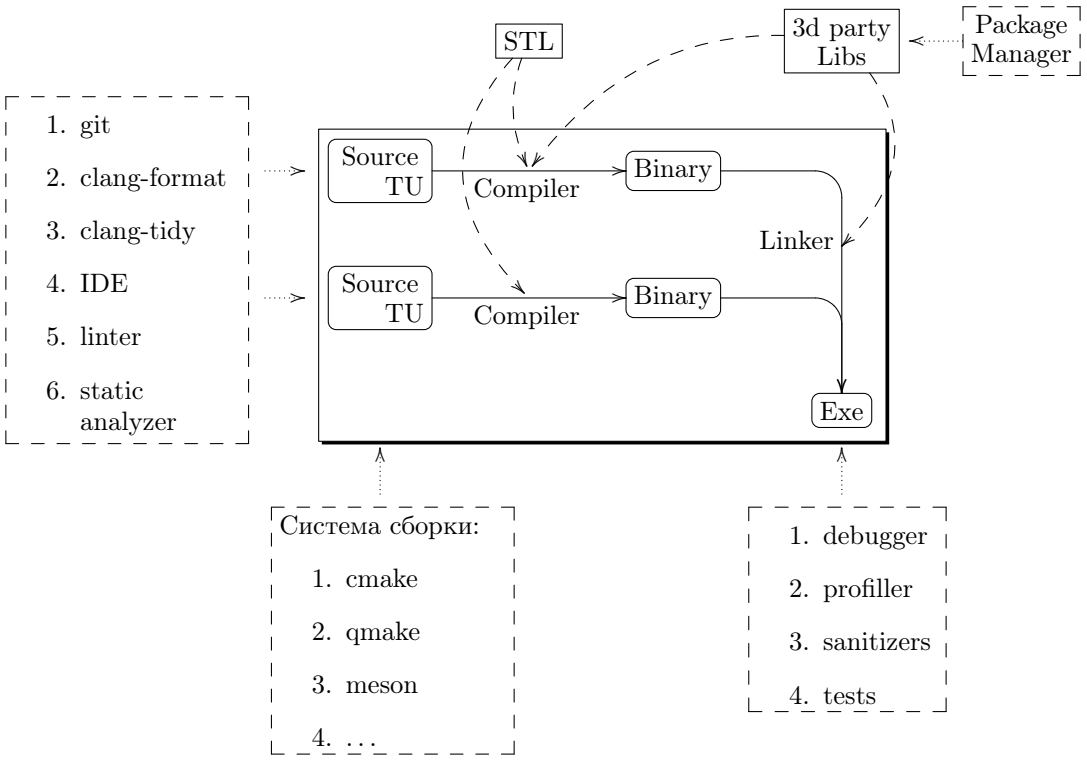
- Прежде всего надо не забывать про cppreference и cplusplus.
- Кроме этого есть FAQ по плюсам с кучей полезной информации по разным вопросам.
- Guide от SEI CERT по написанию надежного и безопасного кода на C и C++.
- Мой персональный плейлист с полезными видео в основном по плюсам.

3 Экосистема C++ и модель компиляции

В начале надо сказать несколько общих слов про экосистему C++.

3.1 Модель компиляции

Изначально C++ был надстройкой вокруг языка Си добавляющий в него новый функционал. Идея была в том, что любой Си код должен компилироваться C++ компилятором. А потому модель компиляции у C++ целиком и полностью позаимствована у Си. Однако, надо понимать, что эти языки очень сильно отличаются друг от друга. Кроме того, с самого начала многие аспекты C++ не были стандартизированы, язык развивался хаотично, что в итоге привело к тому, что поверх старой экосистемы Си был построен совсем новый язык, который только формально совместим с Си и уж точно не предназначен для использования вместо Си. Что эти слова в точности значат я постараюсь объяснить ниже в деталях. А начать я хочу прежде всего с общей картины. На диаграмме ниже изображена общая картинка того, как функционирует сборка приложения в языке C++ (включая информацию о доступных инструментах):



Круглые боксы:

1. Source/TU – исходный код/единицы трансляции
2. Binary – бинарный код
3. Exe – исполняемый код

Пунктирные боксы:

1. Инструменты для исходного кода
2. Инструменты для сторонних библиотек
3. Системы сборки
4. Инструменты для анализа собранного приложения

Прямоугольные боксы:

1. STL – стандартная библиотека
2. 3d party Libs – сторонние библиотеки
3. Бокс с затенением представляет весь проект

Стрелки:

1. ⟶ – поток данных при сборке
2. – – ➢ – показывает точки использования
3. ····➢ – показывает инструменты для работы на этом этапе сборки

3.1.1 Объяснение модели компиляции

Давайте я опишу, что тут происходит. Прежде всего задача сборки любого проекта – преобразование кода C++ в машинный код. Сборка проекта происходит в два этапа:

1. Этап компиляции. Выполняется компилятором (compiler). Преобразует код на C++ в отдельный бинарный файл (binary). Единица компиляции называется translation unit (TU). Обычно это cpp файл.

2. Этап линковки. Выполняется линковщиком (linker). Собирает из нескольких бинарных файлов полученных на этапе компиляции единый бинарный файл с исполняемым кодом (Exe).

Это две независимые стадии, которые выполняются разными программами. Давайте я на модельном примере поясню, что это все значит, и как понимать компиляцию плюсов. Предположим у вас есть следующие файлы:

```
// function.h

void function();

// function.cpp

#include "function.h"

void function() {
}

// main.cpp

#include "function.h"

int main() {
function();
return 0;
}
```

Теперь надо провести компиляцию и линковку кода командами[1]

1. compile `function.cpp` to `function.bin`

2. compile `main.cpp` to `main.bin`

Во время первой команды компиляции `function.cpp` создается так называемая единица трансляции. По простому, выполняются команды препроцессора и вставляются все `#include` копипастой. То есть получается файл

```
// TU for function.cpp

void function();

void function() {
}
```

После чего создается бинарный файл, скажем, `function.bin`, в который складывается информация о функциях и переменных из единицы трансляции.[2]

```
// function.bin

function:
  return;
```

Теперь выполняется компиляция `main.cpp`. В начале составляется единица трансляции.

```
// TU for main.cpp

void function();

```

[1] Я использую условные команды и не использую синтаксис конкретных комиляторов и линкеров.
[2] Я не использую конкретный синтаксис ассемблера, я пишу условный машинный код.

```
5  int main() {
6    function();
7    return 0;
8  }
```

Теперь эта единица трансляции компилируется в машинный код.

```
1  // main.bin
2
3  main:
4    call function();
5    return 0;
```

Обратите внимание, что компилятору не нужно знать определение `function`. Ему достаточно знать, что имя `function` является именем функции и что `function()` – это вызов функции. Об этом компилятору сообщает строчка 3 в единице трансляции для `main.cpp`. Без этой строчки `function` было бы неизвестным именем и вызывало бы ошибку компиляции. А так это функция, которую компилятор не знает, но знает все, чтобы в `main.bin` поместить правила ее вызова.

Теперь у нас есть два бинарных файла

```
1  // function.bin
2
3  function:
4    return;
5
6  // main.bin
7
8  main:
9    call function();
10   return 0;
```

И нам нужно вызвать линковку командой

- Link `function.bin` and `main.bin` to `main.exe`

Точкой входа в программу является функция `main()`. Потому компилятор найдет ее и начнет из всех бинарников подставлять все необходимые функции. Мы видим, что внутри `main` есть вызов функции `function()`. Для этого линкер должен где-то среди всех бинарников найти одну единственную функцию `funcion()` и добавить ее код в `main.exe`. В итоге получится следующее

```
1  // main.exe
2
3  main:
4    call function();
5    return 0;
6
7  function:
8    return;
```

Если линкер не находит имени `function()` ни в одном бинарнике – это ошибка линковки. Если линкер находит две функции `function` в разных бинарниках, то это тоже ошибка линковки, кроме специальных случаев.[3] В специальных случаях мы должны гарантировать линкеру, что все определения для `function` являются одинаковыми, потому что он может выбрать любое из них взамен другого. Соответственно, если это не так, то это undefined behavior.

Для чего сделано такое безобразие? Расчет в такой модели сделан на следующую организацию кода. Единицей кода является файл. И мы специально делим проект на несколько файлов, чтобы в случае внесения изменений нам не надо было перекомпилировать все файлы целиком, а лишь перекомпилировать один-два файла, и потом перелинковать нужные бинарники. Эта идея помогала значительно сократить время компиляции для больших проектов. Такая модель использовалась в Си, и потому плюсы переняли ее, добавив свои дополнительные стадии. Однако, файл, как единица компиляции, это очень плохая идея, особенно учитывая, что в плюсах появились шаблоны. Они как раз и ломают всю идиллию такого подхода.

[3] К специальным случаям относятся функции помеченные словом `inline` и инстанциации шаблонов, о которых я поговорю позже.

Что нужно знать компилятору Обратите внимание, что в плюсах есть два понятия declaration и definition. Declaration – это объявление объекта, которое говорит компилятору, что это за объект (переменная, функция, имя класса, имя шаблона и т.д.). Declaration может быть не достаточно для того, чтобы полностью скомпилировать код с этим объектом. Definition – это declaration, которое полностью определяет объект. Например

```
1  void f(); // declaration
2
3  void f() {} // definition
4
5  static int x; // declaration
6
7  static int x = 1; // definition
8
9  class A; // declaration
10
11 class A {}; // definition
12
13 int y; // definition
```

В строчке 1 мы видим declaration для функции. Его не достаточно, чтобы поместить в бинарник код для функции. Однако, его достаточно, чтобы поместить в бинарник точку вызова функции. В строчке 3 идет defintion для функции, его достаточно, чтобы поместить в бинарник код функции. В строчке 5 идет declaration для статической переменной. Чудесатость таких переменных заключается в том, что эта строчка не помещает переменную в память и не дает ей адрес, но говорит, что имя `x` означает имя переменной типа `int`. А вот строчка 7 уже полностью определяет `x` и после нее у `x` появляется адрес. Строчка 9 – declaration для класса. Ее не достаточно, чтобы создать переменную класса `A`, однако ее достаточно, чтобы создать указатель на этот класс, потому что все указатели имеют одинаковый размер. Строчка 11 – definition для класса `A` и ее уже достаточно, чтобы создать объект этого класса, мы точно знаем какой размер должен занять этот объект. Обратите внимание, что объявление не статической переменной на строке 13 является definition, потому что позволяет полностью определить объект с именем `y` и работать с ним, хранить его в памяти и брать его адрес.

Как мы видели выше на модельном примере, модель компиляции плюсов построена вокруг идеи, что во время компиляции функций, если мы знаем ее определение (definition), то мы его добавляем в бинарник, а если не знаем, то добавляем в бинарник точку вызова этой функции. Главное – надо знать, что имя действительно было именем функции. Для этого мы делаем `#include` header-а содержащего declaration для имени функции. А тело функции будет компилировать в отдельной единице трансляции. И уже задача линкера найти нужное тело функции.

3.1.2 Сборка проекта

Как мы видим в выбранной модели компиляции единицей компиляции является файл. Это очень архаичная система, которая не позволяет внедрить многие фичи, доступные современным языкам программирования. В основном они связаны с тем, что у нас не может быть никакой информации между разными единицами трансляции.

Системы сборки Для управления сборкой проекта нужна система, которая отслеживает изменился ли файл с исходным кодом для проверки нужно ли его перекомпилировать. И если нужно, то перекомпилировать и перелинковать лишь те части проекта, которые необходимо. Такие системы называются системами сборки. Правила хорошего тона использовать систему, которая собирает весь проект по нажатию одной кнопки. Вот примеры систем сборки (это далеко не все примеры):

1. cmake – считается стандартной системой сборки. Постоянно развивается и большинство проектов поддерживают именно ее.

2. qmake – специальная система сборки для экосистемы Qt. Она поддерживает дополнительные стадии компиляции с кодогенерацией для Qt специфичного служебного кода.

3. meson – основанная на Python система сборки. Очень удобна в использовании, так как не требует изучения специального синтаксиса системы сборки.

Виды сборки Важно понимать, что существуют два основных режима сборки:

1. Debug. Сборка без оптимизаций и с дополнительной информацией и проверками при исполнении. Медленная в исполнении, но более удобна для отлавливания bug-ов при исполнении. Обычно используется вместе с Debugger-ом. Debugger – специальная программа, которая следит за состоянием всех переменных программы при исполнении и стеком вызываемых функций. Позволяет пошагово исполнять программу и отслеживать ошибки исполнения.

2. Release. Оптимизированная сборка без дополнительной информации в run-time. Используется для релизных версий программных продуктов.

Обе сборки выполняются компилятором и линковщиком. Отличие только в используемых ключах для компилятора и линковщика. Убедитесь, что ваша система сборки позволяет собрать проект в одном из двух режимах и от вас требуется лишь указать один параметр – вид сборки. Запомните, все настройки должны быть автоматизированы. Вся рутинная работа должна быть поручена компьютеру. Не надо руками консольными командами собирать проект с указанием явных ключей. Даже написание своих make файлов[4] – это плохая практика. Такие проекты тяжело поддерживать. Еще бывает выделяют другие виды сборок. Например сборщик с профилировщиком (специальная программа отслеживающая какой процент времени исполнения тратится на какие части кода). Такая сборка хорошая для тестирования боттлнеков в производительности вашего приложения.

Есть еще один интересный момент в сборке проектов. Есть так называемая кросс-компиляция. Это когда вы собираете на windows приложение для linux. Современные компиляторы и системы сборки позволяют делать и такое. Кроме того, ваше приложение может быть мультиплатформенным в том смысле, что его можно собрать под разные операционные системы. Как такое организовывать, я расскажу ниже.

Роли файлов При сборке любого проекта на C++ все файлы с кодом обычно делятся на две категории:

1. Заголовочные файлы (headers) и имеют расширение h (или hpp).

2. Файлы с кодом (sources) и имеют расширение cpp.

Заголовочные файлы используются для копипасты их текста внутрь других h или cpp файлов. Их задача – снабдить компилятор всей необходимой информацией о символах, чтобы он мог скомпилировать код в бинарный файл. Файлы с кодом или cpp файлы используются как единицы трансляции, то есть их передают компилятору на первой стадии для генерации бинарного кода. Потому вам НИКОГДА НЕ надо отдавать h файлы компилятору.

3.1.3 Использование сторонних библиотек

Сторонние библиотеки бывают следующих типов

1. Header-only библиотеки состоят только из заголовочных h файлов и не содержат бинарных файлов.

2. Библиотека из h и cpp файлов. Подключение таких библиотек проста, вам просто надо трактовать файлы библиотеки как файлы проекта. Использовать h файлы для подключения в другие h и cpp файлы. А cpp файлы надо скомпилировать и потом полученный код слинковать со своим проектом.

3. Библиотека из h файлов и файлов бинарного кода вместо (cpp). Такие библиотеки за вас выполнили компиляцию cpp файлов. Потому вам не надо выполнять ее на вашей машине. Потому вы лишь подключаете h файлы библиотеки куда надо и линкуетесь с бинарниками библиотеки. Надо понимать, что бинарные файлы от разных компиляторов (и даже разных версий компиляторов или от компиляторов одной версии, но с разными флагами компиляции) могут быть НЕ совместимы. Потому вам надо еще найти версию библиотеки именно под ваш конкретный компилятор.

Откуда брать библиотеки? Если во всех современных языках программирования есть своя экосистема вокруг языка, где кроме компилятора у вас есть еще куча библиотек и пакетный менеджер для управления и подключения библиотек, то в C++ ничего такого из коробки нет. Потому приходится опираться на сторонние решения. Есть следующие способы менеджерить сторонние библиотеки:

1. Пакетный менеджер. Это специальные приложения, задача которых автоматизировать подключение сторонних библиотек в ваш проект. Нет какого-то единого пакетного менеджера, которым пользуются все. Но есть несколько распространенных пакетных менеджеров:

[4] Если вы вдруг знаете, что это такое.

(a) conan

(b) vcpkg

Обычно пакетный менеджер имеет свой удаленный репозиторий в интернете, где находится база данных всех поддерживаемых библиотек. Вы должны добавить в проект файл в специальном формате с указанием деталей о требуемых библиотеках и в систему сборки добавить соответствующие шаги по подключение библиотек указанных в таком файле. Такой подход позволяет вам избавиться от указания конкретных путей к библиотекам или необходимости их ручной установки. Пакетный менеджер выполнит всю работу сам.

2. Управление библиотеками с помощью системы сборки. Разные системы сборки имеют функционал покрывающий функции пакетного менеджера. Например cmake способен выполнять функции пакетного менеджера.

3. Использование git submodule. Это позволяет включить стороннюю библиотеку в ваш git репозиторий без копирования ее кода. Вам все равно придется настраивать сборки библиотеки у себя локально. Но использование систем сборки вроде cmake обычно автоматизирует и эту часть процесса. А для header-only библиотек это вообще супер простое решение.

3.1.4 Шаблоны и модель компиляции плюсов

А теперь добро пожаловать в мир интересных интересностей – шаблоны.

Что такое шаблоны Прежде всего отмечу, что бывают шаблоны функций, классов и переменных.

```
1 template<class T>
2 void f(T);
3
4 template<class T>
5 int x;
6
7 template<class T>
8 class A;
```

Самое важное, что надо знать про шаблоны: шаблоны – это не объекты, это правила по которому надо создать объект. Например шаблон функции – это не функция, это правило с параметрами, как создать конкретную функцию, когда вы в него подставите все значения параметров. Аналогично с шаблоном класса – это не класс, это правило как собрать класс, когда вы подставите все параметры. Шаблон переменной тоже не исключение и говорит как собрать переменную со своим значением параметров. Важно понимать, что получающиеся объекты (функции, классы и перменные) определяются именем шаблона и значением параметров, а не только именем шаблона. Шаблон – это лишь рецепт, как собрать.

У шаблонов так же есть declarations и definitions. Declarations лишь говорят, что данное имя является правилом, как собрать объект. Например в строчках 1-2 выше имя `f` является именем для правила собрать функцию. У этого правила есть один параметр `T` и он параметризует тип аргумента. Однако, это не definition, мы не знаем, как именно собрать функцию по этому правил. Вот пример определений

```
1 template<class T>
2 void f(T) {}
3
4 template<class T>
5 int x;
6
7 template<class T>
8 class A {};
```

Как я писал выше для переменных это уже будет definition, потому что шаблон точно знает, что надо собрать переменную типа `int`.

Как компилируются шаблоны Так как шаблоны – это не объекты (не функции, не классы, не переменные), то они не дают кода в бинарниках, проще говоря они просто так не компилируются. Например

```cpp
// file.h

template<class T>
T f(T t) {
  return t;
}
```

При компиляции compile `file.h` to `file.bin` мы получим пустой бинарный файл `file.bin`. Шаблоны компилируются, только если кто-то затребовал создать объект по данному шаблону с конкретными параметрами! Например,

```cpp
// file.h

template<class T>
T f(T t) {
  return t;
}

// main.cpp

#include "file.h"

int main() {
  int x = f(2);
  return 0;
}
```

Если мы теперь запустим компиляцию compile `main.cpp` to `main.bin`, то произойдет следующее. Компилятор дойдет до строчки 13 и увидит, что используется имя `f`, которое является шаблоном функции. По-хорошему, мы должны были указать `f<int>(2)`, чтобы сказать с каким параметром мы строим функцию. Однако, компилятор для функции умеет определять тип параметров. Здесь передается константа 2, которая по умолчанию имеет тип `int`, а значит компилятор определит тип `T = int`. В этот момент, когда компилятор понял, что ему надо использовать `f<int>`, он налету создаст код для этой функции поместит его в бинарник:[5]

```cpp
// main.bin

main:
  int x = call f<int>(2);
  return 0;

f<int>(int t):
  return t;
```

Теперь самое интересное, нельзя разбивать declaration и definition для шаблона на header и source файлы. Давайте я объясню почему. Предположим у нас есть:

```cpp
// file.h

template<class T>
T f(T t);

// file.cpp

#include "file.h"

template<class T>
T f(T t) {
  return t;
}

// main.cpp

#include "file.h"

int main() {
  int x = f(2);
```

[5]Не забываю напоминать дотошных формалистов, что мне глубоко плевать на конкретный синтаксис конкретного ассемблера и на детали имплементации бинарного кода.

```
21    return 0;
22  }
```

Теперь проведем компиляцию обеих единиц трансляции. Когда мы выполним compile `file.cpp` to `file.bin`, то создастся единица трансляции:

```
1  // TU for file.cpp
2
3  template<class T>
4  T f(T t);
5
6  template<class T>
7  T f(T t) {
8    return t;
9  }
```

После чего компилятор видит, что тут одни шаблоны и создает пустой бинарник, ибо ни один шаблон не был затребован для построения конкретной функции. Далее мы делаем compile `main.cpp` to `main.bin` и получаем единицу трансляции

```
1  // TU for main.cpp
2
3  template<class T>
4  T f(T t);
5
6  int main() {
7    int x = f(2);
8    return 0;
9  }
```

Теперь, когда компилятор доходит до строчки 7, он видит, что тут `f` – это имя шаблон и он восстанавливает, что нам нужно построить функцию `f<int>`. Однако, компилятор не видит definition для шаблона и потому не может тут налету сгенерировать код для функции `f<int>`. Но это не страшно, компилятор умный, он знает, что программист тоже не дурак и специально тут вызывает функцию, которая будет определена где-то еще, и генерирует бинарник

```
1  // main.bin
2
3  main:
4    int x = f<int>(2);
5    return 0;
```

А теперь самое интересно, надо линковать. Запускаем Link `file.bin` and `main.bin` to `main.exe`. Линкер получает на вход

```
1  // file.bin
2
3  // main.bin
4
5  main:
6    int x = f<int>(2);
7    return 0;
```

Теперь линкер начинает с функции `main` идет по ее телу и добавляет в `main.exe` код всех функций, которые `main` вызывает. Встречает `f<int>` и понимает, что ни в одном бинарнике для нее нет кода. Это ошибка линковки. Привет. Именно по этой причине шаблоны всегда пишут в header файле. Более того, их часто определяют там же, где объявляют.

Шаблоны и время компиляции О да, шаблоны любят поднасрать в длительность компиляции вашего проекта. Ведь вы же каждый раз как встретили шаблон, должны сгенерировать для него код. Можно надеяться, что компиляторы будут кэшировать информацию о встреченных шаблонах, но по-хорошему, компилятор видит только одну единицу трансляции за раз. Потому без дополнительных костылей разные единицы трансляции должны заново генерировать код для всех встреченных шаблонов. «Это не дело!» решили программисты из Microsoft и придумали precompiled headers. Идея их вот в чем. Создается единый файл *.pch, в котором компилятор хранит какое-то внутреннее представление для шаблонов для их быстрой генерации. Получается такой не маленький файл. Обычно туда отправляют файлы из STL или других внешних библиотек,

которые вы не будете менять ибо при любых изменениях pch файл надо перекомпилировать, а это долго. Но если вы его один раз создали, то можете потом быстро генерировать шаблоны, которые в нем учтены. Более того, таких pch файлов можно делать несколько и подключать к сборке только нужные из них в нужный момент. Вот такая вот технология. Я знаю, что gcc тоже умеет делать что-то подобное, не удивлюсь, что и clang умеет, но врать не буду.

3.1.5 Правила организации кода и пространств имен

Запомните, никогда ни при каких условиях не пишите в global scope. Выберете для проекта namespace и весь ваш код должен быть целиком и полностью внутри этого namespace. Для лучшей грануляции вы можете выделить еще несколько namespace внутри. Например

```
// Project.h

namespace Project {
struct A {
  void g();
};

namespace Impl {
void f();
} // namespace Impl
} // namespace Project

// Project.cpp

namespace Project {
void A::g() {}

namespace Impl {
void f() {}
} // namespace Impl
} // namespace Project

// main.cpp

#include "Project.h"

int main() {
  namespace pj = Project;
  pj::A x;
  x.g();
  pj::Impl::f();
  return 0;
}
```

Кроме того, никогда и ни при каких условиях не пишите в пространство имен `std`. Может быть встретятся ситуации с хэш функциями, когда вам придется так сделать, но все равно не делайте так. Превознемогайте.

Тонкости с cpp файлами Есть еще пара тонкостей критичных и нет. Начнем с критичных вещей. Давайте напомним страшную тайну про линкер – у него только глобальные имена и линкер не любит видеть какое-то имя в количестве отличном от одного (кроме определенных случаев). Вот пример, когда такая проблема может появиться

```
// a.cpp
void f() {}

// b.cpp
void f() {}
```

Если вы скомпилируете эти два файла в бинарники, то в каждом из них будет лежать функция с именем `f` и когда линкер попытается залинковать эту функцию, то он увидит два определения и ругнется. Такая ошибка встречается, когда внутри cpp файла вам нужна вспомогательная функция и вы случайно даете вспомогательной функции одинаковое название (и одинаковую сигнатуру). Эта проблема лечится таким образом. Все определения, которые выделаете в cpp файлах надо делать static или (что более удобно и равносильно для функций) помещать в anonymous namespace:

```cpp
// a.cpp
static void f() {}

// b.cpp
namespace {
void f() {}
}
```

В этом случае создается уникальное для единицы трансляции имя для namespace и потом только в этой единице трансляции выполняется `using namespace`. Но когда имена помещены в namespace компилятор дает им специальные имена, чтобы линкер мог их различать.

Подобная проблема будет и с переменными и классами в cpp файлах. Потому, когда вы определяете хоть что-либо в cpp файле, то обязательно помещайте весь код в anonymous namespace. Даже если это не глобальные имена, просто сделайте anonymous namespace внутри того пространства имен, внутри которого это понадобится. Разницы со словом `static` нет, кроме того, что классы нельзя объявить со словом `static` их придется помещать в namespace. Кроме того, в namespace можно помещать массово в отличие от `static`, которым надо помечать каждое имя.

Тонкости с h файлами Если вы создаете h файл, то это файл, которым будет кто-то пользоваться. Так бывает, что вам надо определить вспомогательные классы, которые пользователь видеть не должен и какие-то другие вспомогательные конструкции. Чтобы не засорять видимость пользователю, предлагается помещать этот код в специальное поле имен `detail` (или даже делать свое специальное имя для каждого случая). Вот пример

```cpp
namespace project {
namespace detail {
void auxiliary_function() {}

struct Proxy {};
}

class A {
  using Proxy = detail::Proxy;
public:
  Proxy get();
};
}
```

Такой подход позволяет вам контролировать, что видит пользователь и давать ему в руки только те инструменты, которые вы хотите раскрыть для него. И ему не придется копаться в кишках вашей имплементации. Не используйте в h файлах anonymous namespace. Это бесполезно и просто увеличивает количество имен в единицах трансляции. Кроме того, это может подгадить вашим глобальным ресурсам (потому что такой ресурс теперь в каждой единице трансляции станет локальным, а это скорее всего не то, что вы хотели.).

Бывают ситуации, когда вы можете спрятать вспомогательные типы внутри вашего класса в приватной части. Однако, вложенные определения классов в плюсах работают с некоторыми ограничениями, которые я не готов обсуждать прямо здесь и сейчас. Но могу сказать одно, бывают ситуации, когда вы просто не можете имплементировать то, что вы хотите с помощью вложенных классов. И тогда приходится делать классы не вложенными и так как это детали имплементации, то их хорошо бы спрятать внутри какого-нибудь пространства имен.

3.1.6 Дефекты языка C++

Тут я хочу просто перечислить некоторые важные моменты, которые важно иметь в виду при работе с языком и которые по сути являются страшными недостатками, которые невозможно обойти.

1. Синтаксис C++ сформировался поверх синтаксиса Си. И так как Си – это всего лишь высокоуровневый ассемблер без какой-то сверх сложной логики, то для него этот синтаксис не вызывает проблем. Хотя даже у него есть Однако, в случае C++ усложнение синтаксиса привело к любопытным дефектам. Например, программу просто невозможно однозначно распарсить. Вот мои два любимых примера:

   (a) Most vexing parse.

```
1  void f(double x) {
2    int y(int(x));
3  }
```

Вот тут строка 2 может трактоваться одним из двух способов:

Переменной `y` присваиваем значение `x`, которое кастуется из `double` к `int`:

```
int y = int(x);
```

Объявлена функция `y` с единственным аргументом `x`.

```
int y(int x);
```

(b) Привет от синтаксиса шаблонов.

```
1  a < b, c > d;
```

Вот эту строчку тоже можно распарсить двумя разными способами.

Вначале выполняются два сравнения, а потом оператор запятая, который вычисляет оба выражения слева направо, а потом откидывает результат первого и возвращает результат второго.

```
(a < b), (c > d);
```

Это объявлена переменная `d` шаблонного типа `a` с параметрами `b` и `c`.

```
a<b, c> d;
```

Локально глядя на код эти две ситуации не возможно отличить. Нужно знать контекст. И все усложняется, если этот код внутри шаблонного кода, где все имена – зависимые параметры.

2. Немного о препроцессоре. Напомним, что у плюсов архаичная система сборки, где единица компиляции – файл, а метод работы с файлами – это текстовый препроцессор и его директивами. Это значит, что перед тем как код вашей программы попадет компилятору его официально предобрабатывает как текст некая другая программа – препроцессор. И это может приводить к совершенно непредсказуемым результатам. Вы будете получать ошибки компилятора, который ругается на код, который вы даже не увидите в программе. Мой любимый пример, с которым я сам сталкивался, когда собирал одну из сторонних библиотек из исходников. При сборке проекта по официальной инструкции получаю ошибку компиляции, где компилятор ругается на строчку кода, которой нет в проекте. После нескольких дней гуглежки нашел решение: «Нужно зайти в файл некой сторонней библиотеки (не той, которую я собираю, а ту которая используется той, которую я собираю), найти там некую строчку и заменить имя переменной `small` на `little`.» Я когда нашел это, то просто не поверил, что сработает. И как же я был удивлен, когда после этой правки проект собрался. Оказывается, официальная версия библиотеки тестировалась на сборку только под Linux. А под Windows она включала один из заголовочных файлов операционной системы Windows. А Windows просто обожает вводить новые макросы среди которых был такой:

```
#define small short
```

И конечно же этот заголовок был подключен до использование сторонней библиотеки, где слово `small` использовалось для имени переменной. И компилятор выдавал ошибку на строчку вида

```
int short = 0;
```

Которой конечно же не было в исходном коде.

3. Шаблоны ломают систему сборки плюсов. Тут хочется сказать сразу две вещи:

   (a) Шаблоны ломают систему компиляции плюсов. Как мы разобрали выше, невозможно делить код шаблонов на заголовочный и имплементацию.

   (b) Язык шаблонов полный. А это значит, что на них можно написать любую программу, которая будет исполняться в compile-time. Например я могу написать какой-нибудь сервер, который будет крутиться в бесконечном цикле и выполнять обработку запросов. А это значит, что компилятор не может знать заранее остановится компиляция программы или нет, потому что это неразрешимая задача. Это не так страшно, и решается time лимитами, но все же неприятно.

4. Важно отметить, что при разработке C++ нужно было усидеть на двух стульях. С одной стороны – это низкоуровневый язык наравне с Си, с другой – все правила языка определены аксиоматически и абстрактно независимо от имплементации. Но хуже, что многие важные вещи не были стандартизированы. Вот два важных примера:

   (a) STL. Стандартная библиотека не имеет стандартной имплементации. Она лишь фиксирует требуемое поведение и интерфейсы. А это значит, что код разных STL просто не совместим на бинарном уровне. Потому что одна и та же структура данных может просто по разному лежать в памяти и занимать разное количество байт.

   (b) Перегрузка функций и операторов а так же введение пространства имен породило еще одну большую беду – разрешение конфликта имен для линковщика. Давайте напомним, что линковщик работает с бинарными файлами, в которых все имена глобальные. Потому если мы имеем функции с одинаковыми именами в коде

```
1 void f(int);
2 void f(int, double);
3 namespace A {
4 int f(char);
5 }
```

   то нельзя их имена передавать линковщику как есть. У него не будет способа их отличить. По простому к имени функции добавляются имена всех scope, где она содержится, и типы всех аргументов. Эта процедура называется Name Mangling. Потому линковщик будет видеть три разных названия вместо этих имен. И как вы понимаете, при сообщении об ошибке о функции на стороне линковщика, вы не увидите привычного имени `f`, вы увидите какую-то мешанину из символов, по которой надо будет догадаться, что это за функция. А самое классное, что сама процедура изменения имени тоже не стандартизирована. Каждый компилятор использует свою. У msvc своя закрытая имплементация (у которой правда есть reverse engineering версия), а у gcc используется Itanium.

3.2 Си и C++

Дизайн языка C++ изначально планировал, что любая программа на Си должна компилироваться компилятором C++.[6] Однако, если вы когда-нибудь настраивали сборку проектов, то заметили, что вы всегда указываете два компилятора – один для Си, другой для C++. Возникает вопрос: а зачем? Если C++ компилятор может компилировать все. И тут мы приходим в суровый мир продуманного Си, как низкоуровнего языка, и прекрасный мир стихийного развития C++ поверх Си.

3.2.1 ABI совместимость и Name Mangling

Одна из основных идей за двухступенчатой системой компиляции – модульная структура приложений на уровне бинарного кода. А именно, мы будем хранить приложение не в виде одного бинарного файла, в виде нескольких бинарных файлов. И если нам надо обновить какую-то компоненту, то мы можем перекомпилировать только нужный бинарник и подменить его. Но для такого подхода важно, чтобы на уровне бинарного кода старая и новая имплементация были совместимы. Такая совместимость в частности предполагает одни и те же имена функций. И потому, если у вас на уровне бинарного кода есть Name Mangling (смотри раздел 3.1.6), который не стандартизирован, то вы просто не можете использовать язык C++ в рамках такой парадигмы.

Давайте вернемся к вопросу зачем нам два компилятора для Си и C++ кода. Если вы имеете код на Си, то он конечно скомпилируется обоими компиляторами. Но бинарный файл будет разные. В случае компилятора Си имена методов будут такими же как их видит программист, а в случае C++ имена будут подвергнуты Name Mangling-у. Подобная проблема в частности не позволяет слинковаться с методами скомпилированными с помощью Си компилятора. Действительно

```
1 // c-file
2 void f(int) {
3 ...
4 }
```

[6] И это почти работает. Точнее есть одно ключевое слово из Си, которое не поддерживается компилятором C++ – это слово `restrict`. Давайте я не буду ничего про него говорить.

```
5  // c++ file
6  void f(int);
7
8  int main() {
9    f(5); // linking error
10   return 0;
11 }
```

Если мы скомпилировали первый файл с помощью Си компилятора на на уровне бинарника имя функции будет `f`. А при компиляции второго файла с помощью C++ компилятора имя объявленной в строчке 6 и вызываемой в строчке 9 функции будет совсем другой, условно пусть будет `_Zfint`. А тогда линковщик не найдет нужную функцию. Для решения этой проблемы есть специальное ключевое слово:

```
1  // c-file
2  void f(int) {
3  ...
4  }
5  // c++ file
6  extern "C" void f(int);
7
8  int main() {
9    f(5); // OK
10   return 0;
11 }
```

Теперь для функции `f` в строчке 6 отключен Name Mangling и линковщик сможет понять какое у нее имя в бинарнике после Си компилятора.

3.2.2 ABI совместимость и отсутствие стандартизации

Если вы думали, что Name Mangling – это единственная причина для отсутствия ABI совместимости, то у меня для вас плохие новости. На этом список не заканчивается. Если вы хотите ABI совместимость, то вы автоматически не можете использовать STL, даже для одно и того же компилятора, потому что имплементация STL не фиксирована на уровне бинарников. И есть известная история со строками в STL для gcc. Изначальная имплементация строк имела copy-on-write оптимизацию, которую потом насильно запретили. И потому теперь есть две несовместимые на уровне бинарника имплементации STL для gcc, между которыми нужно переключаться специальным макросом. Но и это не самая большая проблема. Даже внутри самого языка ничего не стандартизировано. Вот перечень проблем:

1. Exception (смотри раздел 4). Данный механизм языка имплементирован по разному на разных компиляторах. При этом на msvc есть еще возможность переключаться между разными имплементациями исключений по флагу компилятора.

2. RTTI (смотри раздел 4). Имплементация этой фичи тоже не стандартизирована и на разных компиляторах имплементирована по разному.

3. Наследование с виртуальными методами (смотри раздел 4). Имплементация так называемых виртуальных методов также не стандартизирована. Где и как располагается таблица виртуальных функций, какого она размера и прочие и прочие детали не стандартизированы. Да и вообще, что должна быть использована именно таблица для виртуальных методов никак в стандарте не указано (хотя все имплементации используют только такой подход). Тем не менее, конкретное расположение в памяти объектов такого класса может отличаться для разных компиляторов.

4. C-run-time. Если вы изучали АКОС,[7] то знаете, что точка входа в программу – функция start, а не main. Дело в том, что для работы программы на C++ нужны некоторые возможности, которые гарантируются языком, но не операционной системой. Например использование операторов `new` и `delete`. Эти операции зовут библиотечный код относящийся к так называемому C-run-time-у. И конечно же его имплементация не стандартизирована. И у разных компиляторов эти методы работают по-разному и не совместимы.

Все указанные выше проблемы практически уничтожают возможность распространения кода для C++ проектов на уровне бинарников. Потому подключение сторонних библиотек почти всегда означает сборку библиотеки из исходников для своего проекта, либо использовать пакетный менеджер и надеяться, что ваша

[7] Архитектура компьютера и операционной системы

конфигурация найдется и не надо будет собирать все с нуля. Под Windows из-за распространенности компилятора Visual Studio (msvc) вопросы со стандартизации на уровне бинарников в некоторой степени решаются сильно проще, но все же привязаны к версии компилятора.

4 Опасные механизмы языка

Когда у вас в руках очень мощный язык, который умеет делать все и любым образом очень важно отличать какие его механизмы для каких нужд были созданы. Это помогает понять, когда какими механизмами надо и не надо пользоваться. Один из способов смотреть на возможности языка – разделить их на опасные и безопасные. И тогда важно придерживаться следующего правила:

- Безопасными механизмами языка можно пользоваться всегда
- Опасными механизмами языка можно пользоваться только для имплементации какого-то паттерна или структуры данных, который снаружи имеет безопасный интерфейс для использования. Опасные механизмы – это ТОЛЬКО библиотечный код. НИКОГДА не использовать их в бизнес логике.

Стоит отметить, что в языке Rust даже есть ключевое слово `unsafe`, которое показывает, что вы начинаете пользоваться небезопасными механизмами. В языке Rust понятие безопасного и не безопасного формализовано очень просто: если вы используете только безопасный код, то вы не получите undefined behavior. И если вы гарантируете, что ваш `unsafe` код не содержит undefined behavior, то и вся программа его не содержит. Более того, с безопасным кодом, компилятор Rust отдаст свою жизнь за то, чтобы не пропустить UB и прочтет вам лекцию по поводу каждой ошибки и предложит варианты их решения. В случае C++, к сожалению, вам приходится полагаться только на себя и свой опыт. Никакой помощи от компилятора вы не дождетесь. Это осознанное решение комитета. А потому в C++ намного важнее, чем в Rust понимать, чем можно пользоваться, а чем нужно пользоваться осторожно и только в особых случаях. Если вы себя не защитите от использования небезопасного языка, то вас ничто не спасет. А это плохая тактика работы с кодом.

Давайте я начну с перечисления механизмов, которые я бы считал не безопасными, и которым нет места в повседневном коде. А потом я обсужу пару примеров и прокомментирую, как это все должно работать. Список небезопасных механизмов языка:

1. `goto`
2. `new`/`delete`, `fopen`/`fclose`, `malloc`/`free` и любые другие парные операции захвата и освобождения ресурса (не обязательно памяти)
3. `cast`-s, в особенности `const_cast`
4. Exceptions
5. friend
6. mutable
7. Lock примитивы
8. `shared_ptr` (в особенности на неконстантный объект)
9. Глобальные переменные
10. Наследование с виртуальными методами
11. Итераторы, ссылки и указатели в структурах (НЕ в алгоритмах)
12. RTTI

Давайте разберем эти примеры. Я на примере `goto` обсужу важные философские моменты, потому первый пункт прочитайте обязательно:

1. `goto`. Я думаю, вы уже ни раз слышали, что это главное зло всех программистов и его не надо ни в коем случае никогда и ни при каких условиях использовать. Это стало понятно уже больше 50 лет назад еще до создания языка C++. Тем не менее `goto` присутствует в языке и доступен программисту. Проблема с этой командой в том, что вы можете прыгнуть внутри функции из любой строки кода в любую, при этом логика исполнения становится очень запутанной. Очень тяжело отследить состояние программы и что и как может измениться при таких прыжках. Потому эту команду не надо использовать никогда.

   После такого вступления должен возникнуть разумный вопрос: а зачем тогда вообще нужна `goto`? И ответ очень простой: вообще без этой команды написать разумную программу не получится. Любые

условные переходы по `if`/`else`, циклы `while` или `for` используют внутри себя `goto`. И без нее их просто не возможно имплементировать. Именно эта причина и порождает необходимость использования `goto`. Теперь возникает следующий вопрос: почему `goto` внутри условных переходов и циклов – это хорошо, а в коде программы – плохо? И ответ опять очень простой: дело в том, что внутри условного перехода по `if`/`else` или внутри циклов `while` или `for` команда `goto` используется в очень ограниченном наборе сценариев. И потому их можно просто все перебрать и убедиться, что при любом исполнении программы блок, содержащий `goto` будет работать корректно, как и полагается. Именно это ограниченное использование позволяет не просто поверить, а именно проверить (доказать), что программа будет корректной. Если же использовать эту команду в бизнес логике приложения, то сделать такой перебор просто не представляется возможным. Кроме того, снаружи и условные переходы по `if`/`else` и циклы `while` или `for` являются безопасными механизмами. Они не несут в себе проблем, которые несет `goto`.[8]

2. `new`/`delete`, `fopen`/`fclose`, `malloc`/`free` и любые другие парные операции захвата и освобождения ресурса (не обязательно памяти). Очевидная причина почему эти операции опасны – они идут в паре и надо не забыть освободить ресурс и не освободить его дважды. И проблема тут не только в дисциплинированности программиста, а так же в том, что в C++ есть исключения. А потому вы можете выйти из scope в любой точке кода. Давайте рассмотрим пример

```
1 void f() {
2   A* ptr = new A();
3   g(ptr);
4   delete ptr;
5 }
```

Если функция `g` может кинуть исключение, то наличие `delete` в 4 строчке не поможет. Мы выйдем из функции `f` раньше. Потому такие парные операции всегда надо запаковывать в RAII (смотри 6.3.15) обертки:

```
1 void f() {
2   std::unique_ptr<A> ptr = std::make_unique<A>();
3   g(ptr.get());
4 } // ptr destructor is called automatically
```

Теперь деструктор `ptr` вызовется при любом выходе из функции `f` и при нормальном завершении работы и при брошенном исключении.

Давайте я на всякий случай проговорю важную вещь про `new`/`delete`. Они конечно же используются внутри всех динамических контейнеров в STL, иначе вы просто не сможете выделить память под него. Тогда почему внутри контейнера их использовать хорошо, а вне плохо? Потому что внутри контейнера они используются в ограниченном количестве сценариев таком маленьком, что их все можно перебрать руками и доказать, что ваш контейнер работает корректно и снаружи для использования безопасен. Вот эта мысль: опасно внутри, но там мы все проверили, а снаружи безопасно – это главная мысль при использовании опасных механизмов языка. Тут мысль такая же как и при описании `goto`, но я решил, что все равно стоит произнести эти слова еще раз.

3. `cast`-s, в особенности `const_cast`. Касты это очень низкоуровневый механизм. С помощью них можно легко получить UB и не заметить этого. А `const_cast` вообще был создан только для того, чтобы делать UB и никаких других функций у него нет. Потому касты должны быть использованы внутри каких-то паттернов или хотя бы вспомогательных функций (которые снаружи работают безопасно). Хороший пример – CRTP (Curiously Recursive Template Pattern). Про этот шаблон я расскажу позже в разделе **??**.

4. Exceptions. С исключениями есть несколько проблем

   (a) Из-за исключений брошенных в любой точке программы ее исполнение может обвалиться в любой момент времени в любом месте и вы ничего с этим сделать не сможете.

   (b) Бросание исключения при уже висящем исключении приводит к завершению программы по terminate и вы ничего с этим сделать не сможете.

[8] Хорошо держать в голове аналогию: есть нож для хлеба, а есть циркулярная пила. Вы не используете циркулярную пилу, чтобы намазывать масло на хлеб по утрам, но это не значит, что она не нужна. Есть задачи, которые без нее не сделать, где обычный нож не подойдет. Но когда вы работаете с циркулярной пилой, у вас есть огромное количество техник безопасности и правил, тогда как при использовании ножа, вы даже не задумываетесь об этом.

(c) Не все фреймворки поддерживают исключения, например Qt не поддерживает.

(d) При бросании исключений нужно выходить из функций, чьи данные лежат на стеке. А значит надо чистить эти данные. Такой процесс называется stack unwinding. Существуют две версии имплементации исключений:

i. Имплементация gcc хранит для каждого scope, где может быть брошено исключение таблицу из всех деструкторов всех объектов, которые надо позвать. Что раздувает бинарники.

ii. Имплементация msvc параллельно в run-time поддерживает второй стек, куда складываются деструкторы для текущего стека. Это означает, что вы платите за использование исключений, даже когда они не кидаются.

5. friend. Данное ключевое слово нарушает инкапсюляцию данных в классе, то есть позволяет получить доступ к внутренностям класса. Это чревато тем, что вы можете через этот доступ сломать внутренние инварианты класса. Использовать же friend надо тогда, когда два класса шарят один общий инвариант.

   (a) В качестве примера можно привести блокирующий observer pattern (см. раздел 12.3). В этом случае observer и observable должны поддерживать друг на друга согласованные указатели и чтобы уметь восстанавливать этот инвариант, они должны друг у друга дергать не безопасные методы.

   (b) Еще один пример builder pattern (см. раздел **??**). Если сложно объект построить сразу одним конструктором (например потому что для этого надо указывать кучу данных в аргументах и это не удобно) слишком сложно. Тогда можно завести класс builder, который будет строить нужный класс по частям и внутри builder-а нужный класс может быть в некорректном состоянии и builder-у разрешено управлять его инвариантами.

6. mutable. Данное ключевое слово нарушает const correctness в прямом смысле слова. Оно позволяет добавить в класс поле, которое можно менять даже в const методах. Предполагается, что для этого ключевого слова есть два use-case-а:

   (a) Кэш. Действительно, даже при работе константного метода вам может понадобиться записать в специальное поле результат, чтобы потом не пересчитывать его. Однако, есть куда более простой способ. Вспомним, что const не пробрасывается сквозь указатели. А потому кэш можно хранить в `std::unique_ptr`. Либо получать доступ к внешнему сервису кэширования.

   (b) Lock примитивы. Если вам нужны lock примитивы, то вы скорее всего пишите какую-то известную структуру данных, про которую доказано, что она в такой имплементации будет работать корректно. Если это не так, то вы не знаете, что вы делаете. Проблема с хранением lock примитивов в классе лишает класс value semantics (см. раздел 5). Если вам нужна на таком классе синхронизация, возможно вам и не нужна value semantics. Но если нужна, то все так же можно положить все `std::unique_ptr`. Все что касается синхронизации и межпоточного взаимодействия – это отдельная болезненная тема.

7. Lock примитивы. Примитивы синхронизации – это слишком низкоуровневые элементы языка, чтобы они появлялись в бизнес логике. Кроме того, бездумное использование этих примитивов может легко подвесить вашу программу в deadlock или race condition.

8. `shared_ptr` (в особенности на неконстантный объект). Использование `shared_ptr` на неконстантный объект нарушает локальность. У вас может быть две переменные очень далеко друг от друга в коде, которые хранят указатель на один и тот же ресурс. И теперь любые изменения в одной точке программы, могут вызвать любые непредсказуемые изменения в сколь угодно далекой точке программы, про которую вы можете даже не подозревать. Такие связанные воедино данные и раскиданные по всему коду называются Incidental Data Structures. Есть всего два случая для использования `shared_ptr` на неконстантный объект. Их мы обсудим позже в разделе 10.3. Так же не забывайте, что `shared_ptr` из STL имеют скрытую синхронизацию счетчика ссылок, что будет вам стоить производительности всегда и вы не можете ее отключить, только если не напишете свою версию. Это печально.

   С другой стороны `shared_ptr` на константный объект – это дешевый способ сделать данные дешево копируемыми при условии их константности. Это не рассматривается, как опасный механизм языка (потому что он обладает value semantics 5), но даже в этих условиях надо прятать `shared_ptr` внутрь шаблонного класса, потому что это лишь деталь имплементации. Одна из причин в том, что у `shared_ptr` скорее всего не тот интерфейс, который вы хотите иметь.

9. Глобальные переменные. Очевидная проблема с глобальными переменными та же, что и с `shared_ptr` – это создание Incidental Data Structures, то есть данных, которые тесно связаны друг с другом и находятся в неожиданных и совершенно несвязанных друг с другом местах. Но кроме этого у глобальных переменных есть еще одна проблема, которая растет из модели компиляции C++. Так как компиляция происходит в две стадии: сначала компиляция каждой единицы трансляции, а потом линковка всех полученных бинарников в едино, то возникает вопрос: а в каком порядке запускаются конструкторы и деструкторы глобальных переменных? Внутри одной единицы трансляции они идут в том же порядке, в котором они написаны в коде. А вот между единицами трансляциями нет никаких гарантий. А это значит, что если вы хотите использовать глобальную переменную из другой единицы трансляции в конструкторе другой глобальной переменной, то она может быть еще не создана. А значит вы не можете к ней безопасно обратиться. Для таких ситуаций придуман Singleton pattern. Использование их не лучшая практика, но все таки есть ситуации, когда они нужны. Это мы обсудим отдельно в разделе **??**.

10. Наследование с виртуальными методами. Изначально наследование в языке – это лишь вид слияния интерфейсов, или другими словами «горизонтальная композиция» про это можно почитать в разделе 6.3.16. Однако, есть возможность унаследоваться от базового класса с виртуальными методами, что позволяет использовать вместо класса напрямую только интерфейс класса (базовый класс). На этом приеме обычно принято строить полиморфное поведение в run-time (что это такое можно посмотреть в разделе 11). Однако, у этого решения есть огромное количество недостатков и дефектов, которые делают код хуже как по читаемости, так и по поддерживаемости, так и по производительности. На данный момент есть куда более хорошее решение – стирающие типы (Type Erasure, разделы 11.4 и 11.5). К сожалению в C++ нет поддержки стирающих типов на уровне языка и в STL есть только три стирающих типа (точнее даже два с половиной). Детально проблемы с наследованием с виртуальными методами я буду говорить тут 11.3.

11. Итераторы, ссылки и указатели в структурах (НЕ в алгоритмах). Давайте начнем со ссылок. Причина почему они есть в языке – мы не хотим передавать в функцию тяжелые данные по копии. Вместо этого давайте передадим указатель на данные. И чтобы не возиться со случаем, что он может быть нулевой в языке есть отдельный вид указателя со специальным поведением, который имитирует исходный объект, и который называется ссылка. То есть при использовании ссылок, всегда предполагается, что мы можем гарантировать, что данные всегда живы во время работы ссылок. И для этого даже есть специальные правила языка по продлению life-time временных объектов, которые забиндились на ссылки (посмотрите конец раздела 5 про то, как это работает). Итераторы – это по сути более умные указатели, которые просто унифицируют работу с разными контейнерами и используются для передачи данных в алгоритмы. То есть ссылки, указатели и итераторы – это данные для передачи аргументов в методы. Они НЕ предназначены для длительного хранения, чтобы по ним ссылаться на другие объекты. Причина очень простая – объект по указанному адресу мог переехать или мог быть уничтожен. Потому, если только вы не пишите свой итератор, использовать ссылки или указатели для постоянного хранения – плохая идея.

    Есть отдельный специальный случай – Си-шные строковые константы. Они хранятся по `const char*`. И единственная причина почему их безопасно использовать – такие строковые константы создаются и инициализируются до запуска любой программы и в течение времени работы программы лежат в одной и той же области памяти. Потому ссылаться на них по константному указателю имеет value semantics (что это такое смотри в разделе 5), а значит безопасно. Но это касается только констант, а не создаваемых в run-time новых строк.

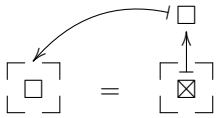
12. RTTI (Run Time Type Information). Это механизм языка, который позволяет во время исполнения программы работать с информацией о типе объекта. На мой взгляд эта фича языка является ошибкой дизайна и крутится вокруг проблем связанных с наследованием с виртуальными методами, иерархиями интерфейсов и прочей хренью, которых не должно существовать в коде. Если мы используем язык со статической типизацией, давайте мы не будем ее ломать. А если вам нужно полиморфное поведение, то стирающие типы решают эту задачу в run-time лучше, чем любое другое кустарное решение.

5 Семантика языка

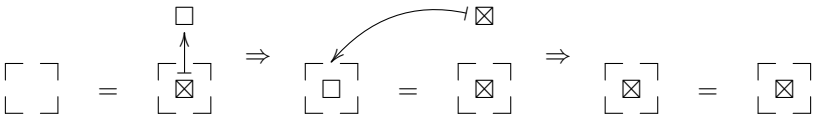
Вопрос семантики – вопрос как интерпретировать присваивание в языке. Есть два подхода:

1. Value Semantics.
2. Reference Semantics.

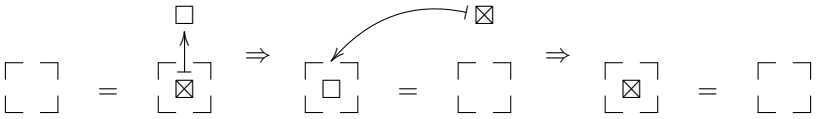
Давайте начнем с первой. В ней надо думать про переменные, как стаканы содержащие данные. А присваивание – это перекладывание данных.



И тут есть две вариации. Данные могут копироваться. Например, модель присваивания C++



Или же данные могут перемещаться. Например, модель присваивания Rust



Важное свойство такого подхода – изменение данных в одном стакане никак не может повлиять на данные в другом. Для элементарных типов такое поведение достигается тем, что они хранятся в разных областях памяти. Однако, value semantics не означает, что это всегда так. Например `shared_ptr` на константный объект, тоже обладает value semantics. Подумайте про работу с целыми числами в Python. Вы ни за что не отличите ее от работы с целыми числами в C++. Вы можете думать про такие переменные как стаканы с данными и все будет работать. Причина в том, что в Python целые числа всегда константы и присваивание просто перевешивает ссылку на объект с новым значением, не меняя старого. Вы не можете вызвать ни одного метода у целых чисел, который бы их изменил.

В языках вроде Python, Java, C# и прочих переменные – это имена указывающие на данные, потому присваивание интерпретируется по-другому. Присваивание $x = y$ означает: «называй именем x то, что называется именем y». Модель присваивания в Python можно представлять себе так:

Очень важно понимать, что value semantics safe by default. А вот reference semantics – контр интуитивная штука.

Тест на интуицию Предположим, нам дана матрица

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

И пусть у нас выполнен следующий код

```
MatA = ...
# = A

print(Determinant(MatA))
print(Determinant(MatA))
```

Что выдаст эта программа? Наша интуиция подсказывает, что должно быть $-1, -1$. Однако, программа выдаст $-1, 1$. Какая ваша первая мысль? Наверное есть баг в `Determinant`. Каково же будет ваше удивление, когда вы проверите и убедитесь, что `Determinant` работает полностью корректно. Проблема в том, что `Determinant` изменяет матрицу `MatA`, когда вычисляет определитель гауссом. Из-за этого во втором вызове у нас будет лежать единичная матрица. Вот такая гадость. Проблема в том, что человеческому сознанию сложно мыслить ссылками. Мы мыслим значениями. Потому мы не ожидаем такого поведения в этом коде. И это главная проблема Python. У этого подхода есть и плюсы, но value semantics я все таки люблю больше. По сути, в reference semantics языках переменные – это `shared_ptr` на не обязательно константные объекты. И если вы читали выше раздел 4 пункт 8 то знаете, почему чем они опасны. Так же рекомендую почитать раздел 10.3 посвященный `shared_ptr`.

Давайте посмотрим с точки зрения семантики на следующие типы. Я их специально поделил на типы с value semantics и reference semantics:

Value Semantics

```
char x;
A x;

const char& x; // almost
const A& x; // almost

const char* x; // almost
const A* x; // almost

std::unique_ptr<char> x;
std::unique_ptr<A> x;

std::shared_ptr<const char> x;
std::shared_ptr<const A> x;
```

Reference Semantics

```


char& x;
A& x;

char* x;
A* x;




std::shared_ptr<char> x;
std::shared_ptr<A> x;
```

Обратите внимание, что константные ссылки и указатели помечены, что они почти являются value semantics. Дело тут вот в чем, если гарантируется, что в течение времени жизни ссылки или указателя объект будет жив, то тогда действительно мы имеем value semantics. Например такие гарантии есть если

1. Строковые константы

```
const char* msg = "Hellow World!";
```

   Глобальные константы живут в статической памяти в течение времени жизни всей программы и потому к ним безопасно получать доступ.

2. Передача аргументов в функцию по константной ссылке. У любого временного объекта, который подставляется вместо такого аргумента life-time продливается до конца вызова функции. Например

```
struct A {
  const int& value() const {
    return value;
  }
  int value;
};

void f(const int& value);

int main() {
  f(A().x); // is OK
  f(A().value()); // is OK
}
```

   Однако, стоит иметь в виду, что функции в этом плане очень особенные. И стоит помнить вот такой пример

```
struct A {
  const int& value() const {
    return value;
  }
  int value;
```

```
6  };
7
8  int main() {
9    const int& x = A().value; // is OK
10   std::cout << x << '\n'; // is OK
11   const int& y = A().value(); // is BAD
12   std::cout << y << '\n'; // y is dangling reference
13 }
```

Если мы говорим об обычных ссылках, то время жизни временной структуры продлевается, если мы биндим ссылку на поле этой структуры (например строки 1 и 2 выше). Однако, если же мы ссылку возвращаем каким-то опосредованным методом, например через метод (строка 3), то life-time для временного объекта не продолжается и в 4 строке у нас y является dangling reference, которая уже не ссылается на живой объект.

ВАЖНО Код должен быть написан в value semantics всегда. Тут полезно понимать, как переделать код со ссылками и указателями, чтобы он стал в value semantics и при этом не потерял в производительности и не стал жрать больше памяти, но этому надо будет научиться всего лишь один раз, а код будет лучше всегда.

6 Классы

6.1 Класс или структура

Технически в языке C++ класс и структура – это одна и та же сущность, но с разными дефолтными квалификаторами доступа. В структуре все публично, а в классе все приватно по умолчанию. Главный вопрос: когда использовать класс, а когда структуру.

- Структура создается когда вам нужно хранить данные рядом в одной переменной.

```
struct Pair {
  int x;
  int y;
};
```

  При этом у структуры НЕ должно быть методов в том числе конструкторов. Технически любая функция имеет доступ к полям структуры, потому нет необходимости делать методы структуры. Идейно мы думаем про структуру, как про пассивный объект, хранящий данные вместе и все. Все манипуляции над данными происходят снаружи. При этом никаких ограничений на данные внутри структуры нет.

- Класс создается, когда хранимые данные должны удовлетворять какому-то инварианту и мы не можем просто положить их в структуру, ибо инвариант будет нарушен.

```
class OrderedPair {
public:
  ...
private:
  // invariant min <= max
  int min;
  int max;
};
```

  Для сохранения инварианта данные класса делаются приватными (это называется инкапсюляция). А для взаимодействия с данными создаются публичные методы, задача которых поддерживать класс в корректном состоянии.

6.1.1 Замечания

1. Наличие инварианта не всегда означает, что вам нужно зашивать данные в класс. Точнее, надо, но вопрос в том на каком уровне. Давайте я приведу пример, на котором будет понятно, что я имею в виду. Предположим вам надо создать класс вектор из линейной алгебры и для хранения размера вектора вам нужно хранить положительное число.[9]

```
class Vector {
private:
  int size_;
  double* data_;
};
```

   Так вот, вопрос заключается в том какой тип должен быть у `size_`. В STL в `std::vector` используется беззнаковое `size_t` потому что все его значения допустимы для размера вектора. Казалось бы разумная идея, но позже добавляется метод `ssize()`, который возвращает размер в виде знакового типа. Давайте разберемся в логике этого явления. При создании класса `Vector` выше, он должен представлять вектор из линейной алгебры и для вектора у нас нет смысла в длине равной нулю или отрицательной. Это инвариант, который мы хотим гарантировать. И главный вопрос: кто должен за этот инвариант отвечать. И тут есть два решения:

   (a) За инвариант должен отвечать новый класс `PositiveInt`

   (b) За инвариант отвечает класс `Vector`

[9] Сразу скажу, если вам такое надо, то используйте `std::vector`, а не имплементируйте контейнер самостоятельно. Пример ниже лишь для наглядности.

Я утверждаю, что второй подход в данном случае лучше. Кроме того, второй подход позволяет хранить `size_` в виде знакового типа. Почему же второй подход на мой взгляд предпочтительнее:

(a) Для написания типа `PositiveInt` нам придется писать новый полноценный класс. Это лишняя работа. Кроме того, по умолчанию новый класс ничего не умеет, а значит всю его функциональность надо будет писать с нуля. И хороший вопрос, что этот класс должен уметь делать. Можно ли например вычитать из одного числа другое? Если можно, что делать при вычитании из меньшего большего? А если нельзя, что делать если хочется все же вычитать?

(b) Новый класс никак не совместим со встроенными типами. Нельзя прибавить к нему `int`, нельзя умножить на `int` и т.д. А это значит, везде, где раньше мы пользовались удобством встроенных типов, наличием операций над ними, нам придется все эти операции дописывать руками, что будет лишней работой. Вот это особенно плохо, потому что мы по сути выкидываем функциональность, которая нам нужна.

(c) Если мы используем `PositiveInt` для указания размера, то мы скорее всего будем аллоцировать память такого размера, а все библиотечные функции принимают сырые встроенные типы. Мы теряем удобство использования переменной нового отдельного типа.

Куда более правильный подход – сказать, что размер `size_` будет храниться внутри `Vector` как сырой знаковый тип, а условие, что `size_` обязана быть положительной – это инвариант класса `Vector`. Это позволит легко составлять любые арифметические выражения, используя `size_` и мы не потеряем в надежности.

2. Еще один пример. Предположим, вы работаете с геометрией и для этого используете библиотеку, которая предоставляет вам классы для матриц и векторов. И вам нужны ортогональные матрицы. Тогда вы можете создать новый класс ортогональных матриц

```
class OrthoMatrix {
public:
  ...
private:
  Lib::Matrix A_;
};
```

Это кажется разумным. Но, это крайне неудобно. Ваши ортогональные матрицы теперь тип незнакомый библиотеке, которой вы пользуетесь. Теперь эти матрицы не знают как умножаться на векторы, к ним не применимы алгоритмы библиотеки, весь функционал придется писать заново. Это крайне не удобно. Потому вводить такой класс – плохая идея. Если же вам нужны гарантии, что матрица будет ортогональная, то скорее всего это значит, что у вас есть другой класс, который хочет использовать эту матрицу и уже он будет поддерживать инвариант:

```
class Camera {
public:
  ...
private:
  // Camera keeps the invariant: LocalBasis_ is orthogonal matrix
  Lib::Matrix LocalBasis_;
  ...
};
```

6.2 Структуры

1. Пусть мы хотим создать структуру вида[10]

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

Не надо делать у структур конструкторы. Вместо этого для структур есть aggregate initialization. Это работает так

[10] Я сейчас игнорирую содержательные названия классов намеренно.

```
A a{5, 1.2, B()};
A a{.x = 5, .y = 1.2, .z = B()};
```

В этом случае поля структуры напрямую инициализируются указанными значениями без вызова конструктора. Если вам нужны дефолтные значения для переменных, то их можно указать напрямую в структуре

```
B getDefault();

struct A {
  int x = 42;
  double y = 1.2;
  B z = getDefault();
};
```

2. Не надо добавлять каких суффиксов и префиксов в название полей структуры вида `x_` или `m_x`. Все поля публичные и пользователь будет видеть именно такие имена. Пусть у них будут нормальные названия.

3. Если вам нужны методы для работы со структурой, то сделайте либо глобальный метод, либо в можно сделать статический метод

```
struct A {
  int x;
  double y;
  B z;

  static int f(const A&);
};

double g(const A&);

A a;
int value = A::f(a);
a.y = 2.2;
double c = g(a);
```

Так как все поля публичные, любой метод может стучаться в структуру.

4. Не надо делать методы доступа к полям структуры. Вы и так их можете инициализировать напрямую. Но вот что бывает полезно сделать. Предположим, что вы собираете дерево на основе нод и указателей и ваша нода выглядит так

```
struct Node {
  Data data;
  Node* parent = nullptr;
  Node* left = nullptr;
  Node* right = nullptr;
};
```

И теперь вы хотите обрабатывать запросы к нодам через указатели на них. Тогда вам придется обрабатывать случай `nullptr`. Вот что имеется в виду

```
Node* node = ...;
if (node)
  node->data = Data{};
```

Чтобы избежать проверки на `nullptr` можно сделать отдельный метод, который проверяет указатель на вершину на `nullptr` и только если она не нулевая, то меняется поле по указателю. Это бывает удобно, бывает не удобно. Но полезно знать, про такую опцию

```
void set_data(Node* node, Data data) {
  if (node == nullptr)
    return;
  node->data = std::move(data);
}

Node* node = ...;
set_data(node, Data{});
```

В последней строчке все равно был ли `node` нулевым или нет.

6.3 Классы

6.3.1 Дефолтные методы

Как я уже написал выше, главная причина создания класса – инвариант на данные. Это не единственная причина (и не всегда достаточная[11]), но основная. Что надо понимать: когда мы создаем класс, то у любого класса (в том числе структуры) есть шесть дефолтных методов, которые автоматически генерируются компилятором. Давайте я их укажу явно

```
class A {
public:
  A() = default;
  A(const A&) = default;
  A& operator=(const A&) = default;
  A(A&&) noexcept = default;
  A& operator=(A&&) noexcept = default;
  ~A() = default;
};
```

Задача этих методов – обеспечить семантику поведения объекта. То есть мы хотим, чтобы объекты класса `A` вели себя как переменные-стаканы, хранящие данные.[12] Давайте я назову эти методы явно

1. Конструктор по умолчанию (default constructor)
2. Конструктор копирования (copy constructor)
3. Копирующее присваивание (copy assignment)
4. Конструктор перемещения (move constructor)
5. Перемещающее присваивание (move assignment)
6. Деструктор (destructor)

Запомните важное правило: если все переменные в классе обладают value semantics, то ни один из этих методов НЕ надо явно задавать. Дефолтные значения сработают прекрасно. В идеальном мире value semantics все выглядит как-то так:

```
class A {
public:
  A() = default;
  A(/*arguments*/);
  /*class public interface*/
private:
  int x_ = /*default value*/;
  std::vector<int> y_ = /*default value*/;
  B z_ = /*default value*/;
};
```

Так же обратите внимание на правильную сигнатуру move операций. Они обязательно помечаются `noexcept`. Это делается для лучшей совместимости с алгоритмами STL. Давайте расскажу про конкретный пример, в котором это важно. Представьте, что вы имплементируете `std::vector`. И вам надо имплементировать в нем реаллокацию. Представим себе, что в векторе лежат элементы $a_1, \dots, a_k$. Тогда разумно было бы выделит новый сегмент памяти, и потом сделать move данных из a_1 в новую ячейку b_1 и так далее вплоть до $a_k \mapsto b_k$. Однако, что если эта операция может кинуть исключение. Тогда вы можете оказаться в ситуации, что часть данных уже переместилась, а часть еще нет. И вам надо восстановить инварианты вектора и почистить лишнюю память до отпускания исключения. Что можно сделать? Надо бы вернуть данные из b_1 обратно в a_1 и т.д. Но если move операции могут кидать исключение, то у нас опять может быть брошено исключение. И тогда программа падает по `terminate` ибо нельзя кидать два исключения за раз. Потому, чтобы обработать корректно такую ситуацию, если move операции не помечены `noexcept`, то `std::vector` при реаллокации будет использовать копирование вместо move! И вы низачто не заметите где же у вас все время идут лишние копирования.

[11] Всегда есть особые случаи, но это не значит, что у нас нет ориентиров на 99.9999% случаев.
[12] Это не всегда желаемое поведение, но так надежно по умолчанию.

6.3.2 Структура класса

Я предпочитаю следующую структуру

```
class A {
  using Type0 = double;
public:
  using Type1 = int;

  A();
  A(Type1);
  A(const A&);
  A(A&&) noexcept;
  A& operator=(const A&);
  A& operator=(A&&) noexcept;
  ~A();

  void make_something() const;

  static  void do_something();

  static constexpr const Type1 default_value = 0;

protected:
  void do_protected_stuff();
private:
  void do_private_stuff();

  static Type0 private_static_data_;

  Type1 private_data_ = default_value;
};
```

Из важного тут следующее. Все переменные класса идут в конце в приватной части. Так их легче найти. перед ними идут статические переменные, чтобы их тоже было легко искать. Публичный интерфейс идет в самом начале класса, потому что это та часть класса, через которую с ним будут взаимодействовать. Перед публичным интерфейсом идут псевдонимы для удобства работы с классами. Не перемешивайте между собой методы и статические методы. Не перемешивайте между собой переменные класса, статические переменные и методы. В начале публичного интерфейса если нужно явно определить находятся шесть методов обеспечивающих семантику класса.

6.3.3 Как инициализировать классы

Для инициализации классов используются конструкторы. Конструктор по умолчанию инициализирует все поля, вызывая у них конструктор по умолчанию если такой есть (у встроенных типов его нет и ничего не вызывается).

```
class A {
public:
  A()
   : x_(0), y_({1, 2}), z_(B(42)) {}
  // inconsistent value of y_
  A(int x)
   : x_(x), y_({1, 2, 3}), z_(B(42)) {}
private:
  int x_;
  std::vector<int> y_;
  B z_;
};
```

```
class A {
public:
  A() = default;
  // values of y_ and z_ are the same
  // for both constructors
  A(int x)
   : x_(x) {}
private:
  int x_ = 0;
  std::vector<int> y_ = {1, 2};
  B z_ = B(42);
};
```

Если вы хотите задать дефолтные значения, то лучше задавать их при объявлении переменных, а дефолтный конструктор задать через `=default`. Кроме того, в примере выше присутствуют три магические константы – значения инициализации переменных. Я предпочитаю либо заводить `constexpr` дефолтные значения, либо методы для инициализации:

```
class A {
```

```
public:
  A() = default;
  A(int x) : x_(x) {}
private:
  static constexpr k_x_default_ = 0;
  static std::vector<int> makeStartingVector();
  static B makeDefault();

  int x_ = k_x_default_;
  std::vector<int> y_ = makeStartingVector();
  B z_ = makeDefault();
};
```

Так же обратите внимание, если вы задали хотя бы один не дефолтный конструктор `A(int)`, то дефолтный не будет сгенерирован. Это сделано намерено, потому что раз у вас объект конструируется из каких-то данных, то наверное нет способа построить корректно дефолтный объект и компилятор специально не генерирует дефолтный конструктор, чтобы у вас случайно не было ошибки. Если вам он все же нужен и его дефолтная имплементация вам подходит, то надо лишь написать `A() = default`. Не пишите `A() {}` или другие версии.

Теперь поговорим о конструировании объектов из данных. Давайте начнем со следующего примера

```
class A {
public:
  A(const B& b, const std::vector<int>& v)
    : b_(b), v_(v) {}
private:
  B b_;
  std::vector<int> v_;
};
```

Я часто вижу такой код и его основная мотивация – передаем по константной ссылке аргументы, чтобы избежать лишнего копирования. Давайте обсудим, какие есть проблемы с таким конструктором, и как их решать. Давайте разберем как это работает на примере кода

```
A a(B(3), {1, 2});
```

Первый вопрос: сколько конструкторов будет вызывано кроме конструктора `A`? Оказывается тут будет аж 5 конструкторов. Давайте проанализируем как приблизительно это работает. Код выше можно условно заменить на следующий фрагмент:

```
const B& b = B(3);
const std::vector<int>& v = {1, 2};
B b_(b);
std::vector<int> v_(v);
```

В начале создастся временный объект `B(3)` и он забиндится на константную ссылку `b`. После чего создастся временный объект `{1, 2}`. Но этот объект имеет тип `std::initializer_list<int>`. После этого, чтобы можно было забиньдить `v` на этот объект временный объект `{1, 2}` конвертируется в `std::vector<int>` (это еще один конструктор). И уже после этого `v` биндится на этот временный вектор. После чего в третьей строчке содержимое `b` копируется в `b_` (еще один конструктор). И аналогично в последней строчке содержимое `v` копируется в `v_`. То есть реально происходит следующее[13]

```
B tmp = B(3); // ctor
const B& b = tmp;
std::initializer_list<int> list = {1, 2}; // ctor
std::vector<int> vec = list; //ctor
const std::vector<int>& v = vec;
B b_ = b; // copy ctor
std::vector<int> v_ = v; // copy ctor
```

Конечно, можно надеяться, что компилятор что-нибудь из этого оптимизирует, но есть вещи, которые он обойти не сможет. Давайте поговорим, что тут плохо, почему и как это исправлять:

1. Первое, что сразу бросается в глаза – плохая идея принимать аргумент в виде ссылки на контейнер. Так как в C++ разрешена одна конвертация для каждого аргумента функции, то происходит следующее,

[13] Здесь последние две строчки хоть и изменились, но делают то же самое, это эквивалентный синтаксис. Добро пожаловать в мир плюсов, где один и тот же вызов копирующего конструктора может быть вызван разным синтаксисом.

мы вместо `std::vector<int>` передали объект, который является типом `std::initializer_list<int>` и чтобы он забиндился на ссылку на константный вектор, приходится явно строить временный вектор. Такую проблему с контейнерами можно решить, принимая явно список инициализации

```
A(const B& b, std::initializer_list<int> list)
  : b_(b), v_(list) {}
```

Так мы избавляемся от одного лишнего конструктора построения временного вектора. Обратите внимание, что `std::initializer_list<int>` всегда передается by value. Так как он не хранит данные, а сам он лишь является умной оберткой вокруг указателя на данные и потому передавать его по копии дешево, и работа с ним через указатель или ссылку стоит дороже за счет одного дополнительного разыменования указателя. Тем не менее, такой подход тоже не является наилучшим, об этом ниже.

2. Другая проблема заключается в том, что мы в каждый аргумент конструктора передаем временный объект и потом копируем из него данные. А именно

```
A a(B(3), {1, 2});
```

Объект `B(3)` создается в памяти и мы бы хотели своровать его содержимое, а не копировать внутрь `b_`, потому что временный объект все равно уничтожится. И давайте сразу обсужу плохое решение, которое никогда не стоит применять – передать по не константной ссылке данные

```
A(B& b, std::vector<int>& v)
 : b_(std::move(b)), v_(std::move(v)) {}
```

Проблема в том, что такая не константная ссылка не биндится на все что угодно, как константная ссылка (это особенности языка). Например, она никогда не биндится на временные объекты (потому что она не продлевает их время жизни). Так же она не биндится на объекты другого типа (и не делает конвертации). То есть

```
class A {
public:
  A(B& b, std::vector<int>& v)
   : b_(std::move(b)), v_(std::move(v)) {}
};

B b = B(3);
std::vector<int> v = {1, 2};
// OK
A a(b, v);
// compilation error, B(3) is temporary
A a(B(3), v);
// compilation error, {1, 2} has wrong type and temporary
A a(b, {1, 2});
// compilation error, both
A a(B(3), {1, 2});
```

3. При использовании `std::initializer_list` имейте в виду следующую особенность: он всегда хранит константные данные. То есть данные из него никогда нельзя своровать и переместить внутрь. Из него данные всегда копируются. Это может быть очень неприятно и дорого для тяжелых объектов. К сожалению это никак не обходится.

4. Обратим внимание, что мы принимаем аргументы в конструкторе ровно того типа, что мы храним внутри и мы хотим просто поглотить аргументы конструктора. Такие аргументы называются sink arguments. Sink argoments можно принимать by value, а именно

```
class A {
public:
  A(B b, std::vector<int> v)
   : b_(std::move(b)), v_(std::move(v)) {}
private:
  B b_;
  std::vector<int> v_;
};

```

```
B z = B(3);
std::vector<int> w = {1, 2};
// OK
A a(z, w);
// OK
A a(B(3), w);
// OK
A a(z, {1, 2});
// OK
A a(B(3), {1, 2});
```

Давайте посмотрим, как работает первый вызов.

```
B b = z; // copy constructor
std::vector<int> v = w; // copy constructor
B b_ = std::move(b); // move constructor
std::vector<int> v_ = std::move(v); // move constructor
```

Как мы видим в этом случае аргументы конструктора снимают копии с переданных `z` и `w`, а потом содержимое аргументов мувается внутрь локальных переменных класса. Теперь давайте посмотрим, как работает последний вызов. Код можно условно заменить следующим

```
B b = B(3); // argument initialization
std::vector<int> v = {1, 2}; // argument initialization
B b_ = std::move(b); // move constructor
std::vector<int> v_ = std::move(v); // move constructor
```

Как вы видите, в этом случае аргументы конструктора напрямую инициализируются временными объектами, а потом их содержимое перемещается внутрь переменных класса. Главный плюс этого метода – мы можем передавать в конструктор как временные объекты, так и переменные. Содержимое временных объектов воруется мувом, а с переменных мы снимаем копию. И вот бывает, что мы не всегда хотим эту копию делать. Этому посвящен следующий метод.

5. Еще один вариант для конструктора – передавать данные по rvalue ссылке:

```
class A {
public:
  A(B&& b, std::vector<int>&& v)
    : b_(std::move(b)), v_(std::move(v)) {}
private:
  B b_;
  std::vector<int> v_;
};

B z = B(3);
std::vector<int> w = {1, 2};
// OK
A a(B(3), {1, 2});
// OK
A a(std::move(z), std::move(w));
// compilation error
A a(z, std::move(w));
// compilation error
A a(std::move(z), w);
```

Этот метод работает так же как предыдущий, только он запрещает передавать «не временные объекты» в качестве аргументов. То есть временные объекты передаются так же, как и выше, а вот переменные без `std::move` передать внутрь такого конструктора не получится. То есть такой конструктор используется, когда надо гарантировать, что данные всегда воруются. А не снимаются копии. Я обычно такие конструкторы делаю приватными и оборачиваю их в named constructor idiom, о которой речь пойдет ниже.

Еще одно отличие этого метода заключается вот в чем. Давайте рассмотрим две имплементации и два вызова

```
class A {
public:
  A(B b, std::vector<int> v)
    : b_(std::move(b)), v_(std::move(v)) {}
private:
  B b_;
  std::vector<int> v_;
};

B z = B(3);

A a(std::move(z), {1, 2});
```

```
class A {
public:
  A(B&& b, std::vector<int>&& v)
    : b_(std::move(b)), v_(std::move(v)) {}
private:
  B b_;
  std::vector<int> v_;
};

B z = B(3);

A a(std::move(z), {1, 2});
```

В обоих случаях содержимое `z` будет поглощено, но стоимость поглощения разная. В первом случае это стоит два мува, первый мув – из `z` в аргумент конструктора `b`, а второй мув – из `b` в член класса `b_`. Во втором случае в конструктор передается ссылка – это копирование адреса объекта и остается один мув – из `b` в член класса `b_`. Если мувы могут быть дорогими, то второй подход работает чуть лучше, ибо использует всего два мува против одного. Имейте в виду это отличие.

6. Главный недостаток подхода со ссылками – что если вы хотите часть аргументов копировать, а часть воровать? Тогда вам придется делать 2^n конструкторов, где n – количество аргументов. Действительно, сравните

```
class A {
public:
  A(B  b, std::vector<int> v)
    : b_(std::move(b)), v_(std::move(v)) {}




private:
  B b_;
  std::vector<int> v_;
};

B z = B(3);
std::vector<int> w = {1, 2};
A a(z, w);
A a(std::move(z), w);
A a(z, std::move(w));
A a(std::move(z), std::move(w));
```

```
class A {
  A(const B&  b, const std::vector<int>& v)
    : b_(b), v_(v) {}
  A(const B&  b, std::vector<int>&& v)
    : b_(b), v_(std::move(v)) {}
  A(B&&  b, const std::vector<int>& v)
    : b_(std::move(b)), v_(v) {}
  A(B&&  b, std::vector<int>&& v)
    : b_(std::move(b)), v_(std::move(v)) {}
private:
  B b_;
  std::vector<int> v_;
};

B z = B(3);
std::vector<int> w = {1, 2};
A a(z, w);
A a(std::move(z), w);
A a(z, std::move(w));
A a(std::move(z), std::move(w));
```

7. Последняя проблема обычно адресуется с помощью шаблонов и техники perfect forwarding так

```
class A {
public:
  template<class TB, class TV>
  A(TB&& b, TV&& v)
   : b_(std::forward<TB>(b)),
     v_(std::forward<TV>(v)) {}
private:
  B b_;
  std::vector<int> v_;
};

std::vector<int> w = {1, 2};
A a(z, w);
A a(std::move(z), w);
A a(z, std::move(w));
A a(std::move(z), std::move(w));
```

Когда `&&` применяется в аргументе функции к чистому шаблонному параметру (имеется в виду, нельзя написать `std::vector<TB>&&`, надо именно `TB&&`), то она может трактоваться и как lvalue ссылка и как rvalue ссылка. А std::forward либо ничего не делает (в случае lvalue ссылки), либо делает std::move (в случае rvalue ссылки).

8. Главный недостаток этого подхода в том, что все данные передаются по ссылке. Это хорошо для тяжелых объектов, но не очень эффективно для встроенных типов. В целом это можно игнорировать, либо если вы знаете, что вы передаете встроенные типы, то передавайте их по значению, а не по ссылке там, где справитесь это сделать.

6.3.4 Инициализация шаблонных классов

Предположим у вас есть шаблонный класс

```
template<class T, class B>
class A {
public:
  ...
private:
  T b_;
  std::vector<B> v_;
};
```

И мы так же хотим написать конструктор для этого класса, который бы эффективно принимал данные. Мы можем обойтись подходом с sink arguments и сделать так

```
template<class T, class B>
class A {
public:
  A(T b, std::vector<B> v)
   : b_(std::move(b)), v_(std::move(v)) {}
private:
  T b_;
  std::vector<B> v_;
};
```

Либо если мы хотим гарантировать, что данные обязательно будут своровны написать

```
template<class T, class B>
class A {
public:
  A(T&& b, std::vector<B>&& v)
   : b_(std::move(b)), v_(std::move(v)) {}
private:
  T b_;
  std::vector<B> v_;
};
```

Обратите внимание, что тут `T&&` это только rvalue ссылка, так как `T` это НЕ шаблонный аргумент конструктора, для конструктора это уже вполне определенный известный тип. А теперь вопрос, что если тип `T = int`, тогда в этом подходе мы будем передавать встроенный тип по ссылке. А ссылка – это все равно что указатель. То есть мало того, что вместо 4 байт мы теперь передаем 8, так еще и для обращения к данным надо пройти по указателю. В целом плохая идея встроенные типы передавать не by value. В этом случае можно завести вспомогательный тип, который решает, как именно принимать аргумент

```
template<class T, class B>
class A {
  template<class X>
  using Arg = If</*type X is small*/>::Then<X>::Else<X&&>;
  using TArg = Arg<T>;
public:
  A(TArg b, std::vector<B>&& v)
   : b_(std::move(b)), v_(std::move(v)) {}
private:
  T b_;
  std::vector<B> v_;
};
```

Здесь в 3-4 строчках описан шаблон, который решает в зависимости от условия на тип `X` какой тип использовать `X` или `X&&`. Для имплементации можно воспользоваться стандартными шаблонами, например так:

```
template<class X>
using Arg = std::conditional_t<std::is_arithmetic_v<X>, X, X&&>;
```

Тогда `Arg<X>` будет `X` для всех арифметических типов C++, и `X&&` для всех остальных типов. Но можно настроить условие более тонко в зависимости от того, чего вы хотите. В целом передавать данные размером в 3-4 указателя можно считать дешево. И потому `std::pair<int, int>` тоже можно передавать by value.

6.3.5 Пример имплементации всех шести и RAII

Предположим я хочу написать класс, который умеет хранить `int` на куче. Тогда мне надо написать

```
class IntOnHeap {
public:
  IntOnHeap();
  IntOnHeap(int);
  IntOnHeap(const IntOnHeap&);
  IntOnHeap& operator=(const IntOnHeap&);
  IntOnHeap(IntOnHeap&&) noexcept;
  IntOnHeap& operator=(IntOnHeap&&) noexcept;
  ~IntOnHeap();
private:
  int* value_;
};
```

Таким образом мы хотим написать класс, который будет сам менеджерить ресурс. Это значит, что класс больше ничего не должен делать кроме этого. Техника использования подобных менеджерящих классов называется Resource Acquisition Is Initialization (RAII, смотри раздел 6.3.15).[14] Давайте напишем имплементации для всех этих методов.

Конструктор по умолчанию Начнем с самого простого – создание объекта по умолчанию. В этом случае полезно инициализировать все поля класса при объявлении и дать дефолтную имплементацию.

```
class IntOnHeap {
public:
  IntOnHeap() = default;
  ...
private:
  int* value_ = nullptr;
};
```

Такой подход предполагает, что у нас объект `IntOnHeap` может ничего внутри не содержать. Это разумно, если мы сделали мув данных из него. Так что мы поддерживаем дефолтный конструктор, который не выделяет ресурсы.

Конструктор от данных и деструктор Теперь чтобы продемонстрировать как писать деструктор, мы дополнительно напишем конструктор от данных. Он формально не входит в шесть автоматически генерируемых методов, но я предпочитаю их писать парой.

```
class IntOnHeap {
public:
  IntOnHeap(int value) : value_(new int(value)) {}
  ~IntOnHeap() {
    delete value_;
  }
private:
  int* value_ = nullptr;
};
```

В конструкторе и деструкторе мы просто создаем и чистим ресурс. Теперь надо заняться копированием и перемещением данных.

[14] На самом деле более обще так называют любую технику, когда надо захватить какой-то ресурс в конструкторе и освободить его автоматически в деструкторе.

Перемещающий конструктор Перемещающий конструктор перемещает данные в еще не созданный объект. То есть поля класса для `this` уже аллоцированы, но объект мы должны все еще построить.

```
class IntOnHeap {
public:
  IntOnHeap(IntOnHeap&& other) noexcept
   : value_(std::exchange(other.value_, nullptr)) {}
  ...
private:
  int* value_ = nullptr;
};
```

С конструктором все просто – надо переместить указатель из `other` в `this`, а потом указатель в `other` надо заменить на `nullptr`. Для удобства мы воспользовались функцией `std::exchange`, которая сохраняет содержимое первого аргумента, потом заменяет значение первого аргумента вторым, и после возвращает старое значение первого аргумента.

Обратите внимание, что перемещающий конструктор (как и перемещающее присваивание) помечено как `noexcept`. Это важная часть сигнатуры перемещаюищих конструктора и присваивания. Дело в том, что многие шаблонные классы проверяют можно ли объект переместить без исключений и в зависимости от этого меняется их поведение. Самый яркий пример – `std::vector`. Если нужно при реаллокации переместить данные из одного буффера в другой, то используется перемещающий конструктор только если он помечен `noexcept`. Если же перемещающие операции могут кинуть исключения, то `std::vector` использует копирующий конструктор. Это может очень сильно ударить по производительности и обнаружить такую проблему очень не просто.

Перемещающее присваивание Принципиальное отличие перемещающего конструктора от перемещающего присваивания в том, что у конструктора объект `this` строится прямо сейчас и потому в нем, во-первых, нет никаких данных и их не надо чистить, а во-вторых, он не может совпасть с `other`. В операторе присваивания придется обработать эти случаи. Существует подход, в котором вам не придется проверять равенство `this` и `other`. Например так:

```
class IntOnHeap {
public:
  IntOnHeap& operator=(IntOnHeap&& other) noexcept {
    IntOnHeap tmp = std::move(other);
    std::swap(value_, tmp.value_);
    return *this;
  }
  ...
private:
  int* value_ = nullptr;
};
```

Первая строчка оператора создает временый объект `tmp` и перемещает в него содержимое `other`. Это всего лишь три присваивания указателей.[15] После этой строчки нам надо выполнить перемещение данных из `tmp` в `this`. Но теперь заметьте, что `tmp` и `this` гарантированно будут разными объектами, значит мы избавились от проверки на самопроисваивание.[16] Если `this` не был равен `other`, то мы просто сделали лишнее перемещение данных. Если же `this` совпадал с `other`, то мы переместили данные из `this` в `tmp` и теперь их надо вернуть обратно. Вторая строчка меняет содержимое `tmp` и `this`. Уничтожение данных, которые находились в `this` будет произведено деструктором `tmp`.[17]

Копирующий конструктор Тут ситуация похожа на перемещающий конструктор, только обычно мы выделяем ресурсы под копию и копирующий конструктор не обязан быть `noexcept`.

```
class IntOnHeap {
```

[15] Если вы думаете почему 3, а не 1, то вспомните, что мы используем `std::exchange`.

[16] В целом плохая идея проверять на самоприсваивание, так как вы вставляете оператор условного перехода в каждой процедуре присваивания. Но при этом самоприсваивание почти никогда не происходит в коде. То есть вы ставите защиту от случая, которого почти нет и при этом бьете по производительности во всех остальных случаях. Так лучше не делать. Справедливости ради, надо сказать, что в этом решении мы все же платим цену за обработку самоприсваивания. Но эта стоимость – один лишний мув конструктор, а это всего лишь несколько лишних присваиваний указателя, что сильно лучше, наличия условного перехода.

[17] Обратите, что тут нельзя вызвать `std::swap(*this, tmp)`, потому что эта функция будет вызывать оператор присваивающего перемещения, который мы и имплементируем, то есть мы вызовем бесконечную рекурсию.

```
public:
  IntOnHeap(const IntOnHeap& other)
   : value_((other.value_ ? new int(*other.value_) : nullptr)) {}
  ...
private:
  int* value_ = nullptr;
};
```

В копирующем конструкторе главное – не забыть обработать случай `nullptr` в `other.value_`. Вообще каждый раз когда вы разыменовываете указатель полезно вспоминать о проверке, что это не всегда можно сделать.

Копирующее присваивание Теперь можно написать копирующее присваивание. Оказывается, что он полностью выражается через предыдущие методы.

```
class IntOnHeap {
public:
  IntOnHeap& operator=(const IntOnHeap& other) {
    return *this = IntOnHeap(other);
  }
  ...
private:
  int* value_ = nullptr;
};
```

Обратите внимание, что мы имплементируем копирующее присваивание через копирующий конструктор и перемещающее присваивание. Такой подход не требует проверки на самоприсваивание, однако, при самоприсваивании, мы перезапимываем в`this` его копию. Такое поведение бывает неприемлемым (например, потому что оно не корректно, если мы хотим в точности сохранить те данные, что были). В таком случае приходится добавлять проверку на самоприсваивание, тут ничего не поделаешь:

```
class IntOnHeap {
public:
  IntOnHeap& operator=(const IntOnHeap& other) {
    if (this == &other)
      return *this;
    return *this = IntOnHeap(other);
  }
  ...
private:
  int* value_ = nullptr;
};
```

Тут надо смотреть по ситуации, какое поведение вам подходит. Бывают разные случаи.

Слияние операторов присваивания Если вам не важна проблема при самоприсваивании, то можно слить копирующий конструктор и перемещающий конструктор в один метод. Хотя я лично не люблю заигрывать с компилятором.

```
class IntOnHeap {
public:
  IntOnHeap& operator=(IntOnHeap other) noexcept {
    return *this = std::move(other);
  }
  ...
private:
  int* value_ = nullptr;
};
```

При такой имплементации при выполнении присваивания

```
IntOnHeap a;
IntOnHeap b;
a = b;
a = std::move(b);
```

В строчке 3 происходит копирование данных из `b` в аргумент конструктора `other` и потом мув данных в `a`. В 3 происходит мув данных из `b` в аргумент конструктора `other` и потом мув данных из него в `a`.

Особенности генерации методов по умолчанию Теперь сделаем важное замечание. Если вы определяете свой move constructor, но не определяете move assignment (или наоборот). То компилятор НЕ сгенерирует move assignment по умолчанию. Сделано это для того, чтобы избежать рассинхрона между вашей операцией перемещения данных и дефолтной. Однако, если вы определите свой copy constructor, но не определите copy assignment (или наоборот), то компилятор сгеренрирует copy assignment по дефолту. Я не знаю зачем так сделали, видимо для того, чтобы компилировался код времен Кеннеди. Но это один из источников проблем. Так как компилятору разрешается подменять copy constructor и copy assignment друг другом, то это один из источников UB. Потому если вы пишете один из этих четырех операторов, то определяйте все четыре. Так же бывают ситуации, когда вам нужно явно удалить возможность копировать или перемещать. Это делается так

```
class A {
public:
 A() = default;
 A(const A&) = delete;
 A& operator=(const A&) = delete;
 A(A&&) noexcept = delete;
 A&operator=(A&&) noexcept = delete;
 ~A() = default;
};
```

В целом хорошо взять за правило, что если вы определяете что-то из дефолтных методов (кроме дефолтного конструктора), то определите все пять (кроме быть может дефолтного конструктора). При этом если вам надо взять имплементацию по умолчанию, используйте `=default`, а если вы хотите убрать соответствующий метод, то значение `delete`.

6.3.6 Аргументы конструктора и Strong Type Aliases

Часто бывает, что конструктор принимает аргументы одинакового типа. Пусть у нас есть класс, для работы с мономами, то есть выражениями вида $x_0^{k_0} \cdot \ldots \cdot x_n^{k_n}$. И пусть его можно построить по индексу переменной и ее степени.

```
class Monomial {
public:
  Monomial(size_t index, size_t degree);
  size_t degree_of(size_t index) const;
private:
  ...
};
```

Здесь есть несколько вещей, которые мне не нравятся.

1. Во-первых, беззнаковый тип. Это я обсуждал в разделе 6.1.1.

2. Но даже, если переделать все в знаковый тип, то становится не лучше, так как тип `int` будет представлять две разные вещи: одна – индекс переменной, другая – степень переменной. Для повышения читаемости стоит ввести локальный псевдоним:

```
class Monomial {
  using Index = int;
  using Degree = int;
public:
  Monomial(Index index, Degree degree);
  Degree degree_of(Index index) const;
private:
  ...
};
```

   Это не позволяет избавиться от ошибок, если вы перепутаете аргументы, но повысит читаемость и поддерживаемость кода. Например, если вы решили, что `Index` занимает слишком много места и вам достаточно одного байта. То вам не придется менять во всех методах сырые типы, вы просто напишете

```
class Monomial {
  using Index = char;
  using Degree = int;
public:
```

```
  Monomial(Index index, Degree degree);
  Degree degree_of(Index index) const;
private:
  ...
};
```

То есть сырые типы, это то же самое, что магические константы, только на уровне типов. Если вы хотите быстро менять подлежащий тип внутри класса, то заводите для всех вспомогательных типов псевдонимы. Так как это не глобальные, а локальные псевдонимы внутри класса, то нет ничего страшного в том, что для компилятора они одинаковые. Ибо в рамках класса мы можем довериться, что программист рассмотрел и проверил все случаи использования. Однако, в публичном интерфейсе все еще могут происходить ошибки. Например, мы можем перепутать индекс переменной и степень местами и компилятор нам не поможет.

3. Давайте посмотрим на вызов конструктора нашего класса

```
Monomial m(1, 3);
```

В этом случае не понятно, что вообще происходит. Читаемость кода на уровне пользователя никакая. Кроме того, мы не застрахованы от ошибки перепутать аргументы местами. Это частая беда, когда аргументы одного типа идут вместе. Тут можно применить технику strongly typed alias-ов, то есть псевдонимов, которые будет различать компилятор. Для встроенных целочисленных типов это делается очень легко.

```
enum Index : char;

class Monomial {
  using Degree = int;
public:
  Monomial(Index index, Degree degree);
  ...
};
```

Тогда на стороне пользователя можно написать

```
Monomial m(Index{1}, 3);
```

Теперь аргументы точно не перепутаешь. Если подставить в первый аргумент целочисленный тип – ошибка компиляции. Однако, название типа `Index` ничего не говорит нам о назначении константы и смысл второго числа не ясен. Тем не менее, я специально остановился на этом решении, чтобы вы имели в виду, что иногда бывает, что не надо делать псевдонимы для всех аргументов, иногда достаточно для части. Но можно сделать и так.

```
enum Variable : char;
enum Degree : int;

class Monomial {
public:
  Monomial(Variable index, Degree degree);
  ...
};

Monomial m(Variable{1}, Degree{3});
```

Другой пример, когда это бывает полезно. Сравните

```
class Picture {
public:
  Picture(size_t width, size_t height);
};

Picture p(300, 400);
```

и имплементацию со строгими псевдонимами

```
enum Width : size_t;
enum Height : size_t;

class Picture {
public:
  Picture(Width width, Height height);
};

Picture p(Width{300}, Height{400});
```

4. Если же надо проделать такую же штуку с числами с плавающей запятой, то это к сожалению не сработает. Так как нельзя создать `enum` с таким подлежащим типом. Придется создавать целую структуру.

```
class Scale {
public:
  explicit Scale(double value) : value_(value) {
  }
  operator double() const {
    return value_;
  }
private:
  double value_;
};

enum Width : size_t;

class Picture {
public:
  Picture(Width width, Scale scale);
};
```

Давайте отмечу пару важных моментов

(a) Конструктор `Scale` специально сделан `explicit`, чтобы избежать неявной конвертации от `double` к `Scale`.

(b) Нельзя сделать `Scale` структурой с публичным полем `value`. Потому что в этом случае `Scale` можно инициализировать с помощью aggregate initialization, а именно

```
struct Scale {
  explicit Scale(double value) : value_(value) {
  }
  double value_;
};

void f(Scale x);

f({1.2});
```

В последней строчке не требуется указывать имя класса. Потому приватность `value` запрещает такой механизм.

5. Оба подхода можно завернуть в шаблоны. В случае целых чисел

```
namespace detail {
template<class T, class Tag>
struct Enum {
  enum type : T;
};
}
template<class T, class Tag>
using AliasE = typename detail::Enum<T, Tag>::type;
```

И использовать так

```
using Width = AliasE<int, struct width_tag>;
using Height = AliasE<int, struct height_tag>;
```

Для чисел с плавающей запятой можно сделать так

```
template<class T, class Tag>
struct Alias {
  explicit Alias(T value) : value_(value) {
  }
  operator T() const {
    return value_;
  }

private:
  T value_;
};
```

И используется так

```
using Scale = Alias<double, struct scale_tag>;
```

В обоих случаях второй аргумент используется для того, чтобы сделать разные типы с подлежащим типом `T`. Если не использовать `Tag`, то все псевдонимы с типом `double` окажутся одним и тем же типом. Это отличает этот тип от `double`, но не отличает их между собой.

6.3.7 Named Constructor Idiom и RVO/NRVO

Предположим, что мы хотим написать класс комплексных чисел и сделать в нем два конструктора. Один создает комплексное число по вещественной и мнимой части, другой по радиусу и аргументу. Тогда мы бы могли написать так

```
class Complex {
public:
  Complex(double re, double im);
  Complex(double r, double phi);
  ...
};
```

Однако, это ошибка компиляции, потому что мы только что объявили одну и ту же функцию. В плюсах имена аргументов не отличают функции, только их типы. Потому в точности так сделать нельзя. В этом случае можно завести специальную функцию, которая будет строить комплексные числа.

```
class Complex {
  Complex(double re, double im);
public:
  static Complex FromReIm(double re, double im) {
    return Complex(re, im);
  }
  static Complex FromRadArg(double r, double phi) {
    return Complex(r * cos(phi), r * sin(phi));
  }
  ...
};
```

В этом случае на стороне пользователя код выглядит так

```
Complex z = Complex::FromReIm(1, 2); // z = 1 + 2i
auto w = Complex::FromRadArg(1, 90); // z = i
```

Тут я предполагаю, что аргумент в градусах, а не радианах. Можно использовать для аргумента комплексного числа два разных псевдонима для работы с радианами или градусами. В этом случае будет две статические функции строящие число по радиусу и аргументу. Обратите внимание, что при вызове функции `FromReIm` в первой строчке тригерится RVO и комплексное число создается конструктором сразу по адресу `z`. Никаких лишних копий создаваться не будет.

У данного метода есть неожиданный плюс. Кроме говорящего имени для конструктора и возможности использовать auto, возникает еще техническое удобство, которое я хочу обсудить. Предположим, мы хотим создать класс

```
class A {
public:
  A(const char * str) : a_(strlen(str)), b_(strlen(str)), str_(str) {}
private:
  std::vector<int> a_;
  std::vector<int> b_;
  std::string str_;
};
```

Смотреть надо на эту операцию так. У нас есть одна дорогая операция – вычисление `strlen(str)`, которую надо выполнить для обоих аргументов. В этом случае синтаксис списка инициализации просто не дает возможности сохранять промежуточные вычисления. Конкретно в этом примере, можно передать длину вместе со строкой, но тогда нет гарантии, что пользователь снаружи подставит честную длину строки, а не случайное число, что не надежно. Да и бывает, что мы не хотим менять интерфейс конструктора и принимать не то, что мы хотим. Тогда можно поступить так. Правильный, но ненадежный конструктор делаем приватным, и потом дергаем его либо в другом конструкторе, либо в статической функции.

```
class A {
  A(const char * str, size_t len) : a_(len), b_(len), str_(str) {}
public:
  A(const char * str) : A(str, strlen(str)) {} // delegates
  static A FromStr(const char * str) {
    size_t len = strlen(str);
    return A(str, len); // here it is safe to call
  }
private:
  std::vector<int> a_;
  std::vector<int> b_;
  std::string str_;
};
```

6.3.8 Приведение типов и ADL

Обычные функции Есть еще одна деталь, которую я хочу обсудить. И связана она с приведением типов друг к другу и на что это влияет. Самый важный элемент языка – ADL или argument dependent lookup. Это механизм, который отвечает за то, какая именно функция будет вызвана для написанного выражения. А именно, предположим что у нас есть две функции:

```
void f(A);
void f(B);
```

И мы хотим позвать функцию так

```
A a;
f(a);
```

Компилятор видит, что используется имя `f` вместе с оператором скобки и ищет все сущности языка, которые сюда могут подойти (это могут быть функции, это могут быть функторы (объект класса с перегруженным оператором скобки), это может быть шаблон функции). Он видит два варианта выше. Теперь ему надо выбрать. В данном случае тип предаваемого выражения совпадает с типом аргумента и компилятор выбирает первую функцию. Однако, если у нас будет вызов вид

```
C c;
f(c);
```

То точного совпадения типов нет. Компилятор на этом не останавливается. Он смотрит, есть ли какой нибудь способ за один шаг сконвертировать передаваемое выражение `c` к типу `A`. За конвертацию типов в языке отвечают два вида функций:

1. Конструкторы от одного аргумента
2. Операторы конвертации типов

Например тип `C` можно конвертировать к `A` в одном из следующих двух случаев

```
class C {};
class A {
public:
  A(C);
};
```

```
class A {};
class C {
public:
  operator A() const;
};
```

Если компилятор не находит способ неявно сконвертировать `C` к `A` или `B`, то это ошибка компиляции, нет нужного метода. Если находит несколько способов сконвертировать – это тоже ошибка компиляции (неоднозначность). В противном случае компилятор делает одну конвертацию. Например, если доступна конвертация `C` к `A`, то

```
C c;
f(c);
// replaces with
f(A(c));
```

и тут используется либо конструктор `A`, либо оператор приведения типа из `C`. Чтобы подавить неявную конвертацию, можно обозначить указанные выше функции методом `explicit` и тогда компилятору будет запрещено вызывать их неявно.

```
class C {};
class A {
public:
  explicit A(C);
};
```

```
class A {};
class C {
public:
  explicit operator A() const;
};
```

Бывают разные ситуации, хотим ли мы неявную конвертацию или не хотим. Потому полезно знать на какие механизмы наличие этой операции может повлиять.

Шаблонные функции Кроме обычных функций в плюсах есть еще шаблонные функции. Важно запомнить, что для шаблонных функций никогда не рассматривается конвертация типов. Потому что компилятор по аргументу пытается подобрать лучший вид шаблонного параметра. Например

```
struct C {
};

template<class T>
struct D {
  D(C) {}
};

template<class T>
void f(D<T>) {}
```

В этом случае вызов

```
C c;
f(c); // compilation error
```

Будет ошибкой компиляции, так как нет шаблонного аргумента `T` для которого `D<T>` в точности совпал с `C`. Но по простому можно запомнить, что для шаблонных функций конвертация типов никогда не происходит.

6.3.9 Делегирование при инициализации

Частой проблемой является написание огромного количества конструкторов от кучи разных данных, например что-то в духе

```
class A {
public:
  A() = default;
  A(int);
  A(double);
  A(Matrix);
  ...
  A(File);
  A(Stream);
```

```
  ...
};
```

Проблема такого подхода не только в раздутом интерфейсе, а в том, что класс начинает заниматься вещами, которыми он заниматься не должен. Например, если вы конструируете объект из файла, то вы этот файл должны парсить в зависимости от формата, потому вы начинаете писать вспомогательные функции для парсинга файла внутри класса `A` и вот теперь вы уже не можете отследить, что относится к классу, а что к вспомогательным иррелевантным для класса функциям. Вместо того, чтобы заниматься такой ерундой, стоит делегировать всю нетривиальную работу другим классам. Давайте разберем пример чтения из файла.

```
class Picture {
public:
  Picture() = default;
  Picture(std::filesystem::path file);
  ...
};
```

Во-первых, можно избавиться от конструктора и заменить его по named constructor idiom статической функцией с говорящим названием.

```
class Picture {
public:
  Picture() = default;

  static Picture FromFile(std::filesystem::path file);
  ...
};
```

А уже внутри метода `FromFile` делегировать работу по чтению и открытию файла специальному объекту, например `ifsream`, работу по парсингу файла специальному классу `Parser`. И уже после того как вы извлечете данные из файла можно создать соответствующий объект `Picture`. Тогда имплементация выглядит как-то так:

```
Picture Picture::FromFile(std::filesystem::path file) {
  std::ifstream stream(file);
  Parser parser;
  Data data = parser(stream);
  return Picture(std::move(Data));
}
```

6.3.10 Сериализация

Что такое сериализация Вопрос чтения и записи в файл или из файла тесно связан с таким вопросом как сериализация данных. По сути сериализация данных – это запись данных в поток, а десериализация – это чтение данных из потока. Есть два синтаксических подхода для сериализации и десериализации:

1. Перегрузка потоковых операторов `operator<<` и `operator>>`.
2. Написание специальных функций чтения и записи.

И тут я хочу сразу начать с предосторожности. Дело в том, что у оператора вывода `operator<<` есть два смысла, а у оператора `operator>>` только один и это приводит к путанице. Давайте поясню на примере. Пусть у нас есть класс

```
class A {
public:
private:
  int x_;
  std::vector<double> vec_;
};
```

И мы для удобства хотим вывести пользователю на печать содержимое данных `A`, например ради дебага. Тогда бы мы написали что-то вроде

```
std::ostream& operator<<(std::ostream& out, const A& a) {
  out << "x = " << a.x_ << '\n';
  out << "vec: ";
  for(double element : a.vec_)
    out << element << ", ";
  return out;
}
```

Смысл этого оператора – вывести данные в поток для чтения человеком, а не для хранения этих данных в потоке. Если же мы хотим записать этот же объект в файловый поток для хранения, мы бы написали

```
std::ofstream& operator<<(std::ofstream& out, const A& a) {
  out << a.x_;
  out << a.vec_.size();
  for(double element : a.vec_)
    out << element;
  return out;
}
```

Так вот, оператор чтения из потока `operator>>` является дополнительным ко второй версии оператора `operator<<`, а у первой версии его просто нет. То есть для чтения из потока мы бы сделали что-то такое

```
std::ifstream& operator<<(std::ifstream& in, A& a) {
  in >> a.x_;
  size_t size;
  in >> size;
  a.vec_.reserve(size);
  double element;
  for(size_t i = 0; i < size; ++i) {
    in >> element;
    a.vec_.push_back(element);
  }
  return in;
}
```

Тут же скажу, что обозначенный выше подход не совсем грамотный, как именно писать имплементацию обоих операторов я скажу ниже. Сейчас важно лишь, что для оператора записи в поток есть две версии: для человека и для хранения, а для чтения только одна. Запись и чтение из потока для сохранения состояния объекта и его восстановление – это две взаимно обратные процедуры и они и называются сериализацией и десериализацией. Именно их я буду обсуждать ниже. Важно отметить, что сериализовать и десериализовать намного легче, если объекты обладают value semantics. В этом случае задача сериализации и десериализации может решаться локально (это может быть не самое эффективное, но в то же время самое простое решение). Под локально я имею в виду, что каждый класс для своей сериализации и десериализации не должен будет знать никакой глобальной информации или контекста.

Как правильно писать сериализацию Будем считать, что мы работаем с классом

```
class Picture {
  ...
private:
  Header header_;
  std::vector<Pixel> data_;
};
```

За чтение и запись в поток должен отвечать отдельный класс. Можно сделать единый класс для чтения и записи, а можно сделать отдельные. Например так

```
class PicReader {};
class PicWriter {};
// or
class PicSerializer {};
```

Потоковые операторы Давайте я выберу первую версию из двух классов. Начнем с синтаксического подхода с операторами `operator<<` и `operator>>`. В начале вам надо перегрузить эти операторы для всех встроенных типов

```
PicWriter& operator<<(PicWriter& out, int x) {
  out << x;
  return out;
}
PicReader& operator>>(PicReader& in, int& x) {
  in >> x;
  return in;
}
```

Всю эту рутинную работу можно сделать с помощью шаблонов и проверки того, что типы удовлетворяют `std::is_arithmetic_v`. После этого надо перегрузить оператор для всех контейнеров, которые вы хотите использовать

```
template<class T, class A>
PicWriter& operator<<(PicWriter& out, const std::vector<T, A>& vec) {
  out << vec.size();
  for(const auto& element : vec)
    out << element;
  return out;
}
PicReader& operator>>(PicReader& in, std::vector<T, A>& vec) {
  vec.clear();
  size_t size;
  in >> size;
  vec.reserve(size);
  T element;
  for(size_t i = 0; i < size; ++i) {
    in >> element;
    vec.push_back(std::move(element));
  }
  return in;
}
```

То есть теперь вы умеете сериализовать и десериализовать все встроенные типы и стандартные контейнеры (если вы все это имплементировали). Теперь, чтобы можно было сделать всю работу рекурсивно, все поля класса должны обладать value semantics (что это такое можно глянуть в разделе 5). Это необходимое, но не достаточное условие. Проблемы могут быть со всякими полиморфными объектами, про которые мы поговорим в разделе 11. Сериализация класса должна выполняться по правилу:

1. Если класс хочет уметь сериализоваться и десериализоваться, то все его поля должны уметь сериализоваться и десериализоваться.

2. Каждый класс должен делегировать свою сериализацию и десериализацию каждому своему полю.

Тогда оператор для класса `Picture` выше будет выглядеть так

```
template<class T, class A>
PicWriter& operator<<(PicWriter& out, const Picture& picture) {
  out << picture.header_;
  out << picture.data_;
  return out;
}
PicReader& operator>>(PicReader& in, Picture& picture) {
  in >> picture.header_;
  in >> picture.data_;
  return in;
}
```

Вот и все. Только для этого надо, чтобы классы `Pixel` и `Header` тоже были сериализуемыми. Но эти классы тоже лишь сериализуют и десериализуют свои поля в нужном порядке. Теперь все используется максимально просто

```
// writing to file
Picture pic = /*initialization*/;
PicWriter w(/*file*/);
w << pic;
// reading from file
Picture pic;
```

```
7  PicReader r(/*file*/);
8  r >> pic;
```

И добавление новых полей в класс не усложняет задачу серизализации или десериализации.

Обратите внимание, что если внутри класса нет value semantics, то такая простая схема уже не подойдет. Например, если класс – это бинарное дерево собранное на указателях. В этом случае решение выше дословно не подходит и нужно писать какое-то отдельное решение. В этом случае сложность написания сериализации пропорциональна сложности устройства класса.

Функции чтения и записи Главный недостаток предыдущего способа заключается в том, что невозможно считать из потока объект, который не default constructible. А такое быть может. Тогда можно прибегнуть к написанию специальных функций, которые сразу возвращают значение объекта, а именно.

```
1  class PicReader {
2  public:
3    template<class T>
4    T read();
5  };
6  class PicWriter {
7  public:
8    template<class T>
9    void write(const T&);
10 };
```

Эти функции идейно выполняют ту же самую работу как и потоковые операторы, то есть надо реализовать их для встроенных типов и стандартных контейнеров, а потом рекурсивно делегировать всю работу для каждого поля в нужном порядке. Тем не менее, давайте я напишу. Для встроенных типов будет что-то вроде

```
1  class PicReader {
2  public:
3    template<>
4    int read() {
5      return /*implementation*/;
6    }
7  };
8  class PicWriter {
9  public:
10   template<>
11   void write(int x) {
12     /*implementation*/
13   }
14 };
```

Технически в 11 строке надо передавать `const int&`, но можно докрутить, чтобы встроенные типы передавались по значению, а остальные по ссылке и тому подобные вещи. Я на это закрываю глаза здесь. Теперь стандартные контейнеры обрабатываются так.

```
1  class PicReader {
2  public:
3    template<class T, class A>
4    std::vector<T, A> read() {
5      size_t size = read<size_t>();
6      std::vector<T, A> vec;
7      vec.reserve(size);
8      for(size_t i = 0; i < size; ++i)
9        vec.push_back(read<T>());
10     return vec;
11   }
12 };
13 class PicWriter {
14 public:
15   template<class T, class A>
16   void write(std::vector<T, A>& vec) {
17     write(vec.size());
18     for(const auto& element : vec)
19       write(element);
20   }
21 };
```

И тогда можно использовать код так

```
PicReader r(/*file*/);
Picture picture = r.read<Picture>();
PicWriter w(/*file*/);
w.write(picture);
```

Все отличие между двумя методами чисто синтаксическое. Но при этом принципиальное отличие этого метода в том, что не надо заводить неинициализированную переменную в дефолтном состоянии и потом перезаписывать поверх ее данными, можно сразу инициализировать нужный объект из потока при конструировании.

Завершить я хочу замечанием, что на сегодняшний день есть много фреймворков умеющих выполнять сериализацию данных. Наверное самый популярный – google protobuf.

6.3.11 Как писать операторы

Операторы можно поделить на 4 класса

	in place	out of place
in class	`A& operator+=(const A&);`	`A operator+(const A&) const;`
out of class	`A& operator+=(A&, const A&);`	`A operator+(const A&, const A&);`
uses	`a += b;`	`c = a + b;`

Оператор `a += b` обычно используется для модификации объекта `a`. И часто его можно сделать эффективно без переаллокации внутренних ресурсов (но не всегда). Оператор же `a + b` используется для построения нового объекта, для которого нужно аллоцировать ресурсы. Сразу скажу, я предпочитаю для in place оператора in class имплементацию, а для out of place оператора out of class имплементацию. Давайте я напишу все 4 и объясню чем они отличаются друг от друга.

Имплементация внутри класса

```
class A {
public:
  A& operator+=(const A&) {
    // do something
    return *this;
  }
  A operator+(const A& other) const {
    A tmp = *this;
    tmp += other;
    return tmp;
  }
};
```

Имплементация снаружи

```
class A {
public:
};
A& operator+(A& left, const A& right) {
  // do something, requires access to A, should be friend
  return left;
}
A operator+(const A& left, const A& right) {
  A tmp = left;
  tmp += right;
  return tmp;
}
```

Обратите внимание, выше приведена стандартная имплементация для out of place оператора через in place оператор. Типичная ошибка в имплементации такая

```
class A {
public:
  A operator+(const A& other) const {
    return A(other) += other;
```

```
  }
};

class A {
public:
};
A operator+(const A& left, const A& right) {
  A tmp = left;
  return tmp += right;
}
```

В исходной версии, когда написано `return tmp` тригерится NRVO. То есть при выполнении операции

```
A x;
A y;
A z = x + y;
```

В третьей строчке сумма `x + y` строится сразу по адресу объекта `z`. Однако, в последних двух версиях этого не просходит. Вместо этого сумма строится во временном объекте `tmp` и потом результат мувается в `z`. Причина этого в следующем. В двух последних имплементациях тип возвращаемый функцией не совпадает с типом выражения в `return`. Действительно, в `return` в обоих случаях последняя операция – применение оператора `+=`, который возвращает `A&`, что отличается от `A`. А это мешает выполнить NRVO и компилятор откатывается до мува. Потому желание сократить в количестве строчек кода, может негативно сказаться на качестве программы.

In class vs. Out of class Давайте поймем разницу между этими подходами. Очевидная разница в том, что операторы вне класса можно добавлять к типам неинтрузивно. То есть не меняя код самого класса, при условии, что у вас есть достаточный уровень доступа. Однако, есть одно содержательное отличие именно для out of place оператора. Рассмотрим следующий код

```
class A {
public:
  A(int);
  A operator+(const A&) const;
};

int main() {
  A x;
  A y;
  y = x + 1; // same as y = x + A(1);
  y = 1 + x; // compilation error
  return 0;
}
```

```
class A {
public:
  A(int);
};
A operator+(const A&, const A&);

int main() {
  A x;
  A y;
  y = x + 1; // same as y = x + A(1);
  y = 1 + x; // same as y = A(1) + x;
  return 0;
}
```

То есть для in class оператора сложения конвертация типов возможна только по второму аргументу, в то время как для out of class оператора сложения она доступна по обоим аргументам. Вопрос позволять или не позволять неявную конвертацию – это уже ваш выбор. Но при этом важно понимать, что если вы позволили конвертацию, второй способ просто универсальнее. Это первая причина, почему я стараюсь определять out of place операторы снаружи и делать их `friend`. То есть мой выбор будет следующая пара

```
class A {
public:
  A& operator+=(const A&);
  friend A operator+(const A&, const A&);
};
A operator+(const A&, const A&);
```

Шаблоны и друзья В начале начнем с описания четырех случаев взаимодействия классов и их друзей.

1. **Ont-to-Many** Не шаблонная функция друг для всех шаблонных классов.

```
template<class T>
class A {
  friend void f();
};
```

Здесь функция `f` умеет получать доступ к внутренностям всех классов вида `A<T>`.

2. **Many-to-One** Все версии шаблонной функции друзья не шаблонного класса.

```
1 class A {
2   template<class T>
3   friend void f();
4 };
```

Каждая функция вида `f<T>` имеет доступ к внутренностям класса `A`.

3. **One-to-One** Одна версия шаблонной функции является другом одной версии шаблонного класса (с теми же параметрами).

```
1 template<class T>
2 void f();
3
4 template<class T>
5 class A {
6   friend void f<T>();
7 };
```

В этом случае `f<T>` имеет доступ только к внутренностям `A<T>`, но не имеет доступ к внутренностям `A<U>` для `U` отличного от `T`. Обратите внимание, что в этом случае вы обязаны сделать forward declaration для `f` перед классом, иначе компилятор не распознает, что написано в строке 6 (он не поймет, что `f` это было имя шаблона, а не просто имя функции и какой-то оператор сравнения потом).

4. **Many-to-Many** Все версии шаблонной функции друзья для всех версий шаблонного класса.

```
1 template<class T>
2 class A {
3   template<class U>
4   friend void f();
5 };
```

Здесь любая `f<U>` может получить доступ к внутренностям любого `A<T>`.

5. **One-to-One*** Теперь неочевидный случай. Давайте попробуем написать первый случай, но добавим класс как аргумент функции.

```
1 template<class T>
2 class A {
3   friend void f(const A&);
4 };
```

В этом случае мы объявляем для класса `A<T>` другом функцию `f(const A<T>&)`. То есть мы объявляем, не одну функцию для всех шаблонных классов, а для каждого шаблонного класса одну уникальную функцию. Потому это отношение **Ont-to-One**, несмотря на то, что синтаксически это очень похоже с первым случаем.

Шаблоны и операторы Теперь поговорим про случай операторов и шаблонов. Мы хотим объявить оператор сложения вне класса шаблона и сделать его другом. Есть принципиально два разных случая.

1. Сделать оператор не шаблонной функцией как в пятом разделе из предыдущего раздела.

```
1 template<class T>
2 class A {
3 public:
4   friend A operator+(const A&, const A&) {
5     /*implementation*/
6   }
7 };
```

Обратите внимание, что в этом случае мы вынуждены написать определение оператора непосредственно внутри шаблонного класса, так как плюсы не предоставляют возможности дать определение каким-либо другим образом для подобного случая.

2. Сделать оператор шаблонной функцией как в третьем случае из предыдущего раздела.

```
1  template<class T>
2  class A;
3
4  template<class U>
5  A<U> operator+(const A<U>&, const A<U>&);
6
7  template<class T>
8  class A {
9  public:
10   friend A operator+<T>(const A&, const A&);
11 };
12
13 template<class U>
14 A<U> operator+(const A<U>&, const A<U>&) {
15   /*implementation*/
16 }
```

Обратите, что мы вынуждены добавить forward declaration для оператора сложения в строках 4-5. Однако, в этом месте мы используем тип `A<U>`, который в 5-ой строке не виден, ибо имя `A` еще не объявлено. Потому надо добавить forward declaration в строках 1-2. Еще из неочевидного – куда пихать `<T>` для оператора? Надо думать так, для оператора `operator+` – это имя функции. А шаблонные параметры надо ставить после имени функции.

Так как в первом решении у нас оператор не шаблонный, а во втором шаблонный. То в первом решении разрешена неявная конвертация аргументов, а во втором нет. Кроме того, во втором случае у нас намного больше шума по отношению к реальному коду. Я в целом предпочитаю случай первый и предпочитаю всегда писать определения шаблонов in place, чтобы не плодить количество шума в коде.

Популярная ошибка с шаблонами и операторами Если написать код

```
1  template<class T>
2  class A {
3  public:
4    friend A operator+(const A&, const A&);
5  };
6
7  template<class U>
8  A<U> operator+(const A<U>&, const A<U>&) {
9    /*implementation*/
10 }
```

То при исполнении

```
1  A<int> x;
2  A<int> y;
3  A<int> z = x + y;
```

У вас вылезет ошибка линковки. Проблема заключается в том, что мы сделали не то, что хотели. А именно, внутри класса в строке 4 мы объявили другом не того оператора, который объявили в строках 6-9. Действительно, когда компилятор видит, что шаблон `A` используется с параметром `int`, он генерирует следующий код

```
1  class A<int> {
2  public:
3    friend A<int> operator+(const A<int>&, const A<int>&);
4  };
```

Теперь относитесь к `A<int>` как к просто имени конкретного класса. Но теперь внутри говорится, что объявлен другом не шаблонный оператор. Потому что перед ним нет определяющего шаблон ключевого слова `template`. Проверьте это, написав

```
1  template<class T>
2  class A {
3  public:
4    friend A operator+(const A&, const A&);
```

```
};

A<int> operator+(const A<int>&, const A<int>&) {
  return A<int>();
}

A<int> x;
A<int> y;
A<int> z = x + y;
```

Теперь все скомпилируется. Если же написать

```
template<class T>
class A {
public:
  template<class U>
  friend A<U> operator+(const A<U>&, const A<U>&);
};

template<class U>
A<U> operator+(const A<U>&, const A<U>&) {
  /*implementation*/
}
```

то вы сделаете все операторы сложения `operator+<U>` друзьями всех классов `A<T>`. А это не очень хорошо, не стоит так делать.

6.3.12 Контроль доступа Safe/Unsafe

Очень часто бывает так, что в интерфейсе вашего класса есть опасные и безопасные операции. Что это значит? Это значит, что какие-то функции работают что бы ни произошло, их вызов всегда корректный и никаких условий на их вызов нет. А не безопасные операции требуют специального состояния объекта, чтобы их можно было вызвать. Например, если мы берем `std::vector`, то метод `size` можно звать у любого состояния вектора, метод `push_back` можно звать на любом векторе. Но методы `front` или `pop_back` требуют, чтобы вектор был не пуст. Как попытаться обезопасить себя и добавить дополнительную проверку, которая бы гарантировала, что вы случайно не вызываете небезопасный метод? В таком языке как Rust есть ключевое слово `unsafe`, без которого просто нельзя позвать небезопасную операцию. Мы можем симулировать этот механизм и в C++. На всякий случай поясню, что я не утверждаю, что это надо делать для `std::vector`. Я лишь хочу рассказать как это можно сделать, и делается это в том случае, когда это действительно вам понадобится для организации надежных интерфейсов.

Доступ к глобальным методам Давайте начнем со случая функций а не класса. Предположим, что у нас есть функции, где часть безопасная и их можно звать всегда, а часть не безопасная.

```
int compute(int data); // safe
bool has_connection(); // safe
void upload(int data); // unsafe
```

Тогда можно в начале закрыть эти функции внутри класса в виде статических методов. В этом случае название класса придаст дополнительный контекст выполняемым функциям.

```
class A {
public:
  static int compute(int data);
  static bool has_connection();
protected:
  static void upload(int data);
};
```

А теперь добавим шаблон

```
template<class T>
class Unsafe;

template<>
class Unsafe : A {
  ~A() = delete;
```

```
public:
  using A::upload;
};
```

Удаленный деструктор не позволит нам создать объект такого типа. Теперь работа с функциями выглядит так

```
int value = A::compute(3);
if(A::has_connection())
  Unsafe<A>::upload(value);
```

Доступ к методам класса Просто так повторить такой же трюк с объектами класса не получится. Обратите внимание, что с помощью наследования мы можем получить доступ к методу, но нельзя кастовать объект к наследнику. Давайте я начну с корректного решения, а потом приведу пример простого и не корректного (но при этом часто используемого). Пусть у нас есть класс:

```
class A {
public:
  int compute(int data);
  bool has_connection() const;
protected:
  void upload(int data);
};
```

Теперь сделаем вспомогательный класс для доступа к методу

```
template<class T>
class Access;

template<>
class Access<A> : A {
  ~Access() = delete;
public:
  using A::upload;
};
```

Как и раньше, удаленный конструктор не позволит нам создавать объекты такого класса. И вот теперь делаем нужный класс

```
template<class T>
class unsafe;

template<>
class unsafe<A> {
  using AccessT = Access<A>;
  static constexpr auto func = &Access<A>::upload;

public:
  explicit unsafe(A& obj) : obj_(&obj) {
  }
  void upload(int x) {
    (obj_->*func)(x);
  }

private:
  A* obj_;
};
```

После чего мы можем написать

```
A a;
int value = a.compute(5);
if (a.has_connection())
  unsafe(a)->upload(value);
```

Вот подход, который опирается на UB

```
template<class T>
class unsafe_instance;
```

```
3
4  template<>
5  class unsafe_instance<A> : A {
6  public:
7    using A::upload;
8  };
9
10 unsafe_instance<A>& unsafe(A& x) {
11   return static_cast<unsafe_instance<A>&>(x);
12 }
13 // using
14 A a;
15 int value = a.compute(5);
16 if (a.has_connection())
17   unsafe(a).upload(value); // UB
```

Причина почему тут UB заключается в том, что мы кастуем ссылку на `A` к ссылке на производный класс `unsafe_instance<A>`. Это пока еще не UB, такое не запрещено. Но в последней строчке мы обращаемся по адресу памяти объекта `A`, как будто там лежит объект типа `unsafe_instance<A>`. Но его там нет, а это UB.

6.3.13 Обработка некорректных данных

Главная задача конструктора – построить объект в корректном состоянии. Потому очень важно проверять переданные пользователем данные и не надеяться ни на что, что не может быть проверено автоматически. Помните, нарушение инварианта класса – это ошибка программиста, которую надо исправлять. Такие ошибки надо проверять через `assert`. Я очень рекомендую завести в классе функцию проверяющую инвариант класса, и после этого делать проверку состояния класса через `assert`. По-хорошему `assert` надо ставить в каждом конструкторе, для проверки, что состояние класса корректно, в начале каждого метода для проверки условий на аргументы. И в конце каждого мутирующего метода для проверки, что метод отработал корректно. Вот пример для хранения упорядоченной пары.

```
1  class A {
2  public:
3    A(int a, int b) : min_(a), max_(b) {
4      assert(min_ <= max_ && "First argument must be not greater than the second.");
5    }
6  private:
7    int min_;
8    int max_;
9  };
```

Другой подход при построении упорядоченной пары – восстановить инвариант. Например можно присвоить их в другом порядке

```
1  class A {
2  public:
3    A(int a, int b) : min_(std::min(a, b)), max_(std::max(a, b)) {
4      assert(min_ <= max_);
5    }
6  private:
7    int min_;
8    int max_;
9  };
```

Для сложного класса, например если мы делаем дерево поиска, стоит выделить отдельную функцию для проверки инварианта

```
1  class BTree {
2  public:
3    void method() {
4      /*mutate this*/
5      assert(is_in_correct_state_(root));
6    }
7  private:
8    static bool is_in_correct_state_(Node* node) {
9      // Must not use asserts inside to avoid infinite recursion
10     return /*condition*/;
11   }
```

```
12
13    Node* root_;
14  };
```

Для удобства можно сделать много разных функций, проверяющих необходимые условия. Это сделает проверяемое условие более читаемым, так как название функции скажет, что именно мы проверяем. Если вы беспокоитесь, что такая проверка дорогая, то вы не зря беспокоитесь. Она может быть дорогой. Но, все `assert`-ы отключаются макросом `NDEBUG`. Потому к нему можно относиться как к автокомментируемому коду. Так же можно сделать свои различные кастомные `assert`-ы так, чтобы их можно было включать и отключать частями.

Бывает, что данные от пользователя нужно предварительно почистить. Например мы хотим хранить внутри класса вектор из положительных чисел, в конструкторе принимаем `std::vector<int>` и пользователь может передать нам вектор содержащий нули и отрицательные числа, тогда стоит вставлять в конструктор методы очистки данных. Причем есть два подхода:

1. сначала вставить грязные данные и потом их почистить
2. сначала почистить и потом уже вставить грязные данные

Давайте покажу, что я имею в виду на примере. Первый подход:

```
1  class A {
2  public:
3    A(std::vector<int> values)
4     : values_(std::move(values)) {
5      auto delete_from =
6          std::remove(values_.begin(), values_.end(), is_non_positive);
7      values_.erase(delete_from, values_.end());
8    }
9  private:
10   std::vector<int> values_;
11 };
```

Второй подход:

```
1  class A {
2  public:
3    A(std::vector<int> values)
4     : values_(clear_non_positive(std::move(values))) {}
5  private:
6    static std::vector<int> clear_non_positive(std::vector<int>&& values) {
7      auto delete_from = std::remove(values.begin(), values.end(), is_non_positive);
8      values.erase(delete_from, values_.end());
9      return values;
10   }
11
12   std::vector<int> values_;
13 };
```

Мне лично импонирует второй подход, где мы сначала данные приводим в порядок и уже корректные данные записываем внутрь класса. Так получается всю инициализацию провести в списке инициализации. Так же работа по очистке вектора выполняется функцией с говорящим названием и лучше читается, что именно мы сделали в конструкторе. Так же видны зависимости данных: из каких данных какие получаются в результате каких действий. Первый подход можно чуть улучшить, завернув метод очистки в приватную функцию, но тогда есть следующая проблема

```
1  class A {
2  public:
3    A(std::vector<int> values)
4     : values_(std::move(values)) {
5      clear_values();
6    }
7  private:
8    void clear_values() {
9      auto delete_from =
10         std::remove(values_.begin(), values_.end(), is_non_positive);
11     values_.erase(delete_from, values_.end());
12   }
```

```
13
14    std::vector<int> values_;
15  };
```

В такой имплементации метод `clear_values()` не принимает ничего и не возвращает ничего. Это не очень хорошо. Мы неявно манипулируем состоянием объекта. В таком случае чуть лучше будет явно передать данные

```
1   class A {
2   public:
3     A(std::vector<int> values)
4      : values_(std::move(values)) {
5       clear(&values_);
6     }
7   private:
8     static void clear(std::vector<int>* values) {
9       assert(values);
10      auto delete_from =
11          std::remove(values->begin(), values->end(), is_non_positive);
12      values->erase(delete_from, values->end());
13    }
14
15    std::vector<int> values_;
16  };
```

Так будет хотя бы видно, что именно вы чистите в вашем классе и не надо будет искать определение метода, чтобы понять на что именно в классе он влияет. Объект передается по указателю, чтобы подчеркнуть, что это out аргумент, который будет меняться. В целом проверку на `nullptr` тут можно не бояться, так как метод приватный и его никто из публичного интерфейса не будет дергать. Потому он не должен обрабатывать случай `nullptr`, этот случай означает ошибку программиста. Потому достаточно поставить `assert`, чтобы случайно не забыть это ожидание для функции очистки.

6.3.14 Хрупкие объекты

Хрупкими я буду называть объекты, которые по тем или иным причинам нарушают поведение value semantics. Давайте я приведу конкретные примеры и объясню, что с этим делать.

1. Объект нельзя перемещать и нельзя копировать. Есть несколько причин, почему объект может обладать этим свойством:

   (a) Это сторонний класс, у которого просто удалены все четыре оператора для копирования и перемещения. Например `std::mutex`.

   ```
   1  std::mutex m; // non-copyable, non-movable
   2
   3  struct B { // non-copyable, non-movable
   4    int x;
   5    double y;
   6    std::mutex m;
   7  };
   8
   9  std::vector<std::mutex> ms; // does not work
   ```

   Проблема полностью решается заворачиванием в `std::unique_ptr`.

   ```
   1  using MutexMovable = std::unique_ptr<std::mutex>;
   2
   3  MutexMovable m; // non-copyable, movable
   4  // copy does not make sense
   5
   6  struct B { // non-copyable, movable
   7    int x;
   8    double y;
   9    MutexMovable m;
   10 }; // copy does not make sense
   11
   12 std::vector<MutexMovable> ms; // does work
   ```

Обратите внимание, что копирование приобрести не получится, так как для мьютекса это просто не имеет смысла.

(b) Объект является адресуемым. Это значит, что в течение работы программы вы будете обращаться к этому объекту по его адресу и потому он должен лежать в точности по одному адресу и никогда не менять его. Это может требоваться для имплементации паттернов взаимодействия между объектов таких как observer-observable.

```
class B {};
class A {
public:
  void send_to(B* address, Message m);
};

A a;
std::vector<B> vec(1);
B* address = &vec[0];
a.send_to(&address, Message());
vec.push_back(B()); // reallocates
a.send_to(&address, Message()); // ups
```

Все опять решается `std::unique_ptr`. Надо заменить старый класс `B` на `B_Impl`, а в качестве `B` сделать обертку.

```
class B_Impl { /*Old class B*/ };
class B {
public:
  B_Impl* operator->() {return impl_.get();}
  const B_Impl* operator->() const {return impl_.get();}
  B_Impl* address {return impl_.get();}
  const B_Impl* address const {return impl_.get();}
private:
  std::unique_ptr<TImpl> impl_;
};
class A {
public:
  void send_to(B* address, Message m);
};

A a;
std::vector<B> vec(1);
auto* address = vec[0].address();
a.send_to(&address, Message());
vec.push_back(B()); // reallocates
a.send_to(&address, Message()); // still good
```

(c) Объект просто развалится, если его скопировать или переместить. Например внутри объекта переменные хранят указатели или ссылки друг на друга (это уже плохой признак, но есть ситуации когда приходится так делать и обычно только для адресуемых объектов).

```
class A {
  void function() const;
  std::function<void()> call_ = [this]() {function();};
};
```

В этом случае поступаем так же, как и в предыдущем случае. Заменяем `A` на `A_Impl`, а вместо `A` пишем обертку. Тут видно, что можно сделать шаблонную обертку для всех трех случаев.

```
template<class TImpl>
class Wrapper {
public:
  TImpl* operator->() {return impl_.get();}
  const TImpl* operator->() const {return impl_.get();}
  TImpl* address {return impl_.get();}
  const TImpl* address const {return impl_.get();}
private:
  std::unique_ptr<TImpl> impl_;
};

```

```
12  using MutexMovable = Wrapper<std::mutex>;
13  using B = Wrapper<B_Impl>;
14  using A = Wrapper<A_Impl>;
```

В случае мьютекса такая обертка лучше тем, что не позволяет сделать `reset` на мьютексе. В целом закрывать ненужный функционал – хорошая идея.

2. Константные поля в класса или структуре.

```
1  struct A {
2    const int x;
3  };
```

В этом случае стоит перейти от структуры к классу, закрыть переменную `x`, убрать `const` модификатор, а условие, что переменная `x` не меняет свое значение сделать инвариантом класса.

```
1  class A {
2  public:
3    A(int x) : x_(x) {}
4    int x() const {return x_;}
5  private:
6    int x_; // x = const is class invariant
7  };
```

3. Класс содержит ссылку на данные.

```
1  struct A {
2    const int& data;
3  };
```

В целом это плохой признак, потому что данные по ссылке могли умереть или переехать в другое место. Тем не менее, такая ситуация бывает встречается. Например, если у вас есть механизм обеспечения, что данные точно живы и данные тяжелые и вы хотите передавать их дешево между функциями. Первый вариант решения проблемы – перейти на указатели

```
1  struct A {
2    const int* data;
3  };
```

Вы тут же приобретаете value semantics. Однако, теперь вам надо обрабатывать случай `nullptr`. Если это проблема, то можно воспользоваться специальным библиотечным классом.

```
1  struct A {
2    std::reference_wrapper<const int> data;
3  };
```

Эта штука ведет себя как указатель, но при этом не бывает `nullptr`. Кроме-того, он неявно кастуется к ссылке. Тем не менее, это все еще плохо из-за наличия ссылки, для которой вам нужны глобальные гарантии существования. Это делает ваш код хрупким и ломает local reasoning. Вам надо не забывать про гарантии существования объекта, на который вы ссылаетесь.

4. Класс собран внутри на указателях и процедура копирования – дорогая операция. Например дерево или граф собранный на нодах с указателями. Если при этом вы хотите построить такой объект один раз и потом никогда не будете его менять, но его надо активно копировать и передавать между потоками, то есть дешевый способ сделать объект легко копируемым, заплатив цену невозможности его менять никогда.

```
1  struct Node {
2    Node* parent;
3    Node* left;
4    Node* right;
5  };
6
7  class Tree {
8  public:
```

```
  ...
private:
  Node* root_;
};

const Tree t;
```

В этом случае можно завернуть объект в `std::shared_ptr<const Tree>`. В этом случае вы можете задешево копировать и даже передавать объект между thread-ами безболезненно. Однако, тут есть одно предостережение. Если вы хотите получить доступ к вершинам этого дерева, то нельзя пользователю выдавать указатель на `Node`. Даже указатель на `const Node` выдавать – плохая идея. Действительно, пусть у вас есть

```
class Tree {
public:
  ...
  const Node* root() const {
    return root_;
  }
private:
  Node* root_;
};

std::shared_ptr<const Tree> t = /*initialization*/;
const Node* croot = t->root();
Node* root = croot->left->parent;
// now you can modify t via root
```

Проблема в том, что `const` не проносится сквозь указатели. Потому для корректной поддержания константности вам надо написать аналог константных итераторов для доступа к элементам такой структуры, чтобы избежать подобных проблем. В целом, для работы с такой структурой и корректного поведения константности вам придется написать аналоги таких итераторов.

6.3.15 Менеджмент ресурсов и RAII

«Resource Acquisition is Initialization» – это волшебная фраза, которая говорит о том, как заворачивать ресурсы в обертки в системе с поддержкой исключений. В разделе 4 мы уже обсуждали зачем нужно заворачивать парные операции в обертки. Напомню, что есть две основные причины:

1. Из-за наличия исключений вы можете выйти из любого scope в неожиданном месте и только выполнение деструкторов локальных объектов гарантируется для исполнения.

2. Автоматизация исполнения парных операций.

Вот соответствующий пример:

```
void g() {
  throw std::runtime_error("Ups!");
}

void f() {
  int* x = new(0);
  g();
  delete x;
}
```

В строчке 6 вы явно выделяете память, а в строчке 7 бросается исключение и 8-я строчка никогда не вызовется. Как вообще работает код в функции `f`. Давайте опишу по строчкам

1. В строчке 6 происходит сразу несколько вещей.

   (a) Выделение памяти на стеке для указателя `x`.

   (b) Вызов оператора `new` для выделения памяти под переменную типа `int`, помещаем туда значение `0`.

   (c) Пишем адрес памяти в переменную `x`.

2. Вызываем функцию `g`.

3. Функция `g` бросает исключение. В этом месте мы вываливаемся из функции `g` в строчку 7 из функции `f`.

4. В строчке 7 после исключения мы должны выйти из функции `f` и будем так выходить до тех пор, пока не встретим `catch`, который поймает исключение. Но чтобы выйти мы должны почистить стек от функции `f`. Это называется stack unwinding. Надо удалить все локальные переменные и аргументы функции `f`. Для этого после строчки 7 вызывается деструктор `x`, а так как это встроенный тип, то никакого деструктора нет и мы просто уменьшаем стек и выкидываем переменную `x` со стека.

5. Почистив стек мы выходим в больший scope из которого вызвана `f` и продолжаем вываливаться дальше.

У этого процесса есть две вещи, о которых надо обязательно поговорить.

1. Безопасное выделение ресурсов. Чтобы безопасно выделить и удалить память, или любой другой ресурс, выделение ресурса надо оборачивать в специальные обертки. Для памяти такой wrapper называется `std::unique_ptr`. Но есть и другие версии. Идея его в следующем.

```
1  struct IntPtr {
2    IntPtr(int value) : ptr_(new int(value)) {}
3    ~IntPtr() { delete ptr_;}
4    int* ptr_;
5  };
6
7  void f() {
8    IntPtr x(0);
9    throw std::runtime_error("Exception");
10 }
```

   Теперь в строчке 8 происходит выделение памяти в конструкторе `IntPtr`. А при выходе из `f` по исключению, во время stack unwinding вызовется деструктор для `x` и удалится выделенная память. Такой механизм позволяет гарантированно выполнять действия при выходе из любого scope.

2. Обратите внимание на тонкий момент в предыдущей схеме. А что если во время stack unwinding вы бросите исключение? По стандарту – это недопустимо. Программа упадет в run-time вызовом `terminate`. Потому никогда не бросайте исключения в деструкторах. Это чревато и не работает. У вас нет шансов сообщить об ошибке в деструкторе безопасно. Их надо обрабатывать внутри самого деструктора. Но лучше не кидать никогда.

Несколько ресурсов Теперь, что делать, если вам надо выделить память для двух указателей? Тупой вариант такой

```
1  struct IntPtrPair {
2    IntPtrPair(int x, int y) : ptr1_(new int(x)), ptr2_(new int(y)) {}
3    ~IntPtrPair() {
4      delete ptr1_;
5      delete ptr2_;
6    }
7    int* ptr1_;
8    int* ptr2_;
9  };
10
11 IntPtrPair x(1, 2);
```

Беда этого подхода вот в чем. Когда вы в строчке 11 вызываете конструктор, то сначала выделяется память под `ptr1_`, а потом под `ptr2_`. И если при первом вызове память выделится, а во втором нет и будет брошено исключение. То конструктор не завершит работу, а значит объект не будет считаться построенным, а значит и не вызовется деструктор для него во время stack unwinding. А это значит, что память потечет. Есть много костылей для этой проблемы, но правильное решение – обернуть каждый указатель в свой wrapper. Например так

```
1  struct IntPtrPair {
2    IntPtrPair(int x, int y) : ptr1_(x), ptr2_(y) {}
3
```

```
  IntPtr ptr1_;
  IntPtr ptr2_;
};

IntPtrPair x(1, 2);
```

И теперь вам вообще не надо беспокоиться о деструкторах и прочих радостях. Все это за вас для каждого ресурса делает wrapper для работы с одним указателем. На практике, конечно, надо пользоваться библиотечными опциями, тут подойдет `std::unique_ptr`.

Несколько зависимых ресурсов Существует еще одна ситуация, о которой стоит рассказать. А что если у нас данные зависимы? Например, я должен сначала выделить память под `x`, а потом, выделяя память под `y`, я должен туда положить информацию про `x`? То есть у вас есть естественная зависимость порядка конструирования данных. А для конструирования по частям как раз и создано наследование. Оно собирает объект в определенном порядке. Давайте приведу такой пример

```
1  struct A {
2    A(int x) : ptr_(std::make_unique<int>(x)) {}
3    std::unique_ptr<int> ptr_;
4  };
5
6  struct B : A {
7    A(int x)
8     : A(x),
9       ptr_(std::make_unique<int*>(A::ptr_.get()) {}
10    std::unique_ptr<int*> ptr_;
11 };
12
13 B x(1);
```

В таком случае у вас в строчке 13 сначала создается базовая часть объекта, то есть выделяется память под `x` и кладется адрес в `A::ptr_`. А потом вы выделяете память под `B::ptr_` и кладете туда адрес `A::ptr_` из базовой части. Понятно, что это бессмысленная деятельность, но демонстрирует зависимость данных. Тут прошу обратить внимание на несколько вещей:

1. Я везде пользуюсь wrapper-ами для указателей и мне вообще не надо задумываться про деструкторы.
2. Обратите внимание, что нигде не пользуюсь `new` явно, вместо этого пользуюсь `std::make_unique` функцией.
3. Если честно, то `std::unique_ptr` не самая высоко оптимизированная вещь.[18] Но для большинства ситуаций этой обертки хватает на ура. Правда деревья я бы на ее основе не собирал, как минимум убьете стэк в деструкторе или получите неописуемые тормоза.

Все что я написал про память релевантно и для выделения других ресурсов. Например для мьютексов используются `std::lock_guard` или похожие механизмы. Любой запрос к операционной системе, который требует освобождения своих ресурсов, должен выполняться через wrapper, где в конструкторе вы запрашиваете ресурс, а в деструкторе освобождаете. Если вы используете наследование для последовательного выделения ресурсов, то вы можете спокойно бросать исключения в конструкторах, ибо деструкторы всех базовых классов отработают как надо. Это уже не искусственный пример, где много разных взаимозависимых ресурсов требуют своего последовательного выделения или освобождения. Наследование позволяет это сделать удобным и контролируемым. Но надо пописать какой-то код. Обратите внимание, что в имплементации методов я даже позволяю себе бросать исключения в конструкторах, при наличии ошибки. Тогда все выделенные ресурсы в базовой части автоматом почистятся и не надо переживать на эту тему. Напоследок хочу сказать еще одно замечание про реализацию выделения памяти под зависимые данные. Есть другая альтернатива

```
1  struct A {
2    A(int x)
3     : ptr1_(std::make_unique<int>(x)),
4       ptr2_(std::make_unique<int*>(ptr1_.get()) {}
5
6    std::unique_ptr<int> ptr1_;
7    std::unique_ptr<int*> ptr2_;
```

[18] Кто-то, читая это, сейчас должен был брызнуть слезами смеха и отчаяния от моей аккуратности в формулировках.

```
};

A x(1);
```

Плюс такого подхода – не вызываются лишние конструкторы, все делается в одном. При сложной цепочке вложенных конструкторов гипотетически можно потратить много на накладные расходы. Опять же, это все нужно измерять и я никогда не испытывал с этим проблемы, но быть они могут. Минус в том, что этот код хрупкий. Вы опираетесь на порядок расположения данных в коде. А именно, вариант ниже

```
1  struct A {
2    A(int x)
3     : ptr1_(std::make_unique<int>(x)),
4       ptr2_(std::make_unique<int*>(ptr1_.get())) {}
5
6    std::unique_ptr<int*> ptr2_;
7    std::unique_ptr<int> ptr1_;
8  };
9
10 A x(1); // incorrect object state
```

порадует вас ошибкой в run-time. Ваш объект будет находиться в некорректном состоянии. Дело в том, что данные конструируются не в том порядке, в каком они идут в списке инициализации (строки 3 и 4), а в каком они объявлены в классе (строки 6 и 7). Так что у вас сначала выполнится строка для инициализации `ptr2_` и в ней вы обратитесь к функции `get` неинициализированной `ptr1_` и получите `nullptr`. И только потом инициализируете `ptr1_`. Как вы видите, после отработы конструктора `ptr2_` не будет содержать адрес данных из `ptr1_`. И такие ошибки хрен найдешь. Причем совершить их очень легко – достаточно переставить данные. А со временем вы попросту забудете, что ваши данные были зависимы, ваш код обрастет дополнительными костылями и вы просто не заметите, что это именно эта ситуация. Причем, код скомпилируется, запустится и даже отработает, пока вы не упадете на каком-нибудь тонком тесте.

6.3.16 Наследование и композиция

Давайте сразу посмотрим на следующие два примера:

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

struct B {
  A a;
  char z;
};
```

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

struct B : A {

  char z;
};
```

В обоих случаях класс `B` содержит внутри себя данные класса `A`. Давайте обсудим разницу между этими двумя подходами и как она проявляется в имплементации и при использования. Если говорить про имплементацию (на сколько это возможно при условии, что в плюсах все аксиоматически определено), то конкретно в этих примерах представление класса `B` в памяти будет одинаковым. Однако, важным отличием будет использование объекта класса снаружи.

```
B b;
b.a.x = 1;
b.a.y = 2.;
b.z = 'c';
```

```
B b;
b.x = 1;
b.y = 2.;
b.z = 'c';
```

Как мы видим, в обоих случаях мы складываем данные вместе в одну структуру, но только в первом примере данные структуры `A` группируются внутри отдельной сущности `b.a`, а во втором случае данные структуры `A` напрямую помещаются внутрь `B` вместе с ее данными. То же самое происходит и при использовании методов.

```
struct A {
  void f() const;
};

struct B {
  A a;
  void g() const;
};
```

```
struct A {
  void f() const;
};

struct B : A {

  void g() const;
};
```

При использовании вы получите

```
B b;
b.a.f();
b.g();
```

```
B b;
b.f();
b.g();
```

Получается, что первый метод объединяет данные и методы двух классов `A` и `B` с наличием иерархии, то есть для доступа к методам `A` надо использовать дополнительное промежуточное имя (имя переменной). А во втором методе, мы просто скидываем все данные и методы в одну кучу и сливаем интерфейсы классов. Обычно принято первый подход называть композицией. Я бы сказал, что оба подхода являются композицией, но просто первый подход – это вертикальная композиция, когда вы создаете иерархию на интерфейсах, а второй подход – горизонтальная композиция, когда вы сливаете интерфейсы.

Важная тонкость – вы не можете унаследоваться дважды от одного и того же класса, то есть вот так написать нельзя

```
struct A {
  void f() const;
};

struct B : A, A {
  void g() const;
};
```

В этом случае при обращении

```
B b;
b.f(); // an issue
```

не ясно к какому из двух базовых классов вы пытаетесь обратиться. В языке нет никакого дополнительного механизма тут для различения базовых классов кроме их имени. Потому использование одного и того же имени создает нерешаемую проблему идентификации. Потому такое запрещено. В случае же вертикальной композиции, у вас есть дополнительный параметр – имя переменной класса `A`, что позволяет помещать в класс много переменных одного и того же типа.

```
struct A {
  void f() const;
};

struct B {
  void g() const;
  A a1;
  A a2;
};
```

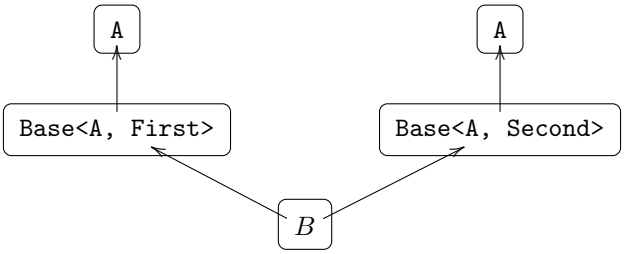
В этом случае при обращении

```
B b;
b.a1.f(); // resolved
b.a2.f(); // resolved
```

На самом деле можно обойти косвенно ограничение на наследование. Для этого надо просто добавить в имя класса некоторый тег, что делается с помощью шаблонных адаптеров.

```
template<class T, class Tag>
struct Base : T {};

struct B : Base<A, struct First>, Base<A, struct Second> {};
```

В этом случае иерархия наследования выглядит так



В любом случае пользоваться потом этим не очень круто, только если это не библиотечный код спрятанный от программиста.

Сравнение Обычно принято противопоставлять «композицию» и «наследование», потому что под «наследованием» подразумевается наследование с виртуальными методами (подробнее стоит посмотреть раздел 11.3). Однако, если вы не используете базовый класс как интерфейс, то это та же самая композиция, при которой вы просто сливаете интерфейсы вместе. Такая техника использования наследования хорошо описана в книге Alexandrescu Modern Design of C++ в первой главе названной Policy Based Design. Другое название у этого подхода Mixings, когда базовые классы используются как компоненты наследника, то базовые классы называются Mixings. Обычно принято говорить, что композиция лучше наследования. Последнее правильнее с моей точки зрения выразить по-другому: композиции (вертикальная и горизонтальная) лучше наследования с виртуальными методами. После такой переформулировки на самом деле видно, что это странное утверждение, потому что они используются для разных целей. А как только вы сравниваете инструменты для разных задач, то ответ будет все время варьироваться от выбора задачи. Тем не менее, композиция в моем смысле (обе версии) обладают value semantics и safe by default. С помощью Policy Based Design можно легко собирать классы из настраиваемых компонент.

У обоих видов композиции есть свои достоинства и недостатки. Мы уже отмечали, что в случае горизонтальной композиции (то есть наследования) сложно слить два одинаковых интерфейса напрямую. С другой стороны, у наследования есть одно важное преимущество, которое делает Policy Based Design таким востребованным. Я говорю про доступ к производному классу. В случае вертикальной композиции это аналогично доступу к классу из его поля. Чтобы было понятно, что имеется в виду, рассмотрим следующие два примера:

Виды композиции

Вертикальная

```
struct A {
  void g() const;
  B b
  int y;
};
```

```
struct B {
  void f() const {
    const A* host =
      /*how to get the host?*/
    /*using the host*/
  }
  int x;
};
```

Горизонтальная

```
struct A : B {
  void g() const;

  int y;
};
```

```
struct B {
  void f() const {
    const A* derived =
      static_cast<const A*>(this);
    /* using derived */
  }
  int x;
};
```

Обратите внимание, что в случае наследования (горизонтальной композиции), на уровне языка можно получить адрес производного класса, который включает в себя базовый класс. Ровно для этих целей в языке и существует конструкция `static_cast`. А вот в случае вертикальной композиции в языке просто нет поддержки получения адреса хозяина. Точнее есть один пережиток из Си, которым можно было бы заменить подобный механизм и называется он `offsetof`. Можно попытаться сделать что-то вроде

```
1 struct B {
2   void f() const {
3     const A* host = this - offsetof(A, b);
```

```
    /*using the host*/
  }
  int x;
};

struct A {
  void g() const;
  B b;
  int y;
};
```

Однако у этой конструкции есть некоторые проблемы. Вот что об этом говорит нам информация с cppreference:

- **until c++11** If `A` is not a POD the result is undefined
- **until c++17** If `A` is not a standard-layout the result is undefined
- **since c++17** If `A` is not a standard-layout use of the offsetof macro is conditionally-supported. Прекрасно...

То есть эта операция нужна в Си чтобы двигаться от поля структуры к базовой структуре и ни для чего больше она не годится. В остальных случаях она может быть даже сработает. Но все приличные компиляторы будут вам пилить глаза предупреждениями при любом использовании. В итоге язык просто не предоставляет возможность из поля класса достучаться в родителя. Эта проблема будет еще часто возникать. Мы ее еще увидим при имплементации observer pattern. Но самое смешное ничего не мешает поддержать эту операцию, потому что у компилятора достаточно информации, для того чтобы решить эту задачу. Эта задача ничего не отличается от downcast-а при наследовании. Ровно с теми же самыми ограничениями и опасностями.

6.3.17 Policy Based Design

Давайте посмотрим на пример использования вертикальной композиции для сборки разных классов из компонент, чтобы избежать повторения кода. Предположим у вас есть структура данных – бинарное дерево и мы хотим для него сделать итератор. У нас есть несколько выборов:

1. Обращение к данным константное или нет.
2. Какой порядок обхода по элементам.

В таком случае можно распилить класс итератора на части и собирать разные виды итераторов без дублирования кода. Прежде всего отметим, что константный или не константный итератор отличаются только типами данных. А вот за навигацию по дереву отвечают разные методы. Предположим, что мы реализовали константный итератор для какого-то фиксироанного метода обхода дерева и пусть он выглядит так:[19]

```
template<class T>
class ConstIterator {
public:
  using value_type = const T;
  using difference_type = ptrdiff_t;
  using pointer = const T*;
  using reference = const T&;
  ...
  pointer operator->() const;
  reference operator*() const;
  ConstIterator& operator++();
  ...
private:
  void move_to_next();
  pointer ptr_;
};
```

Чтобы не дублировать код для константного и не константного итератора можно выделить все псевдонимы в базовый класс. И сделать два базовых класса: для константной и не константной версии.[20]

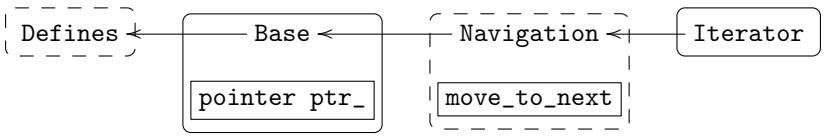
[19] Скорее всего тут будет лежать указатель на вершину дерева, а не на данные в вершине дерева. Но эту мелочь я предлагаю игнорировать сейчас, как неважную деталь.

[20] По-хорошему надо удалять с типа `T` все модификаторы и квалификаторы и прочие трюки, но я это здесь игнорирую.

```cpp
template<class T>
struct ConstDefines {
  using value_type = const T;
  using difference_type = ptrdiff_t;
  using pointer = const T*;
  using reference = const T&;
};

template<class T>
struct NonConstDefines {
  using value_type = T;
  using difference_type = ptrdiff_t;
  using pointer = T*;
  using reference = T&;
};
```

```cpp
template<class Defines>
class Iterator : Defines {
public:
  ...
  pointer operator->() const;
  reference operator*() const;
  Iterator& operator++() {
    move_to_next();
    return *this;
  }
  ...
private:
  void move_to_next();
  pointer ptr_;
};
```

Еще по хорошему справа понадобится переопределить все псевдонимы внутри самого класса. В любом случае это позволяет писать один и тот же код для константной и не константной версии итератора. Чтобы настроить метод обхода дерева, то надо распилить итератор следующим образом.



Здесь на стрелка ведет от наследника к базовому классу. Пунктиром отмечены настраиваемые блоки.

1. `Defines` как и прежде содержит псевдонимы типов для итератора. Этим можно влиять на константность или не константность итератора.
2. `Base` содержит данные итератора, в нашем случае мы предположили, что это всего лишь указатель, но это не принципиально.
3. `Navigation` содержит метод для навигации, который будет вызываться в `operator++`.
4. `Iterator` это финальный класс собранного итератора.

В коде это все может выглядеть так

```cpp
template<class Defines>
class Base : Defines {
public:
  pointer operator->() const;
  reference operator*() const;
protected:
  pointer ptr() const;
private:
  pointer ptr_;
};

template<class Navigation>
class Iterator : Navigation {
public:
  ...
  Iterator& operator++() {
    Navigation::move_to_next();
    return *this;
  }
  ...
};
```

```cpp
template<class Base>
class Navigation1 : Base {
protected:
  void move_to_next() {
    /*Implementation*/
  }
};

template<class Base>
class Navigation2 : Base {
protected:
  void move_to_next() {
    /*Implementation*/
  }
};

template<class Base>
class Navigation3 : Base {
protected:
  void move_to_next() {
    /*Implementation*/
  }
};
```

Тогда создание конкретного итератора выглядит так

```
template<class T>
using ConstIterator1 = Iterator<Navigation1<Base<ConstDefines<T>>>>;

template<class T>
using Iterator1 = Iterator<Navigation1<Base<Defines<T>>>>;
```

Такой подход лучше использования указателей на функцию, так как использует прямой вызов метода по имени, а не по его адресу.

6.3.18 Зависимости класса и Pimpl

Начнем со следующего примера класса

```
// file.h

class A {
public:
  void f(int) const {
    /*Implementation*/
  }
protected:
  void g(double) const {
    /*Implementation*/
  }
private:
  void h(char) const {
    /*Implementation*/
  }
};
```

Теперь давайте подумаем, какие есть зависимости у пользователя этого класса, при его использовании. Есть два вида зависимостей

1. Прежде всего есть синтаксическая зависимость, которая означает, что если я захочу поменять класс `A`, то пользователю этого класса придется что-то менять у себя в коде.

2. Кроме того, есть еще зависимость при компиляции, а именно, если я внесу изменения в класс `A`, то пользователю класса нужно: перекомпилировать свой код, перелинковать свой код.

С перелинковкой мы ничего сделать не сможем. Если уж данные поменяли формат или метод изменил набор инструкций для исполнения, нам надо будет подтянуть новые данные из бинарного файла. А вот перекомпиляции хотелось бы часто избежать, учитывая, что сама модель плюсов рассчитана на минимизацию вызова компилятора и перелинковку изменившихся частей кода. Давайте для определенности напишем какой-то пользовательский код

```
#include<file.h>

class B : protected A {
public:
  void g(double value) const {
    A::g(value);
  }
};

int main() {
  A a;
  a.f(1);
  return 0;
}
```

В начале проигнорируем вопрос компиляции и линковки и посмотрим на синтаксическую зависимость. Если в строчке 5 в листинке класса `A` интерфейс функции не меняется на более узкий, то пользователю класса не надо будет ничего менять в своем коде. Например такое изменение

```
void f(int, int z = 1) const { /*Implementation*/ }
```

не изменит код пользователя, не надо будет ничего переписывать в `main.cpp`. Аналогично обстоят дела с методом `A::g`. Только при изменении его сигнатуры придется вносить изменения не всюду, а только в классах наследниках. Изменение приватной части класса `A` вообще не влияет на пользовательский код, это и есть сила инкапсюляции, которая ставит четкие границы.

А теперь обратим внимание на то, что если весь класс `A` объявлен в заголовочном файле `file.h`, то любые изменения в этом файле приведут к перекомпиляции всех, кто его инклудит. Первый механизм, который приходит на ум – выделение h и cpp файлов так:

```
// file.h

class A {
public:
  void f(int) const;
protected:
  void g(double) const;
private:
  void h(char) const;
};
```

```
// file.cpp

void A::f(int) const {
  /*Implementation*/
}
void A::g(double) const {
  /*Implementation*/
}
void A::h(char) const {
  /*Implementation*/
}
```

Теперь внесение изменений в имплементацию любого метода не вызовет перекомпиляцию на стороне пользователя. Однако, если я добавлю в класс приватный метод, то он вызовет изменение файла `file.h`, а потому автоматически вызовет перекомпиляцию всех пользователей.

```
1  class A {
2  public:
3    void f(int) const;
4  protected:
5    void g(double) const;
6  private:
7    void h(char) const;
8    int w(int) const; // recompilation of users
9  };
```

Несмотря на то, что добавление или удаление приватных методов не меняет код пользователя, он заставляет его перекомпилироваться. Проблему добавления методов можно решить следующим образом. В начале надо метод переделать в статический

```
1  class A {
2  public:
3    void f(int) const;
4  protected:
5    void g(double) const;
6  private:
7    void h(char) const;
8    static int w(const A*, int); // recompilation of users
9  };
```

А теперь надо вынести его и спрятать в `file.cpp` в anonymous namespace.

```
1  // file.cpp
2
3  namespace {
4  int w(const A*, int) {
5    /*Implementation*/
6  }
7  }
8
9  void A::f(int) const {
10   /*Implementation*/
11 }
12 void A::g(double) const {
13   /*Implementation*/
14 }
15 void A::h(char) const {
16   /*Implementation*/
17 }
```

И так можно прятать все вспомогательные функции. Однако, есть еще одна проблема. Что если мы добавим приватное данное в класс `A`?

```
1  class A {
2  public:
3    void f(int) const;
4  protected:
5    void g(double) const;
6  private:
7    void h(char) const;
8    int data_; // recompilation of users
9  };
```

В этом случае пользователю все равно придется перекомпилироваться, ибо это внесение изменения в файл, который инклудится пользователем. Чтобы избежать перекомпиляции в этом случае можно воспользоваться так называемой Pimpl idiom. Расшифровывается это так – pointer implementation. Как понятно из названия, нужно всего лишь спрятать приватную имплементацию класса внутри указателя. Идейно мы бы хотели написать следующий код:

```
1  // file.h
2  #include<memory>
3
4  namespace detail {
5  class ImplA; // forward declaration
6  }
7
8  class A {
9  public:
10   void f(int) const;
11 protected:
12   void g(double) const;
13 private:
14   std::unique_ptr<ImplA> impl_;
15 };
```

```
// file.cpp
#include<file.h>

namespace detail {
class ImplA {
  /*Implementation*/
};
}

void A::f(int) const {
  /*Implementation*/
}
void A::g(double) const {
  /*Implementation*/
}
```

Это решение хорошо всем кроме одного – оно не будет компилироваться. Давайте я объясню, что пойдет не так и как это надо исправлять. Давайте напишем код

```
1  #include<file.h>
2
3  int main() {
4    A a;
5    return 0;
6  } // compilation error
```

Обратите внимание, где происходит ошибка компиляции. Причина в том, что мы используем `std::unique_ptr` на `ImplA` внутри класса `A`. Однако, класс `ImplA` объявлено, но не определен в строке 5 в `file.h`. Теперь вспомним, что компилятор для каждого класса должен сгенерировать его деструктор. При компиляции файла `file.cpp` сгенерировать деструктор не получится, потому что компилятор C++ смотрит только в прошлое. Так как `ImplA` определен после `A`, то при генерации деструктора для `A`, компилятор еще не знает, как устроен класс `ImplA` и не знает как его создавать и разрушать. Теперь при компиляции файла `main.cpp` при выходе из функции `main` должен быть вызван деструктор для `A` и в этот момент компилятор скажет вам, что у данного класса нет деструктора. Решается эта проблема следующим образом. Надо объявить дефолтный деструктор после имплементации класса `ImplA` вот так

```
// file.h
#include<memory>

namespace detail {
class ImplA; // forward declaration
}

class A {
public:
  ~A();  // declaration
  void f(int) const;
protected:
  void g(double) const;
private:
  std::unique_ptr<ImplA> impl_;
};
```

```
// file.cpp
#include<file.h>

namespace detail {
class ImplA {
  /*Implementation*/
};
}
A::~A() = default;  // definition

void A::f(int) const {
  /*Implementation*/
}
void A::g(double) const {
  /*Implementation*/
}
```

Вот теперь это решение работает. Мы можем вносить любые изменения в `ImplA` и это никак не отразится на пользователе класса `A`. Так как в `file.h` остался по сути только интерфейс для `A`: публичный и защищенный, то если вы не собираетесь менять интерфейс, то и перекомпилировать ничего не придется. Обратите внимание, что класс `A` становится легко перемещаемым и даже адресуемым. Однако, его нельзя копироваться. Если такая операция нужна, то придется имплементировать копирующие операции вручную. Не забудьте, что в этом случае нужно определять все дефолтные методы.

6.3.19 Признаки плохого класса

1. Класс содержит указатель на ресурс и занимается чем-то кроме менеджмента ресурса.

```
1  class A {
2  public:
3    double compute_something() const {
4      /*some algorithm with data_*/
5    }
6  private:
7    Data* data_;
8  };
```

   Вы либо менеджерите ресурс, либо имплементируете бизнес логику. В данном случае можно ввести промежуточный класс для менеджмента данных.

```
1  class A {
2  public:
3    double compute_something() const {
4      /*some algorithm with data_*/
5    }
6  private:
7    std::unique_ptr<Data> data_;
8  };
```

   Либо если вам нужна возможность копировать данные, вам придется написать свою обертку аналогичную `std::unique_ptr`, которая будет не только мувать, но и копировать данные.

   Обратите внимание, что бывают исключения из этого правила в следующем смысле. Если вы реализуете структуру данных на указателях, то не всегда хорошая идея отдавать вопрос владения узлам. Давайте я приведу пример. Пусть вы хотите написать бинарное дерево с корнем и не мучиться с владением ресурсами, то можно сделать так

```
1  struct Node {
2    Node* parent;
3    std::unique_ptr<Node> left;
4    std::unique_ptr<Node> right;
5  };
6
7  class Tree {
8  public:
9    ...
10 private:
```

```
11    std::unique_ptr<Node> root_;
12  };
```

Вы действительно избавились от необходимости заниматься менеджментом ресурсов в дереве. Однако, вы приобрели две проблемы:

(a) При вызове деструктора `Tree`, вы должны будете уничтожить `root_`. Внутри деструктор вызовет деструкторы своих детей, они деструкторы свдоих детей и т.д. При большой глубине дерева это может вызвать переполнение стека.

(b) Производительность. Да, я попадал в ситуации, когда замена сырых указателей на `std::unique_ptr` прям кардинально бьет по производительности. Но это вопрос измерений в вашей конкретной ситуации.

Если такое решение вам не подходит, то тогда можно перенести ответственность за менеджмент памяти уже на саму структуру данных `Tree`. В таком случае можно очистить дерево в цикле за $O(n)$.

2. Класс содержит указатель на несколько ресурсов и ведет их менеджмент.

```
1   class A {
2   public:
3     A();
4     A(const A&);
5     A(A&&) noexcept;
6     A& operator=(const A&);
7     A& operator=(A&&) noexcept;
8     ~A();
9   private:
10    B* b;
11    C* c;
12  };
```

Плохая идея менеджерить два ресурса. Ибо вам будет очень больно и сложно написать корректный код, устойчивый к исключениям. Для этого напишите два вспомогательных класса для менеджмента ресурсов, либо используйте `std::unique_ptr` или его налоги, если они подходят.

```
1   class A {
2   public:
3     A() = default;
4   private:
5     std::unique_ptr<B> b;
6     std::unique_ptr<C> c;
7   };
```

Бывают исключения из этой ситуации, вроде `std::vector`, когда задача менеджерить все объекты одновременно в одной области памяти, но даже в этом случае можно сделать имплементацию грамотно, а можно нет.

3. Класс содержит хрупкий объект. Если написание хрупкого объекта – это часть имплементации какого-то шаблона проектирования или структуры, то это не страшно, но снаружи класс должен обладать value semantics. Это значит, что вы должны принять меры, чтобы обеспечить value semantics, как минимум не забыть заняться этим вопросом. В остальных же случаях, вы должны избавиться от проблемы. Как это сделать, можно посмотреть в разделе 6.3.14.

(a) Класс содержит неперемещаемый и некопируемый объект.

```
1   class A {
2     std::mutex m;
3   };
```

Как решать описано в разделе 6.3.14 пункт 1;

(b) Класс содержит константные данные.

```
1   class A {
2     const int x;
3   };
```

Как решать описано в разделе 6.3.14 пункт 2;

(c) Класс содержит ссылку на данные.

```
class A {
  const int& x;
};
```

Для начала посмотрите в разделе 6.3.14 пункт 3, чтобы понять как топорно получить value semantics. Однако, надо понимать, что наличие ссылок на какие-то внешние объекты – это всегда плохой признак. Это скорее всего означает, что ваши объекты должны уметь коммуницировать на расстоянии не видя друг друга. В таком случае обычно нужно использовать подходящий паттерн для общения объектов. Самый популярный является observer pattern. Он как раз гарантирует надежное общение между объектами.

4. Класс содержит `shared_ptr` на неконстантный объект.

```
class A {
  std::shared_ptr<B> b;
};
```

Эта ситуация обычно означает, что вы вообще не понимаете, что вы делаете. Вы по сути говорите в коде, что понятия не имеете кто и когда владеет объектом `b` и черт его знает когда он живет. В большинстве случаев можно просто перейти на `std::unique_ptr`, либо написать аналог, который еще умеет копировать данные (как это делать можно посмотреть в разделе 6.3.5). Другая ситуация – заменить на `std::shared<const B>`. Ситуации, когда же надо действительно использовать `std::shared_ptr` очень редкие и если вы про них не знаете, то это признак того, что вы его точно используете не по назначению (см. раздел 10.3).

7 Декораторы

7.1 Измерение времени выполнения функции

Если по простому, то декоратор, это такой паттерн, который добавляет к какому-либо действию набор автоматически выполняемых действий. Например, если мы хотим замерить время выполнения функции `f`, то мы хотим добавить код засекающий стартовое время перед вызовом функции, а потом посчитать пройденное время после вызова функции

```
1  void f(int, double);
2
3  int main() {
4    Time start = Timer.now();
5    f(5, 42.);
6    Time elapsed = Timer.now() - start;
7    return 0;
8  }
```

Но представьте, что мы теперь хотим измерить в произвольной точке программы время вызова какого-то метода. Для этого нам придется руками прописывать две строчки кода, которые измеряют время. Кроме того, если надо вычислить время работы двух методов в одном scope, нам придется называть по разному переменные `start` и `elapsed`. Как минимум тут видно две проблемы с таким подходом:

1. Мы будем повторять однотипный код
2. Коллизия имен

Кроме этого, у нас есть еще две проблемы, которые я еще буду обсуждать отдельно

1. Нелокальность. Дело в том, что строчки 3 и 5 в коде выше связаны друг с другом по смыслу и по выполняемой работе, но они у нас разделены другим кодом. И да, пусть в этом месте это одна строчка, но это может быть много строчек кода. И потому чтобы понять смысл строчки 5 надо знать строчку 3, которая могла быть далеко до этого. Мы разрываем связанную работу.
2. Нарушение уровня абстракции. Это означает, что в строчке 4 выше вызываем одну функцию, то есть мы даем команду высокого уровня, не производя ее руками. А в строчках 3 и 5 мы вместо вызова функций, которые делают нужную работу, делаем ее прям тут же в коде руками. То есть строчки 3 и 5 выполняют более мелкую работу, чем строчка 4.

Кроме того, важно отметить, что мы не хотим лезть внутрь функции `f`. Во-первых, это не решит проблемы с повторением кода, если мы захотим измерить время работы другой функции. Во-вторых, это может быть сторонняя функция, которая нам не доступна.

Для автоматизации процесса измерения времени предлагается сделать очень простую вещь: давайте заведем промежуточную функцию, которая будет выполнять всю нужную работу по вычислению времени, и звать функцию `f`, а именно так

```
1  void f(int, double);
2
3  void time_of_f(int x, double y) {
4    Time start = Timer.now();
5    f(x, y);
6    Time elapsed = Timer.now() - start;
7  }
8
9  int main() {
10   time_of_f(5, 42.);
11   return 0;
12 }
```

Теперь мы всю работу по измерению времени убрали в отдельный метод. Теперь надо лишь сделать ее generic, то есть сделать так, чтобы не было зависимости от конкретной функции `f`. Передать функцию `f` в метод по измерению времени можно двумя способами: в run-time и в compile-time. Делается это так

```
1  // run-time
2  template<class F, class... Args>
3  void rtime_of(F f, Args&&... args) {
```

```
  Time start = Timer.now();
  f(std::forward<Args>(args)...);
  Time elapsed = Timer.now() - start;
  // print the time
}

// compile-time
template<auto F, class... Args>
void ctime_of(Args&&... args) {
  Time start = Timer.now();
  F(std::forward<Args>(args)...);
  Time elapsed = Timer.now() - start;
  // print the time
}

int main() {
  // run-time
  rtime_of(f, 5, 42.);
  // compile-time
  ctime_of<f>(5, 42.);
  return 0;
}
```

Принципиальная разница между этими подходами в том, что в первом случае мы можем передавать не только функции, но и любой функтор (то есть объект, у которого перегружен оператор `operator()`). Однако, в этом случае лучше сделать perfect forwarding и самого функтора так

```
template<class F, class... Args>
void rtime_of(F&& f, Args&&... args) {
  Time start = Timer.now();
  std::forward<F>(f)(std::forward<Args>(args)...);
  Time elapsed = Timer.now() - start;
  // print the time
}
```

Либо надо считать, что функторы всегда легкие и передавать копию. Второй же способ удобнее тем, что вызов `f(5, 42.);` нужно заменить на `ctime_of<f>(5, 42.);`, что делается просто через автозамену ну и компилятор не будет звать метод по указателю.

Кроме того, если вы собираетесь логировать время куда-то, можно добавить дополнительные аргументы для сообщения или другой информации.

```
template<class F, class... Args>
void time_of(F f, const char* msg, Args&&... args) {
  Time start = Timer.now();
  f(std::forward<Args>(args)...);
  Time elapsed = Timer.now() - start;
  std::cout << msg << ", time = " << elapsed.inSeconds() << '\n';
}

int main() {
  time_of(f, "f(5, 42.)", 5, 42.);
}
```

7.2 Оператор `operator->`

Теперь давайте сделаем все то же самое, только теперь вместо времени работы глобальной функции хочется замерить время работы метода у объекта какого-то класса, то есть хотим автоматизировать процесс:

```
struct A {
  void f(int, double) const;
};

int main() {
  A a;
  Time start = Timer.now();
  a.f(5, 42.);
  Time elapsed = Timer.now() - start;
  return 0;
```

```
11 }
```

Подобные трюки делаются с помощью перегрузки оператора `operator->`. Но для начала давайте поймем как он работает. Когда компилятор видит выражение

```
1 a->f(5, 42.);
```

То он проверяет что стоит слева от оператора стрелки и тут есть следующие варианты:

1. Слева стоит указатель

```
1  struct A {
2    void f(int, double) const;
3  };
4
5  int main() {
6    A a;
7    A* ptr = &a;
8    ptr->f(5, 42.);
9    return 0;
10 }
```

   В этом случае если это указатель на класс, который содержит метод `f`, то строчка 8 эквивалентна вызову

```
1 a.f(5, 42.);
```

   Если же это указатель на класс, где нет метода `f` с нужной сигнатурой или указатель на встроенный тип, то это ошибка компиляции.

```
1  struct A {
2    void f(int) const;
3  };
4
5  int main() {
6    A a;
7    A* ptr = &a;
8    ptr->f(5, 42.); // compilation error
9    int x = 5;
10   int* p = &x;
11   x->f(5, 42.); // compilation error
12   return 0;
13 }
```

2. Слева стоит объект класса, у которого есть перегруженный оператор `operator->`.

```
1  struct A {
2    void f(int, double) const;
3  };
4
5  struct B {
6    A* operator->() const;
7  };
8
9  int main() {
10   B b;
11   b->f(5, 42.);
12 }
```

   В этом случае для `b` вызывается перегруженный оператор `operator->`, а потом к результату применяется рекурсивно вызов оператора стрелка, то есть происходит следующее:

```
1 int main() {
2   (b.operator->())->f(5, 42.);
3 }
```

В данном случае вместо `b.operator->()` подставляется указатель на `A` и потом по первому правилу зовется метод класса `A`. Оператор `operator->` не обязан возвращать указатель, он может вернуть объект другого класса, у которого все еще перегружен оператор стрелка. То есть может быть следующая цепочка:

```
struct A {
  void f(int, double) const;
};

struct B {
  A* operator->() const;
};

struct C {
  B operator->() const;
};

int main() {
  C c;
  c->f(5, 42.);
}
```

В этом случае происходит следующее. В начале для объекта `c` вызывается его перегруженный оператор стрелка, который вернет объект класса `B`. После для него вызовется его перегруженный оператор стрелка и он вернет указатель на `A`. И уже последний вызов позовет по указателю метод `f` с указанными параметрами. То есть этот вызов будет означать

```
int main() {
  ((c.operator->()).operator->())->f(5, 42.);
}
```

Такие вызовы лево ассоциативны по стрелкам и допускают любую глубину рекурсии. Главное, чтобы в конце последний оператор `operator->` возвращал указатель.

3. Если же оператор `operator->` возвращает класс, у которого не перегружен оператор стрелка `operator->`, то это ошибка компиляции.

7.3 Измерение времени выполнения метода класса

Теперь вернемся к нашей задаче по автоматизации измерения времени вызова метода. Пусть у нас есть следующий класс

```
struct A {
  void f(int, double) const;
};
```

В начале, прежде чем замерять время, создадим следующий вспомогательный класс для получения доступа к методам `A`:

```
template<class T>
class Ptr {
public:
  Ptr(T* ptr) : ptr_(ptr) {}
  T* operator->() const {
    return ptr_;
  }
private:
  T* ptr_;
};
```

Теперь мы можем писать следующий код

```
A a;
auto ptr = Ptr(&a);
ptr->f(5, 42.);
//or
Ptr(&a)->f(5, 42);
```

Давайте проанализируем как работает строчка 5. В ней в начале создается временный объект `Ptr(&a)`. После чего у него вызывается оператор `operator->`, который возвращает указатель на объект `a`. После чего у объекта `a` вызывается метод `f(5, 42.)`. И после этого вызова в конце этой строчке вызывается деструктор временного объекта `Ptr(&a)`. Язык гарантирует, что все временные выражения в одной строке живут до конца жизни всего выражения в строке. Это нужно, чтобы указатель `ptr_`, который возвращается перегруженным оператором `operator->`, существовал в момент обращения к нему.

Теперь ясно, что до выполнения функции вызывается конструктор временного объекта, а после выполнения – его деструктор. И есть соблазн добавить код по измерению времени именно в них, то есть написать следующее

```
template<class T>
class Ptr {
public:
  Ptr(T* ptr) : ptr_(ptr), start_(Timer.now()) {}
  ~Ptr() {
    Time elapsed = Timer.now() - start_;
    // print elapsed time
  }
  T* operator->() const {
    return ptr_;
  }
private:
  T* ptr_;
  Time start_;
};
```

Тогда действительно строчка кода

```
Ptr(&a)->f(5, 42.);
```

замерит время вызова функции `f`. Однако, если вы напишете следующее

```
int main() {
  A a;
  auto ptr = Ptr(&a);
  ...
  ptr->f(5, 42.);
  ...
  return 0;
}
```

то теперь вы измерите время от строчки 3 (создание `ptr`) и до выхода из `main`, то есть до строчки 8 (где вызовется деструктор `ptr`). И это будет не то, что вы ожидаете. И ничто не запрещает так использовать этот шаблонный класс обертку. А раз класс можно использовать неправильно и даже не догадываться об этом, то лучше переделать его дизайн. Проблема тут в том, что объект `ptr` может жить дольше, чем время вызова метода. Потому надо вклинить еще один временный прокси-объект, который будет создаваться только в момент обращения к оператору `operator->`. И делается это так, наивную имплементацию выше помещаем в отдельный прокси объект[21]

```
template<class T>
class Proxy {
public:
  Proxy(T* ptr) : ptr_(ptr), start_(Timer.now()) {}
  ~Proxy() {
    Time elapsed = Timer.now() - start_;
    // print time
  }
  T* operator->() const {
    return ptr_;
  }
private:
  T* ptr_;
  Time start_;
};
```

[21] Тут можно сделать конструктор приватным, а класс `Timed<T>` сделать `friend`, чтобы никто не мог просто так создать прокси класс. Так же его стоит спрятать либо внутрь класса `Timed<T>`, либо внутрь служебного `namespace detail`.

После чего делаем наш декоратор

```
template<class T>
class Timed {
public:
  Timed(T* ptr) : ptr_(ptr) {}
  Proxy<T> operator->() const {
    return Proxy(ptr_);
  }
private:
  T* ptr_;
};
```

И теперь мы можем написать

```
A a;
Timed(&a)->f(5, 42.);
// or
auto ptr = Timed(&a);
ptr->f(5, 42);
...
ptr->f(5, 42);
```

Давайте разберемся как работает код выше. В строчке 2 произойдет следующая последовательность действий

1. В начале создается временный объект `Timed(&a)`, который в конструкторе запоминает адрес `a`.
2. Теперь вызывается оператор `operator->` для временного объекта, который возвращает временный объект типа `Proxy<A>`. В его конструкторе запоминается адрес хронящийся в `Timed<A>` объекте (то есть адрес `a`) и текущее время.
3. Теперь вызывается оператор `operator->` для `Proxy<A>`, который просто дергает метод `f(5, 42.)`.
4. Теперь разрушается временный объект `Proxy<A>`, который в деструкторе замеряет текущее время и высчитывает прошедшее.
5. Теперь умирает временный объект `Timed<A>`.

Строчки 5 и 7 работают аналогично за исключением того, что в них не зовутся конструкторы и деструкторы долгоживущего объекта `Timed<A>`. Но при каждом обращении по стрелке у нас создается ровно один вспомогательный прокси объект, который и занимается работой по измерению времени.

Теперь можно обвешивать декоратора функционалом, который нам нужен. Что можно настроить

1. Можно передать для исполнения любую функцию для вызова в конструкторе.
2. Можно передать для исполнения любую функцию для вызова в деструкторе.
3. Можно передать любые дополнительные данные.

Например, можно передать метод куда именно печатать сообщение о времени и строку с сообщением, которую надо добавить к измеренному времени.

7.4 Временные якоря

Раз уж мы затронули вопрос измерения времени, то давайте обсудим еще одну идею, которую можно использовать для измерения времени. Например, если вы хотите измерить время работы какого-то scope.

```
void f(int x, double y) {
  auto t = TimeAnchor();
  ...
} // ~TimeAnchor() counts the time since the constructor was called
```

Если вам надо измерить время с какой-то точки времени до момента выхода из scope и таких точек может быть много (например, может быть несколько `return` выражений в теле функции), то можно написать код как выше, который в стиле RAII (смотри 6.3.15) захватит время при создании и вычислит прошедшее при разрушении якоря. Код для якоря выглядит как-то так

```
class TimeAnchor {
public:
  TimeAnchor() : start_(Timer.now()) {}
  ~TimeAnchor() {
    Time elapsed = Timer.now() - start_;
    // print elapsed time
  }
private:
  Time start_;
};
```

Можно добавить функционал для временного якоря, например добавить метод измерения времени до текущего момента. При этом можно как отключить замер времени при уничтожении, так и настроить его, чтобы он был с текущего момента или с самого начала времени жизни якоря. Можно добавить разные методы куда выводить результат и кучу других модификаций, какие только захотите.

8 Признаки хорошего и плохого кода

8.1 Общие слова

Вы скорее всего встретите много разных советов, что в коде хорошо, а что плохо. И с большой вероятностью все эти правила буду противоречить друг другу. Потому очень важно понимать как к этому относиться. Ситуация с кодом очень похожа на ситуацию со скоростными ограничениями на дороге. Плохая идея гонять по городу 110, в то же время плохая идея ехать по трассе 60. Главная мысль в том, что все сильно зависит от ситуации и обстоятельств. Чтобы хоть как-то пояснить это я хочу начать с примера.

Олимпиадное программирование Вы когда-нибудь смотрели посылки на codeforces? Вы видели какое безобразие люди пишут в коде? Вот один из примеров.

```
1 #define int long long
```

А теперь вопрос: хорошо это или плохо? Раз топовые участники codeforces так пишут, то видимо хорошо. Однако, в то же время топовые программисты со всего мира рассказывают вам, что ни в коем случае так писать нельзя. Возникает вопрос, почему в олимпиадном программировании так писать хорошо, а в промышленном программировании плохо. Дело в том, что задача олимпиадного программиста максимально быстро написать программу, которая продет все тесты системы и все. После этого код никогда не используется, он выкидывается на помойку и про него никто не вспоминает. Его не надо поддерживать, его не надо модифицировать, его вообще не надо даже вспоминать. А потому любые трюки, которые позволяют что-либо сделать быстрее – это хорошо. В частности вместо того, чтобы везде менять int на long long и получить ошибку от того, что вы где-то это забыли сделать, можно просто воспользоваться препроцессором.

С другой стороны в промышленном коде подобные трюки плохи, потому что они мешают поддерживать код. Основная проблема с этим макросом, что он меняет код где угодно после него. И тогда код который видит программист и который видит компилятор – это два разных кода. А это значит, что код начинает врать программисту. Это не локальное изменение, которое тяжело обслуживать.

Математика и алгоритмы Есть хорошее правило, что размер функций должен быть не большим, чтобы можно было понять, что функция делает. Что переменные должны иметь нормальные имена, чтобы было понятно, что они делают. Что не должно быть больше 3 вложенных scope, чтобы не было спагетти кода. Все это хорошо, но вот вы начинаете писать какой-то известный алгоритм. Предположим в этом алгоритме используются обозначения для времени, координаты, скорости, массы и импульса t, x, v, m и p, соответственно. Теперь сравните два кода

```
x = v * t;
p = m * v;
E = 1/2 * m * v * v;
```

```
coordinate = velocity * time;
impulse = mass * velocity;
Energy = 1/2 * mass * velocity * velocity;
```

Обратите внимание на сколько тяжелее читать код справа, хотя мы дали всем переменным говорящие имена. Тут есть две проблемы:

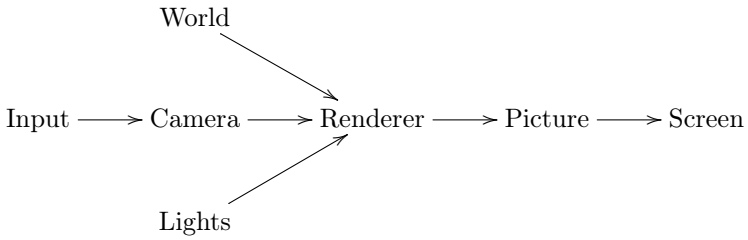
1. Имена справа не привычные, а обозначения слева нам более привычны.
2. Имена справа врут. Дело в том, что переменная `time` не ясно какое именно время обозначает. То есть мы вроде уточнили, что `t` это время, но вот время от какого момента до какого, не уточнили. Или `Energy` справа – это все таки кинетическая энергия, а не просто какая-то энергия. И усложнять эти имена до длинных предложений не имеет смысла.

Аналогичная проблема с именами в математических алгоритмах или в имплементации структур данных. Часто авторы статей используют для обозначений короткие обозначения, которые облегчают понимание алгоритма. Разумно так и записать алгоритм, а перед этим алгоритмом добавить пояснения для всех переменных, которые присутствуют в статье.

Сюда же я хочу добавить замечания по поводу размеров функций и количества вложенных scope. Так часто бывает, что какой-нибудь сложный алгоритм может содержать 4 вложенных цикла. И он не бьется никаким разумным образом на подфункции, в которые можно было бы спрятать внутренние циклы или как-то разбить его на логические части.

Я хочу сказать, что в подобных ситуациях нет смысла пытаться подогнать ваш код под какие-то специфические правила. Вы можете ухудшить читаемость кода и/или его производительность. И причина почему вы можете себе это позволить – вы этот алгоритм пишете один раз. Вы точно знаете, что он так работает корректно и никогда не будете его модифицировать. В худшем случае, вы имплементируете новый алгоритм из другой статьи. А потому лучше для понимания, чтобы код был близок к статье, по которой вы его пишете.

Масштабы проекта Вы можете прочитать кучу разных теорий о том как писать то или иное приложение, о разных подходах к архитектуре или прочие тонкости. Однако, не всегда следует применять все что вы видите, только потому что это было создано для вашей ситуации. Все сильно зависит от масштабов вашего проекта. Если проект маленький, то нет смысла в него внедрять сложные конструкции или парадигмы. Не занимайтесь Over Engeneering. Вот пример, который мне часто приходится видеть на проектах. Ребята имплементируют 3D Renderer с нуля и используя его пишут интерактивное приложение, которое демонстрирует возможности движка. Схема потока данных в интерактивном приложении следующая



Для такой простой схемы, можно даже не городить никакую Message Driven System, не использовать MVC или другие паттерны. Весь код можно засунуть целиком в цикл обработчик событий.

```
class Application {
public:
  Application(); // initializes resources

  void run(); // run event loop
private:
  auto read_user_input();

  World world_;
  Camera camera_;
  Lights lights_;
  Renderer renderer_;
  Screen screen_;
};
```

```
Application::Application() {
  /*Initialization*/
}
void Application::run() {
  ...
  /*somewhere in event loop*/
  auto action = read_user_input();
  camera_.move(action);
  Picture picture = renderer.make(world_,
                                  camera_,
                                  lights_);
  screen_.draw(picture);
  ...
}
```

Заметьте, что нет необходимости делить код на ядро, и другие компоненты. Все хранится непосредственно в классе приложения. И вы просто друг за другом передаете данные от одного объекта другому, как и требуется. В этой простой ситуации нет необходимости использовать межобъектную коммуникацию или какие-то сложные паттерны. Вы даже можете не использовать event loop, вы можете просто крутиться в бесконечном цикле и опрашивать клавиатуру. Нужно только защититься таймером, чтобы цикл не проходил слишком быстро и все.

Недостаток такого подхода – его не возможно масштабировать. Но если вам надо начать масштабировать это решение, то вы можете стартовать с кода выше. У вас уже есть в готовые работающие компоненты и теперь лишь надо распилить `Application` и соединить его компоненты друг с другом в соответствии с MVC. То есть вы делаете эту работу тогда, когда она нужна, а не заранее, когда вы даже не знаете нужно это или нет.

Идеал и реальность Есть хороший вопрос: как понять, когда что-то надо использовать или нет. Есть по сути три простых ситуации

1. Вы знаете что в данной ситуации надо использовать и умеете это имплементировать. Например, вы знаете, что лучше использовать `std::vector` из-за локальности в памяти данных. Или вы умеете имплементировать поиск за $O(\log(n))$ или $O(1)$ с помощью соответствующей структуры данных и вы знаете, что вам это нужно. Ну значит так и делайте. Раз вы знаете, что это тут нужно и умеете, то глупо этого не делать.

2. Вы не знаете, что в данной ситуации надо использовать. Тут все проще, вы даже не поймете, что у вас была такая ситуация. Либо если вы задумаетесь над ней, то это повод что-то почитать и изучить.

3. Вы знаете, что в данной ситуации надо использовать, но не умеете это имплементировать. Вот тут ситуация интереснее. Вы можете поступить двумя способами

   (a) Имплементировать как получится и как вы умеете.

   (b) Изучить, что в этом случае люди делают.

   И тут все зависит от ваших ресурсов.Если вы ограничены во времени и ресурсах, то намного проще и правильнее будет имплементировать как получится. Очень может оказаться, что оно и так сойдет, что не оказывается тут нет никакого узкого горлышка и все и так работает. Вы сэкономите время и силы на то, что не потратите его на ненужную работу. Во втором подходе есть так же проблема в том, что вам без опыта и понимания надо как-то понять какое из предлагаемых решений вам действительно нужно и какое из них действительно хорошо. И часто вы просто придете к тому, что надо было просто имплементировать как есть.

От себя хочу добавить, что работая над проектом анализа качества печати в одиночку, я часто встречался с ситуацией, что есть какая-то второстепенная деталь в проекте, которую надо имплементировать и без нее нельзя продвинуться дальше в содержательных вопросах. И я понятия не имел как это делать. И каждый раз решение – сделать как-то было самым правильным. Оно сохраняло время, силы и желание работать над важной частью проекта. Чем опытнее вы будете становиться, тем больше у вас будет понимания, а как теперь надо писать проект.

8.2 Локальность

Local Reasoning или возможность делать выводы по небольшому куску кода – очень важное свойство. Давайте я поделюсь тут некоторым философским замечанием. Я знаю, что в среде программистов не особенно принято доказывать, что их программа работает корректно. Мы просто пишем код. Потом смотрим, кто на него ругается. Если ругается компилятор – исправляем на что ругается. Если ругаются тесты – исправляем на что ругается. Если ругается начальник – исправляем на что ругается. Все работа выполнена. При таком подходе не возможно ничего гарантировать. И вот тут я бы сравнил написание программ с написанием математических статей. Математические статьи оперируют куда более сложными конструкциями, чем проекты в индустрии. Однако, если вы почитаете хорошие статьи в них нет ошибок. Это не просто привычка писать текст и проверять себя, просто на дисциплине такое держать сложно. Один из важных инструментов – Local Reasoning. Мы разбиваем сложное утверждение на маленькие утверждения, чьи доказательства проверить на корректность легко. А потом собираем все из кусочков так, чтобы ошибке некуда было закрасться. У профессиональных математиков этого достаточно чтобы гарантировать отсутствие ошибок (хотя не на 100% как показывает практика). Однако, даже в этом вопросе программисты пошли другим путем. Есть специальные языки программирования, которые позволяют проверить корректность работы программы. И если она не корректная, то вам на это ругнется компилятор, а если он не ругнулся, то значит проблем нет, это гарантируется. Это очень здорово на самом деле и жаль, что такие инструменты мало распространены и не всегда ими удобно пользоваться. А теперь давайте перейдем все таки от философии к делу.

Вся идея написания большого и сложного программного проекта – разбить его на независимые части поменьше, обозначить четкие границы и правила взаимодействия между ними сквозь эти границы. А дальше мы применяем это правило рекурсивно, бьем полученные компоненты на меньшие части и так далее. Отсюда сразу становится понятно почему важна value semantics. Наличие ссылок и указателей в данных или в объектах друг на друга (если только это не специальный паттерн) – это всегда большая проблема для локальности. Например

```
class A {
public:
  ...
  void f(int);
  void g(int);
private:
  B* b_;
  int data_;
};
```

```
class B {
public:
  ...
  void h(int);
  void w(int);
private:
  A* a_;
  int data_;
};
```

Как мы видим выше у нас два класса и каждый из них хранит указатель на другой. Тут важно понимать, что это не значит, что два объекта обязаны хранить указатели друг на друга. Сырые указатели не спрятанные внутрь шаблона проектирования не дают никаких гарантий. Теперь, представьте, что внутри метода `A::f` вы дергаете метод `B::h` доступный по указателю `b_`. То есть теперь, чтобы понять, что делает класс `A`, вам надо знать, что делает класс `B` и как он коммуницирует с `A`. Но этого мало. Вам надо держать в голове целый граф из всех возможных объектов класса `A` и `B`, потому что они могут ссылаться друг на друга случайным образом. А еще надо не забыть, что указатель может быть нулевым. И чтобы написать код корректно, вам придется обрабатывать все возможные ситуации. Вероятность, что вы это сделаете, почти ноль. А значит ваш код очень быстро начнет падать на всем подряд.

Ссылки для передачи данных В таких ситуациях надо понять зачем вам нужны ссылки и указатели на другие объекты. Очень часто бывает так, что вам надо передать данные из одного объекта в другой. Вот пример кода и как его переделать

```
struct A {
  void do_something() {
    int x = make();
    b->do_something(x);
  }
  int make() const;
  B* b;
};

struct B {
  void do_something(int);
};

int main() {
  A a;
  B b;
  a.b = &b;
  a.do_something();
}
```

```
struct A {
  int make() const;
};

struct B {
  void do_something(int);
};




int main() {
  A a;
  B b;
  int x = a.make();
  b.do_something(x);
}
```

Как вы видите нет необходимости хранить указатель на другой объект, мы просто получаем данные из `A` и отдаем их `B`. Тут есть одна особенность. Что эту передачу данных должен кто-то делать. Если оба объекта лежат в каком-то классе, то за передачу данных отвечает класс. Если оба объекта являются локальными объектами функции (как в примере выше), то за передачу данных отвечает сама функция. Это так называемый мастер объект. Однако, так бывает, что объекты равноправны и не имеют одного мастер объекта, который бы мог передавать данные. В таких ситуациях надо использовать Observer pattern, который будет описан в разделе 12.3. В целом всегда можно организовать любую программу как пайплайн, где объекты поглощают и производят данные, и передача данных между объектами делается посредством Observer pattern-а. Это не всегда нужно, но там где нужно, это позволяет полностью избавиться от каких-либо ссылок и указателей.

Парные операции Еще один пример отсутствия локальности мы видели в разделе 7.1 посвященному измерению времени. Давайте посмотрим на вот такой кусок кода

```
1 void f(int, double);
2
3 int main() {
4   Time start = Timer.now();
5   f(5, 42.);
6   Time elapsed = Timer.now() - start;
7   return 0;
8 }
```

Этот код хорош, что в нем сразу несколько проблем. Но тут же мы видим не локальность. А именно, мы работу по измерению времени делаем не в одном месте, а сразу в двух строчках 4 и 6. И на практике эти строчки могут разъехаться очень далеко. Данную ситуацию можно отнести к случаю, когда у вас есть какие-то парные связанные друг с другом операции, которые должны гарантированно выполняться парами. Раздел 7 про Декораторы как раз был посвящен тому, как стоит организовывать код в такой ситуации.

`shared_ptr` Одна из причин, почему я делаю акцент на `shared_ptr` на неконстантный объект – он нарушает локальность. Два `shared_ptr`-а могут быть сколь угодно далеко друг от друга в коде, но при этом ссылаться на одни и те же данные. Подробнее про это можно почитать в разделе 4 и 10.3.

Глобальные данные Это еще один пример нарушения локальности. Наличие глобального состояния у системы может сломать все что угодно. Любой чих в любой строчке кода может изменить глобальное состояние и все функции могут начать выдавать другой результат из-за этого. Если вы страдали с тем, чтобы выставить правильно все сиды в рандомных генераторах каких-нибудь крупных фреймворков, чтобы получить воспроизводимость результатов, то вы знаете о чем я говорю. Тем не менее, бывает, что без глобальных данных не обойтись. Про примеры и то, как этим пользоваться стоит посмотреть раздел **??**.

Value Semantics Одна из причин почему value semantics важная штука – она улучшает локальность. Потому что по определению данные в двух разных переменных никогда ни при каких условиях не влияют друг на друга, какой бы ни была имплементация. Подробнее про семантику можно глянуть в разделе 5.

8.3 Явные зависимости

Любая программа всегда состоит из двух частей

1. Данные
2. Методы

Это все можно заворачивать в классы и прочие прелести, но у вас всегда есть данные, над которыми вы работаете, и методы, которые эти данные обрабатывают.

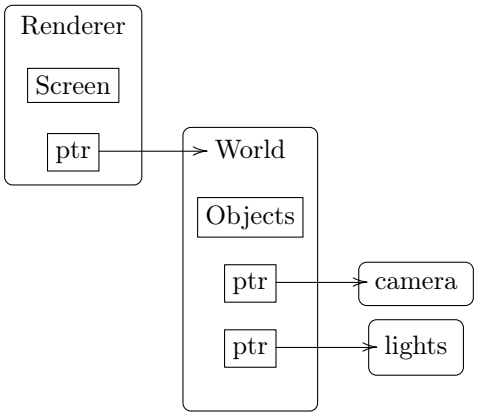
Диаграмма владения Когда вы работаете с данными, то очень важно понять какой объект какие другие объекты в себе содержит. Это позволяет понять зону ответственности объекта и ограничить связанные данные вместе от посторонних. Полезно перед глазами держать диаграмму владения. Это граф, у которого вершины – это объекты и вы проводите ребро из вершины подъобъекта в вершину владеющего объекта. Либо вы можете это визуализировать в виде вложенных друг в друга боксов. Если у вас не получается нарисовать такую диаграмму, значит у вас какие-то проблемы. И часто проблемы связаны с отсутствием value semantics. В данном случае диаграмма описывает явно зависимости между классами, кто кого содержит.

3D Renderer Я вернусь тут к примеру с 3D Renderer-ом с нуля. Пусть у нас есть несколько простых компонент в приложении как ниже. Очень часто вижу приблизительно такой код, где вас должны сильно смутить указатели и отсутствие `Screen`.

```
class Application {
public:
  Application(); // initializes resources

  void run(); // run event loop

private:

  Camera camera_;
  Lights lights_;
  World world_;
  Renderer renderer_;
};
```

```
Application::Application()
  : camera_(initialize_camera()),
    lights_(initialize_lights()),
    world_(initialize_world(),
            &camera,
            &lights_),
    renderer_(600, 800, &world_) {
}
void Application::run() {
  ...
  /*Somewhere in event loop*/
  camera_.move(action)
  renderer_.make();
  ...
}
```

А потом выясняется вот такая схема владения.



В частности выясняется, что GUI код находится внутри `Renderer`. И тут сразу становится понятно зачем надо было в `Renderer` передавать размеры окна. Диаграмма приведенная выше – это на самом деле еще цветочки. У меня были ситуации, когда надо было около 4 часов тщательно изучать код и рисовать сложный граф, содержащий информацию о том, кто на кого содержит какие ссылки, кто реально владеет данными и кто кому как их передает.

Обратите внимание на `run` функцию. Тут я опустил код, читающий пользовательский ввод, но предположим мы считали данные `action`. А дальше происходит какая-то магия в строчке `renderer_.make()`. Потому что метод `make` ничего не принимает, ничего не возвращает, но тем не менее эта строчка делает все. С такой диаграммой владения, когда объекты неявно ссылаются друг на друга, у вас всегда есть возможность начать модифицировать состояние чужого объекта внутри своего метода. А значит вы автоматически начинаете передавать данные неявно. Это порождает кучу проблем с пониманием потока данных.

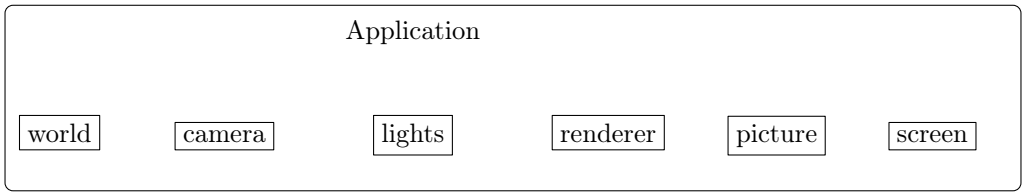
Диаграмма потока данных Другая полезная информация – это поток данных между методами. Просто потому что любое исполнение программы – это выполнение ее методов: построение данных одними методами, передача этих данных в другие методы и так далее. Важно не делать скрытых зависимостей и делать передачу данных явно. Это одна из причин по которой я не люблю всякие ссылки и указатели внутри объектов. Потому что вы могли передать данные в другой объект внутри метода, а пользователь метода снаружи даже не знает о том, что оказывается там какой-то скрытый метод выстрелил. У вас всегда должны быть четкие границы между объектами. Избегайте двустороннего общения между объектами.

Еще раз 3D Renderer Давайте вернемся к примеру 3D Renderer-а из начала раздела. Здесь теперь объекты лежат как надо.

```
class Application {
public:
  Application(); // initializes resources

  void run(); // run event loop
private:
  auto read_user_input();

  World world_;
  Camera camera_;
  Lights lights_;
  Renderer renderer_;
  Picture picture_;
  Screen screen_;
};
```

Диаграмма владения очень простая. Нет никакой необходимости этим объектам знать хоть что-то друг про друга. Всю информацию, которую нужно передать от одного объекта другому сможет передать `Application`

в своих методах.



Но даже в этом случае можно сделать очень плохой поток данных вот такой имплементацией.

```
1  void Application::run() {
2    ...
3    /*somewhere in event loop*/
4    move_camera();
5    make_picture();
6    draw_picture();
7    ...
8  }
9  void Application::move_camera() {
10   auto action = read_user_input();
11   camera_.move(action);
12 }
13
14 vpod Application::make_picture() {
15   picture_ = renderer_.make(world_, camera_, lights_);
16 }
17
18 void Application::draw_picture() {
19   screen_.draw(picture);
20 }
```

Ну а действительно, зачем нам явно подставлять все поля класса в функции, они же видны из любого метода класса. И у вас в классе появляются методы, которые ничего не принимают и ничего не возвращают, но они делают всю работу. И вот как понять по коду выше в строках 4-6, откуда и куда текут данные? Так же обратите внимание, что теперь название метода `move_camera` просто врет программисту. Дело в том, что этот метод не только двигает камеру, но и читает пользовательский ввод. Кроме того, теперь мы еще и новую переменную положили в `Application`, которая вообще то имеет смысл только локально для передачи данных от `renderer` к `screen` и не отражает состояния приложения. Это может быть желательно скажете вы, чтобы не переаллоцировать буффер под картинку. Но это все можно исправить следующим образом.

В начале давайте избавимся от неявных методов только модифицирующих состояние.

```
1  void Application::run() {
2    ...
3    /*somewhere in event loop*/
4    auto action = read_user_input();
5    camera_.move(action);
6    picture_ = renderer.make(world_, camera_, lights_);
7    screen_.draw(picture);
8    ...
9  }
```

А теперь, чтобы переиспользовать память внутри `picture_` заведем ее локально внутри `run`, но будем мувать ее внутрь `make` метода объекта `renderer`, чтобы уже он решал может ли он и хочет ли он переиспользовать буффер. Код будет выглядеть как-то так.

```
1  void Application::run() {
2    Picture picture;
3    ...
4    /*somewhere in event loop*/
5    auto action = read_user_input();
6    camera_.move(action);
7    picture = renderer.make(world_, camera_, lights_, std::move(picture));
8    screen_.draw(picture);
9    ...
10 }
```

И у нас теперь нет необходимости хранить внутри `Application` данные, которые не релевантны для состояния приложения и которые возникают только внутри конкретного метода, и при этом избежать неявного потока данных и переаллокации. Метод `make` уже проверит можно ли переиспользовать буфер внутри `picture` или нет.

Аргументы ссылки Еще один прекрасный источник неявных зависимостей – принимать аргументы методов по не константной ссылке. Посмотрите на следующий код.

```
int x = 1;
double y = 2.;
char z = 'c';
short r = function(x, y, z);
```

Вот какое значение вы ожидаете у аргументов после такого вызова? А функция еще такая умная, возвращает значение, наверное они просто по трем значениям стоит четвертое, ведь так? А вот ее сигнатура

```
short function(int, double&, char);
```

И у вас нет ни шанса узнать на стороне пользователя, что второй аргумент может меняться. Для того, чтобы подсказать на стороне пользователя, что аргумент может меняться, обычно принято передавать аргумент по указателю. В этом случае картина выглядит так

```
short function(int, double&, char);

int x = 1;
double y = 2.;
char z = 'c';
short r = function(x, &y, z);
```

На самом деле такой же походи использует функция `scanf`, которая пишет в аргумент ввод пользователя. Она принимает аргументы по указателю. Это подсказывает пользователю функции, что аргумент может измениться. В рамках такого подхода принято разделять аргументы функции на входные и выходные.

1. Из входных параметров функция читает данные.
2. В выходные параметры функция пишет данные. Бывает еще что некоторые выходные параметры имеют двойное назначение из них данные и читаются и в них пишутся.

В этом случае принято придерживаться следующих правил:

1. Все входные параметры идут в начале, а все выходные параметры в конце.
2. Все входные параметры принимаются либо по значению либо по константной ссылке.
3. Все выходные параметры принимаются по указателю.

В соответствии с этими соглашениями предыдущий пример выглядел бы так.

```
short function(int, char, double&);

int x = 1;
double y = 2.;
char z = 'c';
short r = function(x, z, &y);
```

В этом случае возвращаемое значение может содержать код ошибки, если в этом есть необходимость.

Инкапсюляция Формально инкапсюляция – это противник явных зависимостей. Потому что вы скрываете детали имплементации и не знаете, что внутри происходит. Тем не менее, это не рассматривается как что-то плохое. Дело в том, что вы обычно имплементируете класс с требуемым поведением, а потому наличие каких-то зависимостей внутри класса для вас не важно. Если вы поставили границу так, что класс теперь просто черный ящик с нужным поведением, то у вас все зависимые узлы спрятаны внутри ящика. А если вы занимаетесь имплементацией ящика, то там конечно не стоит делать неявные зависимости между деталями через ссылки и указателями. Обращайтесь с данными явно.

Observer pattern Есть различные мысли о том почему Observer pattern плох. Один из аргументов в том, что он ломает явные зависимости: мы отправляем сообщение и не знаем, что там выполнится. Но я бы не сказал, что это прям беда, это скорее фича этого паттерна. Но вот бездумное использование указателей и ссылок и вызов через них методов – это уже проблема для явных зависимостей.

8.4 Уровни абстракции

При написании любого кода у вас есть мелкая рутинная работа, может быть прям на уровне машинного кода или вызова интринзиков или ассемблерные вставки и т.д. А есть код очень высокого уровня, где вы оперируете сложными сущностями вроде `Application`, `Scheduler`, `Renderer` и тому подобное. Бывает, что код функций выполняет напрямую арифметические операции, а бывает, что вы вызываете другие функции. То есть вы всегда находитесь на некоторой абстрактной линейной шкале между машиной и идейной моделью. И глядя на разный код, можно смело сказать, что один является кодом более высокого уровня, чем другой. Вот примеры

```
for (size_t i = 0; i < data.size(); ++i) {
  data[i] = function(data[i]);
}
```

```
apply_algorithm(function, &data);
```

Обратите внимание, что код слева написан в виде сырого цикла, где мы руками бежим по элементам, руками применяем функцию `function` к каждому элементу, и руками складываем результат в ту же ячейку вектора. Справа же код более высокого уровня, потому что мы просто вызываем функцию, которой делегируем решение как именно исполнять алгоритм над данными.

Что плохо – смешение уровней абстракции. Любой блок кода всегда должен быть написан на одном уровне абстракции, без смешения этих уровней.

Простая функция `main` Вот пример кода с нарушением уровня абстракции.

```
1 int main() {
2   auto data = read();
3   vector<int> result (data.size());
4   for (size_t i = 0; i < data.size(); ++i)
5     result[i] = f(data[i]);
6   print(result);
7   return 0;
8 }
```

Мы видим, что строки 2 и 6 – это делегирование высокоуровневой работы другим функциям. Но в то же время в строках с 3 по 5 идет низкоуровневая работа по построению результата. Для изменения уровня абстракции, проще поднять уровень абстракции у секции с 3 по 5 строки выделением их в функцию:

```
1 int main() {
2   auto data = read();
3   vector<int> result = apply_algorithm(data);
4   print(result);
5   return 0;
6 }
```

Обратите внимание как теперь выглядит функция `main`. Теперь все строки кода на одном уровне: мы в начале читаем данные, потом обрабатываем данные и получаем результат, потом печатаем результат. Нужны детали – смотрим в имплементацию каждой функции.

Измерение времени Давайте еще раз посмотрим на пример из раздела 7.1.

```
1 void f(int, double);
2
3 int main() {
4   Time start = Timer.now();
5   f(5, 42.);
6   Time elapsed = Timer.now() - start;
7   return 0;
8 }
```

В этом случае строчка 5 – это делегирование какой-то вычислительной задачи другой функции. Это высокоуровневая работа. Измерение времени происходит в строках 4 и 6. Причем мы руками создаем две переменные, руками находим разность между началом и концом. Это явно более низкоуровневая работа. Потому в этом примере происходит нарушение абстракции. Решить вопрос можно с помощью декораторов из раздела 7. Выглядеть это будет как-то так:

```
1 void f(int, double);
2
3 int main() {
4   measure_time::call<f>(5, 42.);
5   return 0;
6 }
```

Захват ресурса Очень частая проблема – захват ресурса. Например, если вы хотите защитить данные для много поточки мьютексом вы можете написать что-то такое

```
1 void function() {
2   Data d = read_data();
3   Header h = compose_header();
4   std::scoped_lock(mutex); // too low level
5   write_to_file(h, d);
6 }
```

Если посмотреть на функцию выше, то мы видим, что строки 2, 3 и 5 – это делегирование работы другим функциям. Это код высокого уровня. Однако, в строке 4 идет супер низкоуровневая операция – захват мьютекса. Это явно нарушение уровня абстракции. В бизнеслогике вообще не должны фигурировать всякие низкоуровневые детали имплементации. Вам все равно что там надо залочить на уровне железа, на уровне функции `function` все что нас интересует – мы хотим безопасно записать данные в файл. Потому правильно будет либо создать отдельную функцию `write_to_file_safely` либо воспользоваться декораторами как написано в разделе 7. Вот как будет выглядеть исправленное решение:

```
1 void function() {
2   Data d = read_data();
3   Header h = compose_header();
4   write_to_file_safely(h, d);
5 }
```

Вопрос выбора имен – это разумеется отдельная проблема, но зато теперь весь код написан на одном уровне абстракции.

8.5 Уровни доступа

Давайте посмотрим на следующий пример

```
1 A a;
2 int x = a.b.f();
3 int y = a.b.c.g();
4 a.d.h(x, y);
```

Обратите внимание, что мы создали объект класса `A` и хотим с ним работать. Мы хотим получить из него две целочисленные переменные `x` и `y` и потом их передать опять объекту типа `A`. Однако, мы вместо использования методов класса `A` стучимся в подъобъекты. Это означает, что класс `A` не содержит нужных методов для работы с ним, отчего приходится стучаться в его кишки. Вместо этого надо в классе `A` пробросить соответствующие методы

```
1  class A {
2  public:
3    int f() const {
4      return b.f();
5    }
6    int g() const {
7      return b.c.g(); // BAD
8    }
9    void h(int x, int y) {
10     d.h(x, y);
```

```
  }
private:
  B b;
  D d;
};
```

Вот так уже лучше, но строчка 7 – это тоже нарушение уровня доступа. Теперь уже объект класса `B` не предоставляет нужные методы для работы с ним, а потому надо метод `g` пробросить в интерфейс класса `B`, чтоб написать так

```
int g() const {
  return b.g();
}
```

После чего использование класса `A` становится простым и понятным

```
A a;
int x = a.f();
int y = a.g();
a.h(x, y);
```

Обратите внимание, что есть паттерн Builder (разобранный в разделе **??**), который наоборот использует вызовы вида

```
Builder b;
b.input(3).addLinear(5).add(Sigmoid).addLinear(2).add(ReLu);
Network n = b.extract();
```

Такой подход вызова команд друг за другом называется Chaining. Его использование нисколько не противоречит правилу про уровень доступа, потому что вы просто хотите на одном объекте вызвать много методов друг за другом. Это просто другой стиль использования плюсового синтаксиса.

8.6 Не сваливать все в кучу

Это очень общее замечание, которое по сути говорит, что не надо плодить супер здоровые сущности, а именно:

1. Не пишите очень большие файлы. Если получается слишком много, значит это скорее всего можно декомпозировать на более компактные компоненты.

2. Не пишите очень длинные функции. Если вы пишите слишком большую функцию, скорее всего она состоит из кучи разной работы, которую можно делегировать. Название функции должно в точности отражать ту работу, которую она делает.

3. Не пишите слишком большие классы, которые делают все на свете (см. пример про `std::string` в следующем разделе). Если у вас класс начинает делать слишком много, то скорее всего он начинает делать лишнюю работу. А значит всю эту работу надо кому-то делегировать.

Все эти соображения растут из того, что мы хорошо понимаем короткие куски кода состоящие из простых операций. Это позволяет нам убедиться, что все работает корректно и как мы ожидаем. И важно, что под «простая операция» не имеется в виду простая операция в терминах языка. Это может быть сколь угодно сложный кусок кода, который снаружи выглядит как одна строчка или один простой вызов. Если про каждую операцию вы знаете ее ожидаемое поведение и операций не много, вы можете проверить, что короткий кусок кода работает корректно. Не возможно поддерживать спагетти-код или супер больших монстров, в которых все связано со всем.

8.7 Не размазывайте код

При желании не сваливать все в кучу, есть противоположная сторона – делить все до тех пор, пока это можно сделать. Но это тоже плохая вещь, потому что тогда, синтаксические единицы кода не могут вмещать содержательную работу. Я бы сформулировал следующие три правила.

1. Не пишите слишком много мелких файлов.

2. Не пишите слишком много мелких функций.

3. Не пишите слишком много мелких классов, которые почти ничего не делают.

Это другой пример раздутия кода, когда все на столько микроскопически мелкое, что содержательная работа не помещается целиком в один файл/функцию/класс. Это размазывает содержательную работу по разным компонентам кода. Важно чтобы код соответствовал логике, которая у вас в голове. Это другая версия спагетти-кода.

8.8 Не своя работа

Тут я хочу обсудить пару примеров.

Чтение из файла Пусть у вас есть класс `Picture`, который некоторым образом работает с изображением. Теперь вы хотите добавить возможность читать картинки из файла и писать их в файл. Часто вижу, как люди добавляют конструкторы и функции записи так:

```
class Picture {
public:
  Picture(std::filesystem::path file);
  void write(std::filesystem::path file);
  ...
  /*public interface*/
private:
  /*Implementation*/
};
```

Проблемы с таким подходом заключаются в том, что внезапно класс `Picture`, задача которого обрабатывать, скажем, растровое изображение, начинает разбираться в файловых системах, форматах файлов, парсинге и прочих странных вещах, которые не имеют отношения к его сути. Особенно эту проблему тяжело заметить, если конструктор и метод записи короткие функции, которые легко имплементировать. Но так бывает не всегда. Если вы начинаете добавлять новые форматы, то у вас может получиться такая вот фигня

```
class Picture {
public:
  Picture(std::filesystem::path file) {
    switch(format_of(file)) {
      case BMP:
        read_bmp(file);
        break;
      case JPG:
        read_jpg(file);
        break;
        ...
      default:
    }
  }
  void write(std::filesystem::path file);
  ...
  /*public interface*/
private:
  void read_bmp(std::filesystem::path file);
  void read_jpg(std::filesystem::path file);
  ...
  /*Implementation*/
};
```

И вот тут становится видно, что какого-то черта ваш класс содержит вспомогательные функции, которые не относятся к его сути. Другой признак, что что-то пошло не так – список `#include` директив. Вы увидите, что файл с классом `Picture` почему-то начинает включать в себя совсем не относящиеся к нему заголовочные файлы посвященные файловой системе.

Мы уже обсуждали в разделе 6.3.10, как выносить из класса эту работу. Но я кратко напомню, что должен быть отдельный класс отвечающий за чтение и запись (может быть два класса: один для чтения, другой для записи). А для удобства, его вызов можно спрятать в статический метод или глобальную функцию так.

```
// some other header
class FileWriter {/*Implementation*/};
class FileReader {/*Implementation*/};
```

```cpp

//Picture.h
class Picture {
public:
  ...
  static Picture fromBMP(std::filesystem::path file);
  static Picture fromJPG(std::filesystem::path file);
private:
  /*Implementation*/
};

//Picture.cpp
// includes for FileWriter and FileReader
Picture fromBMP(std::filesystem::path file) {
  FileReader r(file);
  return r.readBMP<Picture>();
}

Picture fromJPG(std::filesystem::path file) {
  FileReader r(file);
  return r.readJPG<Picture>();
}
```

Так же приимплементации `FileReader` если вам понадобится сложная задача парсинга, то выделите для этого отдельный класс. Или даже много классов: по одному под каждый формат файла. Не делайте всю работу в одном месте и сразу, особенно если она вообще не относится к сути класса.

`std::string` Стандартные строки – это пример плохого класса. Давайте посмотрим на список методов этого класса, начиная с модифицирующих методов.

1. `clear`
2. `insert`
3. `insert_range`
4. `erase`
5. `push_back`
6. `pop_back`
7. `append`
8. `append_range`
9. `operator+=`
10. `replace`
11. `replace_with_range`
12. `copy`
13. `resize`
14. `resize_and_overwrite`
15. `swap`
16. `find`
17. `rfind`
18. `find_first_of`
19. `find_first_not_of`
20. `find_last_of`
21. `find_last_not_of`
22. `compare`
23. `starts_with`
24. `ends_with`
25. `contains`
26. `substr`

В особенности мне нравятся последние несколько функций с `find` в названии. Эти алгоритмы можно вызвать с помощью стандартной библиотеки на любом контейнере. Зачем они находятся внутри класса строки – загадка для всех.

8.9 Плохо продуманные интерфейсы

Важно, чтобы объекты и методы, которые вы пишете, могли удобно друг с другом коммуницировать. В частности если один метод возвращает данные в каком-то формате, то удобно и другому методы принимать их в таком формате. Если это не возможно, то значит должен быть промежуточный метод, который делает трансформацию данных. Или же если один класс пользуется другим для своих целей, то хорошо бы, чтобы в интерфейсе этого другого класса был весь тот функционал, который необходим для использования. Либо нужный функционал должен быть написан поверх предоставляемого.

Передача данных Предположим у вас есть две функции

```cpp
struct Vector2D {
  double x;
```

```
  double y;
};

Vector2D produce();
void consume(double x, double y);
```

Обе функции работают с двумерными векторами, но одна функция возвращает вектор как структуру, а другая принимает координаты вектора по-отдельности. Что придется делать, чтобы передавать данные из одной функции в другую? Конвертировать

```
Vector2D vec = produce();
consume(vec.x, vec.y);
```

Либо симметричная ситуация

```
double produce_x();
double produce_y();
void consume(Vector2D);
```

Тогда придется писать

```
consume(Vector2D{produce_x(), produce_y()});
```

Проблема в этом случае в том, что один метод выдает данные не в том формате, в котором их ожидает другой метод. Если вы разрабатывали эти интерфейсы, то вы просто плохо их продумали. Вы знали, какие у вас есть функции и какие данные надо передавать между ними. И раз вы их передаете в несовместимом формате, то это вы сами себе усложнили задачу. В идеале должно быть

```
Vector2D produce();
void consume(Vector2D);

Vector2D vec = produce();
consume(vec);
```

Если же интерфейсы этих функций вы поменять не можете, то вы на самом деле попали в ситуацию проблемы, которая будет адресована в разделе 8.10. Попросту говоря, вам надо создать новую функцию, с нужным интерфейсом, которая будет внутри себя делать конвертацию и потом вызывать уже функцию с неудобным интерфейсом. Такая функция называется адаптером. Про адаптеры и фасады подробнее написано в разделе **??**.

Интерфейс класса Мой любимый пример – пример из STL, а именно имплементация `std::vector`. А точнее метод `size`. Кто писал такой код

```
std::vector<int> vec = { ... };
for (int i = 0; i < vec.size(); ++i)
  vec[i] = ...;
```

тот знает мою боль. Дело в том, что в цикле `for` происходит сравнение знакового `i` и без знакового `vec.size()`. Так лучше не делать, можно легко добиться UB смешивая знаковые и без знаковые типы. Поэтому в C++20 даже добавили `ssize`, которая возвращает знаковое значение для размера вектора. Я считаю это успех.

8.10 Люди носят не достаточное количество шляп

Я думаю, что стоит начать с пояснения названия раздела. Это название я взял из доклада Tony Van Eerd «SOLID, Revisited». Его можно было сформулировать так: «люди используют недостаточное количество функций». Это значит, что какую-то работу, которую можно было погрузить в отдельную функцию, делают на месте руками, что усложняет код и работу с ним. Я бы не стал ограничиваться в этом тезисе только функциями и сформулировал бы это более обще так. Мы в коде часто делаем следующие ошибки:

1. Не вводим переменную для хранения некоторого промежуточного значения. Это очень частая проблема у новичков в программировании. И если вы программируете уже давно, то скорее всего с этой проблемой даже не сталкиваетесь. Вот пример, который приходит из воспоминаний о том, как мои дети учились в школе программированию. Им надо было считать значения ребер прямоугольника и распечатать его площадь и периметр. Вот как это пишут дети

```
double x;
double y;
std::cin >> x;
std::cin >> y;
std::cout << "P = " << 2 * x + 2 * y << '\n';
std::cout << "S = " << x * y << '\n';
```

В данном случае в последних двух строчках мы логически делаем две вещи: и содержательную работу по вычислению значения и работу по отображению этой работы пользователю. В мелких проектах это не страшно, то куда разумнее кажется сделать так:

```
double x;
double y;
std::cin >> x;
std::cin >> y;
double P = 2 * x + 2 * y;
double S = x * y;
std::cout << "P = " << P << '\n';
std::cout << "S = " << S << '\n';
```

2. Мы не вводим функции для выполнения вспомогательных работ. Вот пример который мне приходит в голову. В этом куске кода надо перемешать столбцы матрицы и для этого нужно предоставить вектор с перемешанными индексами.

```
Matrix m = ... ;

std::vector<int> indices;
indices.reserve(m.size());
for (int i = 0; i < m.size(); ++i)
  incides.push_back(i);

std::mt19937 generator = 42;
std::shuffle(indices.begin(), indices.end(), generator);

m.shuffle(incides);
```

В этом коде много чего плохо. Мы в строках 3-6 мы видим нарушение уровня абстракции. В этой части все что мы хотим сделать – создать вектор индексов от 0 до $n-1$. Это должно делаться в одну строчку вызовом одной функции, которую надо написать так/

```
std::vector<int> make_indices(int n) {
  std::vector<int> indices;
  indices.reserve(n);
  for (int i = 0; i < n; ++i)
    incides.push_back(i);
  return indices;
}
```

Теперь код превращается в

```
Matrix m = ... ;

std::vector<int> indices  = make_indices(m.size());

std::mt19937 generator = 42;
std::shuffle(indices.begin(), indices.end(), generator);

m.shuffle(incides);
```

Кроме того, благодаря NRVO объект `indices` будет создан прямо по адресу возвращаемого объекта. А потому это вам ничего не стоит при исполнении. Теперь в строках 5-6 мы хотим просто перемешать элементы этого массива. Мы не хотим тут задавать генератор, не хотим специфицировать как именно мы хотим это делать. Это все мы хотим инкапсюлировать тоже. Однако, тут у нас появляется состояние в виде инициализрующего сида. Тут важно сказать, что никогда НЕ инициализируйте случайным сидом, потому что это дает вам невоспроизводимое окружение. Если вы поймали баг, то вы его потом не

сможете воспроизвести. Передавать сид не имеет смысла, так как это будет по сути делать каждый вызов функции детерминированным, и вся случайность будет перенесена в выбор сида. Потому я просто инициализирую глобальную переменную спрятанную внутри функции, то есть статическую переменную в функции. По-хорошему, надо спрятать все это в класс `Random`, который умеет помнить свое состояние. Но тут я пойду простым путем.

```
std::vector<int> shuffle(std::vector<int>&& data) {
  static std::mt19937 generator = 42;
  std::shuffle(data.begin(), data.end(), generator);
  return data;
}
```

После чего код превращается в

```
Matrix m = ... ;

std::vector<int> indices  = make_indices(m.size());
indices = shuffle(std::move(indices));

m.shuffle(incides);
```

Ну и наконец-то в этом месте все что мы хотим сделать – перемешать столбцы матрицы. И нам не то чтобы важен массив индексов `indices`, потому что это лишь вспомогательные данные, которые нам приходится строить из-за того, что библиотека для работы с матрицами предоставляет такой интерфейс. Давайте просто сделаем нужную функцию, которая будет делать то, что нам надо.

```
Matrix shuffle(Matrix&& matrix) {
  std::vector<int> indices  = make_indices(matrix.size());
  indices = shuffle(std::move(indices));
  matrix.shuffle(incides);
  return matrix;
}
```

И теперь наш код превращается в

```
Matrix m = ... ;
m = shuffle(std::move(m));
```

В конце хочу добавить небольшое замечание. Можно было бы делать перемешивающие методы с помощью out параметров.

```
void shuffle(Matrix* matrix) {
  std::vector<int> indices  = make_indices(matrix->size());
  indices = shuffle(std::move(indices));
  matrix->shuffle(incides);
}
```

И звать код вида

```
Matrix m = ... ;
shuffle(&m);
```

Это тоже вариант и до появления move семантики это был единственный эффективный способ не копировать данные. Сейчас я предпочитаю использовать `std::move`, чтобы был явно виден поток данных. Но это не принципиально.

3. Мы не вводим класс для выполнения вспомогательной работы. Один из лучших примеров, который я знаю – это имплементация `std::vector`. Большинство пытаются имплементировать `std::vector` напрямую следующим образом/

```
template<class T>
class vector {
public:
  ...
private:
  T* data_ = nullptr;
  size_t size_ = 0;
};
```

Что не очевидно, так это то, что в этом случае вектор будет делать менеджмент сразу двух ресурсов:

(a) Сырая память, которую надо выделять и удалять на куче.

(b) Объекты, размещенные в сырой памяти.

Так вот правильно ввести отдельный класс `buffer`, который будет заниматься менеджментом сырой памяти, а сам `vector` будет только отвечать за размещение объектов в памяти.

```
template<class T>
class buffer {
public:
  ...
private:
  T* data_ = nullptr;
  size_t size_ = 0;
};

template<class T>
class vector {
public:
  ...
private:
  buffer<T> buffer_;
};
```

Почему это важно. Когда вы будете создавать вектор из n объектов, то у вас есть две точки бросания исключений

(a) Выделение сырой памяти.

(b) Размещение объекта в сырой памяти посредством placement `new` оператора.

Так вот, если брошено исключение на втором этапе, то вам надо во-первых, удалить все уже построенные объекты, а потом еще и почистить сырую память. Попытки сделать это одним классом порождают жуткую мешанину из `try`/`catch` блоков в коде. Это не значит, что вы не сможете написать так вектор. Но будет сложнее написать его верно, еще сложнее проверить, что оно верное и еще на много более сложная задача – убедить окружающих в этом.

8.11 Комментарии

Комментарии в коде – это почти всегда плохо. Чаще всего если вам нужны комментарии – ваш код не достаточно выразительный и надо не писать комментарии, а переписывать код. Проблема комментариев в том, что их не читают программисты и их не понимают компиляторы. А еще комментарии могут не обновляться вовремя и тогда они начинают врать.

Сложные условия Одна из самых частых ситуаций – сложные условия внутри `if` блока. В этом случае хочется помочь пользователю и добавить комментарий, поясняющий, что же мы проверяем. Например

```
Point p = ... ;
Box b = ... ;
...
// check if p is in the box
if (p.x >= b.leftdown.x && p.x <= b.rightup.y && p.y >= b.leftdown.y && p.y <= b.rightup.y) {
  ...
}
```

В этом случае вместо комментария нужно создать нормальную функцию, чье имя будет говорить, что вы проверяете

```
bool is_in_box(Point p, Box b) {
  return p.x >= b.leftdown.x && p.x <= b.rightup.y && p.y >= b.leftdown.y && p.y <= b.rightup.y;
}
// Then the previous code:
Point p = ... ;
Box b = ... ;
...
```

```
if (is_in_box(p, b)) {
  ...
}
```

И комментарий больше не нужен.

Разделение функции на блоки Другая ситуация, когда у вас длинная функция имеет несколько различных логических блоков и вы хотите отделить один от другого с пояснениями зачем этот блок нужен.

```
F algorith(const A& a, const B& b, const C&c) {
  // Convert data to required format
  D d;
  for (const auto& v : a.data()) {
    d.add(v);
  }

  //Merge b and c together
  E e;
  for (const auto& x : b.data()) {
    for (const auto& y : c.data()) {
      e.add(merge(x, y));
    }
  }

  // Merge e with d;
  F f;
  for (const auto& x : e.data()) {
    for (const auto& y : d.data()) {
      f.add(merge(x, y));
    }
  }
  return f;
}
```

Содержание кода выше не имеет значение. Важно, что наша функция поделена внутри на три логических блока. И мы их даже честно прокомментировали. Но намного лучше будет выделить блоки с комментариями в функции.

```
F algorith(const A& a, const B& b, const C&c) {
  D d = convert(a);
  E e = merge(b, c);
  F f = merge(e, d);
  return f;
}
```

И уже внутри этих функций вы имплементируете каждую стадию. Теперь это разделение на части видит не только программист, то и компилятор. И не надо словами объяснять, что делает код.

Общий принцип, когда не нужны комментарии Примеры выше призваны показать важный принцип – никогда НЕ поясняйте словами, что делает код. Потому что по коду и так ясно что он делает. Не в том смысле, что это обязательно очевидно, а в том смысле, что компилятор только по коду без ваших комментариев сделает машинный код, который делает конкретную работу. А потому если ваш код не очевиден, то надо не комментировать, а переписывать.

Примеры, когда комментарии нужны Есть несколько ситуаций, когда комментарии действительно в коде нужны и важны.

1. Пояснение решений, которые были сделаны, когда решалось как будет писаться код. Вот конкретный пример. Предположим у вас в проекте есть требования по написанию кода, но при тестировании критической для производительности функции вы выяснили, что написанный по этим правилам код не укладывается в требования по производительности. Тогда вы только внутренность нужной функции пишете так, чтобы она укладывалась в требования. Добавляете хаки, трюки и прочие грязные приемы. Теперь, когда ваш коллега придет работать над проектом, увидев именно эту функцию, он сильно удивится. И скорее всего даже захочет переписать данную функцию в соответствии с требованиями проекта. В таких случаях надо снабжать данную функцию комментариям перед ее объявлением, где

вы поясняете причины, почему функция написана так. Что без костылей вы не могли достичь нужной производительности.

2. Когда вы имплементируете алгоритм из конкретной статьи и используете обозначения этой статьи, то очень полезно перед объявлением функции добавить комментарий поясняющий весь контекст. Что вообще дано, какие данные, что они значат. Что делает функция, какие данные для чего используются и прочие важные детали. Так же полезно добавить ссылку на статью.

3. Если вы пишите какой-то библиотечный код, который будет часто использоваться в проекте. То имеет смысл написать инструкцию с примерами, как этим кодом пользоваться, чтобы пользователи понимали, как вы предполагали использовать данный библиотечный код и не ломали голову, почему что-то не компилируется. Пояснение всяких тонких деталей.

8.12 Не обеспечена семантика класса

Это более или менее техническая деталь. Семантику мы подробно обсуждали в разделе 5. Семантика любого класса обеспечивается обычно генерируемыми автоматически методами.

```
1  class A {
2  public:
3    A();
4    A(const A&);
5    A(A&&) noexcept;
6    A& operator=(const A&);
7    A& operator=(A&&) noexcept;
8    ~A();
9  };
```

Если все поля класса имеют value semantics, то ничего кроме конструкторов для построения объекта вам не нужно. Однако, если объект внутри собран скажем из указателей (красно-черное дерево для примера), то снаружи вам нужно обеспечить ему корректное поведение при операциям перемещения и копирования. Это надо не забывать. Потому что, когда вы имплементируете методы объекта, вы обычно думаете, что он прибит гвоздями к одному месту в памяти. Однако, это не обязательно так и нужно поддержать нужное поведение. Это все подробно обсуждалось в разделе **??**.

9 Встроенные типы

9.1 bool

Данный тип используется только для вычисления истинности логических выражений и ни для чего больше. Вот несколько важных пунктов, которые проясняют смысл этого общего заявления

1. Если вам надо проверить какое-то сложное условие и вы хотите завернуть его в функцию, то эта функция должна возвращать `bool`. Например

```
bool is_prime(int);
bool has_element(key);
```

   Если функция возвращает `bool`, то ее название обязательно содержит глагол is или has.[22] Если вы не можете сформулировать название функции в таком виде, то скорее всего вы не должны возвращать `bool`. Вот хороший пример. Есть тесты целых чисел на простоту, которые либо определяют, что число было составным, либо они не знают ответ, например тест Ферма. Есть желание вернуть `bool` из функции, которая проводит тест Ферма:

```
bool run_fermat_test(int);
```

   Проблема с этим решением видна сразу по названию. Из названия функции не ясно, что значит `true` или `false`. Мы можем переформулировать название как-нибудь в духе «проходит ли тест Ферма», но реальное решение заключается в том, что вместо `bool`, мы хотим вернуть статус проверки и статус может быть либо `Prime` либо `ProbablyPrime`. Потому тут лучше завести `enum class` с нормальными именами статуса и возвращать его

```
enum class Status : unsigned char {
  Prime, ProbablyPrime
};

Status run_fermat_test(int);
```

   И теперь вы не можете просто подставить статус внутрь `if`, вам придется тестировать результат с использованием говорящих имен

```
Status test_status = run_fermat_test(107);

if (test_status == Status::Prime)
  // do something
```

2. Никогда НЕ принимайте аргументы функции в виде `bool`. Если вам кажется, что это хорошая идея, то давайте посмотрим на несколько примеров

```
void set_visibility(bool is_visible);
```

   И на стороне пользователя функции

```
set_visibility(true);
```

   Последний вызов даже не так плох, тут можно даже догадаться, что означает значение `true`. Тем не менее, чтобы уточнить, вам все же придется глянуть на заголовок функции. Но что если в h файле функция написана так

```
void set_visibility(bool);
```

   Уже хуже. А cpp может быть недоступен, да и не дело это лезть читать имплементацию, чтобы пользоваться функцией. Но бывают ситуации еще лучше. Вот скажите, что тут значит `true`:[23]

```
Renderer renderer(600, 800, true);
```

[22] Я в курсе про STL `empty` и прочие методы, это печаль.

[23] Первые две магические константы тоже не сильно понятны из вызова кода, но наверное это хотя бы разрешение. Это тоже не лучшее решение. Как с этим бороться рассказано в разделе 6.3.6 пункт 3.

Или вот тут

```
WidgetList list(true);
```

В таких ситуациях надо завести `enum class` с нормальными названиями. Вот как могла разрешиться ситуация выше

```
Renderer renderer(600, 800, Mode::WireFrame);
WidgetList list(Ownership::Enable);
```

Кроме того, если у вас используется `bool`, то у вас только два возможных значения, а их может быть потенциально очень много.

3. Никогда НЕ храните в классах и структурах `bool` переменные. Проблема тут такая же как и в предыдущем пункте – на стороне пользователя будет тяжело понять, что значит присваивание `true` или `false`. Куда лучше завести `enum class` а-ля Статус и использовать хорошие имена для статуса. Например, если вы хотите пройтись по дереву собранному на нодах с указателями и пометить вершину, как посещенную, то есть соблазн сделать так

```
struct Node {
  Data data;
  bool is_visited = false;
  Node* parent = nullptr;
  Node* left = nullptr;
  Node* right = nullptr;
};
```

Однако, при инициализации можно написать так

```
Node node{Data(), true};
```

Что не помогает чтению. Несколько спасает возможность указать имя инициализируемого члена. Куда лучше сделать `enum class` и написать

```
enum class Status : unsigned char {
  NotVisited, Visited
};

struct Node {
  Data data;
  Status status = Status::NotVisited;
  Node* parent = nullptr;
  Node* left = nullptr;
  Node* right = nullptr;
};
```

Дополнительный плюс этого подхода заключается в том, что теперь можно много разных флагов хранить в одной переменной и не тратить по одному байту на разные свойства. Для этого полезно будет перегрузить операторы `operator|`, `operator|=`, `operator&` и `operator&=` для побитовых операций и получится[24]

```
enum class Status : unsigned char {
  None = 0,
  Visited = 1,
  Leaf = 2,
  Bad = 4
};

Status& operator|=(Status& lhs, Status rhs);
Status& operator&=(Status& lhs, Status rhs);
Status operator|(Status lhs, Status rhs);
Status operator&(Status lhs, Status rhs);

struct Node {
```

[24]Если вы поддерживаете побитовые операции, то обязательно укажите, что ваш `enum class` имеет беззнаковый подлежащий тип. Потому что для знаковых типов побитовые операции часто UB. Лучше не играть с этим.

```
14    Node* parent = nullptr;
15    Node* left = nullptr;
16    Node* right = nullptr;
17    Data data;
18    Status status = Status::None;
19  };
```

И теперь можно создавать вершину так

```
1  Node node{.data = Data(), .status = Status::Visited | Status::Leaf};
```

Если использовать имена полей структуры, то не важно в каком порядке они идут в структуре и в списке инициализации.

10 Указатели

10.1 Виды указателей

В C++ существует два владеющих указателя:

- `unique_ptr` для уникального владения объектом, а потому поддерживается только перемещение.
- `shared_ptr` для совместного владения объектом. Данный вид указателя является очень опасным механизмом языка, потому что он ломает сразу несколько принципов и им важно научиться пользоваться правильно.

При этом в связке с ними идут еще два не владеющих указателя:

- Сырой указатель `T*`, который используется для указания на данные, когда гарантируется, что данные будут живы. Такой указатель обычно используется в паре с `unique_ptr`.
- `weak_ptr`, не владеющий указатель, который знает, что он ссылается на данные лежащие в `shared_ptr`. Такой указатель используется в паре с `shared_ptr`.

Все указатели кроме сырого принято называть «умными» указателями. Однако, более правильно делить их на владеющие и не владеющие. Кроме владеющих указателей есть еще один вид указателей, который я буду обсуждать (и которые хорошо было бы называть умными) – это «отслеживающие» указатели. Такие указатели позволяют отслеживать адрес объекта и знать жив ли еще объект. Такие указатели можно использовать в мультиагентной среде для посылки сообщений между агентами, где вы можете не знать, будет ли жив адресат к моменту, когда письмо дойдет. Примером такого указателя является `QPointer` в экосистеме Qt. Ниже в разделе 12.6 я расскажу о таких указателях более детально.

10.2 Владеющий указатель `unique_ptr`

Основное время вы будете использовать переменные нужного класса для хранения данных и нет необходимости пользоваться указателями. Однако, есть ситуации, когда бывает осмысленно поместить объект в памяти на кучу. Давайте поговорим об этих ситуация:

1. Вам необходимо сделать объект адресуемым, то есть чтобы его адрес был постоянен в течение жизни программы, чтобы на него можно было ссылаться (предполагается, что вы как-то организуете гарантию, что объект будет жив при этом). В этом случае проще всего положить объект в `unique_ptr`. При таком подходе вы не теряете value semantics и можете спокойно пользоваться контейнерами вроде `std::vector` для хранения подобного объекта.

2. Хрупкие объекты. Мы уже обсуждали их в разделе 6.3.14. Напомню, что если внутри объекта у вас есть указатели на разные элементы объекта, то очень проблематично писать напрямую копирование и перемещение объекта, вам придется все данные указатели настраивать после снятия копии или перемещения. И если копирование просто так не решается, то вот перемещение можно сделать бесплатным, положив объект в `unique_ptr`.

3. Борьба с тяжелыми объектами. Если объект класса имеет легкую локальную часть и для него move операция дешевая, то можно легко перемещать данные между методами через rvalue reference. Однако, если же данные в объекте все живет локально, как например в `std::array`, то нет разницы между move и copy. С другой стороны, перемещать по ссылке данные не всегда возможно. Например, если вызов метода не блокирующий и будет использовать данные позже или же при передачи данных между потоками. В этом случае для облегчения данных их можно положить в `unique_ptr`. Если вы хотите поддержать копирование, то вам придется написать соответствующую обертку.

4. Освобождение стека. Если вы пишете приложение, то точка входа в программу может выглядеть так

```
1  int main() {
2    try {
3      Application app;
4      app.run();
5    } catch (...) {
6      react();
7    }
8  }
```

В операционной системе функция `main` запускается на своем thread-е со своим стеком. И объект `Application` создается на стеке. Если он слишком тяжелый и содержит все свои данные локально, то может съесть слишком много места на стеке. Вы скорее всего этого не почувствуете, пока не сработает какая-нибудь рекурсия, которая съест стек. Так же надо иметь в виду, что размер стека на разных операционных системах разный и обычно на Windows стек больше, чем на Linux по умолчанию. В таком случае принято `Application` заворачивать в `unique_ptr` как это рассказывалось в разделе 6.3.14 про хрупкие объекты. Или как в разделе 6.3.18 про Pimpl.

10.3 Владеющий указатель `shared_ptr`

Существует всего три и только три случая использования `shared_ptr`:

1. Использование `shared_ptr<const T>`. В этом случае это дешевый способ сделать тяжелые данные легко копируемыми за счет невозможности их модифицировать.

2. Использование `shared_ptr<T>`, где `T` не константный тип. Это более опасный объект и у него есть два случая применения:

   (a) Имплементация механизмов межпотокового взаимодействия, например `future/promise`. В этом случае оба thread-а должны владеть областью памяти для возможности коммуникации. При этом никто из thread-ов не может зарание знать, кто из них умрет раньше, потому никто не может гарантировать, кто должен последним отпустить ресурсы. Причина использования `shared_ptr` в этом случае в том, что без него просто не возможно имплементировать подобное взаимодействие.

   (b) Персистентные структуры данных или структуры, которые должны шарить общие ресурсы для обеспечения более эффективного механизма копирования. В этом случае все копии одной структуры имеют внутри большое количество общих данных, которые они копируют лишь при необходимости. И владение этим ресурсом осуществляется с помощью `shared_ptr`. Самый простой пример – copy-on-write подход. В рамках этого подхода внутри класса хранится `shared_ptr` на данные и только при необходимости изменить данные с них снимается копия (при наличии нескольких владельцев). Мы обсудим один из примеров использования `shared_ptr`, когда классы имеют общий инвариант, ниже в разделе 12.6.

`shared_ptr<const T>` Обратите внимание, что когда вы используете `shared_ptr` на константный объект, то это гарантированно value semantics. Потому это безопасное использование механизма языка. Однако, не стоит пользоваться сырым `shared_ptr` в коде, заверните указатель в класс или шаблон например так

```
1  template<class T>
2  class ConstValue {
3  public:
4    template<class... Args>
5    ConstValue(Args&&... args)
6     : data_(std::make_shared<const T>(std::forward<Args>(args)...)) {}
7    ConstValue(const ConstValue&) = default;
8    ConstValue(ConstValue&&) noexcept = delete;
9    ConstValue& operator=(const ConstValue&) = default;
10   ConstValue& operator=(ConstValue&&) noexcept = delete;
11   ~ConstValue() = default;
12
13   const T* operator->() const {
14     return data_.get();
15   }
16 private:
17   std::shared_ptr<const T> data_;
18 };
```

Тут надо сделать несколько замечаний. Во-первых, зачем производится такое заворачивание. Дело в том, что вы хотите снаружи иметь поведение такое же как и у вашего класса типа `T` и не иметь как минимум лишних методов в интерфейсе объекта. Если использовать `shared_ptr` напрямую, то в интерфейсе у вас будет много методов специфических для указателя, которых не должно быть в классе `T`. Во-вторых, если вдруг вам потом понадобится поменять что-то в коде или добавить какое-то специальное поведение, то проще его добавить в один класс, а не мучиться переписывать все места использования `shared_ptr` напрямую. В-третьих, обратите внимание на удаление методов мува. Это сделано для того, чтобы ваш объект не мог

находиться в пустом состоянии, в этом случае `operator->` никогда не вернет `nullptr` и потому безопасен в использовании. Можно пойти другим путем и просто сделать move равносильным копированию, что тоже обеспечит нужное поведение, при котором оператор `operator->` никогда не падает, и это даже будет работать лучше.

`shared_ptr<T>` Если же вы используете `shared_ptr` на неконстантный объект, то это всегда имплементация какого-то паттерна или структуры, а потому его можно использовать напрямую внутри этой структуры, так как это обычно деталь имплементации. Тем не менее, даже в этом случае имеет смысл заводить псевдоним, чтобы можно было потом легко подменять разные версии `shared_ptr` (если вдруг вы захотите использовать не STL версию).

В разделе 4 я уже обсуждал проблемы с `shared_ptr` на неконстантный объект. Давайте я резюмирую эти соображения тут еще раз

1. Нарушение локальности. У вас могут быть сколь далекие друг от друга переменные `shared_ptr` ссылающиеся на одни и те же данные. Эти переменные могут даже быть в разных единицах трансляции, в разных бинарниках, в разных проектах. И любые изменения одной влекут непредсказуемое поведение в совершенно другой части кодовой базы. Это означает, что эти разбросанные по кодовой базе данные имеют общий инвариант, который не скрыт в класс. А потому пользователь этих переменных обязан всегда при любом использовании следить негласно за выполнением нужных инвариантов, что разумеется не реалистично (вы можете банально не знать о таких инвариантах). Разбросанные в коде данные с общим инвариантом называются Incidental Data Structures. Их надо всегда избегать и заворачивать в классы.

2. Нарушение value semantics. Тут я не буду особенно повторяться, но reference semantics она контринтуитивная и с ней очень тяжело работать. С другой стороны value semantics is safe by default.

11 Полиморфизм

11.1 Что это и каким оно бывает

По определению полиморфизм означает, что один и тот же текст (=код) делает разные вещи. Например в языке C++ можно написать строчку кода:

```
1  z = x + y;
```

В зависимости от типа переменных `x`, `y` и `z` эта строчка будет вызывать разные процессорные инструкции даже для встроенных типов `int`, `double`, не говоря уже о перегруженных операторах для классов. Потому технически, это пример полиморфного поведения кода. Обычно про такие примеры не говорят, но и полиморфизм обычно определяют очень узко, привязывая его только к определенной имплементации или инструменту языка, а не к поведению. На мой взгляд на языковые вещи надо смотреть не с точки зрения имплементации, а с точки зрения поведения.

Прежде чем двигаться дальше я хочу сделать полезное замечание. Языки программирования по отношению к типам данных бывают трех типов:

- Статическая типизация. Тип переменной не может измениться во время исполнения.
- Динамическая типизация. Тип переменной может поменяться во время исполнения.
- Без типизации. Понятия типа просто не существует.

Язык C++ относится к языкам со статической типизацией. Чем мы выше в этом списке, тем надежнее механизмы языка, тем больше гарантий вам может теоретически предоставить компилятор и инструменты разработки. Чем ниже в этом списке, тем более гибким является язык. Это всегда размен надежности на гибкость. Если в языке нет типизации или динамическая типизация, то все поведение полиморфно. Возьмите для примера язык Python с динамической типизацией. Если у вас есть функция вида

```
1  def function(a):
2    a.f()
```

То не важно какого типа аргумент `a` функции `function`. Единственное что нужно для исполнения этого кода, чтобы для `a` имела смысл строчка вызова `a.f()` и все. В эту функцию можно подставлять любой объект, у которого можно вызвать метод `f` без аргументов. А потому про полиморфизм имеет смысл говорить только в языках со статической типизацией, где по определению вы не можете подставить вместо переменной одного типа переменную другого типа. Точнее, в языках со статической типизацией возникает следующий вопрос: «как добиться полиморфного поведения, не ломая систему типов?»

Полиморфизм и функции Давайте для начала обсудим какие есть механизмы в C++ для полиморфизма.

1. Перегрузка функций, методов и операторов. Да, это тоже полиморфизм и именно с этого примера я начал этот раздел.

```
1  void f(int);
2  void f(double);
3  void f(A);
4
5  int main() {
6    int x = 1;
7    f(x);
8    double y = 2.;
9    f(y);
10   A a;
11   f(a);
12  return 0;
13 }
```

   Как вы видите, функция с именем `f` может быть вызвана для переменных разных типов. То есть один и тот же код вида `f(...)` может выполнять разные действия.

   Важно отметить, что в данном случае у нас есть полиморфзим на стороне пользователя функции `f`, но не на стороне автора функции `f`. Действительно, перегрузка требует от вас определить все три вида

функции, а потому вы обязаны написать все три имплементации и у вас нет возможности использовать один и тот же код для этого (я не говорю сейчас про баловство с макросами, потому что это в целом издевательство над текстом программы).

```
void f(int) {
  std::cout << "f(int);\n";
}
void f(double) {
  std::cout << "f(double);\n";
}
void f(A) {
  std::cout << "f(A);\n";
}
```

2. Шаблоны функций, переменных и классов. Этот вид полиморфизма обычно называют compile-time polymorphims. Благодаря шаблонам вы можете добиться полиморфизма и на стороне пользователя кода и на стороне автора кода, то есть когда и пользователь и автор кода пишет один код, который делает разную работу.

```
template<class T>
void f(T);

int main() {
  int x = 1;
  f(x);
  double y = 2.;
  f(y);
  A a;
  f(a);
}
```

В этом случае программист может написать следующую имплементацию

```
template<class T>
void f(T) {
  std::cout << "f(" << ClassPrintName<T> << ");\n";
}
```

Здесь `ClassPrintName` – это вспомогательный шаблон, который перерабатывает класс в читаемое имя класса в формате строки для отображения. По сути шаблоны – это умная кодогенерация, которая делается механизмами языка.

3. Run-time polynorphism. Это вид полиморфизма относится к ситуации, когда вы хотите менять поведение кода во время исполнения, то есть что именно будет исполнено не определяется на этапе компиляции кода. Например, когда вы хотите поменять поведение кода в зависимости от ввода пользователя или от возможностей железа, на котором исполняется программа. Если вы думаете, что run-time polymorphism это сугубо фича объектно ориентированных языков программирования, то вы глубоко ошибаетесь. Даже в языке Си можно добиться полиморфзного поведения во время исполнения и имплементация всех операционных систем использует этот механизм. Для этого используются указатели на функции (код ниже содержит псевдонимы типов из C++ для удобства чтения кода)

```
void f1();
void f2();

using Signature = void();
using FPtr = Signature*;
FPtr f = &f1;

int main() {
  f(); // f1 executes
  f = &f2;
  f(); // f2 executes
  return 0;
}
```

Обратите внимание, что в коде выше мы подменяем указатель на функцию и в зависимости от этого вызов `f()` приводит к выполнению разных функций.

Полиморфизм и классы Теперь давайте обсудим, что означает полиморфизм на уровне классов, а не функций. В этом случае есть две задачи:

1. Мы хотим ссылаться на переменные разных типов и уметь звать у них методы с фиксированными названиями. Например:

```
struct A {
  void f() const;
};

struct B {
  void f() const;
};

void f(Pointer ptr) {
  ptr->f();
}

int main() {
  A a;
  f(&a);
  B b;
  f(&b);
}
```

   Здесь `Pointer` – это какой-то пока какой-то нам неизвестно как устроенный класс, который играет роль указателя, но только в него можно положить указатель на любой класс, в котором есть метод `f`, который можно вызвать без аргументов. Я не знаю, есть ли у этого подхода стандартное название, но я бы по аналогии со следующим пунктом назвал это «стирающим указателем», потому что мы стираем информацию о типе объекта, на который указывает «указатель», и все что нам доступно – это вызов методов из поддерживаемого интерфейса стирающего указателя.

2. Мы хотим хранить в переменной объекты разных типов и обращаться к их методам по фиксированным названиям. Например мы хотим написать что-то в таком духе

```
struct A {
  void f() const;
};

struct B {
  void f() const;
};

int main() {
  std::vector<Any> vec;
  vec.push_back(A());
  vec.back().f();
  vec.push_back(B());
  vec.back().f();
}
```

   В этом случае `Any` – это тип, в который можно положить любую переменную любого типа, главное, чтобы в ее интерфейсе был метод `f`, который можно вызвать без аргументов. Такой подход называется «стирающим типом» или по английски «Type Erasure». В этом случае тип `Any` как раз является стирающим типом, потому что при присваивании ему, стирается информация о типе исходной переменной, и единственный способ взаимодействовать с ней у нас остается лишь посредством интерфейса, который поддерживает стирающий тип. При этом статическая типизация языка позволяет проверять во время компиляции, можно ли положить данную переменную внутрь стирающего типа или нет и выдать ошибку компиляции, если нельзя.

Важное замечание В этом месте я хочу сделать важное замечание, когда я говорю про «общий интерфейс», я НЕ имею в виду наследование или интерфейсы с виртуальными методами. Я имею в виду публичные методы класса и все. То есть стирающие типы и указатели не требуют от вас калечить код всех возможных классов, чтобы они стали используемыми в коде, а наоборот, суть в том, что мы хотим, навесить интерфейс поверх класса неинтрузивно (то есть не вмешиваясь в код класса). Бывает два вида навешивания интерфейса:

1. Утиный принцип. Мы можем присвоить к `Pointer` из примера выше любой адрес любого объекта, главное, чтобы через этот указатель можно было позвать нужные методы. То есть если синтаксически использование возможно, то оно разрешено без каких-то дополнительных ограничений. Этот принцип похож на поведение Python с его динамической типизацией, когда вы можете положить в переменную, что хотите, главное чтобы синтаксически код был корректным.

2. Явное объявление присоединяемого интерфейса. В этом случае где-то в коде должна быть строчка, которая говорит, что мы разрешаем данному классу присоединяться к данному интерфейсу. Данное поведение можно встретить в таких языках как Rust, когда вы присоединяете trait-ы к структуре.

Обратим внимание, что имплементация стирающих типов не обязана использовать наследование в каком-либо виде, но так исторически сложилось, что именно наследование с виртуальными методами рассматривается как тот самый настоящий run-time polymoprhism. Но это просто ошибка.

11.2 Важный пример

Я решил, что прежде чем рассказывать про все методы по одиночке, я хочу рассмотреть конкретный пример, на котором буду обсуждать дальнейшее. Предположим, что у вас есть некоторая функция `f`, которая внутри себя последовательно вызывает три функции `f1`, `f2` и `f3`. И предположим, что вы хотите вместо функции `f2` использовать другие функции. Вот как бы это решение выглядело на Python:

```python
## script.py

def f1():
  pass
def f2():
  pass
def f3():
  pass
def g():
  pass
```

```python
def function():
  f1()
  function.f()
  f3()
function.f = f2

def main():
  function()
  funciton.f = g
  function()
```

Вот как бы это решение выглядело на Си:

```c
void f1();
void f3();

using Signature = void();
using FPtr = Signature*;
FPtr f = &f2;

void function() {
    f1();
    f();
    f3();
}
```

```c
#include "Function.h"

void g() {
}


int main() {
  function();
  f = &g;
  function();
  return 0;
}
```

Предыдущее решение вполне подходит и для C++, но если метод `f` может использовать внутреннее состояние, то есть более удачное решение на C++:

```c
struct Function {
  static void f1();
  static void f2();
  static void f3();
  void operator()() const {
    f1();
    f();
    f3();
  }
  std::function<void()> f = f2;
};
```

```c
#include "Function.h"

void g() {}

int main() {
  Function function;
  function();
  function.f = g;
  function();
  return 0;
}
```

Ну и конечно же классическое решение на C++, которое вы найдете в любой книжке про полиморфизм:

```
// Function.h

struct Function {
  static void f1();
  static void f2();
  static void f3();
  void operator()() const {
    f1();
    f();
    f3();
  }
  virtual void f() const {
    f2();
  }
  virtual ~Function();
};
```

```
struct FunctionG : Function {
  static void g();
  void f() const {
    g();
  }
};

int main() {
  Function* f = new Function();
  (*f)();
  delete f;
  f = new FunctionG();
  (*f)();
  delete f;
  return 0;
}
```

Уже сейчас вы можете оценить как минимум сложность по количеству написания кода для решения задачи каждым из методом. Дальнейшее изложение я хочу начать с обсуждения классического решения, как им предполагается пользоваться, как оно работает и почему оно является плохим во всех возможных смыслах. После я перейду к обсуждению стирающих указателей и типов с объяснением того, как должна выглядеть имплементация.

11.3 Классический подход и наследование с виртуальными методами

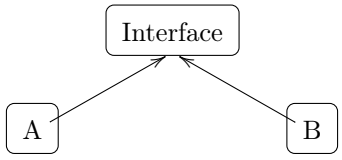
Давайте я начну с описания того, что мы вообще хотим сделать. Мы хотим добиться полиморфного поведения для классов, а именно, мы хотим, чтобы методы класса с одними и теми же именами во время исполнения могли исполнять разный код. Для этого в языке C++ предусмотрен специальный механизм – наследование с виртуальными методами. В начале мы заводим класс-интерфейс, в котором фиксируем интересующий нас публичный интерфейс

```
1 struct Interface {
2   virtual void f() const = 0;
3 };
```

После чего, если мы хотим, чтобы класс подходил под этот интерфейс, мы должны прикрепить этот интерфейс к нужному классу. Происходит это путем наследования ОТ интерфейса вот так:

```
1 struct A : Interface {
2   void f() const override { /*implementation*/ }
3 };
4
5 struct B : Interface {
6   void f() const override { /*implementation*/ }
7 };
```

Тут есть две проблемы. Это точно не утиный принцип, но это еще не большая беда. Главная проблема такого соединения – мы не интерфейс прицепляем к классу, а класс к интерфейсу. То есть мы делаем зависимость не в ту сторону, в которую хотим. Проблемы такого соединения интерфейса и класса мы обсудим ниже, а пока логическая диаграмма наследования между классами:



Теперь, чтобы полиморфизм работал, нам нужно создать объект типа `A`, но ссылаться на него как на объект типа `Interface`. Если нам не надо владеть объектами, то это можно сделать так

```
1 void f(Interface* ptr) {
2   ptr->f();
3 }
```

```
4
5  int main() {
6    A a;
7    f(&a); // casts A* to Interface*
8    B b;
9    f(&b); // casts B* to Interface*
10 }
```

Если же мы напишем так

```
1  int main() {
2    A a;
3    Interface obj = a; // compilation error
4  }
```

То так как класс `Interface` содержит чисто виртуальный метод (тот что равен нулю при объявлении), то невозможно создать объект такого типа и это ошибка компиляции. Давайте поменяем определение так

```
1  struct Interface {
2    virtual void f() const {}
3  };
```

Теперь можно написать код вида

```
1  A a;
2  a.f(); // calls A::f()
3  Interface obj = a;
4  obj.f(); // calls Interface::f()
5  B b;
6  b.f(); // calls B::f()
7  obj = b;
8  obj.f(); // calls Interface::f()
```

Как мы видим, в этом случае полиморфного поведения нет. Вся причина в том, что оператор присваивания `obj = a` строит по объекту типа `A` объект типа `Interface`, то есть от класса `A` отрезается его базовая часть и складывается внутрь `obj`. И мы лишаемся таким образом полиморфного поведения. Обойти эту проблему на уровне value semantics не возможно. Нам придется работать с голыми указателями. А именно, нам надо создать где-то объект типа `A`, и потом кастовать его адрес к адресу на `Interface`. И вот в этом случае обращение к методу `f` через `Interface` гарантирует вызов `A::f`. Теперь пользоваться всем этим можно так

```
1  int main() {
2    std::unique_ptr<Interface> ptr = std::make_unique<A>();
3    ptr->f(); // calls A::f()
4    ptr = std::make_unique<B>(); // calls ~Interface before storing new value
5    ptr->f(); // calls B::f()
6    return 0;
7  } // ~ptr calls ~Interface
```

Однако, в 4-ой строке перед тем, как положить новое значение в `ptr`, будет уничтожено старое значение, а именно вызывается `~Interface`. Но по указателю лежит объект класса `A`, а значит вызовется не верный деструктор. Аналогичная проблема есть в 7-ой строке при выходе из `main` вызывается деструктор для `ptr`, который опять же вызовет деструктор для `Interface`. Для решения этой проблемы надо сделать деструктор в `Interface` тоже виртуальным, чтобы вызвался нужный деструктор.

```
1  struct Interface {
2    virtual void f() const = 0;
3    virtual ~Interface() = default;
4  };
```

Вот теперь это безобразие заработает как надо. Но не как мы хотели. Давайте обсудим, что же не так с этим решением.

Проблемы классического решения

1. Первое и самое главное: мы делаем не то, что хотим. А именно, мы хотим к любому типу привесить интерфейс, но делаем это наоборот, мы к интерфейсу привешиваем тип. А именно у нас зависимость `Interface` ← `A`, а не наоборот. В частности, класс `A` начинает зависеть от интерфейса, а не наоборот. Мы

не можем подвесить интерфейс к классу постфактум, если нет доступа к коду класса, то нет возможности к нему подвесить интерфейс. Для того, чтобы обойти эту проблему придется писать адаптеры.

2. Мы ломаем логику зависимостей в кодовой базе. Так как у нас класс `A` зависит от `Interface`, то мы инклудим в файл с классом `A` файлы, необходимые для работы интерфейса, а не самого класса.

3. Вместо написания самого класса `A`, мы теперь должны думать про виртуальные методы. А именно, мы теперь для каждого интерфейса должны писать лишний код, который делает простой и понятный класс `A`, который до этого имел простую и понятную структуру, сложным и непонятным.

4. Наличие чистых виртуальных функций не согласовано с оператором присваивания. А потому нам приходится работать с объектами через указатели.

5. Наличие виртуальных функций – удар по производительности даже если вы используете класс не полиморфно. Вы всегда платите цену за наличие интерфейса.

6. Работа с объектами через указатели означает, что мы поменяли семантику кода. Если до использования интерфейсов мы могли оперировать переменными и вызывать у них методы, то теперь мы оперируем ссылками и указателями.

7. Работа с объектами через указатели означает так же, что мы должны аллоцировать объекты на куче. А это значит, что пользователь полиморфного класса обязан следить за памятью и lifetime-ом объекта. Нужно заворачивать объекты в адаптеры вроде `unique_ptr`.

8. Потеря value semantics и использование прямых указателей в коде побуждает в объектах ссылаться друг на друга или иметь общее состояние через `shared_ptr`. Это порождает случайные связи между совершенно разными не связанными кусками кода. Код становится сложно оценивать из локальных соображений. Код становится сложно поддерживаемым. Начинаются проблемы с синхронизацией и lifetime-ами.

Если вернуться к примеру из предыдущего раздела

```
// Function.h

struct Function {
  static void f1();
  static void f2();
  static void f3();
  void operator()() const {
    f1();
    f();
    f3();
  }
  virtual void f() const {
    f2();
  }
  virtual ~Function();
};
```

```
struct FunctionG : Function {
  static void g();
  void f() const {
    g();
  }
};

int main() {
  Function* f = new Function();
  (*f)();
  delete f;
  f = new FunctionG();
  (*f)();
  delete f;
  return 0;
}
```

То можно добавить еще одну проблему. Конкретно в этом случае мы не можем использовать любую функцию `g`. Нам приходится заворачивать код этой функции внутрь класса наследника. Потому что теперь объект – это то же самое, что наследник. То есть поведение контролируется НЕ переменной класса, а самим классом. Все переменные одного класса имеют одинаковое поведение и чтобы поменять поведение, надо писать аж целый новый класс.

11.4 Стирающие ссылки

Так получилось, что даже в языке Си есть стирающие указатели. Это называется `void*`. Действительно, вы можете кастануть указатель на любой класс `A*` в указатель типа `void*`. Однако, это стирающий тип с пустым интерфейсом в том смысле, что у вас нет возможности воспользоваться никакими методами этого класса. Все что вы можете – это кастануть обратно к исходному типу и лишь после этого воспользоваться данными по указателю. То есть в языке уже есть стирающий указатель, но с пустым интерфейсом, и теперь наша задача поверх него сделать стирающий указатель но с фиксированным интерфейсом.

11.4.1 Ожидаемое поведение

Что мы хотим В идеале мы хотим следующее поведение.

```
interface A {
public:
  void f(int) const;
};

class B {
public:
   void f(int) const { /*implementation*/ }
};
```

```
void g(A a) {
  a.f(5); // a->f(5);
}

int main() {
  B b;
  g(b); // g(&b);
  return 0;
}
```

То есть мы бы хотели задать интерфейс `A`, который играет роль указателя, такой, что указатель на любой объект (или ссылка на любой объект), который можно использовать с этим интерфейсом биндилась на объект класса `A`. Давайте проговорим все основные пункты этого решения:

1. Нет синтаксической связи между `A` и `B`
2. `A` и `B` связаны именем функции и сигнатурой
3. Размер `A` порядка указателя ($\simeq 1-2$ указателя)

Опционально можно добавить еще несколько фич:

1. `A` знает жив ли объект
2. `A` следит за адресом объекта

Более правдоподобная картина Ключевое слово `interface` мы будем симулирвоать шаблонным классом, а вот интерфейс настраивать шаблонным параметром. Опять же приблизительное идейное решение выглядит так:

```
class A {
public:
  virtual void f(int) const = 0;
};

using C = Interface<A>;

class B {
public:
   void f(int) const;
};
```

```
void g(C a) {
  a->f(5);
}

int main() {
  B b;
  g(&b);
  return 0;
}
```

Таким образом:

- Шаблонный класс `Interface` скрывает весь boilerplate code и симулирует поведение ключевого слова `interface`
- Интерфейс вычленяется из адреса объекта (можно настроить)
- К сожалению прям так пока нельзя, но можно почти так.

Несколько интерфейсов Давайте еще немного помечтаем и поймем, какие интересные штуки можно делать. Классический подход к получению интерфейса – взять адрес объекта и кастовать его к адресу базового класса-интерфейса. Однако, если у нас теперь интерфейс не привязан к наследованию и механизмам языка, то мы можем просто возвращать интерфейс из метода. Причем можно назвать функции возвращающие интерфейс удобным образом и поддерживать много разных интерфейсов.

```cpp
class A {
public:
  virtual void f(int) const = 0;
};

using C = Interface<A>;

class B {
public:
   void f(int) const;
   C interface1() const;
   C interface2() const;
};
```

```cpp
void g(C a) {
  if(!a)
    return;
  a->f(5);
}

int main() {
  B b;
  g(&b);
  g(b.interface1());
  g(b.interface2());
  return 0;
}
```

Важные моменты:

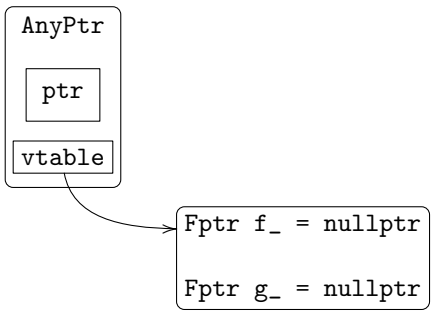
- Объект может сам предоставлять разные интерфейсы с одинаковыми методами. Наследование напрямую не позволяет поддержку такого механизма, но это тоже реализуемо.
- Объект может контролировать количество созданных интерфейсов. Тоже интересная фича, которую можно тут поддержать.
- Предоставляемый интерфейс $\neq$ Публичный интерфейс. То есть у класса может быть свой публичный интерфейс (в смысле публичные методы), а интерфейс, который он предоставляет через метод может быть вполне себе адаптером.

11.4.2 Идея имплементации

Давайте создадим класс `AnyPtr`, который умеет хранить любой указатель на объект, у которого доступны два метода

- `void f() const`
- `void g()`

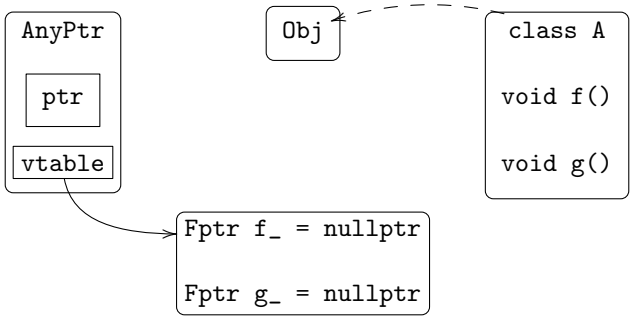
Тогда структуру `AnyPtr` можно графически изобразить так:



Объект `AnyPtr` будет содержать два указателя:

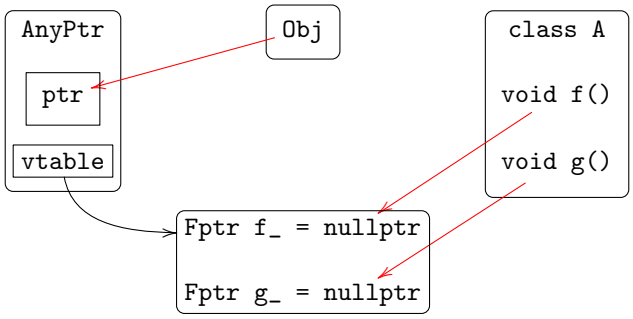
1. Указатель `ptr` в виде `void*` на объект класса, которым мы хотим пользоваться через указатель.
2. Указатель `vtable` на таблицу функций, которые мы будем у него вызывать.

Теперь, если у нас есть объект `Obj` класса `A`, который содержит нужные методы



То при присваивании объекта `Obj` объекту класса `AnyPtr` надо сделать две вещи:

1. Присвоить адрес на `Obj` к указателю `ptr`.
2. Положить в таблицу `vtable` указатели на методы `f` и `g` из класса `A`, которому принадлежит объект `Obj`.



Технически, так как методы `f` и `g` зависят от класса `A` и для всех объектов класса `A` они будут одинаковые, то нет необходимости для каждого объекта хранить на куче свою таблицу. Вместо этого для класса `A` заведем одну статическую таблицу и будем хранить указатель на нее.

11.4.3 Имплементация

Давайте разбираться с деталями имплементации. В начале надо понять, как мы будем работать с таблицами указателей на функцию. Тут мы по сути повторяем то, как имплементирован механизм с виртуальными функциями.

Виртуальные таблицы В начале надо заменить методы на глобальные функции, чтобы у нас не было зависимости от типа `A`. Делается это так:

```
struct A {
  void f() const;
  void g();
};
```

```
void f_(const A* self);
void g_(A* self);
```

Теперь заведем структуру, в которой будут лежать указатели на данные функции:

```
1  struct VTable {
2    using FSignature = void(const void*);
3    using GSignature = void(void*);
4
5    FSignature* f_ = nullptr;
6    GSignature* g_ = nullptr;
7  };
```

Обратите внимание, что настоящий тип функции `f_` есть `void(const A*)`. Однако, в таблице будет лежать функция с сигнатурой `void(const void*)`. А значит, нам надо будет сделать несколько вспомогательных действий. Давайте заведем шаблонный класс, который будет хранить статическую таблицу.

```
1  template<class T>
2  struct Bind {
3  static void f_(const void* self) {
4    static_cast<const T*>(self)->f();
5  }
6  static void g_(void* self) {
7    static_cast<T*>(self)->g();
8  }
9
10 static constexpr VTable vtable{f_, g_};
11 };
12
13 template<>
14 struct Bind<void> {
15
16 static constexpr VTable vtable{nullptr, nullptr};
17 };
```

Здесь для каждого типа `T` мы можем создать структуру `Bind<T>`.[25] Внутри этой структуры мы биндим настоящие методы из класса `T` (который будет `A` в нашем примере выше). Этот бинд делается в строках 3-5 для метода `T::f()` и в строках 6-8 для метода `T::g()`. После чего мы создаем статическую переменную `vtable` внутри этой структуры, которая будет хранить виртуальную таблицу на забинденные методы. Таким образом `Bind<T>::vtable` – это будет таблица с методами для указателя на объект типа `T`. Обратите внимание, что мы принимаем объект `self` в формате `void*` или `const void*`, потому что именно в таком формате он будет лежать внутри `AnyPtr`. Но так как таблица с методами и указатель на объект согласованы, мы знаем, что всегда можно кастовать `self` в тип `T*` или `const T*`. Таким образом мы получаем доступ к настоящим методам `f` и `g` из класса `T`. И если в классе `T` нет возможности вызвать эти методы как в строках 4 и 7, то произойдет ошибка компиляции. Это хорошо, потому что это означает, что мы пытаемся использовать класс `T` не поддерживающий нужные методы и компилятор нашел нашу ошибку. После общего определения идет специализация для таблицы для типа `void`, который используется как sentinel и означает, что эта таблица ни на что не забиндилась.

Имплементация `AnyPtr` Теперь когда мы определились с тем, как устроен вспомогательный класс для работы с таблицами функций, давайте опишем как будет выглядеть основной класс `AnyPtr`. В начале опишем его методы и данные, а потом уже напишем имплементацию всех методов:

```
1  class AnyPtr {
2  public:
3    AnyPtr();
4    AnyPtr(std::nullptr_t);
5    template<NonVoidClass T>
6    AnyPtr(T* ptr);
7
8    AnyPtr(const AnyPtr& other);
9    AnyPtr& operator=(const AnyPtr& other);
10
11   AnyPtr(AnyPtr&& other) noexcept;
12   AnyPtr& operator=(AnyPtr&& other) noexcept;
13   ~AnyPtr();
14
15   operator bool() const;
16
17   void f() const;
18   void g();
19
20 private:
21   void clear();
22
23   const VTable* vtable_ = &Bind<void>::vtable;
24   void* ptr_ = nullptr;
25 };
```

Как мы видим в классе лежат только два указателя:

[25] Название пожалуй не лучшее, но я не знаю как его еще назвать.

1. `vtable_` указывает на таблицу функций. По умолчанию ссылается на пустую таблицу функций определенную для типа `viod`.

2. `ptr_` указывает на объект, у которого мы будем вызывать нужные методы. По умолчанию ссылается на `nullptr`.

В начале разберемся с вопросами обеспечения правильной семантики для `AnyPtr`. Это означает, что этот класс должен правильно создаваться, копироваться, перемещаться и уничтожаться. Для этого имплементируем все конструкторы, деструктор и операторы присваивания. Я в начале сгруппирую дефолтные конструкторы, а уже после них те, что требуют ручной имплементации.

```
1  AnyPtr() = default;
2  AnyPtr(const AnyPtr& other) = default;
3  AnyPtr& operator=(const AnyPtr& other) = default;
4  ~AnyPtr() = default;
5
6  AnyPtr(AnyPtr&& other) noexcept
7      : vtable_(std::exchange(other.vtable_, &Bind<void>::vtable)),
8        ptr_(std::exchange(other.ptr_, nullptr)) {
9  }
10
11 AnyPtr& operator=(AnyPtr&& other) noexcept {
12   AnyPtr tmp = std::move(other);
13   std::swap(vtable_, tmp.vtable_);
14   std::swap(ptr_, tmp.ptr_);
15   return *this;
16 }
17
18 AnyPtr(std::nullptr_t) {
19 }
20
21 template<NonVoidClass T>
22 AnyPtr(T* ptr) : vtable_(&Bind<T>::vtable), ptr_(ptr) {
23 }
```

Дефолтный конструктор настраивается путем инициализации полей класса. Так как все поля класса – это встроенные типы, то деструктор, конструктор копирования и копирующее присваивание тоже могут быть дефолтными. Интерес представляют конструкторы от указателей и мувающие конструкторы. Так как поля класса встроеенные типы, то мувающие операции по дефолту совпадают с копирующими, а я все же хотел бы, чтобы после мува, мы вынули из старой переменной указатель и перенесли его в новую.

```
1 A a;
2 AnyPtr ptr1 = &a;
3 AnyPtr ptr2 = std::move(ptr1);
4 AnyPtr ptr3;
5 ptr3 = std::move(ptr2);
```

Я хочу, чтобы в строке 3 указатель `ptr1` инвалидировался. Аналогично в строке 5 хочу, чтобы указатель `ptr2` инвалидировался. А для этого нужно сделать очистку данных в старом объекте в мувающем конструкторе и операторе присваивания. Тут используются стандартные имплементации, которые обсуждались в разделе 6.3.5.

Конструктор от `std::nullptr_t` нужен, чтобы можно было написать

```
1 A a;
2 AnyPtr ptr = &a;
3 ptr = nullptr;
```

И таким образом мы занулили (инвалидировали) указатель. Конструктор от указателя нужен, чтобы сработала строчка 2 в листинге выше. Но я не хочу создавать `AnyPtr` от указателя на `void*`, а потому мне нужно отключить этот конструктор. Это делается с помощью `concept`-а, который я назвал `NonVoidClass`. По сути этот концепт лишь проверяет ялвяется ли `T` типом `void` или нет. И если является, то этот конструктор не компилируется, а если не является, то можно создавать такой конструктор. Если же класс `T` не предоставляет нужные методы `f` и `g`, то в этом конструкторе в момент обращения к таблице `&Bind<T>::vtable` произойдет ошибка компиляции. Можно поиграться и сделать более адекватные сообщения об ошибках, но я стараюсь делать минимальные примеры.

Теперь напишем имплементацию оставшихся методов:

```
  operator bool() const {
    return ptr_ != nullptr && vtable_->f_ != nullptr && vtable_->g_ != nullptr;
  }

  void f() const {
    assert(*this);
    vtable_->f_(ptr_);
  }
  void g() {
    assert(*this);
    vtable_->g_(ptr_);
  }

  void clear() {
    vtable_ = &Bind<void>::vtable;
    ptr_ = nullptr;
  }
```

Оператор `operator bool() const` нужен, чтобы проверять ссылается ли указатель на кого-то или нет.[26] Метод `clear` просто удобный способ очистки указателя. И методы `f` и `g` являются самым важным элементом публичного интерфейса – это те методы, которые позволяют вызвать нужные методы `f` и `g` на объекте по адресу `ptr_`. Как видите, это происходит через вызов метода по указателю из текущей таблицы. Теперь можно писать такой код

```
A a;
AnyPtr ptr = &a;
ptr.f();
ptr.g();
B b;
ptr = &b;
ptr.f();
ptr.g();
```

Оператор `operator->` Все бы хорошо, но только мы бы хотели у указателя иметь возможность звать методы по оператору `operator->()`. Это можно сделать с помощью простого адаптера поверх `AnyPtr`. Вот простое неинтрузивное решение, то есть без изменения `AnyPtr`. Мы сделаем старый `AnyPtr` интерфейсом, который надо вернуть по оператору `operator->` из нового `AnyPtr2`:

```
class AnyPtr2 : AnyPtr {
public:
  using AnyPtr::operator bool;

  AnyPtr* operator->() {
    return this;
  }
  const AnyPtr* operator->() const {
    return this;
  }
};
```

Теперь можно писать код следующим образом:

```
struct A {
  void f() const;
  void g();
};

struct B {
  void f() const;
  void g();
};

int main() {
  A a;
```

[26] Для имплементации достаточно было бы проверить, то `ptr_ != nullptr`, но я хотел подчеркнуть, что мне важно, чтобы все указатели были не нулевыми. В реальном коде, можно оставить лишь одну проверку для `ptr_` и сделать инвариантом, что для ненулевого `ptr_` указатели на обе функции тоже не нулевые.

```
  B b;
  AnyPtr2 p = &a;
  p->f();
  p = &b;
  p->g();
  return 0;
}
```

Есть популярное решение использующее UB. Давайте я его продемонстрирую.

```
class AnyPtr {
public:
  Interface* operator->() {
    return static_cast<Interface*>(this);  // UB
  }
  const Interface* operator->() const {
    return static_cast<const Interface*>(this); // UB
  }
protected:
  void f() const;
  void g();
};

class Interface : AnyPtr {
public:
  using AnyPtr::f;
  using AnyPtr::g;
};
```

В этом решении мы не делаем адаптер вокруг `AnyPtr`, а добавляем наследника и в операторе `operator->` кастуем `this` к производному классу. Технически это скорее всего будет работать на всех компиляторах, но фактически это UB. Причина следующая, вы в начале кастуете тип `AnyPtr` к производному классу `Interface`. Это еще пока не UB. Но если вы после этого обратитесь к любому методу из производного класса, то это будет UB, потому что нельзя обращаться по адресу к объекту, которого там нет. По адресу `this` лежит объект типа `AnyPtr`, а вызов метода через класс `Interface`, принимает аргумент `this` типа `Interface`. Вот это уже делать нельзя.

11.4.4 Общее шаблонное решение

В предыдущих разделах мы построили класс `AnyPtr`, в который можно складывать указатель на любой объект, у которого можно вызывать два метода `f` и `g` без аргументов и без возвращаемого значения. Однако, мы бы хотели теперь настроить наш тип `AnyPtr` так, чтобы можно было в него передавать любой интерфейс. Основная проблема в автоматизации этого процесса заключается в том, что нам надо научиться передавать имена методов. Но к сожалению в C++ нет способа передавать имя метода в виде параметра. Это возможно с помощью макросов (как кодогенерация), но не возможно методами языка. В идеале мы бы хотели следующее решение

```
class Interface {
public:
  virtual void f() const = 0;
  virtual void g() = 0;
};

int main() {
  A a;
  AnyPtr<Interface> ptr = &a;
  ptr->f();
  ptr->g();
}
```

11.5 Стирающие типы

Стирающим типом принято называть тип, который может хранить внутри себя объект любого типа, если у него в интерфейсе есть определенные методы с заданной сигнатурой. Стирающий тип в отличии от стирающей ссылки (или указателя) владеет объектом, а не просто ссылается на него. В языке сейчас есть небольшая поддержка стирающих типов. Вот что нам доступно

1. `std::any` – это стирающий тип с пустым интерфейсом. То есть он умеет в себе хранить объект любого типа. При этом корректно работают операции копирования и перемещения и уничтожения. Но положив туда данные вы никак не можете ими воспользоваться. Точнее вам надо знать что там лежит и после этого вы можете откастовать `std::any` к нужному типу и достать соответствующий объект.

2. `std::function` – это стирающий тип, который умеет хранить в себе любой объект, для которого определен `operator()` с нужной сигнатурой. В отличие от указателя на функцию, внутри `std::function` можно хранить функторы содержащие данные.

3. Указатель на функцию. Так как функции существуют во время выполнения программы и они их код менять нельзя при исполнении. То указатель на функцию имеет value semantics и ведет себя как будто переменная, в которую вы складываете функцию с указанной сигнатурой.

4. `std::optional` – это не совсем полноценный стирающий тип. Он позволяет положить в него любой объект принадлежащий заранее указанному списку типов. И после этого он предоставляет возможность применения операций к объекту, не вынимая его, с помощью visitor паттерна.

11.5.1 Ожидаемое поведение

Давайте опишем, что бы мы хотели иметь. Прежде всего мы хотим описать требуемый интерфейс как-то так:

```
1 struct Interface {
2   virtual void f() const = 0;
3   virtual int g(const std::string&) const = 0;
4 };
```

Теперь предположим, что у нас есть два класса, которые поддерживают этот интерфейс

```
struct A {
  void f() const;
  int g(const std::string&) const;
};
```

```
struct B {
  void f() const;
  int g(const std::string&) const;
};
```

Мы хотим уметь хранить объекты обоих классов `A` и `B` в одной и той же переменной и вызывать методы `f` и `g` не зная, какой именно там лежит объект. Ожидаемое поведение будет таким:

```
1 int main() {
2   Any<Interface> x;
3   x = A();
4   x->f();
5   x = B();
6   std::cout << x->g("123") << std::endl;
7   return 0;
8 }
```

Есть два способа имплементировать стирающие типы:

1. Ручной менеджмент таблицами виртуальных функций

2. Использование наследования с виртуальными функциями с шаблонными адаптерами

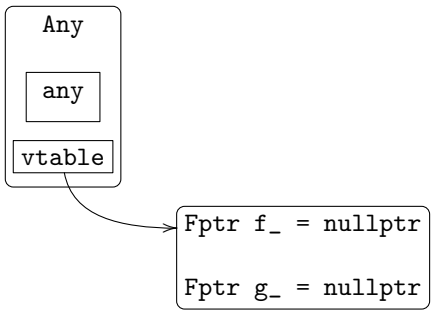
Первый способ по сути повторяет идеи описанные в разделе 11.4, где вместо указателя мы будем использовать произвольный класс, а вместо `void*` – `std::any`. Потому на `std::any` можно смотреть как на владеющий аналог `void*`.

11.5.2 Идея имплементации

Давайте создадим класс `Any`, который умеет хранить любой объект, у которого доступны два метода

- `void f() const`
- `void g(const std::string&) const`

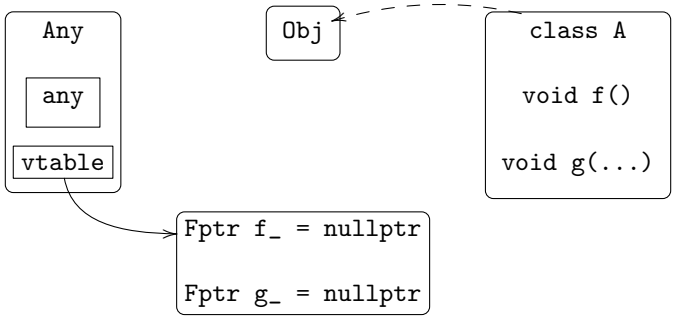
Тогда структуру `Any` можно графически изобразить так:



Объект `Any` будет содержать два поля:

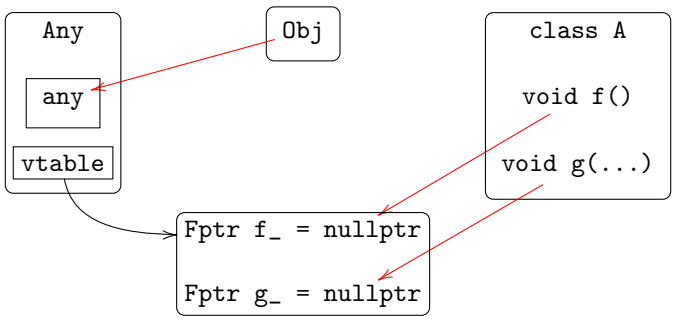
1. Переменная типа `std::any`, содержащая данный объект.
2. Указатель `vtable` на таблицу функций, которые мы будем у него вызывать.

Теперь если у нас есть объект `Obj` класса `A`, который содержит нужные методы



То при присваивании объекта `Obj` объекту класса `Any` надо сделать две вещи:

1. Положить объект `Obj` или его копию внутрь `std::any` поля внутри `Any`.
2. Положить в таблицу `vtable` указатели на методы `f` и `g` из класса `A`, которому принадлежит объект `Obj`.



Технически, так как методы `f` и `g` зависят от класса `A` и для всех объектов класса `A` они будут одинаковые, то нет необходимости для каждого объекта хранить на куче свою таблицу. Вместо этого для класса `A` заведем одну статическую таблицу и будем хранить указатель на нее.

11.5.3 Имплементация

Здесь мы по сути повторяем то, как имплементирован механизм с виртуальными функциями. Однако, объект класса будет приниматься не по `void*`, а по ссылке на `std::any`.

Виртуальные таблицы В начале надо заменить методы на глобальные функции, чтобы у нас не было зависимости от типа `A`. Делается это так:

```
struct A {
  void f() const;
  void g(const std::string&) const;
};
```

```
void f_(const A* self);
void g_(const A* self, const std::string&);
```

Теперь заведем структуру, в которой будут лежать указатели на данные функции:

```
1  struct VTable {
2    using FSignature = void(const std::any&);
3    using GSignature = void(const std::any&, const std::string&);
4
5    FSignature* f_ = nullptr;
6    GSignature* g_ = nullptr;
7  };
```

Обратите внимание, что настоящий тип функции `f_` есть `void(const A*)`. Однако, в таблице будет лежать функция с сигнатурой `void(const std::any&)`. А значит, нам надо будет сделать несколько вспомогательных действий. Давайте заведем шаблонный класс, который будет хранить статическую таблицу.

```
1  template<class T>
2  struct Bind {
3  static void f_(const std::any& self) {
4    std::any_cast<const T&>(self).f();
5  }
6  static void g_(const std::any& self, const std::string& str) {
7    std::any_cast<T&>(self).g(str);
8  }
9
10 static constexpr VTable vtable{f_, g_};
11 };
12
13 template<>
14 struct Bind<void> {
15
16 static constexpr VTable vtable{nullptr, nullptr};
17 };
```

Здесь для каждого типа `T` мы можем создать структуру `Bind<T>`.[27] Внутри этой структуры мы биндим настоящие методы из класса `T` (который будет `A` в нашем примере выше). Этот бинд делается в строках 3-5 для метода `T::f()` и в строках 6-8 для метода `T::g(...)`. После чего мы создаем статическую переменную `vtable` внутри этой структуры, которая будет хранить виртуальную таблицу на забинденные методы. Таким образом `Bind<T>::vtable` – это будет таблица с методами для указателя на объект типа `T`. Обратите внимание, что мы принимаем объект `self` в формате `const std::any&`, потому что внутри `Any` он хранится внутри поля типа `std::any`. Но так как таблица с методами и указатель на объект согласованы, мы знаем, что всегда можно кастовать с помощью `any_cast` переменную `self` в тип `T&` или `const T&`. Таким образом мы получаем доступ к настоящим методам `f` и `g` из класса `T`. И если в классе `T` нет возможности вызвать эти методы как в строках 4 и 7, то произойдет ошибка компиляции. Это хорошо, потому что это означает, что мы пытаемся использовать класс `T` не поддерживающий нужные методы и компилятор нашел нашу ошибку. После общего определения идет специализация для таблицы для типа `void`, который используется как sentinel и означает, что эта таблица ни на что не забиндилась.

Имплементация `Any` Теперь когда мы определились с тем, как устроен вспомогательный класс для работы с таблицами функций, давайте опишем как будет выглядеть основной класс `Any`. В начале опишем его методы и данные, а потом уже напишем имплементацию всех методов:

```
1  class Any {
2  public:
3    Any();
```

[27] Название пожалуй не лучшее, но я не знаю как его еще назвать.

```
  template<class T>
  Any(T&& obj);

  Any(const Any& other);
  Any& operator=(const Any& other);

  Any(Any&& other) noexcept;
  Any& operator=(Any&& other) noexcept;
  ~Any();

  bool has_value() const;

  void f() const;
  void g(const std::string&) const;

private:
  void clear();

  const VTable* vtable_ = &Bind<void>::vtable;
  std::any obj_;
};
```

Как мы видим в классе лежат только два поля:

1. `vtable_` указывает на таблицу функций. По умолчанию ссылается на пустую таблицу функций определенную для типа `viod`.

2. `obj_` содержит объект, у которого мы будем вызывать нужные методы. По умолчанию ничего не содержит, а потому мы поддерживаем метод проверки лежит ли что-либо внутри `Any`.

В начале разберемся с вопросами обеспечения правильной семантики для `Any`. Поле типа `std::any` уже обладает нужным поведением. Проблема есть с `VTable*`. Оно конечно обладает value semantics, но методы перемещения надо прописать вручную, потому что мы хотим очистить данные из объекта из которого мы проделали мув. Для этого имплементируем все конструкторы, деструктор и операторы присваивания. Я в начале сгруппирую дефолтные конструкторы, а уже после них те, что требуют ручной имплементации.

```
  Any() = default;
  Any(const Any& other) = default;
  Any& operator=(const Any& other) = default;
  ~Any() = default;

  Any(Any&& other) noexcept
      : vtable_(std::exchange(other.vtable_, &Bind<void>::vtable)),
        obj_(std::exchange(other.obj_, std::any())) {
  }

  Any& operator=(Any&& other) noexcept {
    Any tmp = std::move(other);
    std::swap(vtable_, tmp.vtable_);
    std::swap(ptr_, tmp.ptr_);
    return *this;
  }

  template<class T>
  Any(T&& obj) : vtable_(&Bind<T>::vtable), obj_(std::forward<T>(obj)) {
  }
```

Дефолтный конструктор настраивается путем инициализации полей класса. Как я писал выше деструктор, конструктор копирования и копирующее присваивание подойдут дефолтными. Мувающие операции вынимают данные из предыдущего `Any`. То есть я хочу достичь поведения

```
A a;
Any obj1 = a;
Any obj2 = std::move(obj1);
Any obj3;
obj3 = std::move(obj2);
```

Я хочу, чтобы в строке 3 объект `obj1` передавал свое содержимое внутрь `obj2`. Аналогично в строке 5 хочу, чтобы объект `obj2` передавал свое содержимое внутрь `obj3`. А для этого нужно сделать очистку данных в

старом объекте в мувающем конструкторе и операторе присваивания. Тут используются стандартные имплементации, которые обсуждались в разделе 6.3.5.

И остался последний конструктор – от произвольного типа объект, чтобы мы могли написать

```
Any x = A();
Any y = B();
```

При этом важное замечание, если мы напишем

```
Any x;
Any y = x;
```

то сработает копирующий конструктор, который мы явно прописали. В этом случае компилятор всегда отдает приоритет нешаблонному методу по сравнению с шаблонным. Если бы мы не определили явно копирующий конструктор через `default` у нас были бы проблемы в этом случае.

Теперь напишем имплементацию оставшихся методов:

```
  bool has_value() const {
    return obj_.has_value() && vtable_->f_ != nullptr && vtable_->g_ != nullptr;
  }

  void f() const {
    vtable_->f_(obj_);
  }

  void g(const std::string& str) const {
    vtable_->g_(obj_, str);
  }
```

Метод `has_value` нужен, чтобы проверять лежит ли что-то сейчас внутри `Any` или нет.[28] И методы `f` и `g` являются самым важным элементом публичного интерфейса – это те методы, которые позволяют вызвать нужные методы `f` и `g` на объекте хранимом внутри `obj_`. Как видите, это происходит через вызов метода по указателю из текущей таблицы. Теперь можно писать такой код

```
A a;
Any obj = a;
obj.f();
obj.g("abc");
B b;
obj = b;
obj.f();
obj.g("123");
```

11.5.4 Общее шаблонное решение

11.5.5 Идея ленивого решения

Выше я описал имплементацию для стирающих типов, не использующую виртуальные функции и наследование. По сути мы руками делаем работу, которую за нас может сделать язык и компилятор. Давайте я опишу здесь альтернативную имплементацию, которая не требует руками возиться с таблицами виртуальных функций.

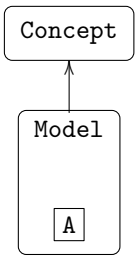
Давайте начнем со следующего вопроса. Пусть у нас задан интерфейс и класс, который удовлетворяет этому интерфейсу, но не унаследованный от него:

```
struct Concept {
  virtual ~Concept() = default;
  virtual void f() const = 0;
  virtual void g(const std::string&) const = 0;
};
```

```
class A {
public:
  void f() const;
  void g(const std::string&) const;
};
```

[28] Для имплементации достаточно было бы проверить, то `obj_.has_value()`, но я хотел подчеркнуть, что мне важно, чтобы все указатели были не нулевыми. В реальном коде, можно оставить лишь одну проверку для `obj_` потому что у нас есть инвариант, что для не пустого `obj_` указатели на обе функции тоже не нулевые и корректные.

Вопрос, как присоединить этот класс к интерфейсу? Для этого можно воспользоваться адаптером между интерфейсом и нужным классом. В качестве адаптера будет использоваться вспомогательный класс `Model`, который будет унаследован от интерфейса `Concept`, будет хранить в себе объект класса `A`, и его задача будет организовать проброс вызовов для виртуальных методов.



В коде это выглядит так

```
struct Concept {
  virtual ~Concept() = default;
  virtual void f() const = 0;
  virtual void g(const std::string&) const = 0;
};

template<class T>
class Model : public Concept {
public:
  Model(const T& data) : data_(data) {}
  Model(T&& data) : data_(std::move(data)) {}

  void f() const final {
    data_.f();
  }

  void g(const std::string& str) const final {
    data_.g(str);
  }
private:
  T data_;
};
```

Теперь мы можем следующим образом хранить любые данные в единой переменно

```
int main() {
  A a;
  std::unique_ptr<Concept> model = std::make_unique<Model<A>>(std::move(a));
  model->f();
  model->g("abc");
  return 0;
}
```

11.5.6 Имплементация ленивого решения

Теперь опишем, как собрать класс `Any`, используя идею из предыдущего раздела. Опишем его интерфейс

```
class Any {
public:
  template<class T>
  Any(T&& object);

  Any() = default;
  Any(const Any&);
  Any(Any&&) = default;
  Any& operator=(const Any&);
  Any& operator=(Any&&) = default;
  ~Any() = default;

  const Concept* operator->() const;
  Concept* operator->();

```

```
  operator bool() const;
private:
  std::unique_ptr<Concept> model_;
};
```

Теперь напишем, как эти методы имплементируются. В начале методы публичного интерфейса

```
const Concept* operator->() const {
  return model_.get();
}
Concept* operator->() {
  return model_.get();
}
operator bool() const {
  return model_.operator bool();
}
```

Обратите внимание, что оператор `operator->` пробрасывает константность. Это связано с тем, что мы хотим, чтобы константность `Any` означало константность данных внутри.

Теперь конструкторы. В начале конструктор для помещения объекта внутрь `Any`

```
template<class T>
Any(T&& object) : model_(std::make_unique<Model<T>>(std::forward<T>(object))) {}
```

Перемещающие методы и деструктор можно оставить по умолчанию. А вот копирование придется писать руками, так как по умолчанию `std::unique_ptr` не имеет операции копирования. Здесь придется виртуализовать копирование. То есть добавить в интерфейс `Concept` метод копирования. Давайте начнем с изменения класса `Concept`.

```
class Concept {
public:
  virtual void f() const = 0;
  virtual void g(const std::string&) const = 0;
protected:
  ~Concept() = default;
private:
  virtual std::unique_ptr<Concept> make_copy_() const = 0;
  friend class Any;
};
```

Обратите внимание, что у `Any` оператор `operator->` возвращает указатель на `Concept`. А мы не хотим, чтобы пользователь мог воспользоваться служебными методами. Поэтому все служебные методы такие как копирование и деструктор, мы прячем. Но с другой стороны, `Any` должен быть способен позвать метод `make_copy_`. А потому его надо сделать `friend` классом. Это хороший пример, когда нужно ключевое слово `friend`. Дело в том, что два класса тесно взаимодействуют друг с другом и хотя поддерживать один инвариант – данное внутри `model_` переменной. При этом `Any` пользуется этим доступом только в копирующем конструкторе и нигде больше. Теперь опишем как изменится модель

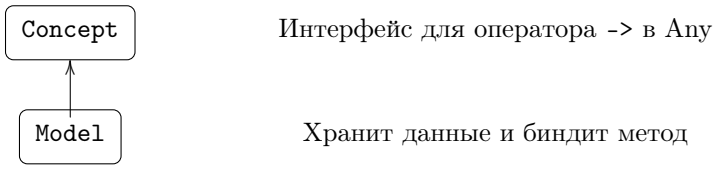
```
template<class T>
class Model : public Concept {
  std::unique_ptr<Concept> make_copy_() const final {
    return std::make_unique<Model>(data_);
  }
public:
  Model(consst T& data) : data_(data) {}
  Model(T&& data) : data_(std::move(data)) {}
  void f() const final {
    data_.f();
  }
  void g(const std::string& str) const final {
    data_.g(str);
  }
private:
  T data_;
};
```

Теперь копирующие операции для `Any` выглядят так

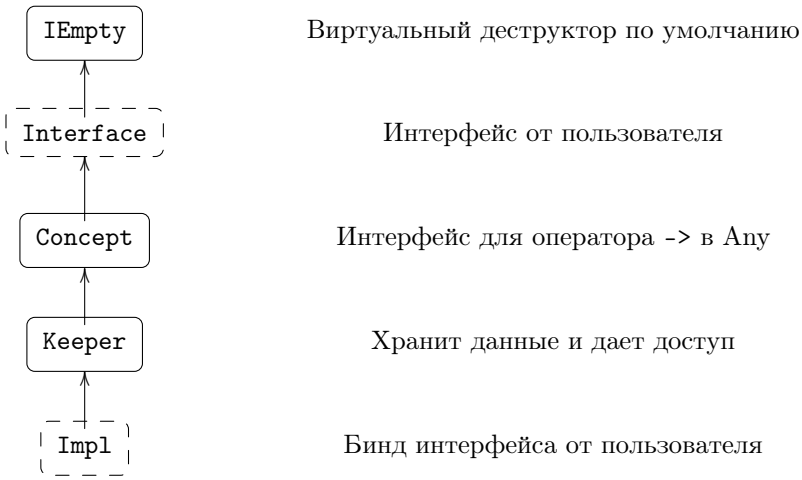
```
Any(const Any& other)
  : model_((other ? other->make_copy_() : nullptr)) {}
Any& operator=(const Any& other) {
  return *this = Any(other);
}
```

11.5.7 Общая шаблонная версия ленивого решения

Теперь давайте посмотрим, как сделать предыдущее решение общим, чтобы оно работало для любого интерфейса, а не только для указанных двух методов с указанными именами и сигнатурами. Давайте поймем, что у нас есть сейчас:



Теперь мы распилим этот класс на следующие компоненты



На этой диаграмме пунктиром отмечены классы, которые мы будем настраивать. А сплошными отмечены классы, которые надо будет написать. Давайте опишем роли этих классов

1. `IEmpty`. Задача этого класса – избавить пользователя от необходимости самостоятельно указывать в интерфейсе виртуальный деструктор. Это позволит не обременять пользователя лишней работой и избежать ошибок в программе, при отсутствии виртуального деструктора.

2. `Interface`. Это шаблон, который предоставляет пользователь, описывающий интерфейс нужного стирающего типа.

3. `Concept`. Это интерфейс, который возвращается по оператору `operator->` в классе `Any`.

4. `Keeper`. Данный класс хранит сам объект и дает доступ к нему.

5. `Impl`. Этот шаблон предоставляет пользователь. Его задача пробросить виртуальные вызовы и назначить их на нужные методы внутри хранимого объекта.

Вот как выглядит имплементация этих классов в коде.

```
template<template<class> class IAny, template<class> class Impl>
class Any;

class IEmpty {
protected:
  virtual ~IEmpty() = default;
};

class Concept : public IAny<IEmpty> {
  std::unique_ptr<Concept> make_copy_() const = 0;
  friend class Any;
};

template<class T>
class Keeper : public Concept {
  std::unique_ptr<Concept> make_copy_() const final {
    return std::make_unique<Impl<Keeper<T>>>(data_);
  }
public:
  Keeper(const T& data) : data_(data) {}
  Keeper(T&& data) : data_(std::move(data)) {}
protected:
  T& object() { return data_; }
  const T& object() const { return data_; }
private:
  T data_;
};
```

Метод `object` нужен для того, чтобы можно было получить доступ к данным в `Impl`, когда мы будем биндить виртуальные методы на методы класса. Теперь в класс `Any` нужно внести следующие изменения

```
template<template<class> class IAny, template<class> class Impl>
class Any {
public:
  template<class T>
  Any(T&& data)
    : model_(std::make_unique<Impl<Keeper<T>>>(std::forward<T>(data))) {}
  /* default methods */
  Concept* operator->();
  const Concept* operator->() const;
  operator bool() const;
private:
  std::unique_ptr<Concept> model_;
};
```

Теперь пользователь должен предоставить два класса

```
template<class IEmpty>
class IAny : public IEmpty {
public:
  virtual void f() const = 0;
  virtual void g(const std::string&) const = 0;
};

template<class Keeper>
class Impl : public Keeper {
public:
  using Keeper::Keeper;
  void f() const final {
    Keeper::object().f();
  }
  void g(const std::string& str) const final {
    Keeper::object.g(str);
  }
};
```

Тут важно обратить внимание, что надо в классе `Impl` пробросить конструкторы из класса `Keeper`, что делается в строчке 11. Теперь можно пользоваться этим следующим образом

```
using AnyFG = Any<IAny, Impl>;
```

```
2
3  struct A {
4    void f() const {}
5    void g(const std::string&) const {}
6  };
7
8  struct B {
9    void f() const {}
10   void g(const std::string&) const {}
11 };
12
13 int main() {
14   AnyFG x = A();
15   x->f();
16   x->g("abc");
17   x = B();
18   x->f();
19   x->g("abc");
20   return 0;
21 }
```

12 Межобъектная коммуникация

12.1 Зачем?

Стандартный код Давайте начнем с двух примеров того, как принято писать код

```
int main() {
  Data1 input = read();
  Data2 result = apply_algorithm(input);
  print(result);
  return 0;
}
```

```
struct A {
  void process(const char* command) {
    parse(command); // set arg
    Data1 data = read(arg.file1);
    Data2 result = apply(data, arg.algorithm);
    write(result, arg.file2);
  }
  Command arg;
};
```

Слева пример стандартного консольного приложения. Оно состоит по сути из трех строчек

1. Считать данные и положить в `input`.
2. Применить к `input` алгоритм и получить `result`.
3. Вывести `result` на печать.

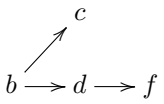
Вся работа на каждом шаге делегируется трем различным функциям. Я хочу обратить внимание на то, кто занимается передачей данных между тремя функциями. За передачу данных и временные переменные отвечает сама функция `main`. В примере справа происходит более или менее то же самое. У класса `A` есть метод `process`, который вызывает такие же три стадии:

1. Считать данные и положить в `data`.
2. Применить к `data` алгоритм и получить `result`.
3. Вывести `result` на печать.

Отличие лишь в том, что теперь у нас есть состояние, хранящееся внутри класса `A`, которое может на что-то неявно повлиять. И в этом случае за передачу данных между методами отвечает по сути класс `A`. Если быть точнее, то надо сказать метод `process`.

Давайте отметим, что это стандартный способ писать код. Мы так привыкли писать функции и классы. Функции внутри себя передают данные между отдельными функциями. Классы сами передают свои внутренние данные между разными методами. В этом случае передача данных осуществляется внешним объектом, который видит всех участников и может осуществить передачу.

Важный пример Чтобы понять зачем нам нужно что-то еще, стоит рассмотреть один важный пример. Давайте представим, что мы хотим логически иметь pipeline из объектов. Чтобы объекты могли передавать друг другу данные вдоль пайплайна. Изобразим схематично такой pipeline:

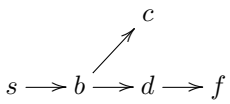


Здесь b, c, d и f – это объекты каких-то классов, а стрелки означают поток данных. Давайте попробуем реализовать это в рамках классического подхода. Чтобы мы могли передавать данные между указанными четырьмя объектами, нужно их всех поместить либо внутрь функции, либо внутрь класса. Давайте поместим внутрь класса.

```
class A {
public:
  void process(int x) {
    b.process(x);
    signal_BC();
    signal_BD();
    signal_DF();
  }

private:
  void signal_BC() {
    int x = b.get();
    c.process(x);
  }
  void signal_BD() {
    int x = b.get();
    d.process(x);
  }
  void signal_DF() {
    int x = d.get();
    f.process(x);
  }

  B b;
  C c;
  D d;
  F f;
};
```

Мы положили в класс `A` все четыре переменные. Для каждой стрелки мы завели соответствующую функцию, которая берет данное из вершины source и отдает эти данные на обработку в вершину target. И есть еще общая функция `process`, которая проталкивает данные сквозь весь пайплайн.

Вроде бы все хорошо и работает как надо. Но давайте попробуем внести изменение и добавить еще одну вершину s следующим образом:



Теперь нужно внести следующие изменения

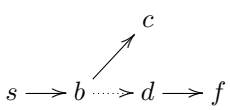
```
class A {
public:
  void process_B(int x) {
    b.process(x);
    signal_BC();
    signal_BD();
    signal_DF();
  }

  void process_S(int x) {
    s.process(x);
    signal_SB();
    signal_BC();
    signal_BD();
    signal_DF();
  }
private:
  void signal_SB() {
    int x = s.get();
    b.process(x);
  }
  void signal_BC() {
    int x = b.get();
    b.process(x);
  }

  void signal_BD() {
    int x = b.get();
    d.process(x);
  }

  void signal_DF() {
    int x = d.get();
    f.process(x);
  }

  S s;
  B b;
  C c;
  D d;
  F f;
};
```

Давайте отметим сразу несколько вещей:

1. Добавить новую переменную `s` и написать метод `signal_SB` мы вынуждены. Так как это новая информация для пайплайна.

2. Метод `process_S` подозрительно дублирует методы из `process_B`, что плохо. Можно переиспользовать `process_B`, но тогда мы не будем использовать `signal_SB`. А это означает, что мы будем имплементацию `signal_SB` вставлять внутрь `process_S`. Это еще хуже, тогда у нас действия отвечающие за ребро $s \to b$ будут размазаны по коду и не будет единого источника правды.

То что мы проделали все еще не страшно. Давайте попробуем удалить ребро $b \to d$ на диаграмме

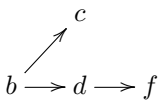


```
class A {
public:
  void process_B(int x) {
    b.process(x);
    signal_BC();
    // signal_BD();
    signal_DF();
  }
  void process_S(int x) {
    s.process(x);
    signal_SB();
    signal_BC();
    // signal_BD();
    signal_DF();
  }
```

```
private:
  void signal_SB() {
  }
  void signal_BC() {
  }
  void signal_BD() {
  }
  void signal_DF() {
  }
  S s;
  B b;
  C c;
  D d;
  F f;
};
```

И вот тут видна еще одна проблема – наши действия не локальны. Код соответствующий ребру находится в двух функциях. А потому удаление ребра $b \to d$ заставляет нам вносить изменения в несколько разных функций. Это тоже очень плохо.

Что мы хотим Мы бы хотели строить системы по схемам из кусочков как на схеме ниже.



При этом мы хотим, чтобы логическая структура диаграммы максимально точно соответствовала коду. Это значит, что простейшая процедура удаления одного ребра должна быть локальной и выглядеть как удаление одной и только одной строчки вне зависимости от размеров и сложности диаграммы.

Проблемы с обычным подходом:

1. Код основан на соглашениях. То есть логически у нас конечно пайплайн, но ментальная модель не соответствует модели в коде. Как пример логически единственное ребро $b \to d$ в коде разбросано сразу в нескольких местах. И логически простая операция удаление ребра или добавление ребра влечет изменение сразу нескольких мест в коде.

2. Статические соединения. Это значит, что все соединения известны на момент компиляции. Это не такая страшная проблема в таком подходе, потому что можно было бы заводить указатели на функции и работать с ними.

3. Подобъекты не могут общаться. Если вдруг внутри `b` есть объект, который бы вы хотели соединить с кем-то, то вам придется пробрасывать его соединение сквозь интерфейс `b`. Либо вываливать наружу кишки `b` и отказываться от инкапсюляции.

4. Нельзя передавать информацию в середине методов. Вот это самая страшная проблема из всех. Так как вы работаете с объектом `b` снаружи, вы не можете оповестить кого-то в середине процесса. Потому что вам не доступны внутренности `b` из-за инкапсюляции. А подобное может понадобится. Например если вы визуализируете структуру данных и хотите сделать анимацию на вставку элемента. То хорошо бы посылать сигнал не в начале и конце операции, а в промежуточные моменты, чтобы отобразить промежуточное состояние структуры.

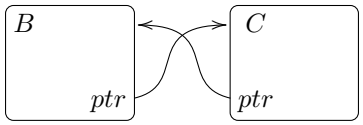
5. Масштабируемость. На примерах выше, мы увидели как много действий приходится делать при малейших изменениях в системе. Причем, чем больше узлов и соединений, чем больше разных путей, через которые могут пройти сигналы, тем сложнее вносить любые изменения. В целом это связано с тем, что функция – это целый путь в пайплайне, а не ребро.

6. Повторяемость кода. Эту проблему мы тоже наблюдали выше. Именно она влечет проблемы с масштабируемостью. Но важнее, что у вас теряется единый источник правды. В этом случае одно и то же действие может быть продублировано в нескольких местах. Что практически не возможно поддерживать.

Если вам нужны подобные пайплайны, то по-хорошему вам нужны примитивы для их построения. Ниже я собираюсь обсудить один из подобных примитивов – observer pattern.

12.2 Прямое взаимодействие объектов

Если два объекта классов `B` и `C` хотят коммуницировать и передавать друг другу данные, то самый простой способ организовать такую связь – хранить указатели друг на друга



Однако у такого решения есть серьезные проблемы как идейные так и технические.

Идейные проблемы

1. В таком подходе `B` зависит от имплементации `C`. Объект `B` должен знать названия методов из объекта `C` и дергать их по назначению. Любые изменения в `C` могут повлечь изменения в `B` и наоборот. По сути в этом случае мы распилили одну структуру данных на два класса, которые должны поддерживать общий инвариант.
2. В таком подходе `B` должен знать об объекте `C`. Мы должны инклудить зависимости объекта `C` в имплементацию объекта `B` и наоборот. Это может сильно ломать и нарушать логику разделения программы на части.

Технические проблемы

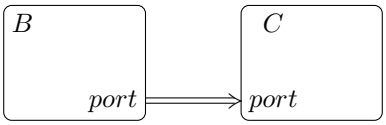
1. Проблема указателя в том, что он может быть нулевым. Это состояние надо не забыть.
2. Объект мог переехать по другому адресу. Действительно, обычные указатели не дают никаких гарантий, есть ли по данному адресу нужный нам объект или нет. Например указатель может ссылаться на мертвый объект.
3. Ну и самое классное – указатели в объектах `B` и `C` могут не ссылаться друг на друга. То есть указатели могут быть не в корректном рассинхронизированном состоянии.

12.3 Observer Pattern

Паттерн описываемый в этой главе по сути призван решить все проблемы прямого взаимодействия и сделать его безопасным.

12.3.1 Общая идея

Мы хотим обеспечить безопасное одностороннее взаимодействие между объектами классов `B` и `C`. Логически это выглядит так:



Идея в том, чтобы завести два вида порта:

1. Выходной порт: observable.
2. Входной порт: observer.

Теперь объект класса `B` будет содержать выходной порт – observable. А объект класса `C` будет содержать в себе входной порт – observer. Отличие этого подхода будет в том, что у этих портов есть некоторый протокол взаимодействия, который гарантирует, что они всегда находятся в корректном состоянии. Но для этого оба порта должны иметь возможность двустороннего общения. Кроме того в этом случае оба порта разделяют общий инвариант, который отражает корректное синхронизированное состояние связи между портами. Так же отметим, что к одной observable можно присоединить множество observer-ов. Но каждый observer прослушивает максимум одну observable.

У каждого порта так же есть тип данных, которые он умеет отправлять. Соединить можно только порты с одинаковым типом данных. Логически observable при поступлении новых данных оповещает всех observer-ов, который на данный момент подписаны на него. Теперь опишем, какой функционал предоставляют нам оба порта.

Observable

1. Конструктор. В конструкторе мы должны указать источник данных для порта. Это то место, где биндятся данные на порт.

2. Метод `subscribe`, который позволяет подписать observer порт. При этом очень важная деталь: сразу после подписания, observable отправляет данные observer-у. То есть observer всегда имеет самые последние данные, которые только есть в observable. Не бывает такого, что ему надо каким-то образом синхронизироваться или он не получил данные. Это push система, где данные толкаются при их появлении получателю.

   Так же стоит отметить, что у данного метода может быть много разных поведений. Например, что делать, если мы пытаемся подписать observer-а, который уже на кого-то подписан? Как правильно поступить? Игнорировать его или отписать и переподписать на себя? Это все вариации поведения, которые можно настраивать и получать разные виды взаимодействия.

3. Метод `notify`, который оповещает всех observer-ов. Все оповещения происходят через этот метод. Технически есть еще метод, который умеет оповестить только одного фиксированного observer-а, но это уже детали.

4. Observable сама менеджерит всех своих подписчиков и знает их полный список. Бывают вариации, когда observable поддерживает только одного подписчика или ограниченное количество подписчиков.

Observer

1. Конструктор. Главная задача принимающего порта – по приходу данных вызвать нужное действие над ними. бывают действия двух типов

   (a) `onSubscribe`. Это действие, которое нужно сделать с данными, когда observer только что подписался на observable и получил от него данные.

   (b) `onNotify`. Это действие, которое нужно сделать с данными, когда observer уже был в подписанном состоянии и прилетели новые данные из observable.

   Оба этих действия биндятся в конструкторе.

2. Метод `unsubscribe`, который позволяет отписаться от `observable`.

3. Метод `isSuscribed` проверки подписан ли в данный момент observable на кого-то или нет.

Кроме этого оба порта должны действовать по некоторому протоколу, который позволяет поддерживать порты в корректном состоянии. Например, если порт observable умирает, то перед смертью он отписывает всех подписавшиеся на него порты. Аналогично, если observer умирает, то он отписывается от своего observable, если таковая имеется.

12.3.2 Что мы хотим в коде

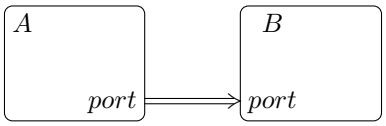
Прежде чем заниматься имплементацией, давайте в начале продемонстрируем на примерах, чего мы ожидаем от паттерна.

Простой пример В этом примере мы хотим показать, как синтаксически будет выглядеть работа с observable и observer в коде.

```
1  #include <iostream>
2  #include "Observer.h"
3
4  void print(int x) {
5    std::cout << x << std::endl;
6  }
7
8  int main() {
9    int x = 42;
10   Observable<int> out([&x] () { return x; }); // data binding
11   Observer<int> in(print, print); // actions binding
```

```
12
13    out.subscribe(&in); // print 42
14
15    x = 24;
16    out.notify();  // print 24
17    x = 42;
18    in.unsubscribe();
19    out.notify(); // does nothing
20
21    return 0;
22  }
```

Пример взаимодействия объектов Теперь давайте обсудим ситуацию соответствующую диаграмме:



Код для `A` выглядит так

```
1  class A {
2  public:
3    void subscribe(Observer<int>* obs) {
4      out_.subscribe(obs);
5    }
6
7    void set(int x) {
8      x_ = x;
9      out_.notify();
10   }
11
12 private:
13   int x_ = 0;
14   Observable<int> out_ = [&x_] () { return x_; }
15 };
```

Здесь у нас есть внутренняя переменная `x_`, которая представляет из себя данные. Кроме этого мы заводим порт `out_` для оповещения. Теперь в методе `set` мы кроме установления нового значения добавляем строчку для оповещения. И метод `subscribe` нужен для того, чтобы можно было пробросить observer порт и подписать его на наш observable порт. Обратите внимание, что порт `out_` перехватывает переменную `x_` по ссылке. Это значит, что объект класса `A` ни в коем случае нельзя перемещать в памяти, иначе биндинг данных сломается. Вот в этом месте бы очень пригодилась возможность по `out_` найти соседнее данное `x_`, а не хранить его адрес.

Код для `B` выглядит так

```
1  class B {
2  public:
3    Observer<int>* port() {
4      return &in_;
5    }
6  private:
7    void print(int x) const {
8      std::cout << x << std::endl;
9    }
10   Observer<int> in_ = Observer<int>([this] (int x) { print(x); },
11                                     [this] (int x) { print(x); });
12 };
```

Обратите внимание, что класс `B` имеет приватный метод `print`, который умеет реагировать на полученные данные. И наша задача вызвать этот метод при получении данных на входной порт `in_`. Для этого в конструкторе данные биндятся на этот метод. В данном случае мы перехватываем `this` и вызываем метод `print` у текущего объекта `this`. Раз мы запоминаем адрес хозяина, то нельзя перемещать объект класса `B` в памяти. Метод `port` нужен для проброски адреса порта, чтобы его можно было подписать на другие порты.

Теперь мы можем использовать эти классы следующим образом

```
int main() {
  A a;
  B b;

  a.subscribe(b.port());  // print 0

  a.set(42);              // print 42
  a.set(24);              // print 24

  return 0;
}
```

12.3.3 Имплементация

Обратите внимание, что ниже приведен минимальный пример, который не содержит никаких проверок. Это все сделано для читаемости. Кроме того, в блокирующем observer pattern можно передавать данные либо по ссылке, либо по копии. И тяжелые данные лучше передавать по ссылке, а легкие по копии. Это все можно настроить и добавить в паттерн, но я намерено игнорирую эту деталь. Например, сигнатура для биндинга данных может быть

```
std::function<const T&()> data_;
```

Observable Начнем с объявления observable.

```
template<class T>
class Observable {
  using Observer = Observer<T>;
  friend Observer;
public:
  template<class Tt>
  Observable(Tt&& data);

  Observable(const Observable&) = delete;
  Observalbe& operator=(const Observable&) = delete;
  Observable(Observable&&) noexcept = delete;
  Observable& operator=(Observable&&) noexcept = delete;

  ~Observable();

  void subscribe(Observer* obs);
  void notify() const;
private:
  void detach_(Observer* obs);

  std::function<T()> data_;
  std::list<Observer*> Observers_;
};
```

В коде выше я использую `std::list`, это один из самых худших вариантов контейнера, который только существует. Потому в реальности лучше использовать что-то другое, но пусть будет так. Первое на что надо обратить внимание – все операции копирования и перемещения удалены. Это значит, что observable будет иметь постоянный адрес в течение всего времени жизни. В целом это хорошее свойство, потому что как мы видели в примерах выше, когда порт складывается внутри класса, то ему приходится биндиться на данные этого класса по адресу. А для корректности такого бинда весь класс тоже должен быть не перемещаемым. Потому помещение observable в любой класс делает его адресуемым в том смысле, что его адрес в течение всей жизни не меняется.

В качестве данных внутри observable лежат всего две переменные.

1. Переменная `data_` – это функциональный объект, который умеет возвращать нужные данные. В целом общая практика такова, что лучше передавать в высокоуровневые объекты не данные, а методы или функторы. Потому мы не храним адрес данных внутри, а храним метод получения данных. Таким образом данные могут лежать где угодно и мы код observable об этом ничего не знает.

2. Переменная `Observables_` – список указателей на всех observer-ов, которые подписаны на данную observable. Так как мы храним указатели на observer-ов, то они должны быть не перемещаемыми в памяти.

Еще важная деталь – класс `Observer` объявлен `friend` для `Observable`. Это сделано потому что эти два класса имеют общий инвариант. Они хранят указатели друг на друга и должны уметь поддерживать эти указатели в корректном состоянии. Например при смерти observer должен гарантировать свое удаление из списка подписчиков. Иначе observable будет хранить не корректный адрес. Для поддержки этого инварианта, эти классы должны дергать друг у друга не безопасные методы, которые никому другому доступны быть не должны. В данном случае это метод `detach_`. Он специально помечен `_` в конце, чтобы подчеркнуть, что он не безопасный. Что он делает будет видно ниже.

Теперь перейдем к имплементации методов. Начнем с шаблонного конструктора

```
template<class Tt>
Observable(Tt&& data) : data_(std::forward<Tt>(data))
```

Здесь имеется в виду, что `data` – это функтор, который является источником данных. Теперь имплементация остальных методов

```
template<class T>
class Observable {
  using Observer = Observer<T>;
  friend Observer;
public:
  /* Constructors */
  ~Observable() {
    while(!Observers_.empty()) {
      Observers_.
        front()->unsubscribe();
    }
  }
  void notify() const {
    for (auto obs : observers_)
      obs->onNotify_(data_());
  }
  void subscribe(Observer* obs) {
    if (obs->isSubscribed())
      obs->unsubscribe();
    Observers_.push_back(obs);
    obs->setObservable_(this);
    obs->onSubscribe_(data_());
  }
private:
  void detach_(Observer* obs) {
    Observers_.remove(obs);
  }

  function<T()> data_;
  list<Observer*> Observers_;
};
```

Давайте пройдемся по методм.

1. Деструктор просто вызывает метод `unsubscribe` у всех observable из списка, пока он не пуст. Обратите внимание, что метод `unsubscribe` является публичным методом интерфейса observer-а. А потому после его вызова и observer и observable находятся в корректном состоянии и никакой дополнительной работы делать не надо. Более того, список observer-ов уменьшится на отписанный порт.

2. Метод `notify` делает в точности то, что должен – он вызывает у всех подписанных observer-ов метод `onNotify_` и сообщает ему данные, которые были забиндины через `data_`.

3. Метод `subscribe` действует чуть интереснее. Мы уже обсуждали, что тут возможно несколько поведений. И по-хорошему настраиваемое поведение должно храниться тоже в переменной функторе внутри observable. Но я для определенности выбрал следующее поведение. В начале мы проверяем подписан ли observer на кого-то сейчас, и если подписан, то тут же его отписываем. Теперь observer свободен. Мы его добавляем в конец очереди в список и так мы указываем на данного observer-а. Но у observer-а

текущий указатель не указывает ни на кого. Для восстановления инварианта, мы зовем не безопасный метод `setObservable_` и складываем туда адрес observable. После чего, мы оповещаем новый порт через `onSubscribe_` действие, которому скармливаем забиндинные данные.

4. Небезопасный метод `detach_` просто удаляет текущего observer-а из списка подписавшихся. Этот метод может сломать инвариант.

Observer Теперь опишем как выглядит объявление класса для observer-а.

```
template<class T>
class Observer {
  using Observable = Observable<T>;
  friend Observable;
public:
  template<class Tt1, class Tt2>
  Observer(Tt1&& onSubscribe, Tt2&& onNotify);
  Observer(const Observer&) = delete;
  Observer& operator=(const Observer&) = delete;
  Observer(Observer&&) noexcept = delete;
  Observer& operator=(Observer&&) noexcept = delete;
  ~Observer();

  void unsubscribe();
  bool isSubscribed() const;
private:
  void setObservable_(Observable* observable);

  Observable* Observable_ = nullptr;
  std::function<void(T)> onSubscribe_;
  std::function<void(T)> onNotify_;
};
```

Как и в случае observable все перемещающие и копирующие операции удалены. Объект становится не перемещаемым в памяти и значит все время его жизни имеет постоянный адрес. Так же `Observable` объявлен `friend` для `Observer`. Это так же связано с тем, что `Observer` должен дергать не безопасные методы для восстановления инварианта. Поля класса ясны из названия

1. `Observable_` – это адрес observable, на которую в данный момент подписан `Observer`. Состояние `nullptr` соответствует отсутствию связи.

2. `onSubscribe_` – действие, которое выполняется при подписании. Биндится в конструкторе.

3. `onNotify_` – действие, которое выполняется при оповещении. Биндится в конструкторе.

Имплементацию начнем с конструктора

```
template<class Tt1, class Tt2>
Observer(Tt1&& onSubscribe, Tt2&& onNotify)
  : onSubscribe_(std::forward<Tt1>(onSubscribe)),
    onNotify_(std::forward<Tt2>(onNotify))) {}
```

В данном случае мы просто передаем два функтора, которые будут выполнять нужные действия на указанные сигналы. Теперь остальные методы

```
template<class T>
class Observer {
  using Observable =
          Observable<T>;
  friend Observable;
public:
  /* Constructors */
  ~Observer() {
    unsubscribe();
  }
  void unsubscribe() {
    if (!isSubscribed())
      return;
    Observable_->detach_(this);
```

```
15      Observable_ = nullptr;
16    }
17    bool isSubscribed() const {
18      return Observable_;
19    }
20
21  private:
22    void setObservable_(Observable* observable) {
23      Observable_ = observable;
24    }
25
26    Observable* Observable_ = nullptr;
27    function<void(T)> onSubscribe_;
28    function<void(T)> onNotify_;
29  };
```

Давайте пройдемся по ним по всем.

1. В деструкторе, прежде чем умереть мы отписываемся от текущей observable. Метод `unsubscribe` является безопасным. Его можно звать внезависимости подписан порт в данный момент или нет. После вызова мы отсоединены от observable и потому можно спокойно умирать.

2. Метод `unsubscribe` в начале проверяет подписаны ли мы на кого-то или нет. Если нет, то просто ничего не делаем. Если же подписаны, то мы вначале зовем небезопасный метод `detach_`, который отцепляет observer от observable. Но теперь для восстановления инварианта, нужно занулить `Observable_`, что и делается в следующей строчке.

3. Метод `isSubscribe` просто проверяет адрес текущей observable.

4. Метод `setObservable_` устанавливает новое значение для переменной `Observable_`. Этот метод не безопасный и может сломать инвариант между портами. Потому он помечен `_` в конце имени.

12.3.4 Модификация observer pattern

Давайте предположим, что у меня есть очень тяжелые данные, которые могут модифицироваться частями.

```
1  BigData data;
```

Например это может быть `std::vector`, в который мы только добавляем данные. Давайте рассмотрим такой пример использования

```
1  BigData data;
2
3  Observable<BigData> port0 = [&data]() { return data; };
4  Observer<BigData> port1([](const BigData& data){ /*do something*/},
5                          [](const BigData& data){ /*do something*/});
6
7  port0.subscribe(&port1); // notify with full data
8  data.update();
9  port0.notify(); // notify with full data
```

Как мы видим, в этом случае при подписке `port1` получил огромную часть данных. После чего в строке 8 данные изменились. И в строке 9 в `port1` опять прилетели все данные целиком. И вот тут вопрос как правильно реагировать на стороне приемного порта. Если просто обновить данные и полностью – это дорогая операция. Если же не обновлять данные полностью, то на стороне observer-а должен быть вычислен `diff` между текущим состоянием и пришедшим в порт и после этого обновлена нужная часть. К сожалению такое вычисления `diff` может оказаться не менее затратной, чем полностью обновить данные. В этом случае лучше при подписании и при оповещении передавать разные данные. Но для этого нужна модифицированная версия `Observable` и `Observer`.

В начале заведем два вида данных

```
1  BigData data;
2  Updata patch;
```

где `patch` представляет из себя `diff` с предыдущим состоянием. На стороне структуры данных мы можем легко поддерживать операцию получения `diff` при любых изменениях структуры.

```
Observable<BigData, Update> port0([&data]() { return data; },
                                  [&patch]() { return patch; });
Observer<BigData, Update> port1([](const BigData& data){ /*do*/},
                                [](const Update& patch){ /*do*/});

port0.subscribe(&port1); // notify with full data
patch = data.update();
port0.notify(); // notify with the patch only
```

Теперь для создания observable нужно биндиться не на одно данное, а на два разных данных. Одно для подписки, а другое для оповещения. Это делается в первых двух строчках. Дальше идет бинд действий для обзервера. Первое действие биндится на подписку, второе на оповещение. Обратите внимание, что теперь у этих действий разный тип данных. Ниже идет пример, в котором при оповещении приходит только `diff`. По-хорошему в этом случае структура должна предоставлять возможность удобно работать с `diff`. Теперь опишем, как выглядят модифицированные версии наших портов.

Observable Обновленный класс выглядит следующим образом.

```
template<class Full, class Update>
class Observable {
  using Observer = Observer<Full, Update>;
  friend Observer;
public:
  template<class TFull, class TUpdate>
  Observable(TFull&& full, TUpdate&& update)
   : full_(std::forward<TFull>(full)),
     update_(std::forward<TUpdate>(update)) {}

  Observable(const Observable&) = delete;
  Observalbe& operator=(const Observable&) = delete;
  Observable(Observable&&) noexcept = delete;
  Observable& operator=(Observable&&) noexcept = delete;
  ~Observable();
  void subscribe(Observer* obs);
  void notify() const;
private:
  void detach_(Observer* obs);
  std::function<Full()> full_;
  std::function<Update()> update_;
  std::list<Observer*> Observers_;
};
```

Как видите мы заменили переменную `data_` на две новые `full_` и `update_` для биндинга на данные. И нужно лишь поправить шаблонные аргументы для типов данных этих биндингов. И обновить конструктор. Методы же модифицируются очевидным образом. Мы подставляем вызов `full_` внутри `onSubscribe_` и вызов `update_` внутри `onNotify_`.

```
template<class Full, class Update>
class Observable {
  using Observer =
    Observer<Full, Update>;
  friend Observer;
public:
  /* Constructors */
  ~Observable() {
    while(!Observers_.empty()) {
      Observers_.
        front()->unsubscribe();
    }
  }
  void notify() const {
    for (auto obs : observers_)
      obs->onNotify_(update_());
  }
```

```
  void subscribe(Observer* obs) {
    if (obs->isSubscribed())
      obs->unsubscribe();
    Observers_.push_back(obs);
    obs->setObservable_(this);
    obs->onSubscribe_(full_());
  }

private:
  void detach_(Observer* obs) {
    Observers_.remove(obs);
  }

  function<Full()> full_;
  function<Update()> update_;
  list<Observer*> Observers_;
};
```

Observer Обновленный класс выглядит следующим образом.

```
template<class Full, class Update>
class Observer {
  using Observable = Observable<Full, Update>;
  friend Observable;
public:
  template<class TFull, class TUpdate>
  Observer(TFull&& onSubscribe, TUpdate&& onNotify)
    : onSubscribe_(std::forward<TFull>(onSubscribe)),
      onNotify_(std::forward<TUpdate>(onNotify)) {}
  Observer(const Observer&) = delete;
  Observer& operator=(const Observer&) = delete;
  Observer(Observer&&) noexcept = delete;
  Observer& operator=(Observer&&) noexcept = delete;
  ~Observer();
  void unsubscribe();
  bool isSubscribed() const;
private:
  void setObservable_(Observable* observable);
  Observable* Observable_ = nullptr;
  std::function<void(Full)> onSubscribe_;
  std::function<void(Update)> onNotify_;
};
```

Здесь ровно такие же изменения как и в `Observable`. Мы заменяем сигнатуры действий `onSubscribe_` и `onNotify_` в соответствии с новыми типами. Добавляем эти типы в виде двух шаблонных параметров. И обновляем конструктор для биндинга двух разных действий. Другие методы `Observer`-а не требуют никаких изменений.

12.4 Message Driven Systems

12.4.1 Зачем все это?

В предыдущем разделе мы обсудили как устроен блокирующий observer pattern. Давайте я поясню, что это значит. Пусть у нас есть код:

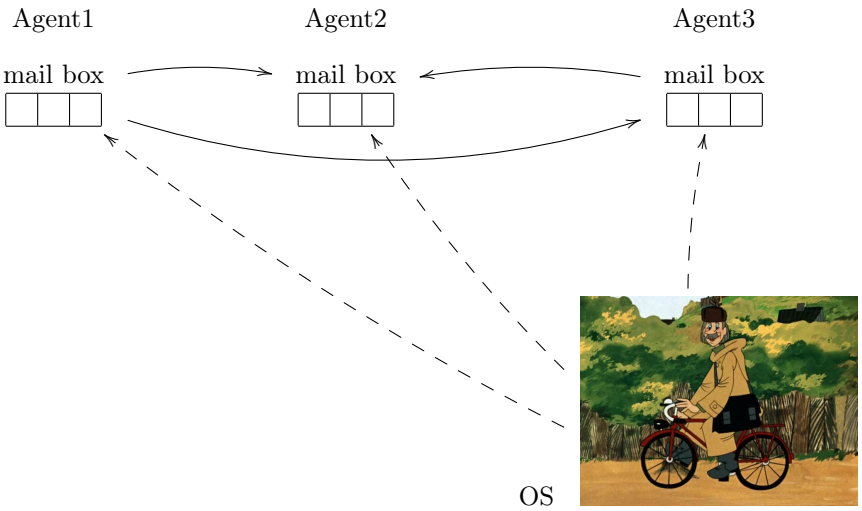
```
1  int x = 42;
2  Observable<int> out([&x] () { return x; }); // data binding
3  Observer<int> in(print, print); // actions binding
4
5  out.subscribe(&in); // print 42
6
7  x = 24;
8  out.notify();  // print 24
9  x = 42;
10 in.unsubscribe();
11 out.notify(); // does nothing
```

Я хочу обратить внимание на строки 5 и 8. Когда строчка 5 исполнилась и мы перешли на строчку 6, то порт `in` уже исполнили свое действие `onSubscribe`. То есть `subscribe` ждет соответствующего observer-а, пока он не завершит обработку на своем порте. Аналогично в строчке 8, когда мы перемещаемся к строке 9, то все observer-ы подписанные на `out` уже выполнили свои действия `onNotify`. То есть метод `notify` ждет, пока все observer-ы подписанные на observable выполнят свою работу. Поэтому данная имплементация и называется блокирующей.

Теперь возникает вопрос: «а что такое не блокирующий observer?» Оказывается просто так неблокирующего ничего быть не может. Действительно, если вы посмотрите на любую свою программу, она всегда идет сверху вниз и выполняет команды по-порядку. Если у вас голая функция `main`, то сложно даже представить, что означает фраза «неблокирующий вызов функции». Банально в какой момент эта функция будет исполняться? Куда она запишет свое значение? Когда мы ее результат прочитаем? Чтобы это все имело смысл, нам нужно иметь некую экосистему в рамках которой можно определить неблокирующие вызовы. В качестве такой экосистемы как раз и подходит среда функционирующая на основе передачи сообщений или Message Driven Systems.

12.4.2 Что такое Message Driven System

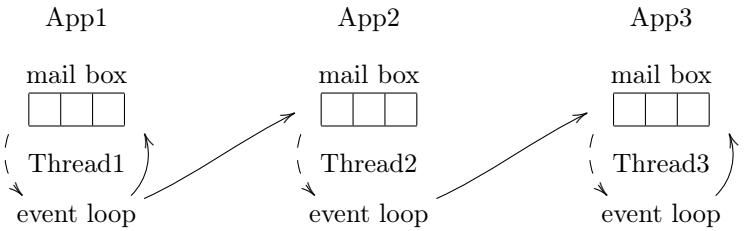
В рамках Message Driven System определено понятие агента. Это минимальная сущность функционирующая в системе. Агенты такой системы имеют свои собственные почтовые ящики и умеют выполнять какую-то работу и посылать себе или друг другу новые сообщения. В такой среде агент работает только, когда к нему в почтовый ящик что-то прилетело, если он пуст, то агент засыпает. Так же есть система, которая преобразует нативные сигналы операционной системы в сообщения среды. Это сделано для того, чтобы агенты могли реагировать на события операционной системы. Графически такую среду можно изобразить так



Здесь в роли почтальона Печкина выступает система, которая перерабатывает сообщения операционной среды в сообщения экосистемы и посылает их нужным адресатам (обозначено пунктирными стрелками). Сплошные стрелки – это сообщения, которые агенты посылают друг другу. Есть разные виды подобных сред с разными гарантиями на передачу сообщений. Но я буду рассматривать только те, которые встречаются в рамках одной физической машины. Для меня будет важно, что системы бывают с агентами двух видов: активными и пассивными.

MDS с активными агентами Напомню, что в компьютере работу совершают ядра процессора. На уровне операционной системы есть понятие thread-а. Это такая сущность, которая содержит работу для ядра процессора и все необходимые данные. И можно думать, что операционная система подсоединяет thread к ядру процессора и/или отсоединяет. Любой thread запускается с указанием стартовой функции, которая и будет описывать работу исполняемую на ядре.

В качестве примера Message Driven System-ы с активными агентами подходит операционная система Windows. Оказывается, что все поведение ОС из коробки построено на сообщениях. Все сигналы в приложение от операционной системы приходят в виде сообщений. Обмен информацией между приложениями ведется с помощью сообщений. Когда запускается очередное приложение в операционной системе, оно запускается на новом thread-е. При этом на этом thread-е создается почтовый ящик для получения сообщений и требуется запустить event loop. Event loop – это цикл, который обрабатывает сообщения если они есть в почтовом ящике, а если ничего нет, то нужно уснуть. Соответствующую картинку можно нарисовать так:

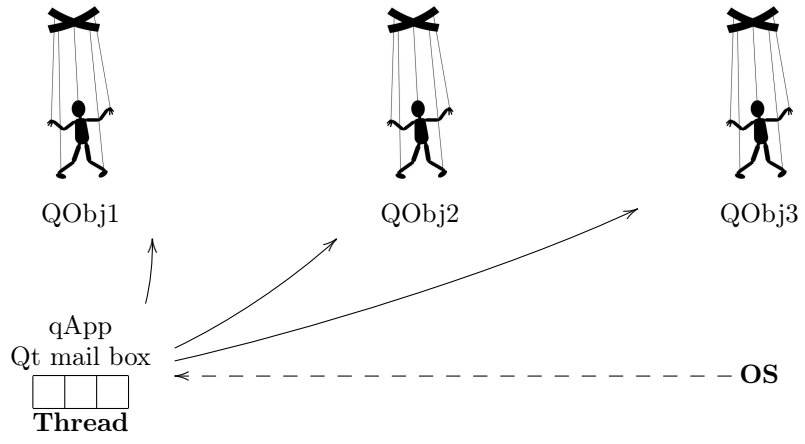


Здесь каждый агент – это отдельное приложение, которое имеет свой почтовый ящик и крутит на своем thread-е event loop. Для полноты картины напишу как приблизительно выглядит event loop

```
int runEventLoop() {
  CMessageStatus Status;
  MSG CurrentMessage;
  while ((Status = ::GetMessage(&CurrentMessage, hwnd(), 0, 0)) != Quit) {
    if (Status == Error)
      handleError(&CurrentMessage);
    else
      ::TranslateMessage(CurrentMessage);
      ::DispatchMessage(CurrentMessage);
  }
  return exitCode(&CurrentMessage);
}
```

Надо сказать, что перед запуском такого цикла обязательно должна быть команды создающие почтовый ящик, регистрирующие этот почтовый ящик в системе и прочие детали для внедрения в среду Windows. Я весь этот код игнорирую. Но давайте поймем логическую структуру данного цикла. В самом начале у нас есть две переменные для статуса и для сообщения. Внутри `while` цикла мы считываем текущее сообщение и записываем его статус в переменную `Status`. Далее если статус это `Quit`, то мы выходим из цикла и программа завершается. Если функция `GetMessage` видит пустой ящик, то она усыпляет данный thread и цикл зависает. Эта функциональность поддерживается автоматически из коробки. Внутри цикла идет в начале обработчик ошибок. А потом две стандартные функции, которые обеспечивают доставку сообщений в Windows. Не спрашивайте меня почему их две, и почему у них такие дурацкие названия, это все следствие legacy. Именно при вызове функции `DispatchMessage` сообщение достается и отправляется в обработку.

MDS с пассивными агентами Примером Message Driven System-ы с пассивными агентами является экосистема Qt (читается cute). На сегодняшний день это один из самых известных и мощных фреймворков для работы с GUI на C++. Но как и C++, она максимально кривая и сложная для безопасного использования. В рамках этой экосистемы выделяется один thread для event loop-а и на нем создается почтовый ящик. Это все инкапсюлировано в объект `QApplication`. Агентами в этой системе выступают `QObject`. Это все возможные компоненты Qt графические и не только. Например окно является агентом, кнопочка на нем является агентом, формы для ввода данных являются агентами и т.д. Но все эти агенты не имеют своего thread-а для выполнения своих методов. Вместо этого они лежат пассивно в памяти, а всю работу выполняет `QApplicaion`. Думать про это надо так. `QApplication` – это Папа Карло. `QObject`-ы – это куча Буратин лежащих в комнате. Что делает Папа Карло? Во-первых, он ловит сообщения операционной системы и перерабатывает их в Qt сообщения и складывает в свой почтовый ящик. Во-вторых, он вынимает из почтового ящика сообщения, смотри какая Буратина должна его получить. Идет к этой буратине, и вызывает у него нужный метод. В результате работы этого метода может быть появятся другие письма, которые будут сложены в почтовый ящик. Давайте снабдим это описание картинкой.



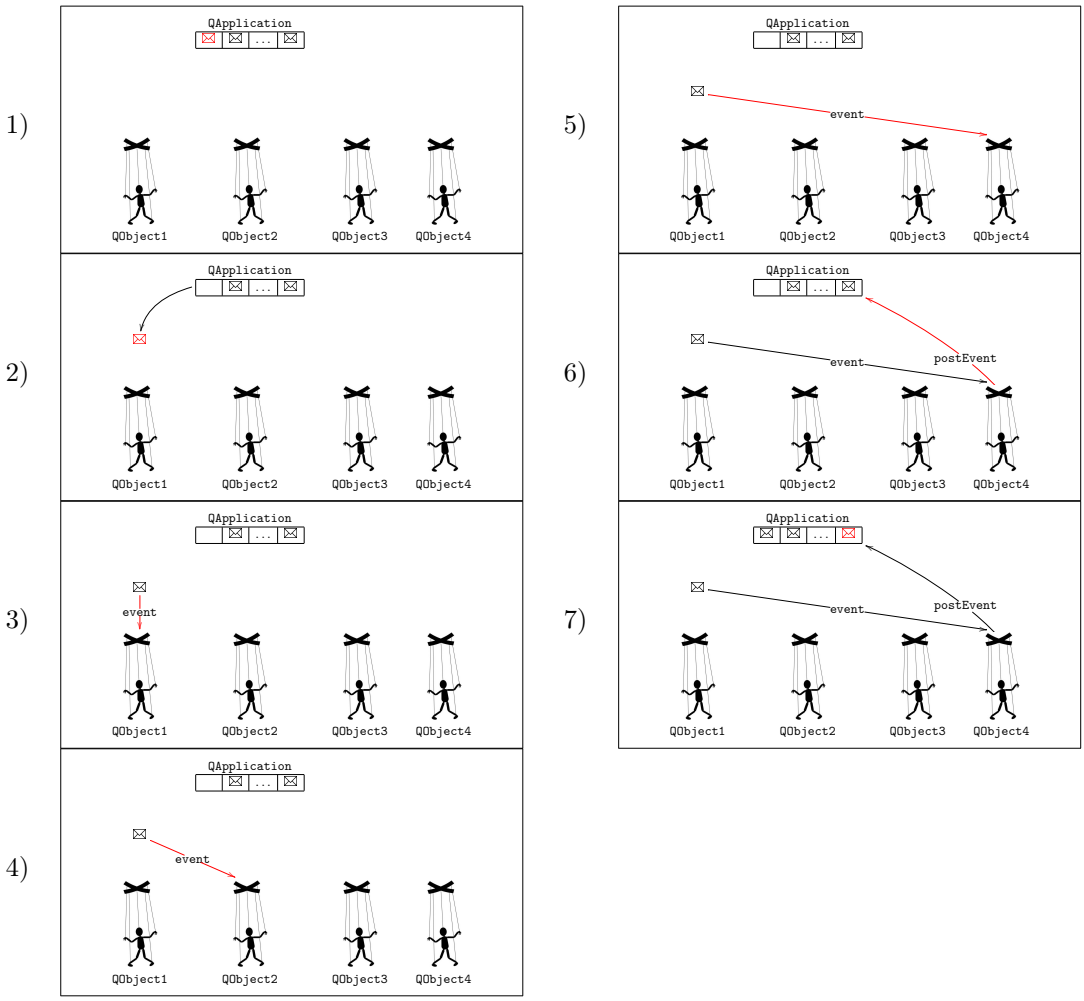
Так же для полноты картины давайте я приведу пример event loop в Qt экосистеме.

```
int main(int argc, char* argv[]) {
  QApplication QRunTime(argc, argv);
  Application application;
  QRunTime.exec();
```

```
5     return 0;
6 }
```

Вторая строчка создает Qt экосистему для данного thread-а. Третья строчка создает объект приложения. Важно, что его компоненты будут взаимодействовать с Qt экосистемой. В этой строчке само приложение просто положили в память, корректно инициализировали и корректно подключили к Qt экосистеме. И четвертая строчка запускает event loop. Именно в этой функции крутится цикл, который выполняет всю работу. В частности внутри этого цикла и будет выполняться работа при получении сообщения адресатом.

Функционирование Qt экосистемы можно представлять себе следующим образом:



Прокомментируем поведение системы выше:

1. `QApplication` проверяет пуст ли почтовый ящик или нет. Если почтовый ящик пуст, то система засыпает пока не появятся сообщения от операционной системы. Эти сообщения будут трансформированы в Qt сообщения и сложены в почтовый ящик.

2. Если почтовый ящик не пуст, то вынимаем первое сообщение из ящика. Теперь можно совершать доставку сообщения адресатам

3. В Qt экосистеме обработчик сообщения у адресата – функция `event`. Вызываем эту функцию для `QObject1` и передаем в нее текущее сообщение.

4. Теперь вызываем `event` у `QObject2` и передаем в нее текущее сообщение.

5. На этом шаге вызываем `event` у `QObject4` и передаем в нее текущее сообщение.

6. В результате обработки сообщения `QObject4` сформировал сообщение и отправляет его в почтовый ящик `QApplication`.

7. Сообщение от `QObject4` положили в конец очереди в почтовый ящик. На этом одна итерация event loop-а завершается и весь процесс повторяется с самого начала.

12.4.3 Идея имплементации

Я уже приводил пример event loop-а для двух Message Driven System. Давайте я в общих чертах скажу, как строится произвольная такая система. Прежде всего у нас должен почтовый ящик и event loop, которые выглядят как-то так.

```
MailBox gBox;

int main() {
  Event e;
  while(getEvent(&e)) {
    dispatchEvent(e);
  }
  return 0;
}
```

Тут важно понимать, что почтовый ящик является глобальным ресурсом для данного thread-а. Как мы видим весь event loop устроен очень просто, мы вынимаем сообщение из почтового ящика в функции `getEvent`. А потом запускаем доставку сообщений в методе `dispatchEvent`. При такой имплементации цикл заканчивается, если `getEvent` вернет `false`. В этой имплементации отсутствует код, перерабатывающий сообщения операционной системы в сообщения экосистемы.

Функция доставки сообщений работает приблизительно так

```
void dispatchEvent(const Event& e) {
  for (auto obj : addresseesOf(e)) {
    obj->handleEvent(e);
  }
}
```

То есть для каждого сообщения составляется список его адресатов, а потом мы просто зовем служебную функцию `handleEvent` у всех адресатов блокирующим образом.

Теперь, что из себя представляет сообщение? Обычно это данные и некоторая метаинформация, которая помогает понять, что это за данные. Ведь передавать мы должны уметь все что угодно. Кратко можно думать про это так

```
enum class EventType {
  Mouse,
  Keyboard,
  /* ... */};

class Event {
public:
  /*Constructor*/
  EventType type() const;
  const void* data() const;

private:
  EventType type_;
  std::any data_;
};
```

Здесь `EventType` – это заранее известный набор констант для всех возможных типов сообщений. Многие системы имеют выделенный диапазон типов для создания пользовательских сообщений. Само сообщение содержит информацию о типе и данные. Я использую `std::any` как стирающий тип, который умеет хранить что угодно.

Теперь перейдем к обработке сообщений. Для хранения списка адресатов, который используется в функции `dispatchEvent` можно использовать интерфейсы или стирающие ссылки.

```
std::vector<EventHandler> addresseesOf(const Event& e)
```

здесь `EventHandler` – это стирающий указатель на объект, которому доставляется сообщение. Для попадания в этот список в системе должны быть свои механизмы, которые я тут не собираюсь обсуждать. Теперь обработка сообщений выглядит как-то так

```
class Obj1 {
public:
  void handleEvent(const Event& e) override {
    switch(e.type()) {
      case EventType::Mouse:
        MouseEvent& Data = cast<MouseEvent&>(e.data());
        // handle mouse event
      break;
      case EventType::Keyboard:
        KeyboardEvent& Data = cast<KeyboardEvent&>(e.data());
        // handle keyboard event
      break;
      ...
    }
  }
};
```

Каждый объект должен прочитать тип сообщения, а потом в зависимости от этой информации вынуть нужные данные и отреагировать на них. Возможно, что `handleEvent` игнорирует какие-то сообщения, а может быть даже и все. Функция `cast` выше – это псевдокод, который призван показать, что мы вынимаем из сообщения данные, которые там зашифрованы в зависимости от информации о типе сообщения.

12.4.4 Неблокирующие операции

Теперь, если мы орудуем в рамках Message Driven System, у нас есть два способа отправить данные из одной функции в другую.

1. Прямой блокирующий вызов. Этот случай – это просто обычный вызов функции внутри другой функции. Например, если работает обработчик сообщений `Obj1`, то он может дернуть (если знает адрес) какой-нибудь метод из `Obj2`. В этом случае реакция на данные происходит немедленно.

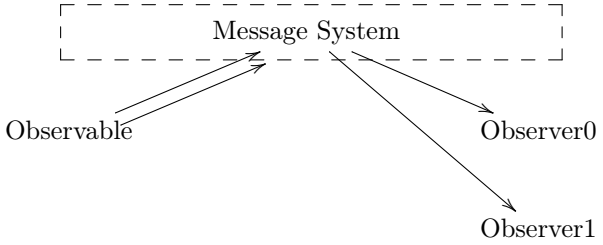
2. Посылка сообщения. В этом случае если обработчик сообщений `Obj1` произвел какие-то данные для `Obj2`, то он не вызывает метод `Obj2`. Вместо этого он создает сообщение, куда складывает эту информацию, пишет в качестве адресата `Obj2` на конверте и складывает письмо в почтовый ящик. В этом случае реакция на сообщение произойдет не сразу же во время отправки сообщения, а позже, когда экосистема дойдет до обработки сообщения и передаст его адресату. Такое поведение называется неблокирующим вызовом. Это поведение важно держать в голове, потому что важно поместить в сообщение все данные, чтобы они были живы, когда сообщение дойдет до адресата. Нельзя вкладывать ссылки и указатели на локальные данные, просто потому что они все умрут к моменту передачи сообщения по назначению.

Второй подход нужен для того, чтобы не блокировать на долго одной задачей ядро процессора. Чтобы на одном ядре успевали покрутиться разные обработчики сообщений. Это делается для того, чтобы GUI был отзывчивым и окна не подвисали. В рамках такой экосистемы полезно иметь неблокирующий observer pattern, в котором observable не сразу дергает методы `onSubscribe` и `onNotify`, а посылает сообщения через Message Driven System.

12.5 Неблокирующий Observer Pattern

12.5.1 Общая схема

Для неблокирующего observer pattern нам прежде всего нужна некоторая Message Driven System, которая будет использоваться для пересылки сообщений. Грубо общую модель можно описать следующей диаграммой.



В рамках такой системы `Observable` не дергает методы `Observer` напрямую. Вместо этого он посылает данные в виде сообщения каждому `Observer`-у. Давайте отметим несколько интересных изменений, которые возникают из-за такой системы.

- Прежде всего, нужен механизм адресации для того, чтобы можно было отправлять сообщения конкретным объектам. Обычно в любой Message Driven System такая возможность имеется из коробки и тут не надо ничего изобретать.
- В зависимости от возможностей Message Driven System-ы бывает нужно уметь проверять жив ли объект, которому отправляется сообщение. Причем жив ли он не в момент отправки, а в момент доставки сообщения. Например в системе Qt таких гарантий нет.
- Так как сообщения посылаются разными сообщениями каждый своему адресату, то не возможно пересылать данные по ссылке. Приходится в каждое сообщение от `Observable` вкладывать копию данных, которые надо пересылать.
- Когда же сообщение доставляется `Observer`, то мы гарантируем, что получатель ровно один и больше сообщение не понадобится. В этом случае данные можно безболезненно вынуть (мувнуть) из сообщения.
- Обратите внимание, что общение между `Observer` и `Observable` теперь происходит не мгновенно блокирующим образом, а с задержками. Они не реагируют друг на друга мгновенно. В этом случае можно думать, что общение портов происходит как бы через сеть. А все сетевое взаимодействие принято описывать с помощью State Machine.

Важно помнить, что конкретная имплементация зависит от выбора Message Driven System, от ее возможностей и предоставляемых гарантий.

12.5.2 Передача данных поверх Qt

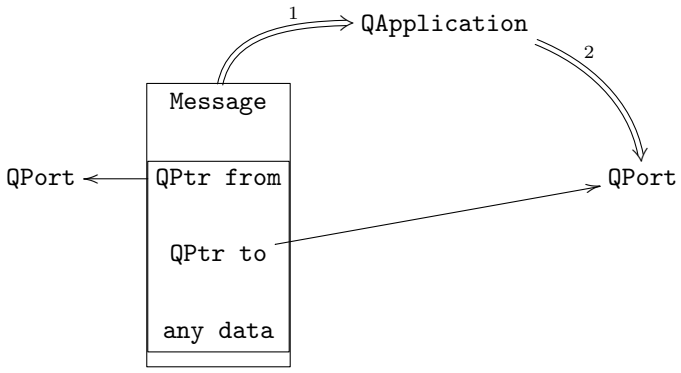
Дефекты Qt Начать я хочу с важного замечания по поводу системы Qt. Когда `QApplication` доставляет сообщения адресатам

$$\texttt{QApplication} \xrightarrow{\texttt{event}} \texttt{QObject}$$

он дергает блокирующим образом метод `event` по адресу объекта получателя. Однако, нет никаких гарантий, что объект, у которого мы дергаем `event` еще жив. Вокруг этой проблемы есть огромное количество костылей в Qt, включая отложенное удаление объекта. Получается, что в Qt экосистеме нельзя просто так слать сообщение `QObject` напрямую. Нужно перед дерганьем метода `event` проверить жив ли объект. И в Qt есть специальный следящий указатель (который по сути является другой имплементацией следящего указателя из раздела 12.6) и называется он `QPointer`. Так же важно, что объект `QApplication` всегда жив (на основном thread-е конечно). Тогда правильная схема – отправить сначала сообщение `QApplication` на своем thread-е. Если получатель живет на этом же thread-е, то `QApplication` в начале проверяет жив ли адресат и после этого дергает у него `event`. Если же нужно послать сообщение другому thread-у, то тут надо огранизовывать взаимодействие между `QApplication` на разных thread-ах. Благо Qt дает некую поддержку из коробки для этого. Но я не хочу это обсуждать.

QPort Обратите внимание, что сейчас обсуждается задача безопасной пересылки сообщения в экосистеме Qt. Этой проблемы может не быть в других экосистемах, но так и хорошо, я смогу продемонстрировать как подобные вещи решаются. Так как это задача более низкоуровневая, чем observer pattern, то здесь речь будет идти о более простом примитиве, который я хочу назвать `QPort`. Его задача будет состоять лишь в том, чтобы уметь передавать безопасно данные из от одного порта к другому не блокирующим образом. Ничего другого эти порты уметь не будут. И уже поверх этих портов мы построим observer pattern. Давайте схематично

изобразим, что из себя представляет сообщение и как оно передается.



1. Сообщение `Message` содержит три поля
   (a) Адрес порта отправителя в виде `QPointer`. Адрес отправителя нам понадобится для двусторонней коммуникации в observer pattern.
   (b) Адрес порта получателя в виде `QPointer`.
   (c) И сами данные в виде стирающего типа `std::any`. Это означает, что пользователь портов должен знать, что он отправляет и написать весь функционал поддерживающий типизацию поверх этой системы.

   Тут надо сказать зачем данные отправляются в виде `std::any`. Почему мы не используем шаблонный порт `QPort<T>`, что гарантировало бы доставку сообщений определенного типа. Тут к сожалению выстреливает еще одна особенность именно экосистемы Qt. Дело в том, что для правильной работы Qt выполняет генерацию кода для каждого `QObject`. Для этого есть специальный метакомпилятор `moc`. И метакомпилятор всегда применяется к `cpp` файлам и никогда не применяется к хедерам. Но шаблон нельзя разделить на хедер и сурс. Потому приходится обходиться без шаблонов с помощью стирающих типов.

2. Порт отправитель не шлет сообщение напрямую порту получателю, вместо этого сообщение отправляется `QApplication` по стрелке (1). Это действие всегда безопасно, потому что Qt экосистема гарантируется, что `QApplication` будет жить, если жив на данном thread-е дольше всех остальных объектов. Он создается первым, и удаляется последним. Межthread-овая коммуникация требует некоторой отдельной работы.

3. Теперь нам надо будет внести модификацию в поведение `QApplication`, чтобы он умел переотправлять наши сообщения порту получателю. Идея в том, что в начале `QApplication` проверяет `QPointer` получателя. Так как получатель живет на этом же thread-е, то у нас нет гонки между деструктором получателя и обработкой сообщения. Либо одно либо другое происходит раньше. Потому `QApplication` может спокойно проверить `QPointer` и если он не `nullptr`, то можно дернуть по его адресу метод `event`.

Сообщения Все сообщения в Qt экосистеме должны быть унаследованы публично от `QEvent`. Как было изображено на диаграмме мы должны иметь

```
1 class Message : public QEvent {
2 public:
3   using CPointer = QPointer<QEvent>;
4   ...
5 private:
6   CPointer from_;
7   CPointer to_;
8   std::any data_;
9 };
```

Кроме того, у каждого сообщения есть идентификатор `QEvent::Type`, который представляет из себя целое число. Это просто тег, который говорит получателю, что это за сообщения, чтобы потом суметь вынуть данные. Экосистема Qt позволяет зарегистрировать автоматически первый свободный идентификатор. Для этого выделим статический метод, который зарегистрирует тип при первом вызове и будет возвращать его значение всюду далее.

```
class Message : public QEvent {
public:
  static QEvent::Type type() {
    static int message_type = registerEventType();
    return QEvent::Type{message_type};
  }
  ...
private:
  ...
};
```

Напишем полностью интерфейс класса

```
class Message : public QEvent {
public:
  using CPointer = QPointer<QObject>;
  static QEvent::Type type();

  Message(CPointer from, CPointer to, const std::any& data);
  Message(CPointer from, CPointer to, std::any&& data);

  std::any&& extract();

  bool isReceiverAlive() const;

  CPointer receiver() const;
  CPointer to() const;
  CPointer from() const;
private:
  CPointer from_;
  CPointer to_;
  std::any data_;
};
```

В начале покажем, как нужно написать конструкторы

```
Message(CPointer from, CPointer to, const std::any& data)
  : QEvent(type()), from_(std::move(from)), to_(std::move(to)),
    data_(data) {}
Message(CPointer from, CPointer to, std::any&& data)
  : QEvent(type()), from_(std::move(from)), to_(std::move(to)),
    data_(std::move(data)) {}
```

Тут все просто, мы вызываем базовый конструктор, в который передаем тип сообщения вызывая статическую функцию `type`. Нам нужны два конструктора, чтобы была возможность положить новые данные в сообщения или мувнуть туда уже существующие. Теперь имплементация оставшихся методов

```
class Message : public QEvent {
public:
  ...
  std::any&& extract() {
    return std::move(data_);
  }
  bool isReceiverAlive() const {
    return to_;
  }
  CPointer receiver() const {
    return to_;
  }
  CPointer to() const {
    return to_;
  }
  CPointer from() const {
    return from_;
  }
private:
  ...
};
```

Тут все просто кроме быть может метода `extract`. Дело в том, что когда сообщение приходит в порт назначения, то оно больше использоваться не будет. А значит, нам надо вынуть данные из сообщения. Для этого нужно мувнуть данные наружу. Тут есть две опции:

1. Вернуть `std::any&&`.
2. Вернуть `std::any`.

Но в любом случае нужно будет вокруг `data_` добавить `std::move(data_)`. Это чуть ли ни единственный пример, когда нужно писать `std::move` в `return`. Без этого каста, будет сниматься копия с данных во второй имплементации. А первая имплементация выдаст ошибку компиляции.

Имплементация QPort Начнем с интерфейса

```
class QPort : public QObject {
  Q_OBJECT
public:
  using Message = detail::Message;
  using CPointer = Message::CPointer;

  void send(CPointer from, CPointer to, std::any data) const;
  bool event(QEvent* event) override;

protected:
  virtual void action(CPointer from, std::any&& data) = 0;

private:
  static constexpr bool k_is_processed = true;
};
```

Давайте прокомментируем, что тут происходит. Прежде всего, чтобы быть участником Qt экосистемы и уметь получать сообщения от системы нужно публично унаследоваться от `QObject`. Однако, этого мало. Из-за кодогенерации, нужно для каждого такого участника экосистемы сгенерировать некоторые служебные данные, без которых взаимодействие с системой не возможно. Для этого служит макрос `Q_OBJECT`, который добавляется сразу же в приватную часть класса.

Обратите внимание, что я страшный сторонник type erasure и ненавистник наследования с виртуальными методами внезапно начинаю использовать виртуальные методы. Связано это с тем, что в основе дизайна Qt лежит возможность наследоваться от базового класса, модифицируя его поведение переопределением виртуальных методов. Если фреймворк использует такой подход, то использовать стирающие типы будет очень сложно, код будет менее читаемым и громоздским и скорее всего еще и будет менее эффективным. Потому раз этот фреймворк использует такой подход, то нужно придерживаться его при взаимодействии с этим фреймворком. Но вот снаружи я могу использовать любые парадигмы, которые захочу.

Метода `action` является чисто виртуальным. Это действие, которое будет выполняться при получении сообщения. Обратите внимание, что данные получаются по rvalue ссылке, что означает, что мы будем вынимать данные из сообщения и передавать в этот метод через мув. Этот метод должен быть перегружен у наследника, какими будут `Observable` и `Observer`.

Теперь как имплементируется метод отправки сообщения

```
void QPort::send(CPointer from, CPointer to, std::any data) const {
  QCoreApplication::postEvent(
      QCoreApplication::instance(), // to whom
      new Message(std::move(from), std::move(to), std::move(data)));
}
```

Тут надо обратить внимание на то, что внутри вызывается всего лишь одна функция Qt экосистемы `postEvent`, которая просто помещает сообщение в почтовый ящик. Эта функция принимает два аргумента:

1. Адрес получателя, как адрес объекта. Здесь мы пишем адрес текущего `QApplication`. Этот объект обеспечивает существование и функционирование Qt экосистемы на данном thread-е и потому ему письмо всегда можно доставить безопасно.
2. Сообщение передаваемый по указателю. Сообщение обязано быть унаследовано от `QEvent`. Пусть вас не смущает голое `new` в коде. Данная функция захватывает владение сообщением и гарантирует его корректный менеджмент. При работе Qt всегда приходится думать кто кем владеет, чтобы случайно не было утечек данных.

И в теперь надо обсудить обработку входящих сообщений портом в функции `event`:

```
static constexpr bool k_is_processed = true;

bool QPort::event(QEvent* event) {
  if (event->type() == Message::type()) {
    Message* msg = static_cast<Message*>(event);
    action(msg->from(), msg->extract());
    return k_is_processed;
  }
  return QObject::event(event);
}
```

Все что тут происходит мы смотрим на получаемое сообщение и если оно нужного нам типа, то мы вынимаем из него данные и уже скармливаем эти данные функции `action`, которую мы обсудили выше.

Поддержка сообщений со стороны Qt На самом деле, чтобы эта схема работала нужно еще внести дополнительные изменения в `QApplication`, чтобы он знал, как реагировать на наши сообщения. Это тоже делается наследованием от `QApplication`. Все что мы сделаем – добавим правило обработки сообщений в самое начало. Это делается с помощью функции `eventFilter`. В коде это выглядит так

```
class QRunTime : public QApplication {
public:
  QRunTime(int& argc, char** argv);
private:
  bool eventFilter(QObject* obj, QEvent* event) override;
};
```

Я специально меняю название на `QRunTime`, потому что по смыслу это именно run-time системы на данном thread-e, а не одно приложение. Конструктор должен установить на себя же фильтр, который представляет из себя действие, которое будет производиться над сообщением до всех остальных действий.

```
QRunTime(int& argc, char** argv) : QApplication(argc, argv) {
  installEventFilter(this);
}
```

Функция `installEventFilter` прикрепила к `this` его же самого как фильтрующий объект. А метод, который будет вызываться для фильтрации надо имплементировать в `eventFilter`.

```
bool eventFilter(QObject* obj, QEvent* event) override {
  using Message = QApp::Library::QPort::Message;
  if (event->type() == Message::type()) {
    Message* msg = static_cast<Message*>(event);
    if (msg->isReceiverAlive())
      msg->receiver()->event(event);
    return true; // stop processing the event
  }
  return QApplication::eventFilter(obj, event);
}
```

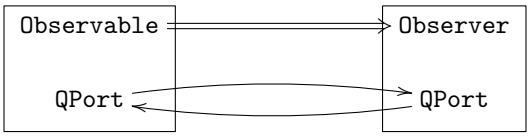
Все что мы делаем – проверяем тип сообщения. Если это наше особое сообщение для порта, то мы проверяем жив ли получатель, что делается через `QPointer` внутри метода `isReceiverAlive`. После чего напрямую дергаем обработчик сообщения у получателя и говорим, что это сообщение больше обрабатывать не нужно. И в конце вызываем дефолтный `eventFilter` для `QApplication`.

12.5.3 Observer и Observable поверх Qt

Прежде чем описывать имплементацию, давайте я напомню, что решение ниже пусть и не блокирующее, но не является много поточным. Я специально не рассматриваю сложности связанные с межпоточным взаимодействием. Более того, сами `QPort`-ы в моей имплементации не способны корректно работать в многопоточке.

Идея Идейно Observer и Observable будут внутри себя содержать `QPort`-ы, с помощью которых они будут пересылать данные друг другу. Снаружи коммуникация идет односторонняя, всегда Observable оповещает

Observer. Однако, для реализации корректного одностороннего протокола, на нижнем уровне все равно требуется двусторонее общение. Оно в этом случае не является проблемой, потому что это общение происходит в рамках конкретного паттерна, где можно перебрать все случаи.



Функционал данных классов можно описать так:

`Observable`

- Конструктор – бинд данных
- `subscribe`
- `notify`
- `unsubscribeAll`

`Observer`

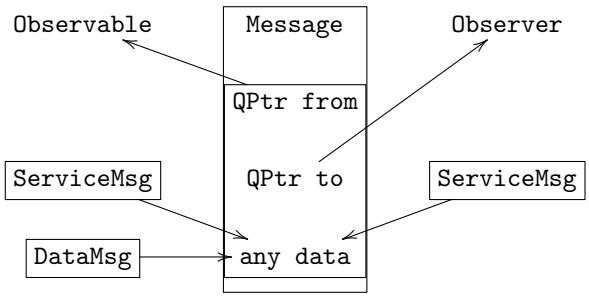
- Конструктор – бинд callback
- `unsubscribe`
- `subscribeTo`
- `isSubscribed`

Важно подчеркнуть, что теперь обмен сообщениями не блокирующий. Потому у этих классов теперь нет общего инварианта. Мы должны про них думать так, как будто они общаются по сети друг с другом.

Формат сообщения Для обеспечения двустороннего общения нам понадобится три поля в сообщении

1. Адрес отправителя.
2. Адрес получателя.
3. Данные.

Графически можно это изобразить так:



Данные при общении между Observable и Observer бывают двух видов:

1. Сервисные сообщения для установки и обрыва связи друг с другом. Эти данные могут посылаться в обе стороны.
2. Данные, которые Observable передает Observer при оповещении. Эти данные всегда идут от Observable к Observer.

В простейшей экосистеме Qt, где гарантируется доставка сообщений и их порядок нам не нужно большого количества сервисных сообщений. Достаточно использовать

1. `Subscribe`
2. `Unsubscribe`

Однако, чтобы имплементировать надежное соединение, которое бы давало корректное соединение между портами по хорошему нужно использовать State Machine внутри Observable и Observer. Но эта тема совсем далека от цели данного текста и потому мы просто напишем очень простую логику.

Observable Вот как будет выглядеть класс `Observable`

```
1  template<class T>
2  class Observable : protected QPort {
3    using GetAction = std::function<const T&()>;
4  public:
5    using ServiceData = ServiceData;
6    using CPointer = QPort::CPointer;
7  private:
8    using CListeners = std::list<CPointer>
9    using Data = std::variant<T, ServiceData>;
10 public:
11   template<class TF>
12   Observable(TF&& data);
13
14   Observable(const Observable&) = delete;
15   Observable(Observable&&) noexcept = delete;
16   Observable& operator=(const Observable&) = delete;
17   Observable& operator=(Observable&&) noexcept = delete;
18   ~Observable();
19
20   void notify();
21   void subscribe(CPointer obs);
22   void unsubscribeAll();
23 private:
24   // QPort virtual method
25   void action(CPointer from, std::any&& data) override;
26   GetAction data_;
27   CListeners listeners_;
28 };
```

Давайте прокомментируем, что тут происходит. Сам класс унаследован от `QPort`, потому что этого требует парадигма использоваться порта и в строке 25 мы как раз будем имплементировать функцию `action`, которая отвечает за обработку полученного сообщения. Так как общение `QPort` идет по адресу, мы удаляем возможность перемещать и копировать данный объект. Это ломает value semantics. Если мы хотим сохранить value semantics, то нам придется использовать следящие указатели, а не просто `QPointer`.

Внутри класса мы храним два поля:

1. Список тех, кого он в данный момент считает на себя подписанным. Это просто список `QPointer`-ов. Не обращайте внимание, что это `std::list`.

2. Метод получения данных, который хранится в `data_`. Этот метод будет возвращать новую копию данных, которые надо переслать `Observer`-у. Вызов этого метода будет единственным местом, где происходит копирование данных. Без этого копирования обойтись не возможно. Но его можно сделать дешевым, если данные могут быть только константными (см. раздел 10.3).

Данные передаваемые по порту у нас – это либо сервисное сообщение `ServiceData`, либо шаблонный тип `T`. Раз `QPort`-ы не различают типов данных, то мы просто будет передавать `std::variant` и проверять тип сообщения перед распаковкой.

Теперь пройдемся по методам. В начале конструктор. Его задача очень простая – забиндиться на данные, то есть запомнить метод получения данных в переменной `data_`.

```
1  template<class TF>
2  Observable(TF&& data) : data_(std::forward<TF>(data)) {}
3
```

А в деструкторе надо отписать от себя всех перед смертью, чтобы от нас не ждали сообщения и не думали, что на нас все еще подписаны.

```
1  ~Observable() {
2    unsubscribeAll();
3  }
```

Оповещение тоже вполне себе предсказуемо:

```
1  void notify() {
2    for (CPointer obs : listeners_)
3      sendData(obs);
4  }
```

Здесь мы будем использовать два приватных метода, которые посылают либо сервисные сообщения, либо сообщения с данными. Метод `sendData` посылает данные адресату, его имплементация будет ниже. Метод отписывающий всех observer-ов на самом деле оповещает всех, кто подписан о том, что надо от нас отписаться.

```
1 void unsubscribeAll() {
2   while (!Listeners_.empty()) {
3     sendServiceMessage(Listeners_.front(), {Unsubscribe});
4     Listeners_.pop_front();
5   }
6 }
```

Обратите внимание, что метод отправляет служебное сообщение `Observer` порту, что надо отписаться. И потом исключает порт из своего списка подписчиков. В этот момент `Observer` порт все еще думает, что он подписан (если с ним никто не взаимодействовал и не поменял его статус). Но нам это не важно. Служебные сообщения посылаются с помощью функции `sendServiceMessage`, которая будет описана ниже.

Метод `subscribe` теперь надо понимать так: пригласи такого-то `Observer` порта подписаться на наш `Observable` порт.

```
1 void subscribe(CPointer obs) {
2   if (obs == nullptr)
3     return;
4   if (!contains(obs)) {
5     Listeners_.push_back(obs);
6     sendServiceMessage(obs, {Subscribe});
7   }
8   sendData(obs);
9 }
```

Обратите внимание, что в строке 6 мы посылаем сервисное сообщение, приглашающее порт `obs` начать нас прослушивать. И потом в строке 8 сразу посылаем данные не дожидаясь никакого ответа. Так можно делать, потому что Qt экосистема гарантирует доставку сообщений и гарантирует их в правильном порядке. По-хорошему, нужно в начале установить на обоих портах соединение и уже после его установки пересылать данные. Я не хочу тут вдаваться в эти детали.

Теперь сервисные функции для посылки сообщений.

```
1 using Data = std::variant<T, ServiceMessage>;
2
3 void sendData(CPointer to) {
4   QPort::send(this, to, Data(data_()));
5 }
6 void sendServiceMessage(CPointer to, ServiceData data)  {
7   QPort::send(this, to, Data(std::move(data)));
8 }
```

Тут мы просто зовем методы `QPort`-ов. Обратите внимание, что по умолчанию третий аргумент принимается как `std::any`, а потому вначале надо данные кастануть к общему `std::variant`. В строчке 4 вызов `data_()` возвращает rvalue копию данных. А потому она мувается внутрь `std::variant`. Который уже мувается внутрь `std::any`. В случае сервисного сообщения мы принимаем данные по значению и муваем их внутрь руками.

Теперь, как выглядит обработчик приходящих сообщений. Так как мы принимаем `std::variant`, то у нас есть известная стратегия обработки данных с помощью visitor.

```
1 struct MsgVisitor {
2   void operator()(T&&) const;
3   void operator()(ServiceData data) const;
4 };
5
6 void action(CPointer from, std::any&& data) override {
7   Data msg = std::any_cast<Data>(std::move(data));
8   std::visit(MsgVisitor(this, from), std::move(msg));
9 }
```

То есть мы принимаем данные и вынимаем данные из `data`, которая есть `std::any`, с помощью `any_cast` и складываем в `msg`. А далее отправляем это сообщение в визитера `MsgVisitor`. Визитер должен знать `Observable`, поэтому он передается в качестве первого аргумента в конструктор, и порт от кого пришло сообщение, для корректной реакции на сообщение. Имплементация визитера следующая.

```
class MsgVisitor {
public:
  MsgVisitor(Observable* host, CPointer from) : host_(host), from_(from) {
  }
  void operator()(T&&) const {
  }
  void operator()(ServiceData data) const {
    host_->handleServiceData(data, from_);
  }
private:
  Observable* host_;
  CPointer from_;
};
```

Данные нам прийти не могут, но мы просто делаем заглушку в виде пустого метода. А на сервисное сообщение визитер дергает у текущего `Observable` нужный приватный метод.

```
void handleServiceData(ServiceData data, CPointer from) {
  switch (data.cmd) {
  case Subscribe:
    subscribe(from);
    break;
  case Unsubscribe:
    unsubscribe(from);
    break;
  default:
    break;
  }
}
```

А в нем идет просто перебор по возможным служебным сообщениям.

Observer Теперь посмотрим на `Observer` в коде.

```
template<class T>
class Observer : protected QPort {
  using Action = std::function<void(T&&)>;
public:
  using ServiceData = ServiceData;
  using CPointer = QPort::CPointer;
private:
  using Data = std::variant<T, ServiceData>;

public:
  template<class TF>
  Observer(TF&& onNotify);

  Observer(const Observer&) = delete;
  Observer(Observer&&) noexcept = delete;
  Observer& operator=(const Observer&) = delete;
  Observer& operator=(Observer&&) noexcept = delete;
  ~Observer();

  void unsubscribe();
  void subscribe(CPointer obs);
  bool isSubscribed() const;
private:
  // QPort virtual method
  void action(CPointer from, std::any&& data) override;
  Action onNotify_;
  CPointer observable_ = nullptr;
};
```

Как и в случае `Observable` мы удаляем все операции мува и копирования ибо порты общаются с помощью адресов и их менять нельзя.

Внутри класса мы храним два поля:

1. Действие, которое должно выполняться при прилете новых данных.

2. Адрес `Observable`, на который мы подписаны в данный момент. Этот адрес хранится в виде `QPointer`, чтобы уметь проверять жив ли объект перед посылкой данных.

Теперь пройдемся по методам. Начнем с конструктора и деструктора.

```
1 template<class TF>
2 Observer(TF&& onNotify) : onNotify_(std::forward<TF>(onNotify)) {}
```

Тут все прямолинейно, мы просто сохраняем метод внутри `OnNotify_`. В деструкторе же мы просто отписываемся перед смертью. Точнее, мы посылаем сообщение, что хотим отписаться прежде чем умереть.

```
1 ~Observer() {
2   unsubscribe();
3 }
```

Метод проверки подписаны мы или нет имплементируется тривиальной проверкой.

```
1 bool isSubscribed() const {
2   return observable_ != nullptr;
3 }
```

Теперь методы подписки и отписки.

```
1  void unsubscribe() {
2    if (!isSubscribed())
3      return;
4    sendServiceMessage(Observable_, {Unsubscribe});
5    Observable_ = nullptr;
6  }
7
8  void subscribe(CPointer obs) {
9    if (obs == nullptr)
10     return;
11   if (isSubscribed())
12     unsubscribe();
13   Observable_ = obs;
14   sendServiceMessage(Observer_, {Subscribe});
15 }
```

Тут важно обратить внимание, что метод отправки сервисных сообщений не блокирующий. А потому в строке 4 отправляется оповещение, что мы отписались. А в строке 14 отправляется приглашение на то, чтобы нас подписали. Сервисный методы отправки сообщений устроены так

```
1 using Data = std::variant<T, ServiceMessage>;
2
3 void sendServiceMessage(CPointer to, ServiceData data) {
4   QPort::send(this, to, Data(std::move(data)));
5 }
```

Логика абсолютно такая же как и в случае `Observable`. Тут важно не забыть обернуть данные в `std::variant`. При получении сообщений, мы реагируем в виртуальной функции `action`.

```
1 void action(CPointer from, std::any&& data) override {
2   Data msg = std::any_cast<Data>(std::move(data));
3   std::visit(MsgVisitor(this, from), std::move(msg));
4 }
```

Так как сообщения бывают двух типов: сервисные и с данными, то нам надо правильно распаковать `std::variant`, что мы делаем с помощью визитера. Имплементация визитера выглядит так:

```
1  class MsgVisitor {
2  public:
3    MsgVisitor(CObserver* host, CPointer from) : host_(host), from_(from) {
4    }
5    void operator()(T&& data) const {
6      host_->handleData(std::move(data), from_);
7    }
8    void operator()(ServiceData data) const {
9      host_->handleServiceData(data, from_);
10   }
```

```
11 private:
12   Observer* host_;
13   CPointer from_;
14 };
```

И как и в случае `Observable` визитер лишь перенаправляет запрос нужной функции из `Observer`. Обработка сервисных сообщений устроена так.

```
1  void handleServiceData(ServiceData data, CPointer from) {
2    switch (data.cmd) {
3    case Subscribe:
4      if (isSubscribed() && Observable_ != from)
5        sendServiceMessage(Observable_, {Unsubscribe});
6      Observable_ = from;
7      break;
8    case Unsubscribe:
9      if (Observable_ == from)
10       Observable_ = nullptr;
11     break;
12   default:
13     break;
14   }
15 }
```

Здесь имплементировано немного нетривиальное поведение. А именно, если мы нас приглашают подписаться и мы уже подписаны на другой порт, то мы отписываемся от другого порта. И после этого встаем на прослушку. При отписывании, мы проверяем, нам предложил отписаться тот самый порт, который мы слушаем или нет. И отписываемся, только если это тот порт, который мы слушали. Обратите внимание, что поведение на сервисное сообщение содержащее `Subscribe` запрос отличается от поведения при вызове метода `subscribe`. Это может быть баг, это может быть намеренно разное поведение. Если вы хотите одинаковое поведение, то внутри метода `subscribe` нужно вызвать `handleServiceData`. Главное понимать, что вы хотите, и что код делает, что нужно.

Обработчик данных выглядит так.

```
1 void handleData(T&& data, CPointer from) const {
2   if (!isSubscribed())
3     return;
4   if (Observable_ == from)
5     onNotify_(std::move(data));
6 }
```

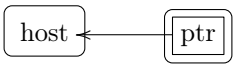
Здесь мы в начале проверяем, а слушаем ли мы сейчас кого-то или нет. Просто так могло получиться, что мы уже решили отписаться от порта, а он нам в это время успел отправить данные, и это прилетели запоздавшие данные, которые мы уже слушать не хотим. Потому если мы не подписаны, мы никого не слушаем и игнорируем данные. Если мы подписаны, мы проверяем от нужного `Observable` или нет пришли данные. И только если данные пришли от нужного порта, мы их передаем в метод `onNotify_`. Передаем все данные через мув.

12.6 Host/Handle

В этом разделе я хочу обсудить по-настоящему умные указатели. А именно, это такие указатели, которые отслеживают адрес объекта при его перемещении в памяти, знают жив ли все еще объект или нет. Как и выше тут приводится однопоточный код. Не надо думать, что в многопоточном случае надо делать то же самое но сложнее. Можно попробовать, а можно просто поставить правильные границы взаимодействия между потоками.

12.6.1 Что хотим

Если мы хотим в одном объекте пользоваться другим сторонним объектом, то самое простое – это хранить на него указатель.



Однако при таком наивном подходе есть несколько проблем.

Проблемы:

Вот список этих проблем:

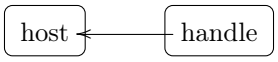
1. Объект мог переехать по другому адресу.
2. Объект мог умереть.
3. Самое интересное: а какое поведение мы хотим при копировании объекта, на который мы ссылаемся?

Оказывается, что первые две проблемы – технические и решаемые. Как раз решению этих проблем и будет посвящен весь этот раздел. А вот третья проблема требует отдельного рассмотрения, потому что она раскрывает важные проблемы при отсутствии value semantics.

Что плохого в копировании Давайте использовать следующие имена

1. host – объект, на который мы ссылаемся.
2. handle – умный указатель, который ссылается на host.

И предположим, мы смогли имплементировать умный указатель handle, который может отслеживать своего host. И пусть теперь у нас есть два таких объекта в связанном состоянии.

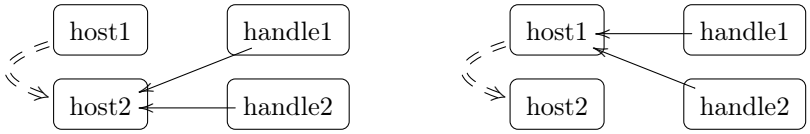


Теперь предположим, что мы хотим эту пару скопировать. Логически это означает, что мы бы хотели, чтобы новая пара была тоже связана стрелкой между собой. Однако, мы хотим так же поддерживать копирование только host или только handle. Давайте разберем следующие случаи:

1. Копирование handle. Есть по сути два варианта поступить:

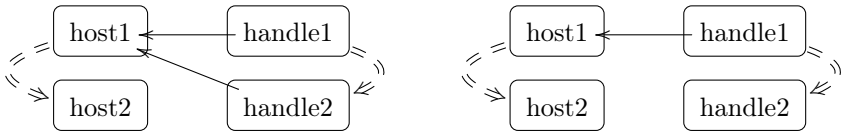
   Мы делаем копию и сохраняем стрелку на хоста или просто делаем абсолютно новую копию, которая ни с кем не соединяется. Если вы когда-нибудь имплементировали скажем бинарное дерево на указателях, то когда вы создаете новый узел, то обычно все указатели в нем устанавливаются в `nullptr` и это соответствуют второй картинке. В случае handle оба поведения выглядят разумными не ломают ничего. Это в частности связано с тем, что мы допускаем, что на одного host может ссылаться несколько handle. И если думать про handle, как про аналог умного указателя, то первое поведение будет более ожидаемым.

2. Копирование host. Вот тут начинаются интересные вещи, потому что мы handle может ссылаться только на одного host. Что мы могли бы сделать:



   В первом случае стрелки переезжают за хостом, а во втором мы просто создаем новую копию не соединенную ни с кем. И вот тут первое поведение не допустимо. Потому что оно ломает уже существующее соединение. А потому хоста мы можем лишь копировать без каких-либо соединений.

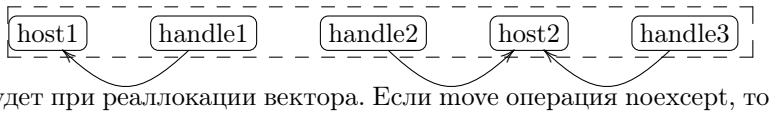
3. Копирование соединенной пары host←handle. Теперь когда мы знаем, какое поведение у копирующих операций для отдельно host и handle у нас допустимо, давайте посмотрим, что будет при копировании пары.



Как мы видим ни в одном из этих случаев не возможно только копирующими операциями каждого из узлов восстановить нужное состояние для копии пары. Наличие соединения между host и handle означает наличие у них общего инварианта. А это означает, что структурой данных является не отдельно host или отдельно handle, а вся связанная пара host←handle. Что в свою очередь значит, что любой scope, где эта пара находится, должен поддерживать связь при копировании.

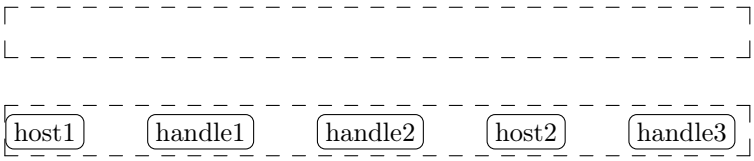
Примеры выше объясняют, что невозможно добиться правильного копирования соединенных объектов без внешнего вмешательства. Однако, есть еще очень важный пример, который показывает, что копирование может случайно разрушить существующие пары.

Опасность копирования Предположим, что мы умеем хранить host-ов и handle-ов в одном векторе. Это допустимая операция, так как handle-ы умеют следить за перемещением своего host.



Давайте посмотрим, что будет при реаллокации вектора. Если move операция noexcept, то после реаллокации получится следующая картина:

Объекты и связи переместятся как ни в чем не бывало. Однако, что если внезапно host или handle не nothrow movable. В этом случае вектор не имеет права использовать move операции и будет использовать копирование. А это означает, что после реаллокации у вас сломаются связи при любой имплементации копирования обсуждаемой выше.



Если бы копирующие операции были удалены, то при необходимости реаллоцировать вектор компилятор бы ругнулся, что он не может использовать такой объект в векторе. А так вы можете просто сломать связи между объектами и даже не заметить этого.

Финальные мысли про копирование Давайте резюмируем кратко, что мы осознали из проделанного:

- Копирование – опасная процедура. Она разрушает связи между объектами.
- Разрушение связей не фиксится локально. То есть невозможно имплементировать копирующие операции host и handle так, чтобы при копировании правильно копировалась связь между ними. Для проведения такого копирования нужен внешний арбитр, который доложен следить за состоянием таких связей. Это плата за отсутствие value semantics.
- Из выше сказанного следует, что если мы хотим уметь ссылаться на другие объекты, то нас есть два пути:

1. Мы убираем возможность копировать объекты неявно. Это решение не требует глобальной информации и все делается локально.
2. Мы оставляем возможность неявного копирования, но тогда должен быть внешний объект, который все время проверяет корректность ссылок между объектами. Это приблизительно как при имплементации структуры данных основанной на узлах и указателях на эти узлы. Только проблема в таком решении в том, что вам теперь в любом месте использования ваших указателей придется руками делать кучу лишней работы. Это решение не локальное и годится только внутри какой-нибудь структуры данных, но никак не подходит для проекта целиком.

- Логически и технически с операцией move проблем нет. Такую операцию можно легко поддержать на стороне host и handle.
- Обратите внимание, что даже если мы сделаем host не копируемым, то это еще не означает, что у него обязательно будет постоянный адрес. Он вполне может перемещаться в памяти и за его адресом нужно следить.
- Если вы хотите получить вместе постоянный адрес и value semantics, то это делается с помощью `std::unique_ptr`.

12.6.2 Что хотим в коде

Прежде чем имплементировать что-то очень полезно понимать, а какой код вы хотите писать, чтобы он работал. Давайте начнем со следующего примера. Мы хотим к любому классу `A` подключить возможность его отслеживать. Выглядеть это должно как-то так

```
1 class A : public Host<A> {
2 public:
3   void f() const;
4   void g();
5 };
```

Заметьте, чтобы отслеживать объект, мы должны вмешаться в операции перемещения и разрушения, чтобы изменять информацию об адресе. Для этого мы можем добавить следящий класс в качестве базового. Публичное наследование нужно только для того, чтобы были доступны методы `Host` снаружи из `A`. Но это не обязательно, можно просто пробросить эти методы наружу. Теперь ожидается следующее использование.

```
1 A a;
2 Handle<A> h = a.handle();
3 ConstHandle<a> ch = a.chandle();
4 if (h) // true
5   h->g();
6 A b = std::move(a);
7 if (ch) // true
8   ch->f(); // OK
```

В строках 2 и 3 создаются handle и константный handle на объект `a`, соответственно. В строке 4 проверяем жив ли объект, на который мы ссылаемся и вызываем метод `g`. В строке 6 перемещаем `a` по новому адресу в объект `b`. После чего в строке 7 handle ссылается уже на новый объект в переменной `b`. Еще один пример, если handle живет дольше, чем host.

```
1 Handle<A> h;
2 {
3   A a;
4   h = a.handle();
5 }
6 if (h) // false
7   h->f();  // f is not called
```

В этом случае при выходе из внутреннего scope в строчке 5 объект `a` умирает и `h` теперь не ссылается на живой объект. И как раз проверка в строчке 6 будет ложна и метод `f` не будет вызван на несуществующем объекте. Так же мы ожидаем защиту const correctness. Например если мы создали константный handle, то нельзя вызвать неконстантный метод.

```
A a;
auto ch = a.chandle();
ch->g(); // compilation error
```

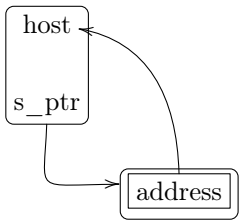
Или присваивание одного handle к другому должна тоже уважать const correctness.

```
const A a;
Handle<A> h = b.handle(); // compilation error
```

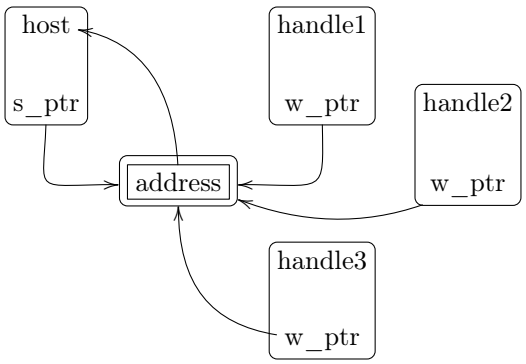
Нельзя внезапно преобразовать константный handle к неконстантному.

12.6.3 Идея имплементации

Идея на самом деле жутко простая и это один из хороших примеров использовать `shared_ptr` и `weak_ptr` Давайте будем хранить указатель на host в `shared_ptr` следующим образом



Тогда handle-ы будут хранить `weak_ptr` на данный `shared_ptr`.

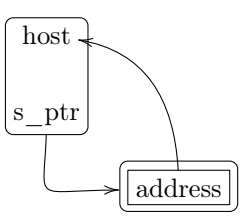


Важно сказать, что host может быть как константным, так и не константным объектом. Но адрес всегда хранится как на НЕ константный объект. Одна из причин почему это делается так – возможность создавать одновременно константные и не константные handle-ы на один и тот же объект. А потому надо быть осторожным при имплементации, потому что можно случайно получить неконстантный доступ к константному объекту и получить UB. Эта опасность НЕ возникает снаружи, когда вы уже будете пользоваться host и handle-ами.

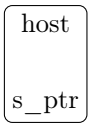
Имплементация методов Host

1. Конструктора. Здесь есть две тактики:

   (a) Явная инициализация. В этом случае мы при создании хоста сразу же инициализируем `shared_ptr` с адрессом хоста.

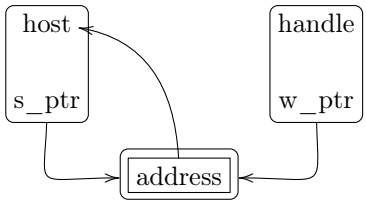


(b) Ленивая инициализация. В этом случае мы не инициализируем `shared_ptr` адресом хоста, а просто создаем пустой `shared_ptr`. В этом подходе инициализация наступает лишь при создании handle. Это позволяет не платить накладные расходы на аллокацию тогда, когда это не нужно.

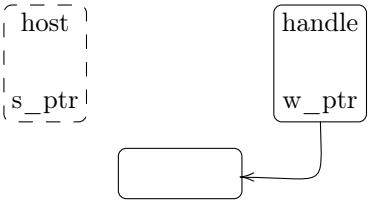


Я буду использовать ленивую инициализацию.

2. Деструктор. Предположим, что мы имеем host-а и к нему прикреплен handle.

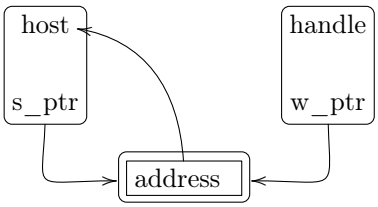


Тогда при смерти хоста, умирает и `shared_ptr`. Но тогда у `weak_ptr` есть возможность проверить жив или мертв объект.

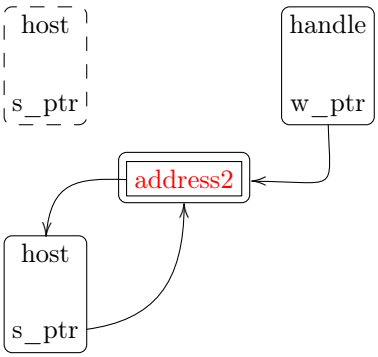


А потому ничего делать не надо.

3. Перемещающий конструктор. Если `shared_ptr` не был инициализирован, то мы просто ничего не делаем. Однако, если же `shared_ptr` с адресом был инициализирован, то надо перезаписать текущий адрес объекта после перемещения. То есть если мы стартовали в такой ситуации:

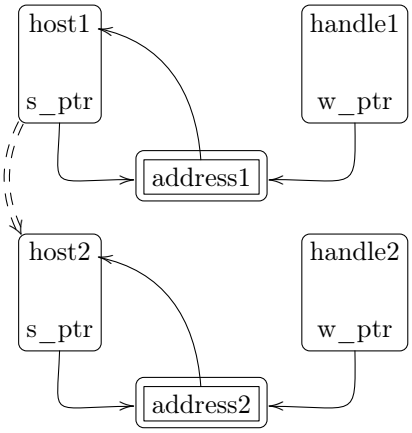


То после перемещения будем иметь

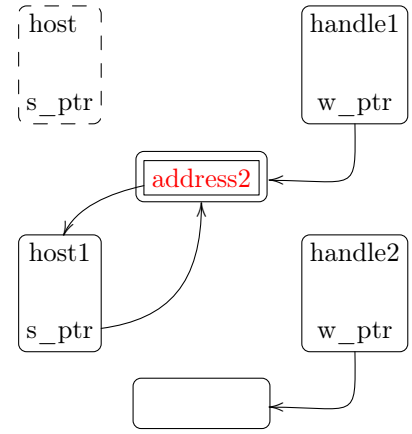


4. Перемещающее присваивание. В этом случае мы хотим переместить объект не в новое место в памяти, а в уже существующий объект. А потому в начале надо этот существующий объект убить. Так же надо

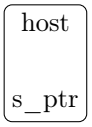
разобрать случаи когда `shared_ptr` инициализирован или нет. Я рассмотрю интересный случай, когда надо сделать изменение адреса. Пусть мы имеем следующую картину



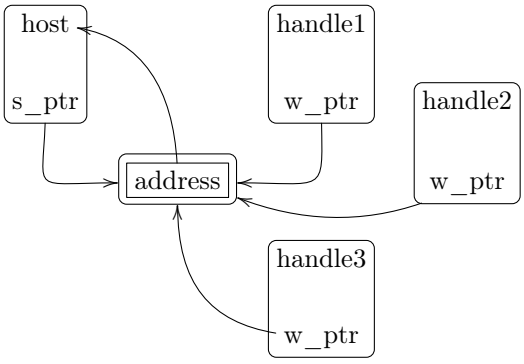
После перемещения `host1` на место `host2` поле `shared_ptr` в `host2` просто удалится и значит `handle2` будет видеть удаленный объект. После этого надо лишь перезаписать адрес на текущее положение `host1`.



5. Создание handle. Пусть у нас есть host с еще не инициализированным `shared_ptr`.



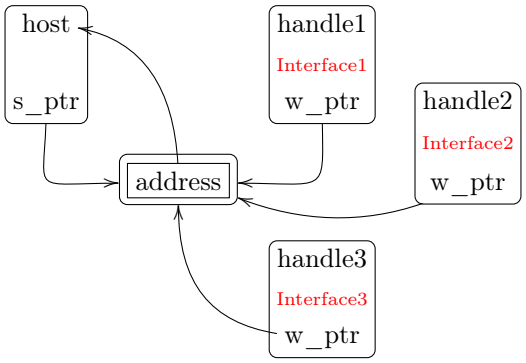
Прежде чем выдать новый handle, надо создать `shared_ptr`, в который запишется текущий адрес объекта. После чего на этот `shared_ptr` делается `weak_ptr`, который складывается в handle.



Имплементация Handle На удивление имплементация handle не содержит каких-то интересных деталей. Поэтому можно сразу переходить к разделу 12.6.5, где представлена имплементация в коде.

12.6.4 Разные интерфейсы

Подход с host/handle позволяет не просто делать ручку, которая умеет следить за хостом. Можно так же контролировать какой интерфейс предоставляет данная ручка.



Вот как это могло бы выглядеть в коде. У нас может быть класс `A`, к которому мы хотим присоединить два интерфейса.

```
class A : public Host<A> {
public:
  void f() const;
  void g();
  void h() const;
  void w();

private:
  ...
};
```

```
class Interface1 {
public:
  void f() const;
private:
  ...
};

class Interface2 {
public:
  void g();
  void h() const;
private:
  ...
};
```

Тогда можно ожидать такое использование

```
1  A a;
2  Handle<Interface1> h1 = a.handle();
3  ConstHandle<Interface2> h2 = a.handle();
4
5  h1->f();// OK
6  h1->g();// Compilation Error
7  h1->h();// Compilation Error
8
9  h2->f();// Compilation Error
10 h2->g();// Compilation Error
11 h2->h();// OK
12
13 h1 = h2;// Compilation Error
14 h2 = h1;// Compilation Error
```

Теперь ручка контролирует не только const correctness, но и предоставляемый интерфейс. При этом ручки с разными интерфейсами не конвертируются друг в друга. Последняя мысль подсказывает, что можно сделать иерархию интерфейсов, чтобы более широкие могли конвертироваться к более узким. Например как-то так.

```
class A : public Host<A> {
public:
  void f() const;
  void g();
  void h() const;
  void w();

private:
  ...
};
```

```
class Interface1 {
public:
  void f() const;
private:
  ...
};

class Interface2 : public Interface1 {
public:
  void h() const;
private:
  ...
};
```

Тогда ожидаемое использование будет выглядеть так.

```
1  A a;
2  Handle<Interface1> h1 = a.handle();
3  Handle<Interface2> h2 = a.handle();
4
5  h1->f();// OK
6  h1->g();// Compilation Error
7  h1->h();// Compilation Error
8
9  h2->f();// OK
10 h2->g();// Compilation Error
11 h2->h();// OK
12
13 h1 = h2;// OK
14 h2 = h1;// Compilation Error
```

И как вы видите теперь есть конвертация между ручками в одну сторону.

Теперь остается вопрос: кто решает какой интерфейс выдавать. Это можно делать на стороне хоста или на стороне ручки.

На стороне пользователя

```
A a;
Handle<Interface1> h1 = a.handle();
Handle<Interface2> h2 = a.handle();
```

Здесь `handle` решает какой интерфейс запросить.

На стороне хоста

```
A a;
auto h1 = a.handle<Interface1>();
Handle<Interface2> h2 = a.handle<Interface2>();
```

Здесь `host` решает какой интерфейс отдать.

12.6.5 Имплементация

Host Начнем с описания того, как будет выглядеть хост.

```
1  template<class T>
2  class Host {
3  protected:
4    Host() = default;
5    Host(const Host&) = delete;
6    Host(Host&&) noexcept;
7    Host& operator=(const Host&) = delete;
8    Host& operator=(Host&&) noexcept;
9    ~Host() = default;
10 public:
11   Handle handle();
12   ConstHandle handle() const;
13   ConstHandle chandle() const;
14   void detach();
15   T* ptr();
16   const T* ptr() const;
17   T& ref();
18   const T& ref() const;
19 private:
20   shared_ptr<T*> host_address_;
21 };
```

Давайте обратим внимание на некоторые особенности класса. Все дефолтные методы создания, копирования, перемещения и удаления делаются защищенными. Это сделано для того, чтобы нельзя было просто создать объект класса `Host<A>`. От него можно только наследоваться. Приватное поле – адрес текущего объекта. Обратите внимание в каком виде хранится адрес. Мы предполагаем, что `Host<T>` буде базовым классом для `T`. А это означает, что реальный адрес объекта должен быть `T*`, а не `Host<T>*`. Когда в базовом классе вы знаете информацию о наследнике, такой подход обычно называется CRTP и обсуждается **??**. Публичный интерфейс содержит методы выдачи ручек константных и не константных. Методы получения указателя и ссылки на отслеживаемый объект. И еще метод для отсоединения всех.

Теперь пройдемся по имплементации методов. Из стандартных методов мы удаляем операции копирования, операции перемещения надо имплементировать, а остальные дефолтные. Перемещающий конструктор будет выглядеть так.

```
  Host(Host&& other) noexcept
  : host_address_(std::move(other.host_address_)) {
    if (host_address_)
      *host_address_ = static_cast<T*>(this);
  }
```

То есть мы перемещаем единственное поле `host_address_` и так как у нас ленивая инициализация, то мы записываем туда новый адрес только если адрес был инициализирован.

Теперь идет присваивающее перемещение.

```
Host& operator=(Host&& other) noexcept {
  *host_address_ = std::move(other.host_address_);
  if (host_address_)
    *host_address_ = static_cast<T*>(this);
  return *this;
}
```

То есть мы выполняем дефолтный мув и только если адрес хоста был инициализирован, то мы его перезаписываем.

Теперь имплементация метода выдачи ручек. Вначале нам понадобится вспомогательный метод:

```
static shared_ptr<T*> sharedOn(Host& host) {
  return make_shared(static_cast<T*>(&host)); } // Downcast
};
static shared_ptr<T*> sharedOn(const Host& host) {
  return make_shared(static_cast<T*>(&const_cast<Host&>(host))); } // Downcast
};
```

Этот приватный метод позволяет получить `shared_ptr` с адресом на текущий объект, который мы отслеживаем. Для этого приходится делать каст базового класса, то есть нашего класса `Host<T>`, к наследнику, которым является `T`. Обратите внимание на имплементацию второго метода. Он нужен чтобы в константных методах избавиться от константности адреса `this`. Если бы мы инициализировали `shared_ptr` в конструкторе всегда, то этой имплементации не понадобилось бы, потому что внутри конструктора объект никогда не константный. Однако, ленивая инициализация требует от нас возможности получения адреса на неконстантный объект даже в константных методах. Это означает, что при разработке такого класса надо быть аккуратным и убедиться, что вы не получаете UB. Но как только вы в этом убедились, то все безопасно. Тогда выдача ручек имплементируется так

```
Handle handle() {
  if (!host_address_)
    host_address_ = make_shared(*this);
  return Handle(host_address_);
}
ConstHandle handle() const {
  if (!host_address_)
    host_address_ = make_shared(*this);
  return ConstHandle(host_address_);
}
ConstHandle chandle() const {
  if (!host_address_)
    host_address_ = make_shared(*this);
  return ConstHandle(host_address_);
}
```

Уже конструктор handle преобразует `shared_ptr` в `weak_ptr`. Метод отсоединения всех ручек имплементируется так.

```
void detach() {
  host_address_.reset();
}
```

Методы доступа к адресу объекта

```
  T* ptr() {
    return static_cast<T*>(this);
  }
  const T* ptr() const {
    return static_cast<const T*>(this);
  }
  T& ref() {
    return *ptr();
  }
  const T& ref() const {
    return *ptr();
  }
```

Из-за ленивой инициализации мы не обязаны хранить адрес внутри `host_address_`. И запрос данных из него требует прыжка по памяти. Явный каст получается лучше и удобнее.

Опасная деталь Напомню про опасную деталь еще раз. Мы храним адрес хоста всегда как указатель на неконстантный объект, даже если хост на самом деле константный. Давайте посмотрим на следующий код.

```
class A : public Host<A> {};

int main() {
  const A a;
  Handle<A> h1 = a.handle(); // compilation error
  ConstHandle<A> h2 = a.chandle(); // OK
  return 0;
}
```

Как мы видим, из-за того, что `a` константный, мы не можем вызвать метод `handle`. А так как доступ к хосту есть либо через его методы, либо через handle. То const correctness будет соблюдена. Мы не сможем создать неконстантную ручку на константного хоста.

Handle Теперь посмотрим на имплементацию handle-ов.

```
template<class T>
class Handle {
  friend class Host<T>;
protected:
  Handle(weak_ptr<T*> host_address);
public:

  operator bool() const;
  bool operator==(nullptr_t) const;
  bool operator!=(nullptr_t) const;

  Handle& operator=(nullptr_t);
  void reset();

  T* ptr() const;
  T& ref() const;
  T* operator->() const;
private:
  weak_ptr<T*> host_address_;
};
```

Обратите внимание, что у `Handle` защищенный конструктор. Это сделано для того, чтобы никто кроме `Host` не мог его создать. Так как объявлен один конструктор, то дефолтного конструктора нет. Операции копирования, перемещения и удаления подходят дефолтные, потому они даже не объявляются. Следующие операции проверяют, что `Handle` действительно ссылается на живой объект.

```
operator bool() const {
  return !host_address_.expired();
}
bool operator==(nullptr_t) const {
  return !(*this);
}
bool operator!=(nullptr_t) const {
  return *this;
}
```

Операции сброса ручки.

```
void reset() {
  host_address_.reset();
}
Handle& operator=(nullptr_t) {
  reset();
  return *this;
}
```

Операции доступа к данным.

```
T* ptr() const {
  shared_ptr<T*> host_address = host_address_.lock();
  if (!host_address)
    return *host_address;
  return nullptr;
}
T& ref() const {
  return *ptr();
}
T* operator->() const {
  return ptr();
}
```

Обратите внимание, что операции доступа к данным не безопасные. Кроме того, `weak_ptr` не позволяет обратиться к данным, несмотря на то, что он знает где они лежат. Приходится вначале делать конвертацию к `shared_ptr`и только потом дергать данные. Можно было бы хранить внутри ручек `shared_ptr`, но тогда пришлось бы делать лишнюю работу по занулению указателя хранящегося внутри `shared_ptr` при удалении или перемещении объекта. Так же несмотря на то что `Handle` ссылается на неконстантные данные, все методы доступа у него снабжены `const` квалификатором. Это связано с тем, что `const` не пробрасывается сквозь указатели. А `Handle` у нас просто умная версия указателя.

Версия `ConstHandle` пишется абсолютно аналогично с двумя маленькими отличиями. Первое отличие – методы доступа возвращают константные ссылки и указатели. Второе отличие – возможность конвертировать `Handle` к `ConstHandle`. Для этого нужно добавить два конструктора. Давайте для полноты картины приведем код класса и имплементацию новых методов.

```
template<class T>
class ConstHandle {
  friend class Host<T>;
protected:
  ConstHandle(shared_ptr<T*> host_address);
public:
  ConstHandle(const Handle& other);
  ConstHandle(Handle&& other) noexcept;

  operator bool() const;
  bool operator==(nullptr_t) const;
  bool operator!=(nullptr_t) const;

  ConstHandle& operator=(nullptr_t);
  void reset();

  const T* ptr() const;
  const T& ref() const;
  const T* operator->() const;
private:
  weak_ptr<T*> host_address_;
};
```

Операции доступа к данным. Тут `const_cast` не обязателен, так как `T*` конвертируется к `const T*`. Я его добавил для того, чтобы подчеркнуть место, в котором происходит конвертация.

```
const T* ptr() const {
  shared_ptr<T*> host_address = host_address_.lock();
  if (!host_address)
    return const_cast<const T*>(*host_address);
  return nullptr;
}
const T& ref() const {
  return *ptr();
}
const T* operator->() const {
  return ptr();
}
```

И два конструктора от `Handle`.

```
ConstHandle(const Handle& other) : host_address_(other.host_address_) {
}
ConstHandle(Handle&& other) noexcept : host_address_(std::move(other.host_address_)) {
}
```

Эти конструкции работают потому что и `Handle` и `ConstHandle` хранят внутри себя адрес в виде `T*`.

12.7 Model View Controller (MVC)

В этом разделе наша задача обсудить один из важных паттернов используемых для GUI – MVC паттерн.

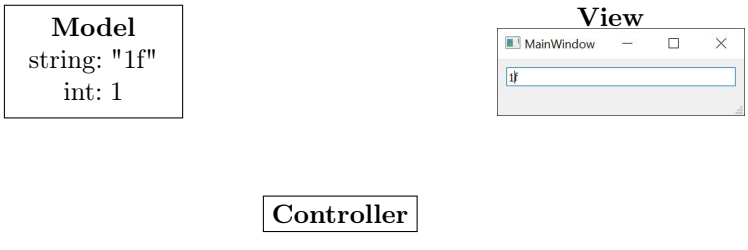
12.7.1 Что происходит

Если мы хотим написать интерактивное приложение, то полезно писать код так, чтобы ядро приложения и графические библиотеки были отделены друг от друга. Если вам нужно внести изменение в пользовательский интерфейс, ядро приложения не должно никак от этого пострадать. Если вы вносите изменения в ядро, которое не нарушает контракт соединения с графической библиотекой, то это не должно ломать ваш код. Для того чтобы обеспечить такое поведение принято выделять 3 основные компоненты.

1. Model. Это внутреннее представление данных. Данный объект содержит все состояние.
2. View. Графическое представление данных. Этот объект использует данные модели и отображает их пользователю.
3. Controller. Объект, управляющей моделью, на основе сигналов от пользователя.

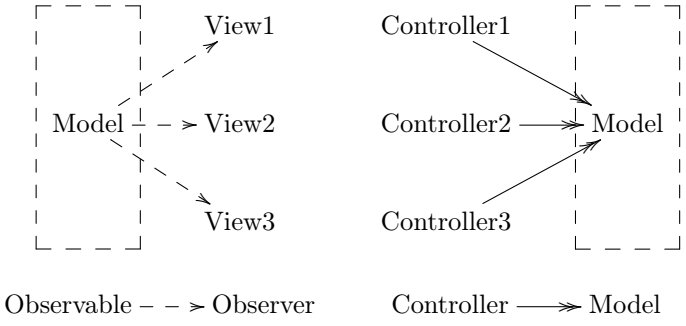
Важно понимать, что это не значит, что вся программа содержит одну модель, одну view и один контроллер. Подобное разделение на роли может происходить на более мелком уровне.

Пример Предположим, что у вас есть простейшее приложение состоящее из окна и поля для введения и редактирования текста. В этом случае у нас есть следующие три компонента:



Обратите внимание, что модель в этом случае не просто строка, а строка и позиция курсора. Потому что окно для редактирования умеет показывать положение курсора для редактирования. Логически данные (или состояние) хранится только в модели. View ничего не хранит (логически не хранит, всякое кэширование данных и прочие вспомогательные структуры могут использоваться). Важно, что только данные из модели являются настоящей правдой. То есть без модели View не знает что рисовать. У нее нет дефолтного состояния. Контроллер – это просто кусок кода, который принимает команды от пользователя и на основе них вызывает методы у модели.

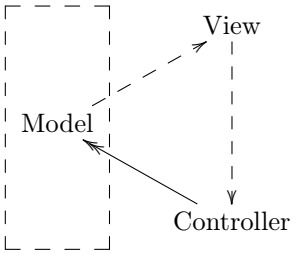
Виды соединений Теперь когда у нас есть три компоненты надо понять как они друг с другом связаны. Бывают два типа соединения:



Соединение представленное слева – это push уведомления, которые представляют из себя Observalbe/Observer соединения, которое обсуждалось в разделе 12.3. Его главная особенность в том, что у нас один источник данных и много слушателей для этих данных. Соединение справа – это соединение Host/Handle, которое обсуждалось в разделе 12.6. Его основная особенность, что наоборот может быть много контроллеров на одну модель.

12.7.2 Модель для MVC

Обычная модель Обычно принято рисовать такую общую картинку для описания того, как соединяются компоненты между собой.



Пунктиром тут обозначены границы ядра программы. Эта часть вообще не зависит ни от каких графических библиотек. Эта часть даже не знает, что она является частью интерактивного приложения.

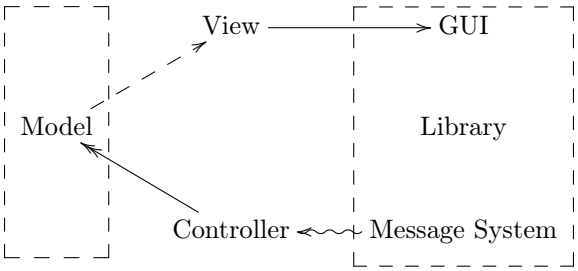
- Модель внутри ядра – основной источник данных. Как только View присоединилась к Model, то Model тут же отправляет ей свое текущее состояние и View тут же его отрисовывает таким, какое оно есть. Если у модели что-то поменялось, то она тут же оповещает свою View (или несколько View), чтобы та отобразила ее в текущем состоянии. К одной модели может быть подсоединено много разных View. Вы можете пытаться отображать данные в разных форматах в разных окнах или вообще принципиально разных способом.

- Контроллер же наоборот управляет моделью по указателю. Точнее, чтобы соединение было надежным это надо делать через handle. В данном случае Model является host. А внутри контроллера содержится handle для управления моделью. У одной модели может быть много контроллеров. Например, один контроллер от клавиатуры, а другой от GUI. Потому тут не подходит Observable/Observer соединение.

- И последняя часть – соединение между View и Controller. Обычно представляют, что это тоже push уведомления по типу Observer/Observable. Почему это так. Ну потому что если пользователь нажал на кнопку экрана, то это возникло какое-то действие, о котором знает View. И значит она должна сообщить теперь об этом Controller. Однако так бывает не всегда. Бывает что источник сигнала не внутри View. Например ввод с клавиатуры.

Когда вы пишите свое интерактивное приложение, вы вряд ли пишите все GUI и Message Driven System-у с нуля сами, вы скорее всего используете готовые решения. А тогда есть подозрения, что ни одна из библиотек не поддерживает возможности соединяться по Observable/Observer соединению, в особенности используемому вами в вашем проекте. Потому полезно эту концептуальную картину немного представить в другом виде.

Другая модель Вот еще чуть более реалистичная картинка.



Здесь виды соединений:

$$\text{Observable}(1) - \succ \text{Observer}(n) \qquad \text{View}(1) \longrightarrow \text{GUI}(1)$$

$$\text{Controller}(n) \longrightarrow\!\!\succ \text{Model}(1) \qquad \text{Message System}(1) \rightsquigarrow \text{Controller}(n)$$

Давайте прокомментируем соединения.

1. Model→View – как и раньше Observable/Observer соединение для поставления данных из модели во View. Соединение от одной модели к многим View.

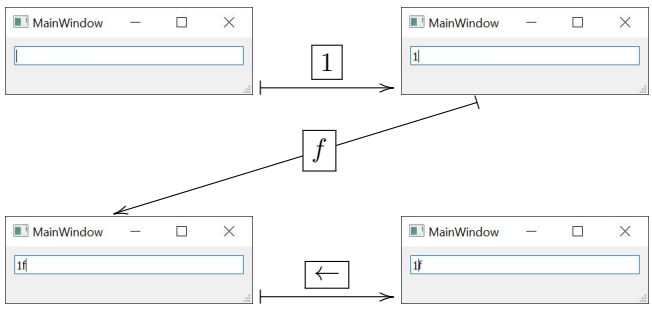
2. Controller→Model – как и раньше соединение по типу Host/Handle для управления моделью. Соединение от нескольких контроллеров к одной модели.

3. View→GUI. Если вы используете готовую GUI библиотеку, то для каждого View объекта вы будете использовать библиотечный код для имплементации GUI. Обычно для одной View вы создаете один объект, представляющей состояние View (состояние графического изображения для пользователя, но НЕ состояние модели). Это обычно один в один соединение.

4. Message System→Controller. В интерактивных приложениях обычно используются Message Driven Systems, которые обсуждались в разделе 12.4. Любые действия пользователя тогда перерабатываются в сообщения системы. А потому контроллер должен прослушивать данные сообщения и после этого совершать нужные действия с моделью. Тут есть такая тонкость. Все ли необходимые для совершения действия данные прилетают внутри сообщения. Бывает так, что не все. А тогда у контроллера должна быть возможность эти данные получить. А это значит, что он может хранить какие-то указатели или лучше handle-ы на графические компоненты, где должны быть данные.

12.7.3 Демонстрация как работает MVC

Давайте вернемся к первой модели в маленьком масштабе из предыдущего примера. Напомню, что у нас есть простое окно с полем для ввода и изменения текста. Предположим, что мы поставили курсор и нажали друг за другом клавиши «1», «f» и «←». Тогда видимое еоведение будет таким



Давайте посмотрим, как такая система должна работать по внутренней логике MVC паттерна. У любой интерактивной программы есть две стадии:

1. Инициализация всех ресурсов для начала исполнения.

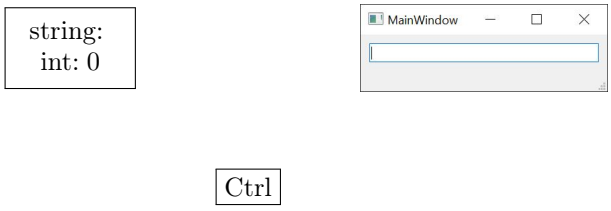
2. Исполнение программы.

Во время первой стадии нам нужно проделать следующее:

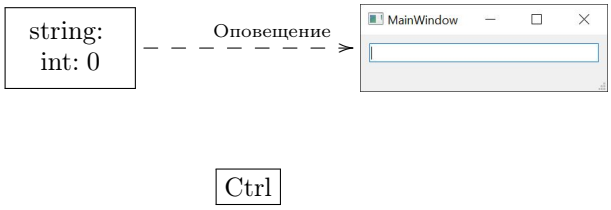
1. Создать Model.
2. Создать View.
3. Создать Controller.
4. Соединить Model→View.
5. Соединить View→Controller.
6. Соединить Controller→Model.

Во время второй стадии собранная система реагирует на сообщения Message Driven System-ы.

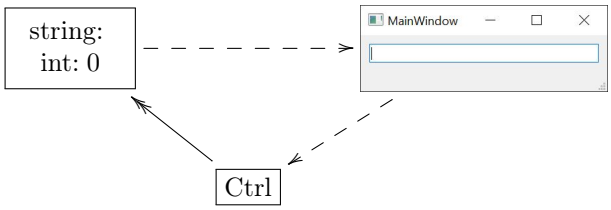
Начнем с создания объектов. В конструкторе нашего приложения мы создаем Model, View и Controller. Логически будем представлять их так.



Обратите внимание, что сейчас у меня показывается окно с пустым полем для ввода и позицией коретки на нуле. Однако, это не очень правильное представление. По хорошему в этом состоянии View просто показывает белое поле без информации. View лишь знает геометрию поля для ввода, но не его содержание. И лишь в момент соединения Model и View, данные от модели поступают во View и она их отрисовывает.



Полностью собранная схема будет выглядеть так

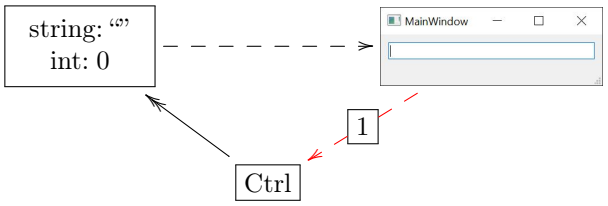


На этом закончилась стадия конструирования. Перейдем к стадии обработки сообщений. На этой стадии мы считаем, что пользователь последовательно выполнил следующие команды

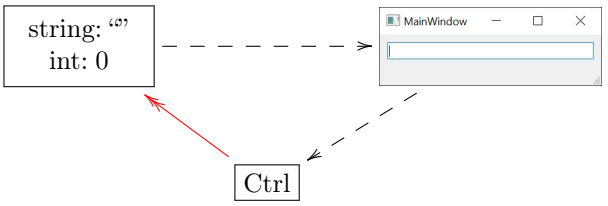
1. Нажата клавиша «1».
2. Нажата клавиша «f».
3. Нажата клавиша «←» (стрелка влево).

Ниже показана поэтапная схема, отражающая, что происходит в MVC контуре. Красным обозначается текущая в данный момент операция.

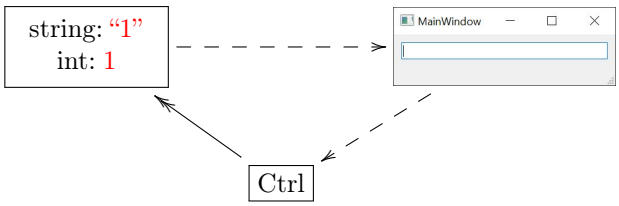
1)



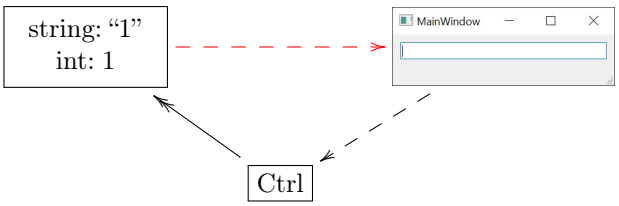
2)



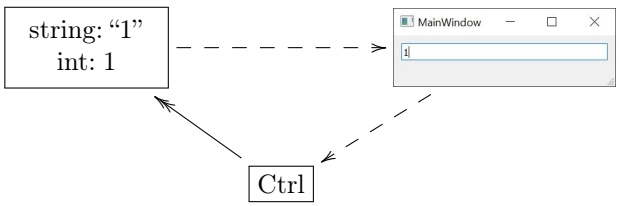
3)



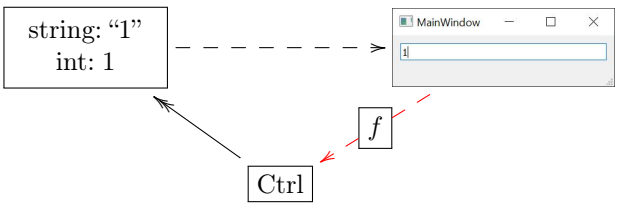
4)



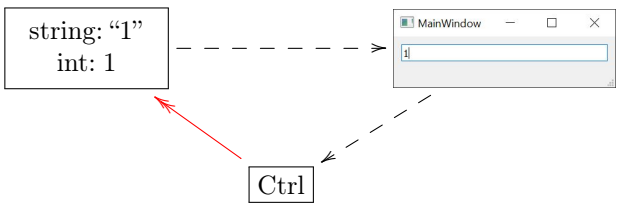
5)



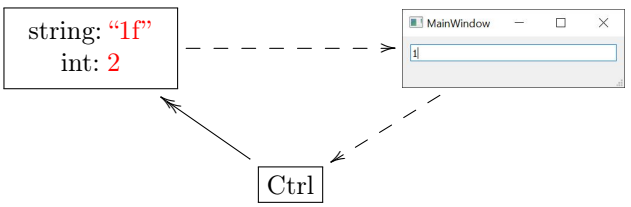
6)



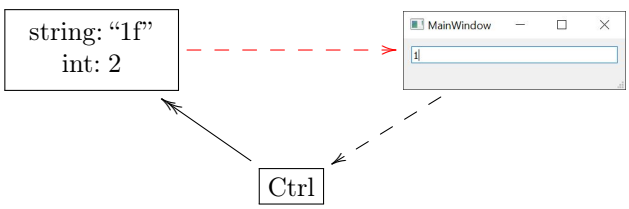
7)



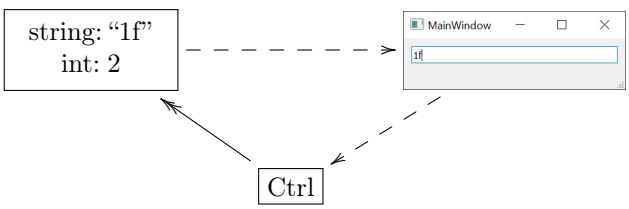
8)



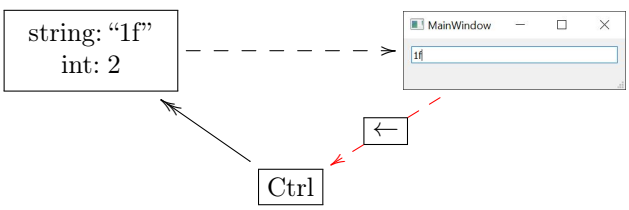
9)



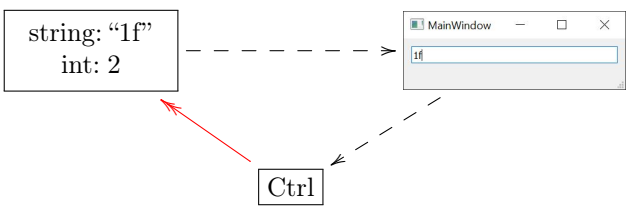
10)



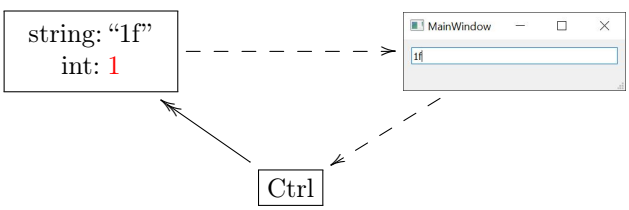
11)



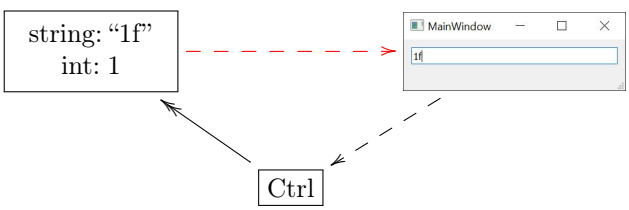
12)



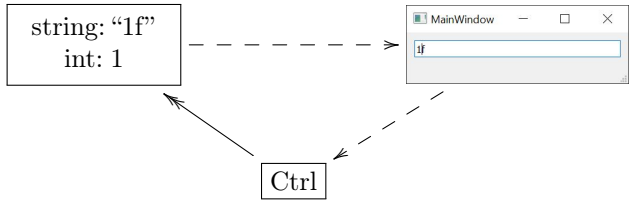
13)



14)



15)



Давайте прокомментируем как это работает.

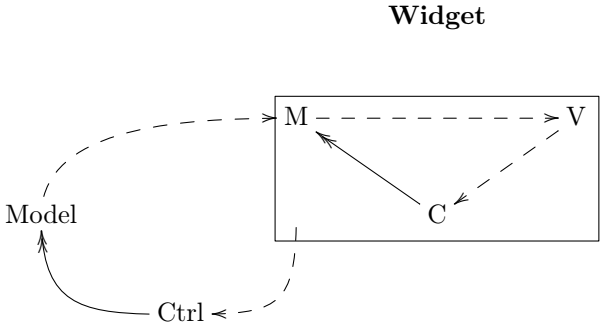
1. После нажатия на клавишу «1» оповещается соответствующий графический компонент, в нашем случае это поле для ввода текста. В этот момент поле ввода для текста НЕ меняется, потому что у него нет внутреннего состояния, оно всегда отражает состояние модели. Вместо этого этот компонент посылает сообщение контроллеру, что нажата клавиша «1».
2. На этом этапе контроллер видит, что клавиша «1» дает символ “1”, который надо поместить в строку внутри модели и сдвинуть коретку на 1 вперед.
3. На этом слайде красным помечено изменение состояния модели.
4. Как только модель изменила свое состояние, она оповещает всех, кто на нее подписан, чтобы они пришли с ней в одно состояние. В частности оповещается поле для ввода текста о новом состоянии модели.
5. И вот на этом шаге поле для ввода текста получив новое состояние модели отражает его в виде строки «1» и положения картки после цифры 1.
6. После нажатия на клавишу «f», оповещается графически элемент – поле для ввода текста. После чего это поле посылает сигнал контроллеру.
7. Контроллер видит, что эта клавиша дает символ. И зовет метод модели, который добавляет этот символ в месте расположения каретки.
8. Здесь красным обозначено новое состояние модели. Теперь хранится строка «1f» и новое положение каретки 2.
9. Теперь модель оповещает текстовое поле о своем новом состоянии.
10. Текстовое поле отражает новое состояние модели. То есть оно отображает строку «1f» и новое положение коретки в конце строки.
11. Теперь после нажатия пользователем клавишы «←» – стрелка влево, оповещается соответствующий графический компонент – поле ввода для текста. Этот компонент посылает сигнал контроллеру.
12. Контроллер видит, что это клавиша отвечает за навигацию по тексту и вызывает у модели метод сдвига коретки влево.
13. Новое положение коретки в модели отмечено красным. Строка не меняется.
14. Теперь модель оповещает всех о своем новом состоянии.
15. И наконец-то текстовое поле отображает текущую строку «1f» и положение коретки между этих двух символов.

Этот пример хорошо показывает общую логику работы MVC. Все данные хранятся в модели, она единственный источник информации. View всегда отражает только то, что ей прилетело от модели и логически не имеет своего состояния. Контроллер же управляет моделью напрямую.

Замечание про Controller В такой схеме может показаться странным, что нам нужно использовать View и Controller как две разные сущности. Потому что все равно все данные из View идут в Controller и потом по этим данным дергается модель. Причина почему выделяется контроллер – мы не хотим, чтобы в GUI коде была бизнес логика управления моделью. Мы не хотим, чтобы GUI зависело от ядра. Мы хотим, чтобы GUI зависило от данных, которые прилетают из ядра по Observer паттерну, но мы не хотим чтобы были какие-либо другие зависимости. Чтобы отрезать GUI код от ядра и вводится контроллер. В этом случае View лишь зависит от двух типов данных: получаемые от модели и отсылаемые контроллеру. View не принимает никаких решений, не умеет управлять моделью, не содержит никакой логики. Вся логика управления уходит в Controller.

12.7.4 MVC и реальность

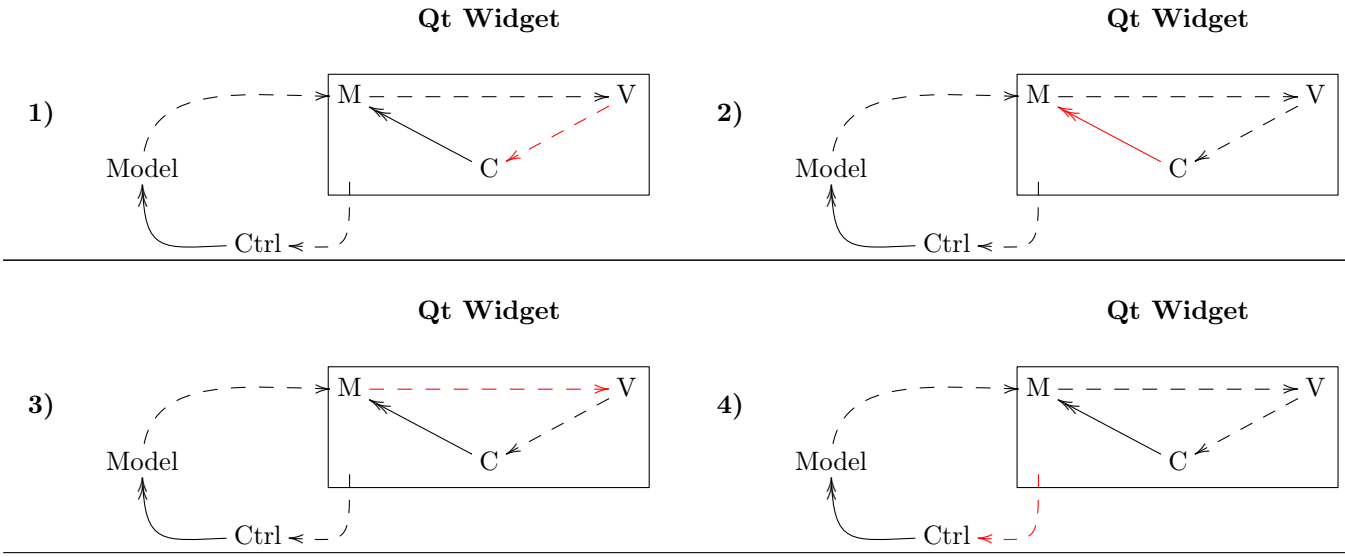
Если вы работаете с готовыми графическими библиотеками, то к сожалению или к счастью, они не предоставляют stateless (без состояния) графических компонент (или не всегда их предоставляют). А потому любой графический компонент или widget в реальной библиотеки сам из себя представляет уже готовый MVC контур запеченый в один объект. И если вы пытаетесь использовать этот widget как View, у вас получится какая-то такая картинка:

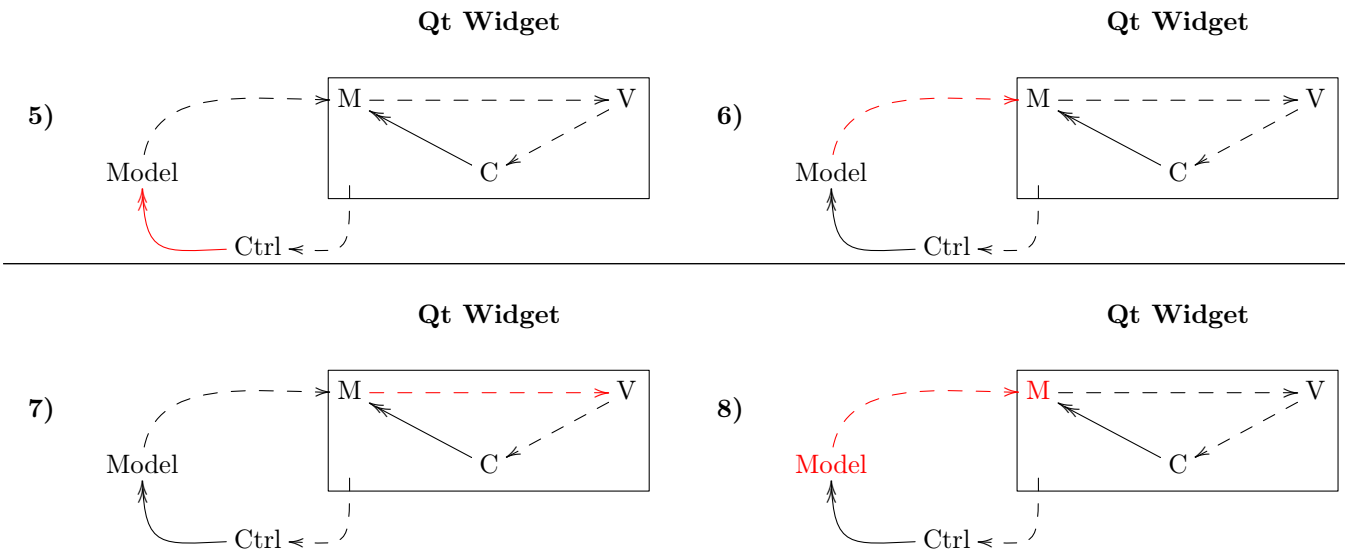


В бокс справа на картинке объединены внутренности Widget. При использовании View, которая имеет состояние, вы по сути имеете две модели, которые вам придется согласовывать. А согласование двух моделей при двустороннем общении – это всегда огромная проблема. У вас возникают следующие проблемы

1. Двойное обновление модели.
2. Гашение обратной связи.

Давайте промоделируем одно оповещение контроллера в схеме представленной выше. Предположим на пользователь совершил какое-то действие, которое заставило View внутри Widget-а оповестить контроллер внутри Widget-а.





Давайте прокомментируем, что тут происходит. Если мы верим, что Widget функционирует как MVC контур (раз он содержит состояние), то мы ожидаем следующее поведение.

1. В начале Widget получает от экосистемы сообщение о действиях пользователя. А именно View объект внутри Widget-а оповещается об этом событии. После чего View объект должен оповестить внутренний контроллер, чтобы обновить свое состояние.

2. Когда внутренний контроллер получает оповещение, он дергает внутреннюю модель, чтобы она обновилась.

3. Когда модель обновилась она совершает два действия. Первое – оно оповещает внутренний View объект, чтобы отразить свое новое состояние.

4. Второе действие – послать сигнал наружним объектам. В данном случае внешнему контроллеру Ctrl.

5. Теперь контроллер оповещает нашу модель Model.

6. Модель меняет свое состояние и оповещает Widget.

7. Widget меняет состояние внутренней модели в соответствии с состоянием, которое пришло от Model. А раз обновилось внутреннее состояние, то надо опять оповестить внутренюю View.

8. Как мы видим, в ребре между двумя моделями оказалось, двойное обновление внутренней модели одними и теми же данными. Из-за чего происходит двойное обновление View внутри Widget. Эта проблема связана с тем, что модель снаружи и внутри Widget общаются в двустороннем порядке.

Поведение описанное выше может быть как желательным, так и не желательным. Например, если мы использовали statefull Widget мы не хотим двойное обновление View объекта, чтобы экран лишний раз не мерцал. Ну и в целом это логически плохо.

Какие есть методы решения этой проблемы.

1. Одно из частных решений – поставить гаситель обратной связи или Suppressor на стороне M. Это значит, что когда мы оповещаем всех, мы перестаем слушать входящие сигналы. К сожалению это работает только для блокирующих вызовов. Кроме того, может быть опасность пропустить нужный сигнал от модели.

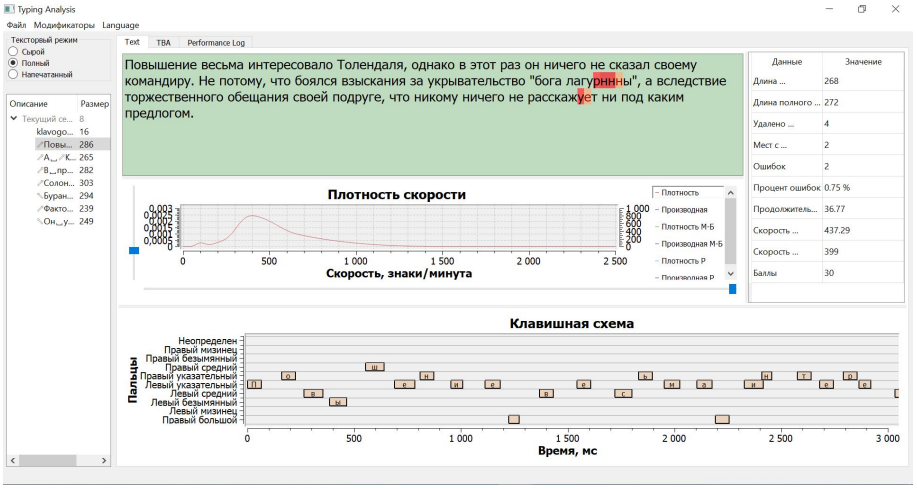
2. Другой подход – поместить некую логику оповещения на стороне Model. То есть теперь Model должна знать кого оповещать а кого нет. Это достаточно сложная задача, потому что контроллеру надо передавать модели данные об отправители и Model теперь должна не просто обновлять свое состояние, но еще и принимать решение кого оповещать. Это опасно рассинхроном состояний.

3. Можно поместить логику обработки оповещений на стороне внутренней модели M. То есть внутренняя модель смотри прилетевшие данные и проверяет отличается ли ее состояние от пришедшего. Это логически самое правильное решение, но тогда нам надо уметь очень быстро считать разницу между состояниями. Иначе этот подход может оказаться очень дорогим для исполнения.

На удивление все становится сильно проще и удобнее, если мы переходим к неблокирующим соединениям между компонентами. В данном случае неблокирующий observer паттерн (можно посмотреть раздел 12.5). В этом случае общение компонент можно рассматривать как общение по сети. И для корректного общения нужен грамотный протокол общения, который решает все проблемы подобного характера. Никакие кустарные заплатки в виде suppressor-ов в нем не потребуются.

12.7.5 Пример 1 использования MVC

Здесь я хочу привести пример программы по анализу качества печати, которую я разрабатывал. Начну со скриншота, чтобы можно было что-то объяснять.

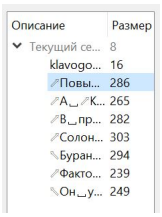


Давайте в общих чертах объясню, что здесь происходит. Данная программа представляет из себя всего одно окно, которое вы видите на скриншоте выше. Программа умеет находиться в двух режимах:

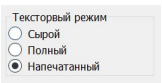
1. Режим перехвата клавиатуры. В свернутом режиме, программа перехватывает весь ввод пользователя.
2. Режим анализа. В развернутом режиме, программе не прослушивает клавиатуру, вместо этого она дает возможность посмотреть информацию о уже записанных данных, которые были перехвачены ранее.

При переходе от прослушивающего состояния в состояние анализа все накопленные данные сгружаются в ядро программы. Вся информация о наборе хранится в виде сессий. Теперь опишем компоненты графического интерфейса.

1. Слева вы видите список сессий, где отражено несколько первых клавиш нажатых в данную сессию, ее длина в количестве нажатых клавиш и какая сессия сейчас активна.



2. Над панелью для выбора сессий находится панель выбора режима для отображения перехваченного набора.



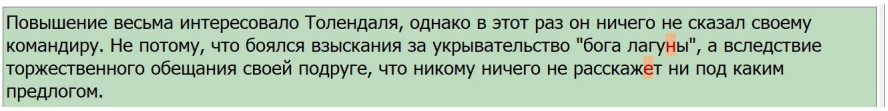
Данный режим влияет на отображение сессии в текстовом окне описанном в пункте 3. Есть три режима

(a) Сырой. В нем отображаются все клавиши которые были нажаты независимо от того дают они символы или нет.

(b) Полный. В этом режиме отображаются только напечатанные символы включая все удаленные. Удаленные символы подсвечиваются специальным образом.

(c) Напечатанный. В этом режиме отображается только конечный текст, который был напечатан без учета стертых символов.

3. Сверху по центру расположено текстовое окно предоставляющее информацию о перехваченной сессии.



4. Ниже идет график плотности скорости.

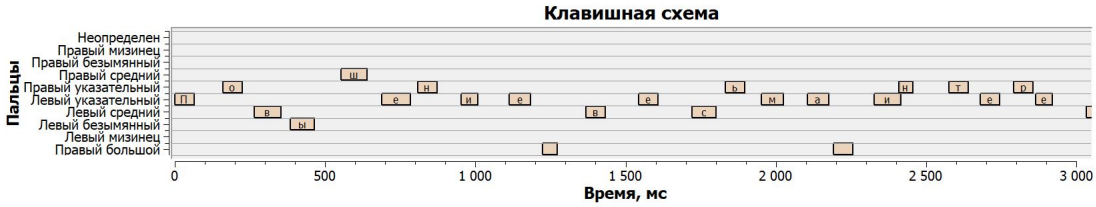
Программа трактует скорость нажатия клавиши как случайную величину. В данном окне приведена плотность распределения данной случайной величины.

5. Справа от текстового окна располагается окно статистики.

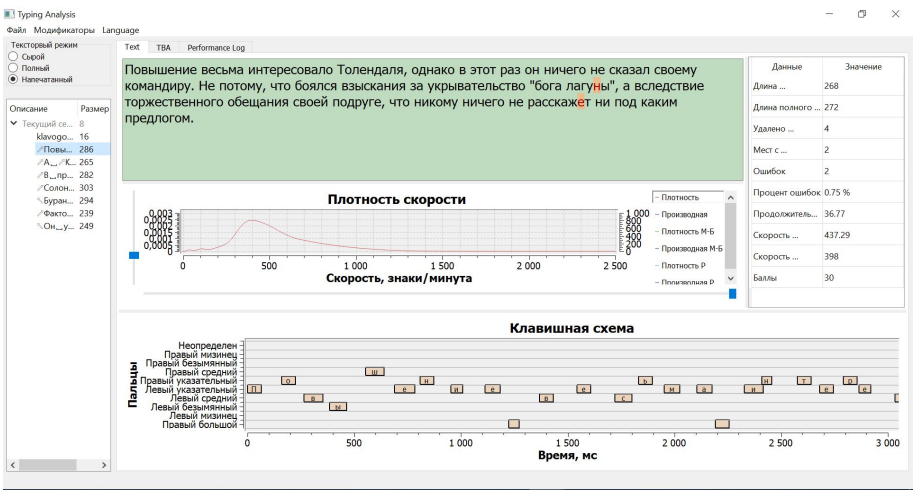
Данные	Значение
Длина ...	268
Длина полного ...	272
Удалено ...	4
Мест с ...	2
Ошибок	2
Процент ошибок	0.75 %
Продолжитель...	36.77
Скорость ...	437.29
Скорость ...	398
Баллы	30

Самый интересный параметр – предпоследний параметр – это пик плотности распределения скорости. Этот параметр хорошо характеризует скорость физических возможностей наборщика.

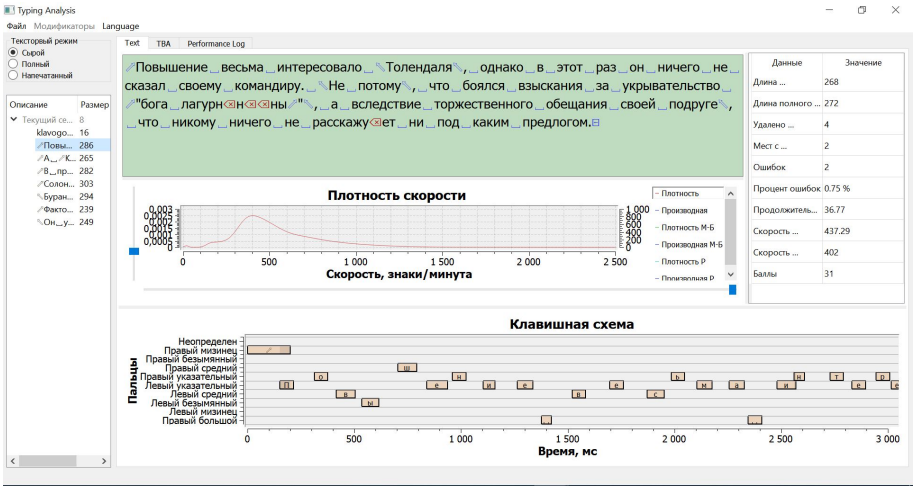
6. В самом низу находится клавишная схема, на которой отображается каким пальцем какая клавиша и в какой временной промежуток была нажата.



Тут стоит сказать, что от выбора текстового режима зависят: текстовое окно, график плотности и клавишная схема. Вот пример как будет выглядеть та же самая сессия а напечатанном режиме.

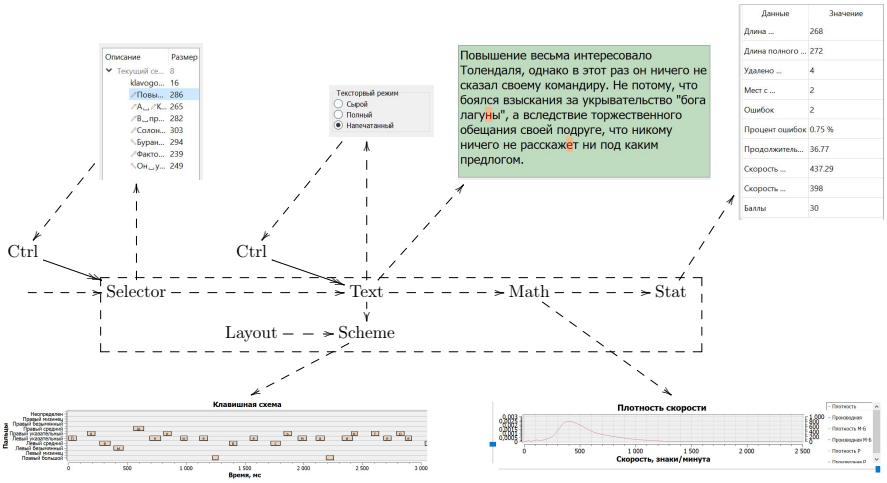


И в сыром режиме



Видно, что в этом режиме появляются символы пробела, символы для backspace-а, для шифтов и прочие.

Давайте я продемонстрирую как выглядит часть приложения с точки зрения MVC.



Давайте я прокомментирую диаграмму выше.

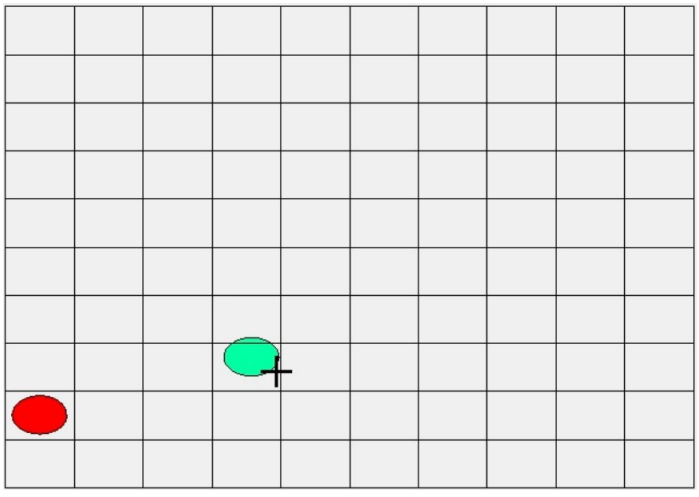
- Часть выделенная пунктиром в центре – это часть ядра программы. Эта часть ничего не знает про графические библиотеки и способна работать без GUI.

- Пунктирная стрелка слева ведет из другой части ядра, которая нам сейчас не очень важна, ибо в той части не используется MVC.
- Ядро из себя представляет пайплайн, по которому данные проходят слева направо и после этого сообщаются пользователю через GUI.
- Блок Selector выбирает текущую сессию для анализа. Можно видеть MVC контур вокруг этого блока. Selector является моделью в этом контуре.
- Блок Text – текстовый модуль. Его задача хранить текущий режим и текст, который необходимо отобразить в текстовом окне. Тут видно, что текстовое окно просто получает новые данные и через него нет возможности управления. А выбор текстового режима – это еще один MVC контур.
- Блок Scheme – модуль, который по распальцовке заданной пользователем и текущему режиму показывает диаграмму распределения нажатий по пальцам во времени. Сама распальцовка хранится в модуле Layout и задается пользователем заранее.
- Блок Math вычисляет плотность распределения скорости набора. Он оповещает View отображающую плотность.
- И в конце идет блок Stat, который является модулем статистики. Он получает всю информацию из всех предыдущих модулей, вычисляет некую статистику и отображает ее в окне для статистики.

На этой диаграмме не отображены некоторые мелочи, как например выбор способа вычисления плотности распределения или локализация текста в элементах GUI. Однако, я надеюсь, что эта схема хотя бы в общих чертах дает понять, как можно собирать подобное интерактивное приложение. Так же отмечу, что все вызовы между компонентами блокирующие. Это не очень хорошо, но это не сложно исправить имея value semantics для данных пересылаемых в observer паттерне. Если вам интересна более детальная информация, то можно изучить репозиторий с проектом. Там можно найти дизайн документацию с более подробным описанием происходящего.

12.7.6 Пример 2 использования MVC

Еще один пример, который я хочу привести – это простая программа, которая показывает пользователю поле с двумя фишками и эти фишки можно двигать только на соседние клетки по вертикали и горизонтали. Больше ничего делать нельзя. Как обычно приведу скриншот в самом начале.



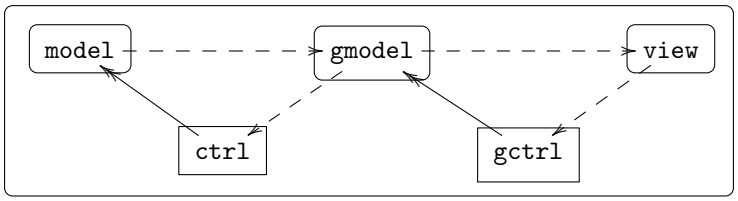
Ожидаемое поведение программы такое. Вы можете подцепить курсором фишку и перемещать ее держа зажатой левую клавишу мыши. В момент перемещения захваченной фишки проигрывается настраиваемая анимация. При отпускании фишки программа пытается поставить ее на указанное поле. Если шаг сделать не возможно, то фишка возвращается в исходное состояние, откуда она была захвачена.

Важно понять, что в данном случае у нас не одна, а две модели.

1. Первая модель знает размер поля, расположение на нем фишек и правила, по которым фишкам можно ходить. Эта модель оперирует целыми координатами фишек, обозначающие столбец и строку, где находится фишка. Она ничего не знает про отображение поля, про фишки, про цвета, анимацию и прочее. Это как бы «внутренняя логика игры».

2. Вторая модель – это геометрическая модель. Дело в том, что в первой модели не достаточно информации для того, чтобы ее отрисовать. Например не ясно каким цветом и как рисовать поле, какие должны быть размеры клеток, какие размеры, цвет и форма фишек. Кроме того, при захвате фишки курсором она может находиться в состоянии, не имеющем смысла для модели игры. Потому для отображения геометрии мы используем вторую модель.

Схематически устройство программы выглядит так.

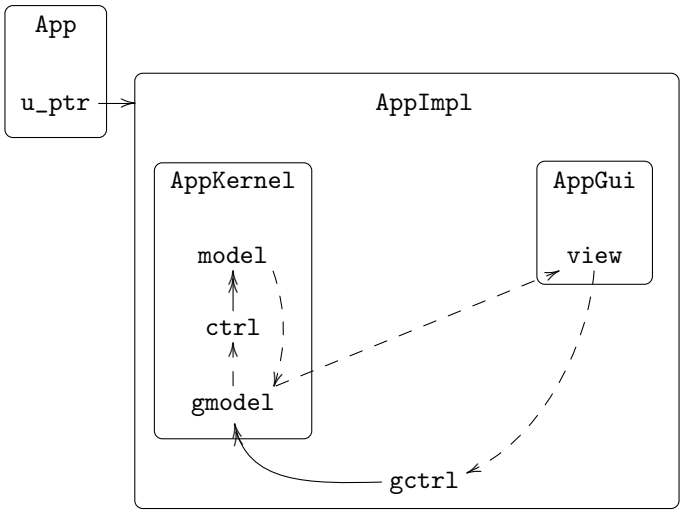


Давайте опишем все ее компоненты.

- model. Это первая модель которая отвечает за
  1. состояние поля
  2. позиции фишек на поле
  3. правила как фишки могут ходить
- gmodel. Это вторая геометрическая модель. Она отвечает за:
  1. Система координат на плоскости.
  2. Изображение поля
  3. Изображение фишек
  4. Анимация фишек при захвате
  5. Информация какая фишка сейчас выбрана и является активной
- view. Это View компонент. Он не содержит состояния и лишь отрисовывает текущее состояние gmodel.
- gctrl. Контроллер управления геометрической моделью. Позволяет
  1. Захватить фишку при зажатии левой клавиши мыши. При этом данная фишка становится активной в геометрической модели.
  2. Двигать гоеметрическое представление активной фишки.
  3. Отпустить захваченную фишку. При этом фишка перестает быть активной.
- ctrl. Это контроллер, который идет от геометрической модели к обычной модели. Когда пользователь перетащил изображение фишки куда-то, гометерическая модель должна попросить обычную модель попытаться сделать предлагаемый ход. И это управление делается через данный контроллер.

Давайте скажу пару слов по поводу поведения в такой схеме. Если пользователь отпускает фишку в геометрической модели, то она оповещает View, чтобы та показала, где фишка остановилась. Но сразу после этого оповещается контроллер модели, который дергает у модели метод, пытающийся сходить фишкой так, как просит геометрическая модель. Далее модель либо ходит фишкой как сказано, либо отвергает действие. Но в любом случае после этого она оповещает всех, кто на нее подписан. В данном случае она оповещает геометрическую модель. Которая при получении новых данных начинает отражать состояние модели, где фишка либо перемещена на новое поле, либо вернулась назад. После чего оповещается View, чтобы отрисовать состояние геометрической модели. Таким образом мы как бы оповещаем View дважды, но это нас устраивает, потому что одно оповещение – это точка сброса фишки, а другое оповещение – это исправленное состояние геометрической модели после попытки сделать ход.

Архитектура приложения Визуально архитектуру приложения можно представлять себе так.



Давайте про комментируем, что тут происходит.

- Объект `App` – это главный объект нашего приложения. Для него используется Pimpl (см. раздел 6.3.18), чтобы разгрузить стек вызовов.
- Вся имплементация спрятана внутрь `AppImpl`. При этом для улучшения локальности мы выделяем в имплементации два раздела посвященных ядру и GUI.
- `AppKernel` – кусок `AppImpl`, в котором находятся компоненты ядра. Технически это базовый класс `AppImple`. В данном случае наследование используется как композиция. Тут живут model, gmodel и ctrl.
- `AppGui` – кусок `AppImpl`, в котором находятся все GUI компоненты. В данном случае это просто view компонента.
- Контроллер gctrl должен соединять GUI и ядро. Потому он располагается в `AppImpl` непосредственно.

В коде это выглядит так.

```
class AppKernel {
  AppKernel()
   : ctrl_(&model_) {
    model_.subscribe(gmodel.port());
    gmodel_.subscribe(ctrl_.port());
  }
protected:
  Model model_;
  GModel gmodel_;
  Ctrl ctrl_;
};
```

```
class MainWindow {
public:
  View* view();
private:
  View view_;
};

class AppGui {
protected:
  MainWindow window_;
};
```

```
1  class AppImpl : AppKernel, AppGui {
2  public:
3    AppImpl() : gctrl_(&gmodel_) {
4      gmodel_.subscribe(view()->port());
5      view()->subscribe(gctrl_.port());
6    }
7  protected:
8    GCtrl gctrl_;
9  };
```

Структура здесь в точности такая как описано выше. Надо лишь прокомментировать как работают конструкторы.

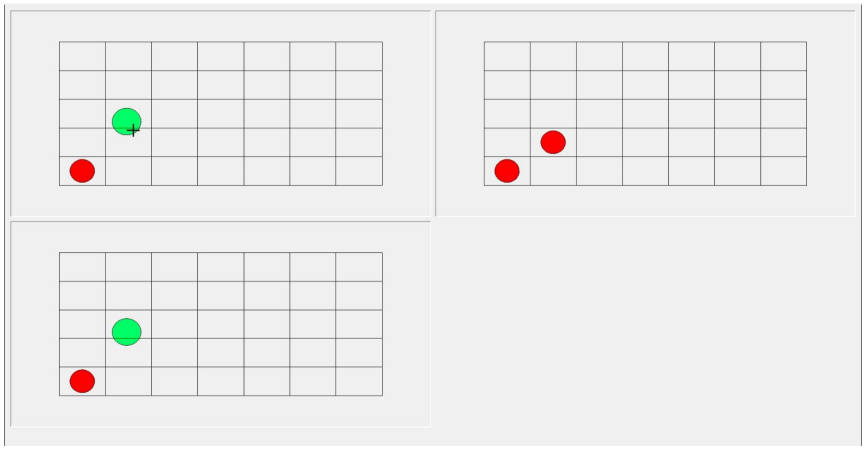
1. `AppKernel` в своем конструкторе соединяет компоненты ядра. Причем в этом дизайне у меня контроллер приваривается к модели в своем конструкторе. А observable/observer порты надо соединить в теле конструктора.

2. `AppGui` содержит одну компоненту – основное окно, в котором располагается `view` объект. Основное окно собирается из Qt кода.

3. Далее все компоненты соединяются в `AppImpl`. С помощью наследования я включаю ядро и GUI часть в имплементацию. А потом в конструкторе делаю все оставшиеся соединения.

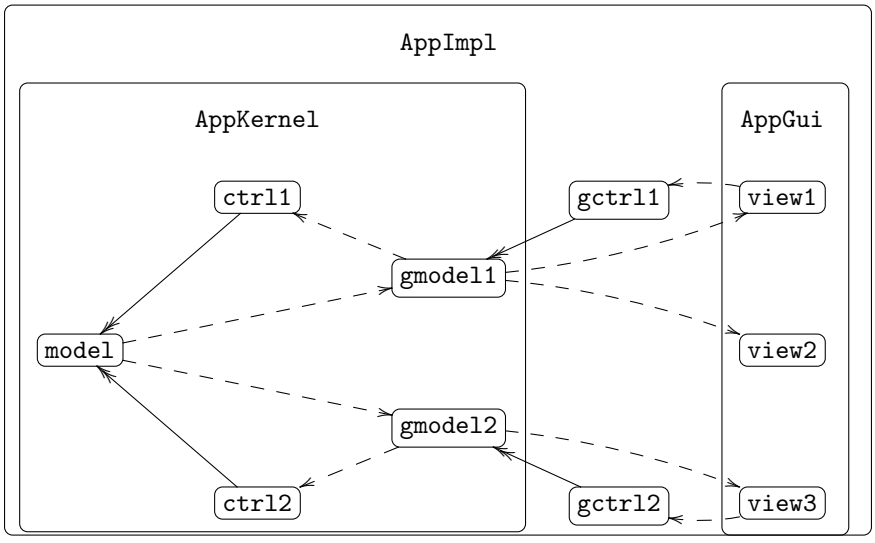
Код приложения можно найти в репозитории.

Другая вариация приложения У этого приложения можно легко сделать вариацию с любым количеством view и геометрических моделей. Очень полезно понимать, что будет происходить при разных вариациях их соединения. Начну как обычно со скриншота.



В этом случае поля слева синхронизированы на уровне геометрической модели. А поле справа имеет свою геометрическую модель. Нижнее поле без контроллера, через него управлять нельзя. Если захватить фишку на левом поле и двигать захваченную фишку по полю, то ее движение будет видно только на двух левых полях. Но если фишку отпустить на клетку допустимого хода, то обновятся все три поля разом. Аналогично, если мы захватим фишку на правом поле, то ее перемещение будет видно только на нем, но когда мы сделаем ход, то обновятся все три поля.

Вот схема соединения объектов.



Эта диаграмма напрямую транслируется в код, потому я не буду его тут приводить.

Точка входа в программу Данное приложение собиралось в экосистеме Qt с использованием не блокирующего Observer Pattern. Вот как выглядит функция `main`

```
#include "Application.h"
#include "Except.h"
#include "QRunTime.h"

int main(int argc, char* argv[]) {
  QApp::QRunTime runtime(argc, argv);
  try {
    QApp::Application app;
    runtime.exec();
  } catch (...) {
    QApp::Except::react();
  }
  return 0;
}
```

Сделаем несколько замечаний:

1. `QRunTime` – это наша надстройка над Qt runtime-ом, которая позволяет общению неблокирующего observer-a. Этот объект в начале функции `main` инициализирует всю экосистему и запускает event loop вызовом функции `exec`.

2. `Application` – это класс нашего приложения. Он является пассивным объектом в Qt экосистеме. Он ничего не выполняет сам, он и его компоненты лишь реагируют на Qt сообщения и посылают свои.

3. Обратите внимание, что функция `main` ничего не делает сама. Она всю работу делегирует другим объектам. Даже обработку исключений.

4. Одна из задач функции `main` – не допустить утекания исключений в OS. Это нужно для того, чтобы отличить ошибки которые возникли в программе из-за самой программы от ошибок, которые возникли в программе из-за операционной системы.

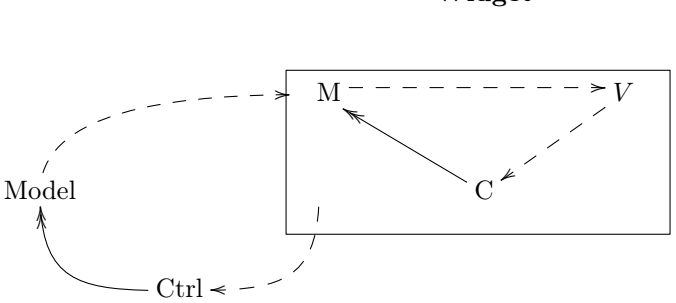
При этом обработка исключений выглядит так

```
#include "Except.h"
#include <QDebug>
#include <exception>

namespace QApp {
namespace Except {
void react() noexcept {
  try {
    throw;
  } catch (std::exception& e) {
    qDebug() << "Exception: " << e.what();
  } catch (...) {
    qDebug() << "Enknown Exception!";
  }
}
} // namespace Except
} // namespace QApp
```

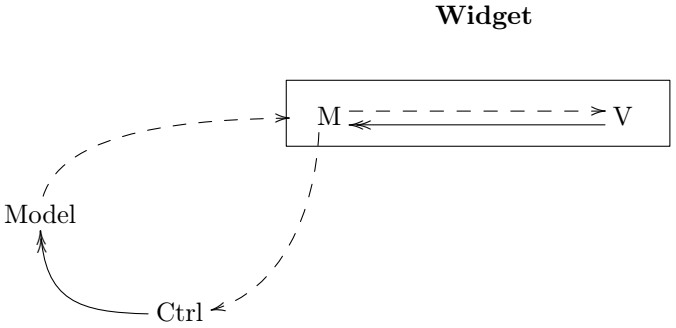
Функция react будет вызвана внутри блока `catch`, а значит в этот момент есть активное исключение. Чтобы его еще раз поймать, мы просто внутри `try` блока делаем `throw` без аргументов. И уже после в разных видах `catch` ловим исключения и реагируем на них. Все это замечательно за одним большим исключением: Qt не разрешает бросать исключения в event loop. Это связано с тем, что Qt изначально не планировал поддерживать исключения. Как мы понимаем сейчас это большая ошибка. Все новые компоненты Qt пишутся в парадигме exception safe. Однако, все ключевые компоненты ядра все еще не поддерживают бросание исключений и это большой геморрой.

12.7.7 MVVM

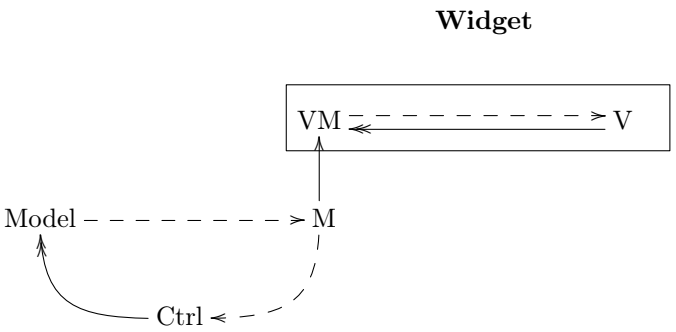
Теперь я хочу поговорить о паттерне Model/View/View-Model. Прежде чем к нему переходить, давайте вспомним наш пример с Widget-ом, который имел внутреннее состояние.



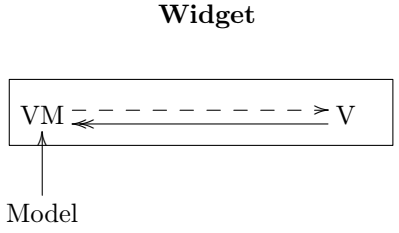
Теперь я хочу помодифицировать чуть-чуть эту конструкцию. Так как внутри Widget-а у нас view оповещает только один контроллер, то в самом начале их можно слить в одну сущность.



Теперь можно вынуть внутреннюю модель наружу, оставив общение с внутренностями Widget-а через фиксированный интерфейс.



Ну а теперь просто нет необходимости во второй модели. И мы получаем.



Давайте я прокомментирую, что тут произошло. Для того, чтобы сделать view stateless нам нужно как-то внутри view получать данные из модели. Понятно, что их можно получать по observer паттерну, но проблема в

том, в каком формате они должны быть. Так вот формат данных для view – это первое что надо фиксировать. Кроме того, если мы хотим управлять произвольной моделью не зная, как она устроена, то нам нужна вспомогательная прослойка, которая на картинке изображена как VM. Это по сути интерфейс для общения с моделью. Это не обязательно интерфейс в смысле наследования с виртуальными функциями. Это может быть стирающий указатель (см. раздел 11.4). В такой модели интерфейс View Model соединяется напрямую с View, потому что ей больше не с кем соединяться. А Model подключается к View Model уже внешне. Такой подход называется Model/View/View-Model. Важно понимать, что он ни в коем случае не отменяет MVC.

13 Обработка ошибок

Очень рекомендую посмотреть доклад Peter Muldoon Exceptions in C++: Better Design Through Analysis of Real World Usage. Значительная часть этого раздела была написана под влиянием этого доклада.

13.1 Виды ошибок и доступные средства

Виды ошибок

1. **Ошибка программиста.** Это bug-и программы, которых в коде быть не должно. Такие ошибки надо исправлять.

2. **Ошибка при обращении к ресурсам среды исполнения.** Например вы пытаетесь выделить память, получить доступ на чтение или запись файла, получаете доступ на прослушивание клавиатуры, получаете доступ к вычислению на видеокарте и так далее. Бывают разные причины, почему такие операции могут быть не успешными. В любом случае это ошибки, которые могут встретиться в программе и которые не связаны с некорректностью кода.

Обработка ошибок Теперь давайте обсудим какие есть механизмы в языке для работы с ошибками. Существуют следующие механизмы:

1. Макрос `assert`. Технически `assert` это макрос, которые ведет себя следующим образом:

   - Если определен макрос `NDEBUG`[29] , то он не дает никакого кода.
   - Если не определен макрос `NDEBUG`, он делает проверку условия внутри макроса и если оно ложно, то валит программу с сообщением об ошибке и указанием строчки кода, на которой все упало.

   Вот как выглядит его использование

```
1  int f(int* x) {
2    assert(x != nullptr && "The argument must not be nullptr!");
3    return *x;
4  }
5
6  int main() {
7    int x = 1;
8    assert(x == 1 && "x must be 1!");
9    return 0;
10 }
```

2. Конструкция `static_assert`. Это по сути то же самое, что и просто `assert`, только данная конструкция выполняет проверки во время компиляции, а не во время исполнения. Все проверки условий известных на этапе компиляции должны выполняться этой функцией, а не просто `assert`-ом. Вот как выглядит его использование

```
1  template<class T>
2  class A {
3    static_assert(std::is_arithmetic_v<T>, "The type T must be arithmetic!");
4    ...
5  };
```

   Строчка в начале класса будет проверяться при компиляции для каждого типа `T`. И если тип `T` не является арифметическим, то компиляция будет падать с ошибкой, где будет указана строчка, где она произошла, и сообщение.

3. Error codes или коды ошибок. Данный подход часто применяется в Си для обработки ошибок. Суть его в том, что мы используем возвращаемое значение функции для указания статуса операции, а для возвращения данных используем out аргументы (см. раздел 8.3 часть «Аргументы ссылки»). Вот как это выглядит

[29] Компилятор от Microsoft автоматически определяет этот макрос в релизе, а gcc и clang нет. Бывает полезно проверить условия в релизе, потому что пути оптимизации компилятора неисповедимы. Тогда в msvc вам надо убрать при компиляции в релизе объявление макроса `NDEBUG`.

```
enum class Status {
  OK, Error
};

Status algorithm(const Data& input, Result* output);

int main() {
  Data data = ...;
  Result result;
  Status status = algorithm(data, &result);
  if (status == Status::OK) {
    ...
  } else {
    ...
  }
}
```

4. `std::expected`. Начиная с C++23 вам наконец-то доступна возможность возвращать из функции желаемое значение или ошибку. По сути это `std::variant<ValueType, ErrorType>` но только с приятным интерфейсом. Суть его в том, чтобы использовать коды ошибок, но при этом и ошибку и результат возвращать через `return`. В коде это выглядит как-то так

```
class Error {
  ...
};

std::expected<Result, Error> algorithm(const Data& data);

int main() {
  Data data = ... ;
  auto result = algorithm(data);
  if (result.has_value()) {
    Result r = std::move(result).value();
    ...
  } else {
    ...
  }
}
```

5. Исключения – Exceptions. Это специальная фича языка, которая позволяет генерировать исключения с помощью ключевого слова `throw`. Перехватывать их можно внутри блока `try` и реагировать на них внутри блока `catch`. Исключения являются опасным механизмом языка, потому что они могут выйти не только из текущего scope, но и в целом подниматься по стеку до тех пор, пока их не поймают или пока исключение не улетит в операционную систему. Из-за исключений вы должны всегда думать, что вы можете выйти из любого scope в любой точке. Из-за этого весь менеджмент ресурсов доложен быть автоматизирован с помощью RAII (см. раздел 6.3.15). В коде это выглядит как-то так

```
Result algorithm(const Data& data) {
  ...
  if (/*some error*/)
    throw std::runtime_error(/*error information*/);
  ...
  return result;
}

int main() {
  Data data = ... ;
  Result result;
  try {
    result = algorithm(data);
  } catch (std::exception& e) {
    /*handle known exceptions*/
  } catch (...) {
    /*handle unknown exceptions*/
  }
  return 0;
}
```

Методы выше можно сгруппировать так:

1. Задача `assert` и `static_assert` уронить программу на первой же ошибке. Программа не должна продолжать работу в некорректном состоянии. Предполагается, что вы как раз ищете ошибку в коде и это должно помочь ее найти. Эти методы для дебага.

2. Все остальные методы: коды ошибок, `std::expected` и исключения, предполагают, что вы хотите как-то отреагировать на ошибку при исполнении. Самое главное, что вы хотите и можете продолжить исполнение дальше (возможно вы все равно хотите выйти по `terminate`, но у вас есть возможность не падать). Эти методы предполагают штатное исполнение программы.

Давайте оценим методы по информативности.

1. `assert` и `static_assert` по умолчанию сообщают вам информацию о строчке кода, на которой произошла ошибка. У вас так же есть возможность сообщить любую другую информацию (хотя у `assert` добавление информации делается костылями). По умолчанию, нет информации о стеке вызовов для `assert` (для `static_assert` это не имеет смысла, так как она вызывается при компиляции, а не при исполнении). Однако начиная с C++23 появилась возможность использовать `std::stacktrace`. Он, правда, не на столько хорошо отображает информацию, как привычные для нас компиляторы.

2. Коды ошибок и `std::expected`. Так как и коды ошибок и тип ошибок внутри `std::expected` пишите вы сами, то тут нет информации, которая будет доступна по умолчанию. Получить информацию о стеке вызовов вы также можете с помощью `std::stacktrace`. Начиная с C++20 информацию о строчке кода, где произошла ошибка можно получить с помощью `std::source_location`.

3. Исключения. По умолчанию класс для исключения пишите вы сами. Потому вы можете поместить в него все ту же самую информацию, которая доступна и другими методами, а именно: `std::stacktrace` и `std::source_location`. Однако, если вы используете дебагер, то при падении на исключении, которое не было поймано, вам дебагер сообщит информацию о стеке вызовов и текущее состояние вашей программы в памяти. Это может быть удобно, но это по сути фича дебагера, а не возможности исключений.

Ожидаемое поведение Давайте поймем, какое поведение от программы мы ожидаем при возникновении ошибок. Можно выделить следующие случаи:

1. Программа падает с сообщением об ошибке.

2. Программа сообщает об ошибке, корректно ее обрабатывает, продолжает находиться в корректном состоянии и корректно работает дальше.

Первое поведение хорошо подходит для разработки приложения, потому что вы хотите найти ошибку как можно раньше, упасть как можно ближе к ней, чтобы найти ее и исправить. Нет необходимости работать дальше. Второе же поведение желательно на машине пользователя. Какой бы ни была ошибка, пользователь не ждет, что программа упадет. В худшем случае он ждет информации, что что-то пошло не так. В связи с этим `assert` и `static_assert` подходят только для первого поведения и совсем не подходят для второго. Их суть ровно в том, чтобы уронить программу как можно раньше. В свою очередь коды ошибок, `std::expected` и исключения технически могут быть использованы для обеих ситуаций. Это может быть не очень разумно, но никто вам не мешает это сделать.

13.2 Доступные инструменты

Давайте поговорим об инструментах, которые позволяют искать ошибки программиста. Тут обычно вспоминают два:

1. Дебагер. Это специальная программа, которая позволяет позволяет следить за выполнением программы. При этом вам обычно доступна следующая информация

   (a) Текущий стек вызовов

   (b) Текущие переменные и присвоенные им значения. При этом доступна информация о типе переменной и ее адрес (если это класс, то какие поля у переменной и их значения).

   (c) Текущая команда в программе.

При этом дебагер позволяет поставить точку в коде на которой вы хотите остановить исполнение. Позволяет делать пошаговое исполнение с возможностью проваливаться внутрь функций или выполнять их как единое действие.

2. Санитайзеры. Сразу оговорюсь это все инструменты доступные под Linux и MacOS и не доступные под Windows. Санитайзеры – это специальные внешние инструменты, которые следят за исполнением программы и выявляют в ней ошибки. Например для проверки корректности работы с памятью, вся выделяемая память в вашей программе имеет специальные служебные блоки памяти вокруг каждого вызова `new`. И если ваша программа обращается к служебным блокам, значит вы не верно обратились к данным.

Хочу сказать несколько важных вещей по поводу этих инструментов. Давайте посмотрим на диаграмму экосистемы C++ еще разок (см. раздел 3). Обратите внимание, что дебагер и санитайзеры находятся в самом конце пайплайна по работе с кодом. То есть дебагер – это ваша последняя линия обороны. На этом этапе вы работаете напрямую с бинарным кодом. И для того, чтобы дебагер мог делать свою работу, вам приходится компилировать программу в специальном режиме, в котором становятся доступны имена и типы объектов, расположенных в памяти. Обычно убираются разные оптимизации, чтобы бинарный код лучше сопоставлялся с исходным кодом. Как бы то ни было, если вы пользуетесь только дебагером и санитайзерами – вы совершаете большую ошибку, потому что на уровне исходного кода у вас куда больше возможностей предотвратить свои ошибки и получить помощь со стороны компилятора.

13.3 Информация об ошибке

Информация об ошибке нужна для того, чтобы выявить и исправить проблему. Это может быть как ошибка на стороне программы, так и на стороне среды исполнения. В идеале следующая информация должна быть доступна:

1. Текстовое описание ошибки. Технически эту информацию можно хранить в `std::string`. Данная информация должна быть модифицируемой, чтобы вы могли добавлять информацию об ошибке, при передаче ошибки при раскручивании стека.

2. Строчка в коде, на которой произошла ошибка. Для получения этой информации с C++20 можно использовать `std::source_location`. Эта информация должна быть константной.

3. Состояние стека вызовов во время произошедшей ошибки. Для получения этой информации с C++23 можно использовать `std::stacktrace`. Эта информация должна быть константной.

4. Какая-то специфическая для этой ошибки информация. Это произвольный тип, который настраивается под каждую ошибку. Данная информация должна быть модифицируемой, чтобы вы могли добавлять информацию об ошибке, при передаче ошибки при раскручивании стека.

Независимо от метода, который вы используете и которые описаны ниже, независимо от того ошибка это программиста или ошибка исполнения среды, вам всегда достаточно информации выше. При бывает, что можно обойтись меньшим количеством информации, например, в простых `assert`-ах.

Как собрать информацию о строке кода Давайте начнем с простого примера.

```
#include <source_location>

void f() {
  std::cout << std::source_location::current() << '\n';
}

int main() {
  f();
  return 0;
}
```

Где для вывода информации о точке кода делается с помощью следующего оператора:

```
std::ostream& operator<<(std::ostream& out, const std::source_location& location) {
  out << location.file_name() << "(" << location.line() << ":"
      << location.column() << "), function '" << location.function_name()
```

```
      << "'";
  return out;
}
```

Вывод будет выглядеть как-то так

```
main.cpp(4:38), function 'void __cdecl f(void)'
```

То есть функция `current` вызвана в строчке 4 и ее имя находится в столбце 38. А что если мы теперь хотим автоматизировать печать информации и завернуть ее во вспомогательную функцию. Тогда можно было бы попытаться написать так

```
#include <source_location>

void print_location() {
  std::cout << std::source_location::current() << '\n';
}

void f() {
  print_location();
}

int main() {
  f();
  return 0;
}
```

Но в этом случае будет показана информация о 4-ой строке, а не о 8-ой. Другой вариант передавать эту информацию явно:

```
#include <source_location>

void print_location(const  std::source_location& location) {
  std::cout << location << '\n';
}

void f() {
  print_location(std::source_location::current());
}

int main() {
  f();
  return 0;
}
```

Но это ничуть не лучше, потому что все равно надо писать вызов функции `current`. Завернуть этот вызов можно только в макрос, иначе она покажет точку в функции обертке, а не где мы ее вызываем. Макрос – плохая идея, потому что вы вводите глобальное имя, которое может потенциально конфликтовать с чем угодно. А для избежания конфликта, вам все равно придется давать длинное имя макросу. В этом случае есть следующий трюк – аргументы по умолчанию.

```
#include <source_location>

void print_location(const  std::source_location& location = std::source_location::current()) {
  std::cout << location << '\n';
}

void f() {
  print_location();
}

int main() {
  f();
  return 0;
}
```

Теперь вывод будет как надо. В этом случае в строчке 8 при вызове функции `print_location` логически аргументы по умолчанию считаются в этой строчке до вызова функции. Вывод будет:

```
main.cpp(8:3), function 'void __cdecl f(void)'
```

Как собрать информацию о состоянии стека В этом случае работают все те же правила и трюки, что использовались в случае информации о точке исполнения в коде. Единственное – нам надо научиться печатать информацию о стеке. Начнем опять с простого примера.

```
#include <stacktrace>

void f() {
  std::cout << std::stacktrace::current();
}

int main() {
  f();
  return 0;
}
```

В этом случае оператор вывода для состояния стека определяется так:

```
std::ostream& operator<<(std::ostream& out, const std::stacktrace& backtrace) {
  for (auto iter = backtrace.begin(); iter != (backtrace.end()); ++iter) {
    out << iter->source_file() << "(" << iter->source_line()
        << "):" << iter->description() << '\n';
  }
  return out;
}
```

Вывод будет выглядеть следующим образом:

```
PathToProject\main.cpp(5599):Project!f+0x29
PathToProject\main.cpp(5682):Project!main+0xB
..\exe_common.inl(79):Project!invoke_main+0x39
..\exe_common.inl(288):Project!__scrt_common_main_seh+0x132
..\exe_common.inl(331):Project!__scrt_common_main+0xE
..\exe_main.cpp(17):Project!mainCRTStartup+0xE
(0):KERNEL32!BaseThreadInitThunk+0x14
(0):ntdll!RtlUserThreadStart+0x21
```

Обратите внимание, что нижние 6 строк о состоянии стека – это вызов системных функций и лишь верхние две строчки показывают стек вызова для вашей программы. Если вы хотите убрать эти последние 6 строк, то можно сделать так:

```
std::ostream& operator<<(std::ostream& out, const std::stacktrace& backtrace) {
  static constexpr int bound = 6;
  if (backtrace.size() < bound)
    return out;
  for (auto iter = backtrace.begin(); iter != (backtrace.end() - bound); ++iter) {
    out << iter->source_file() << "(" << iter->source_line()
        << "):" << iter->description() << '\n';
  }
  return out;
}
```

Что превратит вывод в

```
PathToProject\main.cpp(5599):Project!f+0x29
PathToProject\main.cpp(5682):Project!main+0xB
```

Получилось сильно лучше. Если вы хотите автоматизировать сбор информации о стке, то надо так же воспользоваться аргументами по умолчанию.

```
#include <stacktrace>

void print_stack(const std::stacktrace& stack = std::stacktrace::current()) {
  std::cout << std::stacktrace::current();
}

void f() {
  print_stack();
}

int main() {
  f();
```

```
13    return 0;
14  }
```

13.4 Ошибки программиста

13.4.1 assert

Как использовать К этому макросу можно относиться как к автоматически комментируемым проверкам. Работает это так

```
1  void f(int* x) {
2    assert(x != nullptr);
3    *x = 42;
4  }
```

Если вы хотите проверить preconditions для некоторой функции `f`, которая ожидает не нулевой указатель, то можно поставить в начале тела функции проверку, что указатель `x` не нулевой. При компиляции с макросом `NDEBUG` вторая строчка не даст никакого кода. А без этого макроса выполняется проверка условия внутри и если оно истинно, то исполнение переходит дальше к строке три. Если же оно ложно, то программа падает с сообщением на какой строчке кода она упала. Сообщение об ошибке пишется в `stderr`. Чтобы руками добавить сообщение к `assert` можно писать в поток `std::cerr`.

По какой-то странной причине в отличие от `static_assert` макрос `assert` не поддерживает второй аргумент – сообщение об ошибке. Потому приходится прибегать к следующему трюку.

```
1  void f(int* x) {
2    assert(x != nullptr && "The argument x must not be nullptr!");
3    *x = 42;
4  }
```

В этом случае мы передаем адрес статической константы (который всегда не ноль) и он конвертируется к `true` и потом делается логическое «и» с проверяемым условием.

Информация о стеке вызовов Макрос `assert` печатает строчку на которой он упал. Потому если строчка содержит текстовую константу, то вы увидите ее содержимое. Например

```
1  #include <cassert>
2
3  int main() {
4    int x = 0;
5    assert(x != 0 && "x must not be 0!");
6    return 0;
7  }
```

Вывод будет:

```
1  Assertion failed: x != 0 && "x must not be 0!", file PathToProject\main.cpp, line 5
```

Если же вы хотите добавить неконстантную информацию, то вам нужно писать в `std::cerr` перед вызовом `assert`. Давайте покажем, как это можно сделать на примере добавления информации о стеке вызовов. Введем вспомогательную функцию

```
1  bool stack_state(const std::stacktrace& backtrace = std::stacktrace::current()) {
2    std::cerr << "Call Stack:\n" << backtrace << '\n';
3    return true;
4  }
```

Теперь можно написать следующее:

```
1  #include <cassert>
2
3  int main() {
4    int x = 0;
5    assert(stack_state() && x != 0 && "x must not be 0!");
6    return 0;
7  }
```

Обратите внимание, что вызов функции `stack_state` должен быть самым первым из-за ленивых вычислений в логических операторах. Если первое условие ложно в коньюнкции, то второе даже не вычисляется. Так же я размещаю вызов этой функции внутри `assert`, чтобы при отключении `assert`-ов, эта функция не вызывалась. В этом случае вывод будет:

```
Call Stack:
PathToProject\main.cpp(5):Project!main+0x25

Assertion failed: stack_state() && x != 0 && "x must not be 0!",
file PathToProject\main.cpp, line 5
```

Аналогично можно добавлять любую информацию к `assert`. Информацию о строке кода добавлять не надо, так как она и так выводится.

Рекомендации Мои рекомендации по использованию `assert` кратко выражаются так: чем больше, тем лучше. Чем ближе к багу программа упадет на ошибке, тем быстрее вы его найдете и поправите. С `assert`-ами вам скорее всего придется забыть про использование дебагера, так как не понадобится. Вот пара мыслей об использовании:

1. Ставьте `assert` в начале каждой функции, чтобы проверить preconditions. Например у оператора деления второй аргумент должен быть не ноль. Причем это не ошибка исполнения, это программист должен был проверить этот случай перед вызовом деления. То есть программист не предусмотрел валидный случай в программе, на котором падает текущее исполнение.

```
class Int {
  ...
};

Int operator/(Int first, Int second) {
  assert(second != 0 && "The second argument of operator/ must be non-zero!");
    return first.value / second.value;
}
```

2. Ставьте `assert` в конце каждой мутирующей функции для проверки postcondition. Вдруг вы где-то ошиблись с имплементацией.

```
template<class T>
class Tree {
  using Node = Node<T>;
public:
  ...
  void add_value(T value) {
    /*place the value in the tree*/
    assert(is_correct());
  }
private:
  bool is_correct() const:
  Node* root_ = nullptr;
};
```

3. Все конструкторы класса должны проверять не только preconditions для аргументов, но и инварианты класса. Выделите в классе приватные функции по проверке инвариантов и заворачивайте их внутрь `assert`.

```
template<class T>
class Tree {
  using Node = Node<T>;
public:
  ...
  void add_value(T value) {
    /*place the value in the tree*/
    assert(is_correct());
  }
private:
  bool is_correct() const {
    return is_correct(root_);
```

```
  }
  static bool is_correct(Node* node) {
    if (node == nullptr)
      return true;
    bool is_node_good = /*do checks for node*/;
    return is_node_good && is_correct(node->left) && is_correct(node->right);
  }
  Node* root_ = nullptr;
};
```

4. Не пишите слишком длинные и сложные выражения внутри `assert`, вы иначе никогда не поймете, что вы проверяете. Выделите отдельные вспомогательные функции с говорящими названиями. Например

Плохо

```
void f(int x, int y, int z) {
  assert(x > y && y <= z + x &&
         (x & !y) == 1 && "Ups!");
  /*function code*/
}
```

Получше

```
void f(int x, int y, int z) {
  assert(is_condition_true(x, y, z) &&
         "Ups!");
  /*function code*/
}
```

Вы конечно можете добавить комментарий и сообщение об ошибке. Но все же имя функции скажет больше, чем комментарий. Только важно в функциях, которые вызываются внутри `assert` не делать проверок с помощью `assert`, а то вы можете уйти в бесконечную рекурсию вызовов `assert`.

5. Обязательно вставляйте `assert` перед обращением через указатель, он может быть нулевой.

```
void f(A* a, B* b) {
  assert(a != nullptr && "a must not be nullptr!");
  assert(b != nullptr && "b must not be nullptr!");
  b->consume(a->produce());
}
```

Обратите внимание, что две разные проверки разнесены в разные `assert`-ы, чтобы при падении было понятно на какое именно условие вы упали.

Если в классе перегружен оператор `operator->()`, то соответствующую проверку можно выполнить внутри него. Однако, для указателей нельзя этот оператор перегрузить, а потому вам придется делать эту проверку заранее. В случае же класса и перегруженного оператора `operator->()` проверка может выглядеть так.

```
class A {
public:
  ...
  I* operator->() const {
    assert(ptr_ != nullptr && "The object has invalid pointer!");
    return ptr_;
  }
private:
  I* ptr_ = nullptr;
};
```

В данном примере мы можем спокойно звать

```
int main() {
  A a;
  a->f();
  return 0;
}
```

Если в объекте `a` хранится нулевой указатель, то в третьей строчке все упадет на проверке по `assert`.

6. НЕ делайте `assert` на ошибки, которые разрешимы на этапе компиляции (например в шаблонной магии), для этого используйте `static_assert`. Следующий пример возможно не самый лучший, тут можно было бы воспользоваться концептами или `std::enable_if` (если вы динозвар из прошлого), но он хорошо иллюстрирует проблему.

Плохо

```
template<class T>
void f(T x) {
  assert(std::is_arithmetic_v<T> &&
         "The type must be arithmetic!");
  /*function inplementation*/
}
```

Получше

```
template<class T>
void f(T x) {
  static_assert(std::is_arithmetic_v<T> &&
         "The type must be arithmetic!");
  /*function inplementation*/
}
```

7. И конечно же не забывайте сообщить об ошибке текстом или добавить информацию о стеке вызовов или любую другую полезную для дебага информацию. Вам же будет проще.

Плохо

```
#include <cassert>

int main() {
  int x = 1;
  assert(x == 1);
  return 0;
}
```

Получше

```
#include <cassert>

int main() {
  int x = 1;
  assert(x == 1 && "x must be 1!");
  return 0;
}
```

13.4.2 Кастомные assert-ы

Главный недостаток `assert` заключается в том, что макрос `NDEBUG` отключает сразу все проверки во всех компилируемых файлах. Если вам нужно выборочное отключение проверок, то можно сделать свои кастомные макросы на основе `assert` и управлять включением и отключением проверок с помощью дополнительных макросов. Такое может понадобиться, если вы захотите запустить часть недорогих проверок (как например проверка указателя на `nullptr` перед разыменованием) в релизе (который идет в прод) и отключить все дорогие проверки. Вот как могут выглядеть кастомные `assert`-ы.

```
1  // file custom_asserts.h
2  #include <cassert>
3
4  #ifndef DISABLE_ASSERTS
5  #define DISABLE_ASSERTS 3
6  #endif
7
8  #if DISABLE_ASSERTS < 0
9  #define ASSERT0(expr, msg) ((void)0)
10 #else
11 #define ASSERT0(expr, msg) assert(expr && msg)
12 #endif
13
14 #if DISABLE_ASSERTS < 1
15 #define ASSERT1(expr, msg) ((void)0)
16 #else
17 #define ASSERT1(expr, msg) assert(expr && msg)
18 #endif
19
20 #if DISABLE_ASSERTS < 2
21 #define ASSERT2(expr, msg) ((void)0)
22 #else
23 #define ASSERT2(expr, msg) assert(expr && msg)
24
25 #endif
26
27 #if DISABLE_ASSERTS < 3
28 #define ASSERT3(expr, msg) ((void)0)
29 #else
30 #define ASSERT3(expr, msg) assert(expr && msg)
31 #endif
```

В этом примере мы вводим 4 вида `assert`-ов:

- ASSERT0
- ASSERT1
- ASSERT2
- ASSERT3

Думать про них надо так: чем номер меньше, тем важнее `assert`. Макрос `DISABLE_ASSERTS` отключает `assert`-ы с уровнем выше заданного. То есть `-1` отключает все сообщения об ошибках, `0` оставляет только самые важные `ASSERT0` и так далее. Используется это так

```
#include "custom_asserts.h"

int main() {
  int x = 0;
  ASSERT0(x == 0, "x must be 0!");
  int y = 1;
  ASSERT1(y == 1, "y must be 1!");
  int z = 2;
  ASSERT2(z == 2, "z must be 2!");
  return 0;
}
```

При необходимости отключить `assert`-ы после уровня `0` нужно сделать так:

```
#define DISABLE_ASSERTS 0
#include "custom_asserts.h"

int main() {
  int x = 0;
  ASSERT0(x == 0, "x must be 0!");
  int y = 1;
  ASSERT1(y == 1, "y must be 1!"); // is disabled
  int z = 2;
  ASSERT2(z == 2, "z must be 2!"); // is disabled
  return 0;
}
```

Так же можно сделать `assert`-ы не по уровням, а по управляемому макросу. Например, вы хотите сделать проверки для конкретного класса или для конкретного модуля в коде.

```
// file my_assert.h
#include <cassert>

#ifndef DISALBE_MY_ASSERT
#define MY_ASSERT(expr, msg) assert(expr && msg)
#else
#define MY_ASSERT() ((void)0)
#endif
```

Тогда используется это так

```
#include "my_assert"

int main() {
  int x = 1;
  MY_ASSERT(x == 1, "x must be 1!");
  int y = 2;
  ASSERT0(y == 2, "y must be 2!");
  return 0;
}
```

И для отключения нужно добавить макрос отключения

```
#define DISALBE_MY_ASSERT
#include "my_assert.h"

int main() {
  int x = 1;
  MY_ASSERT(x == 1, "x must be 1!"); // is disabled
  int y = 2;
  ASSERT0(y == 2, "y must be 2!");
  return 0;
}
```

Так же можно сделать свитч между тремя опциями: `assert`, исключение или ничего. Например так

```
// file my_check.h
#include <cassert>
#include <exception>

#ifdef MYASSERT
#define MyCheck(expr, msg) assert(expr && msg)
#else
#ifdef MYEXCEPT
#define MyCheck(expr, msg)       \
  if (!expr)                     \
  throw std::runtime_error(msg)
#else
#define MyCheck(expr, msg) ((void)0)
#endif
#endif
```

Теперь этот код можно использовать в трех случаях:

Без проверок

```
#include "my_check.h"

int main() {
  int x = 1;
  MyCheck(x == 1, "Ups!");
}
```

Проверка через `assert`

```
#define MYASSERT
#include "my_check.h"

int main() {
  int x = 1;
  MyCheck(x == 1, "Ups!");
}
```

Бросаем исключение

```
#define MYEXCEPT
#include "my_check.h"

int main() {
  int x = 1;
  MyCheck(x == 1, "Ups!");
}
```

13.4.3 Exceptions

Если вы хотите проверять ошибки программиста с помощью исключений, то вам надо валить вашу программу соответствующим исключением. То есть надо кидать исключение и не перехватывать его. Тогда в режиме дебага вы сразу увидите текущее состояние стека вызова и состояние программы в памяти при исполнении. Наличие стека вызовов позволит лучше понять происхождение ошибки. Однако при таком подходе вам не доступна информация лежащая в самом исключении.

Так же не пытайтесь такие ошибки обрабатывать в блоках `catch`. Иначе вы получите управление логикой программы с помощью исключений. Лучше завести какой-то служебный класс для таких ошибок и в перехватчике исключений его выбрасывать из программы. Например так (см. также самый конец раздела 12.7.6):

```
// file react.h
struct Die {};
void react();

// file react.cpp
void react() {
  try {
    throw;
  } catch (std::exception& e) {
    /*react to standard exception*/
  } catch (Die& die) {
    throw; // rethrow die exception
  } catch (...) {
    /*react to unknown exception*/
  }
}
```

```
#include "react.h"

namespace {
void f() {
  throw Die{};
}
}

int main() {
  try {
    f();
  } catch(...) {
    react();
  }
  return 0;
}
```

Обычно дебагер позволит вам не только увидеть стек вызовов, но и прыгнуть в строчку кода, где возникло исключение. Так же из плюсов вы увидите текущее состояние всех переменных на стеке.

13.5 Ошибки исполнения среды

Ошибки исполнения среды – это ошибки, которые возникают не из-за багов в программе, а из-за того, что есть какие-то внешние проблемы. В основном это проблемы с получением доступа к ресурсу среды. Вот примеры таких ошибок:

1. Memory Corruption. Нарушение целостности памяти. Это фатальная ошибка, с которой не возможно бороться и нет смысла бороться. Тем не менее, это ошибка исполнения среды.

2. Невозможно открыть файл на чтение или запись. Это по сути отсутствие доступа к ресурсу. Такие ошибки могут быть как фатальными (без этого файла не возможно исполнение), так и не фатальными (вы знаете, что делать и можете работать дальше, без доступа к этому файлу).

3. Нет запрашиваемого устройства. Это по сути отсутствие доступа к ресурсу. Такая ошибка может быть как фатальной, так и не фатально. Все зависит от вашего приложения и устройства.

4. Процессор не поддерживает нужный вид инструкций. Это тоже по сути доступ к ресурсу. Вы не можете получить доступ к операциям на процессоре. Обычно это не фатальная проблема, вы просто откатываетесь к имплементации, которая не использует недоступные операции.

5. Ошибка при парсинге файла. Вам предоставили не корректный ресурс.

Ошибки связанные с целостностью памяти мы просто игнорируем. Если вашей системе битая память, то вам надо менять железо, а не латать программу. Конечно мы можем поговорить на тему «программируй как NASA», но все же это слишком специальный случай.

Ошибки при исполнении можно классифицировать следующим образом:

1. Фатальная ошибка, которая вызывает завершение процесса по `std::terminate`

2. Фатальная ошибка, которая вызывает перезапуск event loop на процессе, но сам процесс не умирает.

3. Локальная ошибка при исполнении, например нет доступа к файлу или некорректное содержимое файла.

Под первые два вида ошибок хорошо подходят исключения, потому что они позволяют быстро умереть или быстро подняться по стеку (при этом параллельно накапливая информацию об ошибках и/или логируя их). Последний подход плохо решается исключениями. Если их решать исключениями, то вы начнете управлять control flow с помощью исключений, а это ошибка дизайна. Ошибки третьего типа лучше решать через Error Codes или `std::expected`, в особенности если расстояние между бросающим и ловящим исключение кодом на стеке вызовов маленькое. Тем не менее, многие сторонние библиотеки по умолчанию обрабатывают локальные ошибки с помощью исключений. Один из таких примеров и что с ним делать можно глянуть ниже 2. Потому для использования методов из таких библиотек, к ним приходится писать обертки, которые перехватывают исключения и на выходе используют Error Codes или `std::excpected`. О том, как писать такие обертки можно посмотреть ниже в примере 4.

13.5.1 Error Codes

Большая часть ошибок может быть обработана здесь и сейчас во время вызова функции. Например, если вы пытаетесь открыть файл на чтение, а файла нет, то вы просто принимаете решение сообщить об этом пользователю и продолжать работать. У вас нет необходимости посылать эту ошибку куда-то далеко по стеку вызовов. Или если вы опрашиваете наличие видеокарты, чтобы понять можно ли ее использовать для вычислений. Если вам приходит ответ, что ее нет, то вы просто запоминаете это и не используете видеокарту. Нет необходимости добираться до `main` раскручивая стек вызовов полностью. Для таких ошибок еще в Си используют Error Codes.

Давайте посмотрим как это работает. Пусть у вас есть функция

```
1 void func(int a, const B& b);
```

и вы хотите сообщить об ошибке из нее. Для этого надо понять какие могут быть все виды ошибок, которые могут произойти. Для этого заводим `enum class`, содержащий статус операции после вызова функции. Статус возвращается через `return`.

```
1 enum class Status {
2   OK, Fatal, Error, Ingored, Pending
3 };
4
5 Status func(int a, const B& b);
```

Теперь вы можете проверять наличие ошибок так

```
Status status = func(1, B{});
if (status != Status::OK) {
  /*handle an error*/
}
```

Что делать, если функция возвращает данные через `return`? В этом случае надо сделать выходные аргументы (см. раздел 8.3 часть «Аргументы ссылки»). Например, если функция имеет вид

```
struct Result {
  std::string description;
  int operations = 0;
};

Result func(int a, const B& b);

int main() {
  ...
  Result result = func(1, B{});
  ...
}
```

Мы переделаем функцию в

```
Status func(int a, const B& b, Result* result);

int main() {
  ...
  Result result;
  Status status = func(1, B{}, &result);
  ...
}
```

Главный недостаток такого подхода в том, что если `Result` не является default constructible, то есть его нельзя построить неинициализированным, то вы не сможете просто так создать «пустой» `result` и потом заполнить его данными. В языке Си конечно же есть только структуры и там такой проблемы не бывает. В C++ же проблемы все таки есть. Многие Сишные библиотеки используют такой подход. В операционной системе Windows функции используют такой подход.

13.5.2 `std::expected` и `std::optional`

Данный подход вертится вокруг идеи про Error Codes. Мы не хотим использовать out аргументы у функции. В этом случае мы хотим иметь возможность сообщать об ошибке и инициализировать данные, которые не default constructible. В любом случае данные должны быть хотя бы movable. Если у вас есть хрупкий класс, то его всегда можно сделать movable, если завернуть его имплементацию в `unique_ptr` (стоит поглядеть в 6.3.14). Главная идея в том, что теперь по `return` мы можем вернуть и данные и ошибку.

Простой пример Самый простой вариант – ошибка не содержит никакой информации. В этом случае можно использовать `std::optional`. Пусть для примера вам нужно прочитать файл. При этом нет гарантии, что файл, который вы читаете вообще существует. Давайте предположим, что у вас есть функция, которая пытается прочесть данные из файла. Пусть функция имеет следующую сигнатуру:

```
Data read_from_file(const std::filesystem::path& name) ;
```

То есть она всегда по `return` возвращает данные. Что тогда должна делать эта функция, когда данных нет (например из-за отсутствия файла или ошибки чтения)? Она должна бросить исключение. В этом случае использование функции становится неудобным, приходится писать

```
Data data;
try {
  data = read_from_file(file_name);
} catch (...) {
  /*react to the error*/
}
```

И к сожалению это не работает, если `Data` не deafult constructible. Так как логически такая функция возвращает либо данные либо ошибку, что данных нет, то можно вернуть `std::optional<Data>`.

```cpp
std::optional<Data> readFromFile(const std::filesystem::path& name) {
  try {
    return read_from_file(name);
  } catch (...) {
    return std::nullopt;
  }
}
```

В этом случае проверка на стороне пользователя выглядит как-то так

```cpp
auto data = readFromFile(file_name);
if (data.has_value()) {
  use_data(*data);
} else {
  /*react to the error*/
}
```

Обратите внимание, что мы не переписываем существующую функцию для чтения из файла (у нас может не быть такой опции, если это сторонняя функция).

Чуть менее простой пример В этом примере мы считаем, что ошибка может содержать какие-то данные. По сути нам нужен тип `std::variant`, который умеет хранить два типа: тип данных и тип ошибки. Только при этом мы хотим эти типы трактовать по разному. Тип `std::expected` по сути и является частным случаем `std::variant`, с подправленным интерфейсом. Начнем с примера

```cpp
struct Data { ... };
struct Error { ... };

using ExpectedData = std::expected<Data, Error>;
```

Теперь функция возвращающая ошибку может выглядеть так

```cpp
ExpectData parse_file(const std::filesystem::path &name);
```

В этом случае мы можем не просто узнать, что была ошибка, но и узнать детали об ошибке:

```cpp
auto data = parse_file(file_name);
if (!data.has_value()) {
  /* deal with data.error()*/
  return;
}
use_data(*data);
```

В данном случае есть не симметрия. Мы ожидаем, что ошибки это редкость, потому по умолчанию по оператору `operator*` из `std::expected` вынимаются данные. В такой парадигме бывает два вида функций:

Без ошибок

```cpp
R function1(const A&);
```

С ошибками

```cpp
std::expected<R, E> function2(const A&);
```

Функции первого типа – это функции, которые всегда могут любой вход типа `A` переработать в выход типа `R`. Функции второго типа не всегда могут отработать и на каких-то входах дают ошибки. Потому функции второго типа могут вернуть либо тип `R` в случае успеха, либо тип `E` в случае ошибки.

Компонирование Теперь нужно научиться компонировать функции. Если у нас только функции, которые не могут совершать ошибки, то в этом нет ничего сложного, это делается на уровне языка композицией:

```cpp
B function1(const A&);
C function2(const B&);

C c = function2(function1(a));
```

Однако, если мы хотим работать с ошибками, то у нас периодически будут появляться типы вида `std::expected`. Давайте обсудим какие есть опции.

1. `transform`. В данном случае мы хотим выполнить операцию над содержимым `std::expected`. В этом случае сигнатура функции следующая

```
R function(const A&);
```

И нам дано

```
std::expected<A, E> data;
```

Если внутри `data` лежит объект типа `A` мы хотим применить к нему функцию `function` и запаковать результат в `std::expected<R, E>`. Если же внутри `data` лежит ошибка типа `E`, то мы хотим не трогать ее и вернуть `std::expected<R, E>` инициализированную ошибкой из `data`. Ясно что эту операцию можно сделать руками следующим образом:

```
std::expected<R, E> result;
if (data.has_value()) {
  result = function(*data);
} else {
  result = data.error();
}
```

Эти действия делаются автоматически с помощью метода `transform`.

```
std::expected<R, E> result = data.transform(function);
```

То есть `transform` умеет применять к данным любую функцию, которая не совершает ошибок.

2. `and_then`. В этом случае мы тоже хотим применить к данным функцию, но только теперь функция может ошибаться, то есть сигнатура функции теперь такая

```
std::expected<R, E> function(const A&);
```

При этом данные нам даны в следующем виде

```
std::expected<A, E> data;
```

Мы ожидаем следующее поведение:

   (a) Если внутри `data` лежат данные, то мы применяем к ним `function` и возвращаем этот результат.

   (b) Если внутри `data` лежит ошибка, то мы возвращаем `std::expected<R, E>` инициализированный ошибкой из `data`.

Обратите внимание, что тип ошибки должен быть единый как для данных, так и для функции. Ясно, что можно реализовать такое поведение руками следующим образом

```
std::expected<R, E> result;
if (data.has_value()) {
  result = function(*data);
} else {
  result = std::unexpected(data.error());
}
```

Такая операция автоматически делается с помощью `and_then` так:

```
std::expected<R, E> result = data.and_then(function);
```

Таким образом `and_then` умеет применять к данным с ошибкой любую функцию, которая может совершить ошибку того же типа.

13.5.3 Exceptions

Зачем? Прежде всего давайте поймем зачем нам в принципе в языке нужны исключения. Казалось бы можно использовать коды с ошибками или даже `std::expected`. Однако, есть два случая, когда вы не можете воспользоваться этими механизмами.

1. Конструкторы. Давайте начнем с примера кода

```
1  class A {
2  public:
3    A() = default;
4    A(int x) : x_(x) {}
5  private:
6    int x_ = 0;
7  };
8
9  int main() {
10   A a;
11   A b(4);
12   return 0;
13 }
```

Как мы видим в строчках 10 и 11, в синтаксисе использования конструкторов у нас нет возможности вернуть код ошибки. Потому что конструктор должен вернуть сконструированный объект. Точнее, это не совсем правда. Технически устроены конструкторы объявленные в строчках 3 и 4 можно представлять себе так

```
1  class A {
2  public:
3    static void ctor(A* this) = default;
4    static void ctor(A* this, int x);
5  private:
6    int x_ = 0;
7  };
```

Так вот, конструктор получает указатель на адрес, где надо создать объект и возвращает `void`. Техническую часть можно было бы изменить, но синтаксически не понятно, как это выразить. Как в строчках 10 и 11 написать что-то, чтобы вернуть сообщение об ошибке? Думаю тут можно было бы придумать особый синтаксис, но что если вы используете вложенные конструкторы или напрямую используете их как аргументы функции

```
1  function(A(), B(A(), 5), C{1, 2});
```

Кто и куда тут должен возвращать ошибку? Тут вызывается 4 конструктора. Как такое должно сообщать об ошибке?

2. Операторы. Давайте посмотрим на следующий код

```
1  class A {
2  public:
3    friend operator+(const A& first, const A& second);
4  private:
5    ...
6  };
7
8  int main() {
9    A a, b, c;
10   std::cin >> a >> b;
11   c = a / b;
12   std::cout << c << '\n';
13   return 0;
14 }
```

В строчке 11 вызывается оператор деления. Что если разделить на `b` нельзя? Куда сообщить об ошибке? И опять, можно было бы придумать разные синтаксисы, но подумайте, как вы это будете применять, если у вас большое сложное арифметическое выражение. Кроме того, тут еще есть два оператор `operator>>` и `operator<<` для потокового ввода вывода. Они тоже не имеют возможности вернуть код ошибки. Тем не менее, даже потоковые операторы для `std::out` и `std::cin` имеют альтернативный механизм, в рамках которого устанавливаются биты ошибок в специальном глобальном объекте.

Как это работает Давайте посмотрим на следующий пример:

```
1  void g() {
2    throw std::runtime_error("Ups!");
3  }
4
5  void f() {
6    /*some work*/
7    g();
8    /*some work*/
9  }
10
11 int main() {
12   try {
13     f();
14   } catch(std::exception& e) {
15     std::cout << "Error: " << e.what() << '\n';
16   } catch(...) {
17     std::cout << "Unknown exception!\n";
18   }
19 }
```

Давайте опишу работу кода по строкам. Точка входа в программу – функция `main`, потому начинаем с нее.

1. Исполнение начинается в строке 13. Тут вызывается функция `f`. Так как мы ожидаем, что она кидает исключения, оборачиваем это в блок `try`, чтобы его можно было поймать.
2. Далее выполняется код в строке 6 (тут может быть много кода).
3. После этого в строке 7 вызывается функция `g`.
4. Управление переходит к функции `g` и выполняется строчка 2. В этой строчке бросается исключение. В этом месте мы вываливаемся из функции `g` в строчку 7 внутри функции `f`.
5. В строчке 7 после исключения мы должны выйти из функции `f` и будем так выходить до тех пор, пока не встретим `catch`, который поймает исключение. Но чтобы выйти мы должны почистить стек от функции `f`. Это называется stack unwinding. Надо удалить все локальные переменные и аргументы функции `f`. Для этого после строчки 7 вызываются деструкторы всех локальных переменных на стеке, которые создала функция `f`.
6. Почистив стек мы выходим в больший scope из которого вызвана `f`. В данном случае это строка 13. Так как мы вышли в блок `try`, то мы ловим исключение и для реакции на него нужны следующие блоки `catch`.
7. Мы идем по блокам `catch` один за одним и смотрим, есть ли хотя бы один блок, под который подходит наше исключение. В данном случае исключение типа `std::runtime_error` унаследовано от `std::exception` и так как мы принимаем исключение по ссылке, то включается полиморфизм и мы можем поймать это исключение в первом блоке. Если бы было несколько подходящих блоков, то исключение ловится первым, а остальные не реагируют. Если нет ни одного блока, который может поймать исключение, то оно улетает дальше из текущего scope.

Таким образом, если вы работаете с исключениями, то вообще говоря, вы можете выйти из любого scope в любой точке функции.

13.5.4 Гарантии поведения при исключениях

Предположим, мы выполняем следующий код

```
Data d = ...;
Result r;
try {
  r = algorithm(d);
} catch(...) {
}
```

Вопрос, если во время выполнения алгоритма было брошено исключение, то в каком состоянии находятся `d` и `r`? Если `d` принимается по константной ссылке, то оно гарантировано не меняется, потому нас скорее всего интересует состояние объекта `r`. Бывает три вида гарантий:

1. Strong Exception Guaranty. В этом случае все объекты находятся в состоянии в каком они были до вызова функции бросающей исключение.

2. Basic Exceptioni Guaranty. В этом случае все объекты находятся в корректном состоянии, но неизвестно в каком. Объектами можно пользоваться, но конкретное состояние не известно и скорее всего не представляет интереса.

3. No Exception Guaranty. Никаких гарантий. Например, у `r` из примера выше могут быть нарушены инварианты класса.

Когда вы пишите код, то важно его писать так, чтобы выполнялись Basic Exception Guaranty. Это должно быть опцией по умолчанию. Хорошо было бы иметь Strong Exception Guaranty, но это может оказаться дорого. Последняя опция допустима только при отключенных исключениях. Однако, даже если вы не используете исключения, если вы пишите код с соблюдением Basic Exception Guaranty, то под получается лучше. Связано это с тем, что у вас появляются ограничения, которые направляют написание кода в лучшую сторону.

13.5.5 Технические детали и логическая модель для исключений

Технические детали, которые надо знать Прежде чем обсуждать какие есть сложности при использовании исключений, давайте обсудим технические детали связанные с имплементацией. Когда вы кидаете исключения надо помнить о следующем:

1. Как аллоцируется память для исключения в стандарте не определено. На практике это означает, что вы всегда аллоцируете память на куче даже для стандартных типов вроде `int`.

2. Для того, чтобы произвести stack unwinding вам нужно для каждой точки программы знать, какие деструкторы позвать. Как это должно делаться в стандартне не определено, но на практике есть два способа:

   (a) Frame-based. Данный метод по сути поддерживает в run-time второй стэк, на который складываются деструкторы объектов для текущего scope. Этот метод означает, что вы платите за исключения даже, когда их не кидаете. Этот подход использует компилятор от Microsoft.

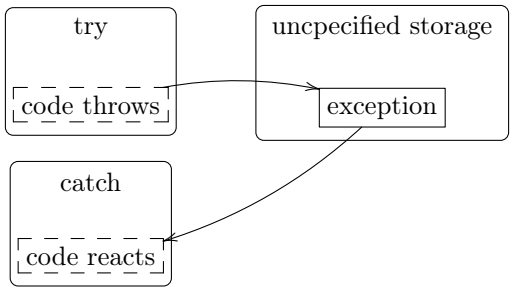
   (b) Table-based. В данном подходе, во время компиляции программы, вы составляете таблицы для вызова нужных деструкторов из каждой точки вашего кода. И в run-time используете эти таблицу для их вызова. Такой подход очень сильно раздувает бинарники программы, потому что нужно хранить все эти таблицы. Такой подход используют gcc и clang.

Когда мы ловим исключения надо понимать следующее:

1. Язык предоставляет полиморфное поведение. То есть если вы унаследовали класс `A` от класса `B`, то по ссылке на `B` можно ловить и исключение класса `A`. В стандарте не прописано как это должно быть имплементировано, но на практике всегда используется RTTI, что дорого.

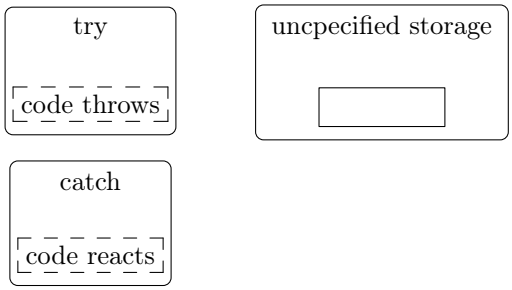
2. Деаллокация памяти для исключения не специфицирована в стандарте. На практике это значит удаление объекта с кучи и это дорого.

Возможные состояния Здесь я хочу немного поговорить про ментальную модель для исключений и в каких состояниях может находиться ваша программа логически. Давайте начнем с картинки:

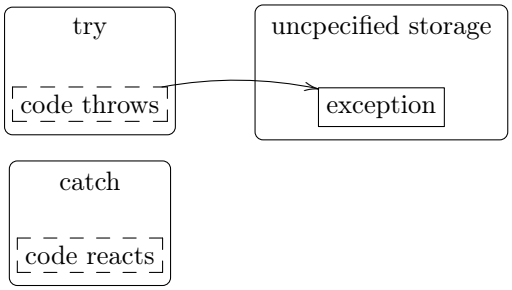


Думаю, что данная диаграмма более или менее не требует объяснений. Процесс бросания исключений в штатном режиме можно представлять себе так:

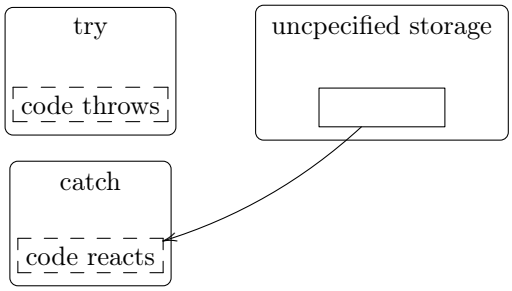
1) Исполнение в блоке `try`



2) Бросаем исключение



3) Перехватываем исключение в `catch`

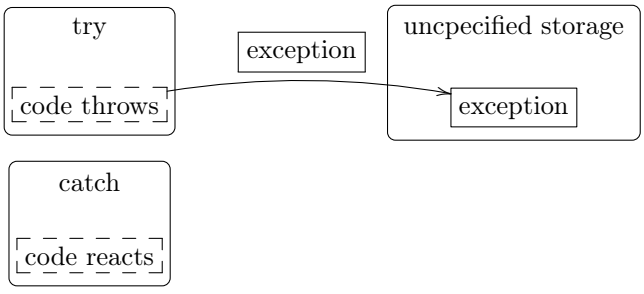


Давайте проговорим, что тут происходит:

1. Начинаем исполнение программы в блоке `try`.
2. Код в блоке `try` бросает исключение. В этот момент объект исключения аллоцируется неспецифицированным образом.
3. Как только исключение долетело до блока `try`, для него ищется нужный обработчик среди присоединенных `catch` блоков. Если нужный обработчик нашелся, то мы вынимаем объект исключения оттуда, где он хранился и отдаем его в распоряжение `catch` блока. В конце обработки `catch` блок сам удалит объект исключения, вам не надо об этом заботиться. В этом случае исключение логически не в брошенном состоянии, а в пойманном. И система помнит это исключение. Его можно перебросить по команде `throw` без аргументов.

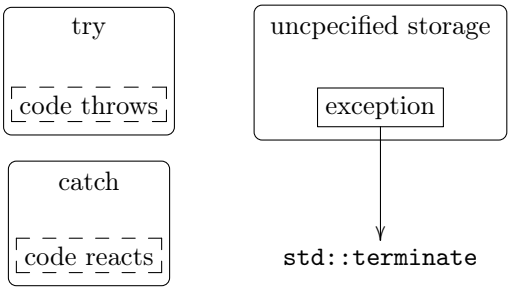
Проблемы А теперь поговорим, что в этой схеме может пойти не так.

1. Вы можете кинуть исключение, когда другое исключение все еще висит не обработанное. Такое возможно, если вы бросите исключение во время stack unwinding. По этой причине нельзя кидать исключения в деструкторах.



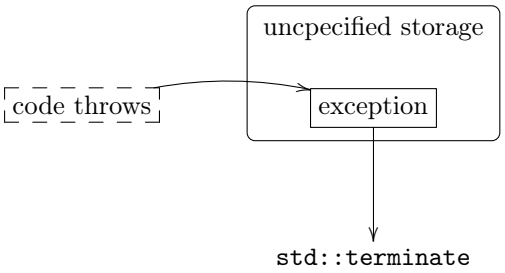
В этом случае программа падает по `std::terminate`.

2. Если не нашлось ни одного обработчика среди `catch` блоков.



При этом был ли почищен стэк с помощью stack unwinding не специфицировано.

3. Исключение брошено не внутри блока `try`. В этом случае мы никогда не найдем соответствующий перехватчик для исключения. И мы опять падаем по `std::terminate`.



13.5.6 Важные примеры и ожидаемое поведение для исключений

1. Самый первый пример, который обычно приводя – выделение памяти. Например, вы пытаетесь выделить память под `std::vector` и ее не хватило. Это плохой пример для использования исключений по следующим причинам:

   (a) Если вы запросили память и ее больше в системе нет, то что вы собираетесь делать? Скорее всего у вашей системы проблемы покруче, чем невозможность продолжить работать вашей программы. Смысла тут делать что-то нет.

   (b) Память запрашивает чуть ли не каждый кусок кода в любой библиотеке. Тогда придется проверять на исключение каждое обращение к оператору выделения памяти. В таком случае программа только и будет делать, что проверять исключения и ничего больше. Это очень глупо.

2. Разумный пример, чтобы умереть. Скажем, вы хотите работать в программе с клавиатурой через какой-нибудь интерфейс операционной системы. Делаете запрос, на доступ к этому интерфейсу, а операционная система говорит, что нельзя и не дает доступ.

```
auto handle = access_keyboard(...);
if (!handle) {
  // cannot access the keyboard
}
```

   То есть системная функция вернула ошибку и вы не можете даже начать выполнение программы в таком случае. Вот такую ситуацию как раз надо ловить исключением. И политика тут такая, если все получилось, то хорошо, если не удалось получить доступ, то сообщаем пользователю, что у нас проблемы с доступом и завершаем работу. Обычно за получение доступа к клавиатуре у вас будет отвечать специальный объект. И доступ к клавиатуре вы запросите в его конструкторе.

```
class Keyboard {
public:
  Keyboard() {
    auto handle = access_keyboard(...);
    if (!handle)
      throw std::runtime_error("Cannot access the keyboard!");
  }
private:
  ...
};
```

При этом для приличия такое исключение надо поймать в `main`, чтобы не падать по `std::terminate`.

3. Разумный пример, чтобы умереть и воскреснуть. Предположим у вас в потоке крутится сервис, который может упасть в результате любой ошибки и вы хотите этот сервис перезапустить. Код для сервиса может выглядеть как-то так:

```
int main() {
  Server server;
  server.run();
  return 0;
}
```

Если метод `run` бросает исключение, то можно сделать следующее:

```
int main() {
  for (Status status = Running; status != Quit;) {
    try {
      Server server;
      status = server.run(); // if returns normally sets status to be Quit
    } catch(...) {
      /*log the error*/
    }
  }
  return 0;
}
```

В этом случае мы крутимся в бесконечном цикле, пока флаг `status` не имеет значение `Quit`. Если сервер заканчивает работу в штатном режиме, то метод `run` устанавливает `status` в состояние `Quit` и так мы понимаем, что надо выйти из цикла. Если же конструктор `server` или метод `run` умирают по исключению, то мы логируем ошибку и пытаемся еще раз инициализировать `server` и запустить его. В таком методе полезно еще установить границу максимального количества перезапусков сервера и максимальную частоту перезапусков.

4. Разумный пример для продолжения работы. Предположим, что вы парсите файл какого-то сложного формата. И в рамках этого парсинга вы вызываете много вспомогательных функций. И если возникла проблема с парсингом, то это значит, что файл прочитать нельзя, формат не верный и вы хотите быстро выйти наверх в самую старшую функцию.

```
Object parse(std::filesystem::path name) {
  File file = open_file(name);
  auto header = parse_header(file);
  auto data = parse_data(file);
  return Object(data);
}

auto parse_data(File& file) {
  ...
  parse_something(file);
  ...
  return ...;
}

auto  parse_something(File& file) {
  ...
  if (error occure)
    throw std::runtime_error("parsing error in parse_something!");
  return ...;
}
```

Только теперь надо обработать эту ошибку грамотно

```
std::optional<Object> try_parse(std::filesystem::path name) noexcept {
  try {
    return parse_impl(name);
  } catch (...) {
    return std::nullopt
  }
}
```

```
8
9  Object parse(std::filesystem::path name) {
10   File file = open_file(name);
11   auto header = parse_header(file);
12   auto data = parse_data(file);
13   return Object(data);
14 }
```

Причем мы это делаем неинтрузивно, то есть без вмешательства в имплементацию старой функции. Обратите внимание, что старая функция всегда возвращает объект нужного типа `Object`, но об ошибках она сообщает с помощью исключений, которые надо ловить снаружи. Это может быть не удобно, так как засоряет ваш код. Вы не хотите управлять логикой программы с помощью исключений. Для этого мы оборачиваем старую имплементацию в новую. Новая имплементация уже не может кидать исключения, и об ошибке она сообщает уже через возвращаемое значение. Я использовал `std::optional`, но можно и лучше использовать `std::expected`. В таком случае вы просто пытаетесь распарсить файл и если была ошибка возвращаете пустоту, а если не было ошибки, то возвращаете данные. После использования этой функции надо просто проверить удалось ли считать данные или нет. И если удалось, то надо вынуть их. Давайте сравним два обработчика:

```
try {
  Object obj = parse(file_name);
  run_program(obj);
} catch (std::exception& e) {
  print_error(e.what());
} catch(...) {
  print_error("Unknown error!");
}
```

```
std::optional<Object> obj =
                    try_parse(file_name);

if (!obj.has_value()) {
  print_error("Cannot parse file_name");
  return 0;
}
run_program(*obj);
```

Обратите внимание как проще выглядит workflow во втором случае. Во втором подходе, если вы забыли, что у вас могут быть невалидные данные после парсинга и вы попытаетесь написать такое:

```
1 auto obj = try_parse(file_name);
2 run_program(obj);
```

То компилятор вам подскажет об ошибке, что вы забыли распаковать данные и передаете `std::optional` вместо `Object` в функцию `run_program`. Однако, если в первом подходе, вы забыли поставить `try/catch` блок или забыли добавить в `catch` блок нужный перехватчик, то ваша программа в этом случае упадет по `std::terminate` и никто вам в этом не поможет.

Давайте я подведу итоги написанному выше. Исключения можно использовать в двух сценариях:

1. На критическую ошибку, которая не позволяет продолжать работать, умереть и сообщить причину смерти пользователю. Это был пример 2.
2. На не критическую ошибку перехватить исключение и пойти по плану «Б» для данной ошибки. Это был пример 4. При этом обратите внимание, что на стороне пользователя лучше иметь функцию, которая не кидает исключения, а возвращает либо объект, либо ошибку. Потому что в этом случае попытка управления через исключения – это ошибка дизайна. Вы не должны управлять логикой программы с помощью исключений. Кроме того, вам придется при каждом использовании бросающей исключения функции писать `try/catch` блоки, что сделает код менее читаемым. Лучше инкапсюлировать эту работу в другую не бросающую исключения функцию.

Если вы используете исключения для чего-то еще, то скорее всего вы делаете что-то не так.

13.5.7 «Идеальная» имплементация

Богатая иерархия наследования между исключениями – большая головная боль. Вы никогда не запомните (а может быть даже и не узнаете), кто от кого унаследован и унаследован ли и никогда не поставите `catch` блоки в правильном порядке. Потому система должна быть максимально простой. Нужен обязательно один базовый класс, чтобы можно было ловить сразу все исключения, а все остальные от него унаследованы и отличаются только данными. На уровне интерфейса я бы ожидал следующее:

```
template<class T>
class Exception {
public:
  Exception(std::string description, T data);

  const std::string& what() const;
  std::string& what();
  const std::source_location where() const;
  const std::stacktrace& stack() const;
  const T& data() const;
  T& data();
private:
  std::string description_;
  std::source_location location_;
  std::stacktrace backtrace_;
  T data_;
};
```

Обратите внимание, что любое исключение содержит информацию, которую мы обсуждали в разделе 13.3, а именно:

1. Текстовое описание ошибки.
2. Точка в коде, где ошибка произошла.
3. Текущее состояние стека.
4. Специфические данные типа `T`.

В данном подходе все исключения параметризуются типом `T` и для одинакового типа `T` исключения не отличимы.

Заметим, что для `T = void` нужно сделать специальную имплементацию, которую можно так же использовать в качестве базового класса, а именно:

```
template<>
class Exception<void> {
public:
  Exception(std::string description);

  const std::string& what() const;
  std::string& what();
  const std::source_location where() const;
  const std::stacktrace& stack() const;
private:
  std::string description_;
  std::source_location location_;
  std::stacktrace backtrace_;
};
```

```
template<class T>
class Exception : public Exception<void> {
public:
  Exception(std::string description, T data);

  const T& data() const;
  T& data();
private:
  T data_;
};
```

Давайте обсудим некоторые важные детали.

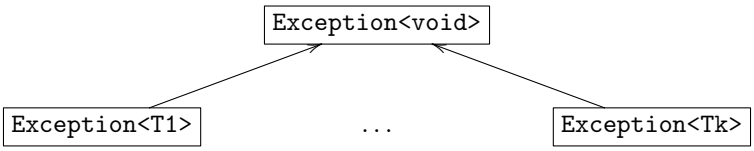
1. Данные о точке кода, где произошла ошибка, и текущее состояние стека являются константными (это инвариант класса). Они создаются в конструкторе и потом доступны только для чтения. Поля класса не делаются константами, чтобы обеспечить хорошее поведение value semantics.
2. Строка с описанием и данные типа `T` доступны для модификации. Это нужно, если при перебрасывании исключения вам нужно добавить информацию. Выглядит это как-то так

```
try {
  function(); // throws
} catch(Exception<void>& e) {
  e.what() += "info"; // adds extra information
  throw;
}
```

Аналогично может понадобиться модификация кастомных данных типа `T`.

3. Перехват всех возможных исключений нашего типа делается с помощью перехвата базового класса `Exception<void>`, который не содержит кастомных данных.

Таким образом логическая схема наследования получается очень простой:



Теперь приведем имплементацию классов. В начале базовый класс.

```
template<class T>
class Exception;

template<>
class Exception<void> {
public:
  Exception(std::string description,
            std::source_location location = std::source_location::current(),
            std::stacktrace backtrace = std::stacktrace::current())
      : description_(std::move(description)), location_(std::move(location)),
        backtrace_(std::move(backtrace)) {
  }
  std::string& what() {
    return description_;
  }
  const std::string& what() const {
    return description_;
  }
  const std::source_location where() const {
    return location_;
  }
  const std::stacktrace& stack() const {
    return backtrace_;
  }
private:
  std::string description_;
  std::source_location location_;
  std::stacktrace backtrace_;
};
```

Обратите внимание, что тут приходится использовать трюк из раздела 13.3, чтобы правильно перехватить строку, в которой произошла ошибка и состояние стека. Нам важно, чтобы это был не конструктор для исключения, а точка, где вызван конструктор. Это возможно с помощью аргументов по умолчанию, потому что они конструируются в точке вызова функции (в данном случае конструктора). Теперь классы с дополнительной информацией выглядят так:

```
template<class T>
class Exception : public Exception<void> {
  using Base = Exception<void>;

public:
  Exception(std::string description, T data,
            std::source_location location = std::source_location::current(),
            std::stacktrace backtrace = std::stacktrace::current())
      : Base(std::move(description), std::move(location), std::move(backtrace)),
        data_(std::move(data)) {
  }
  const T& data() const {
    return data_;
  }
  T& data() {
    return data_;
  }
private:
  T data_;
};
```

Обратите внимание, что тут нельзя вызвать базовый конструктор с аргументами по умолчанию, потому что они покажут точку в коде – текущий конструктор. Потому здесь придется руками еще раз прописать аргументы по умолчанию и все аргументы передаем в базовый конструктор.

Теперь для удобства ввода вывода нужно правильно перегрузить потоковые операторы. Для отображения места в коде, где произошла ошибка, и текущего состояния стека мы уже писали имплементацию в разделе 13.3, однако, для полноты картины я их еще раз тут повторю:[30]

```
std::ostream& operator<<(std::ostream& out, const std::source_location& location) {
  out << location.file_name() << "(" << location.line() << ":"
      << location.column() << "), function '" << location.function_name()
      << "'";
  return out;
}

std::ostream& operator<<(std::ostream& out, const std::stacktrace& backtrace) {
  static constexpr int bound = 6;
  if (backtrace.size() < bound)
    return out;
  for (auto iter = backtrace.begin(); iter != (backtrace.end() - bound); ++iter) {
    out << iter->source_file() << "(" << iter->source_line()
        << "):" << iter->description() << '\n';
  }
  return out;
}
```

Теперь нужны удобные операторы для отображения исключений. Придется написать оба по-отдельности, потому что в базовом классе отсутствует данное, которое есть в производных классах.

```
std::ostream& operator<<(std::ostream& out, const Exception<void>& e) {
  std::cout << "Failed to process: " << e.what() << '\n'
            << e.where() << '\n'
            << e.stack();
  return out;
}

template<class T>
std::ostream& operator<<(std::ostream& out, const Exception<T>& e) {
  std::cout << "Failed to process with code (" << e.data() << ") : " << e.what()
            << '\n'
            << e.where() << '\n'
            << e.stack();
  return out;
}
```

Теперь этим можно пользоваться так:

```
void function() {
  throw Exception<int>("Some error!", 1);
}

int main() {
  try {
    function();
  } catch (const Exception<int>& e) {
    std::cout << e;
  } catch (const Exception<void>& e) {
    std::cout << e;
  } catch (...) {
  }
}
```

13.5.8 Рекомендации

Рекомендации по использованию Тут я бы хотел сформулировать несколько соображений о том, как я вижу обработку исключений.

[30] Если вам интересно, что значит `bound`, то загляните в конец раздела 13.3.

1. По-хорошему у вас должна быть одна точка по обработке исключений (или несколько, но не много). Чаще всего – это event loop в вашей программе, где вы можете сообщить об ошибке и умереть. Или функция `main`, которая является последней линией обороны до выхода в операционную систему или падения по `std::terminate`.

2. Исключения никогда не должны быть в критической для производительности части. Более того, рекомендуется помечать код `noexcept` (даже если он кидает исключения, например на запрос памяти у системы), чтобы компилятору было дозволено больше оптимизаций.

3. Исключения не должны ни в каком виде контролировать workflow вашего кода. За это должна отвечать логика программы.

Технические рекомендации

1. Исключения кидаются by value.

```
1  throw A(1, "abc");
```

2. Исключения ловятся по ссылке или константной ссылке. Это нужно для обеспечения полиморфного поведения.

```
1  catch(const A& a)
```

3. Повторное кидание висячего исключения делается командой `throw` без аргументов. Этот трюк используется для того, чтобы отложить реакцию на исключение.

```
1  void react() {
2    try {
3      throw;
4    } catch (std::exception& e) {
5      /*react to std::exception*/
6    } catch (...) {
7      /*react to unknown exception*/
8    }
9  }
10
11 int main() {
12   try {
13     run();
14   } catch(...) {
15     react();
16   }
17 }
```

В этом примере, когда мы попадаем в блок `catch`, мы перехватили текущее исключение. Оно больше не висит, но система знает это последнее исключение внутри блока `catch`. И чтобы перекинуть его мы в функции `react` используем команду `throw` без аргументов в строке 3. Однако, ее надо помещать внутри `try` блока, чтобы после перекидывания, можно было поймать исключение.

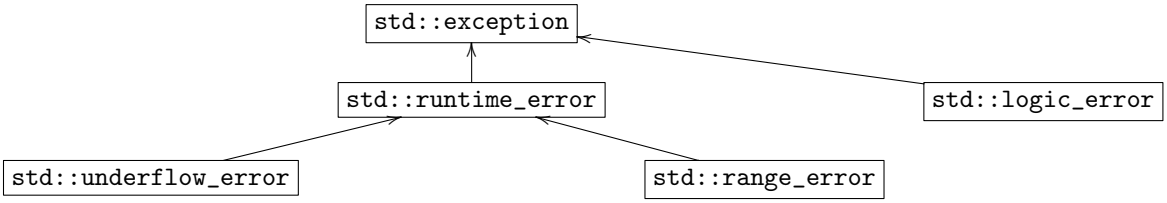
4. Производные классы исключений среди `catch` обработчиков должны идти перед базовыми классами, чтобы поймать наиболее специализированный тип.

```
1  class A {};
2  class B : public A {};
3
4  try {
5    run();
6  } catch (const B& e) {
7    /*reacto to B*/
8  } catch (const A& e) {
9    /*reacto to A*/
10 } catch(...) {
11   /*reacto to everything left*/
12 }
```

Напомню, что блоки `catch` проходятся сверху вниз по одному, и на первом подходящем блоке исключение обрабатывается и последующие `catch` блоки игнорируются. Проблема с этим подходом в том, что вы должны знать отношение наследования между классами, чтобы правильно добавлять обработчики. Вот отличный пример

```
1 try {
2   ...
3 } catch (const std::runtime_error& e) {
4 } catch (const std::underflow_error& e) {
5 } catch (const std::range_error& e) {
6 } catch (const std::logic_error& e) {
7 } catch (...) {
8 }
```

проблема в том, что диаграмма наследования для исключений выглядит так



Это означает, что мы никогда не поймаем исключение в строках 4 и 5, потому что они унаследованы от исключения, перехватываемого в строке 3. Помнить наизусть это все невозможно. Иерархия на исключениях делает код хрупким. Куда проще было бы сразу перехватывать базовый класс.

```
1 try {
2   ...
3 } catch (const std::exception& e) {
4 } catch (...) {
5 }
```

В идеале для работы с исключениями можно пользоваться имплементацией описанной в разделе 13.5.7.

13.5.9 Проблемы при использовании исключений

Прежде чем вы прочитаете как жизнь с исключениями невыносима, я хочу напомнить про главную пользу исключений. Когда вы пишите код, подразумевая, что у вас могут быть исключения, вы находитесь в рамках серьезных ограничений (пусть компилятор их на вас не накладывает, но вы должны это понимать). А когда у вас есть серьезные ограничения, то вы всегда пишите код лучше.

Менеджмент ресурсов Так как исключение может возникнуть в любой строчке кода то это значит, что если исполнилась одна строчка кода, то у вас нет гарантии, что выполнится следующая. Давайте посмотрим на следующий пример:

```
1 void g() {
2   throw std::runtime_error("Ups!");
3 }
4
5 void f() {
6   int* x = new(0);
7   g();
8   delete x;
9 }
```

В этом примере при вызове `f` мы выйдем из нее в строчке 7, где функция `g` кинет исключение. Потому операция `delete` никогда не вызовется. Потому когда вы используете исключения, все ресурсы должны менеджериться автоматическими методами с использованием RAII (см. раздел 6.3.15).

Исключения в деструкторах В разделе 13.5.5 мы обсудили, что нельзя бросать исключения, когда есть активное не пойманное исключение в системе. Такое может произойти, когда мы освобождаем стек при выходе из текущего scope. В этот момент вызываются деструкторы всех локальных переменных из данного scope.

А это означает, что ни в коем случае нельзя кидать исключения в деструкторах. Если вы это сделаете, то программа упадет в run-time вызовом `terminate`. Потому никогда не бросайте исключения в деструкторах. Это чревато и не работает. У вас нет шансов сообщить об ошибке в деструкторе безопасно. Их надо обрабатывать внутри самого деструктора. Но лучше не кидать никогда.

Правила инициализации объектов и несколько ресурсов Теперь, что делать, если вам надо выделить память для двух указателей? Тупой вариант такой

```
1  struct IntPtrPair {
2    IntPtrPair(int x, int y) : ptr1_(new int(x)), ptr2_(new int(y)) {}
3    ~IntPtrPair() {
4      delete ptr1_;
5      delete ptr2_;
6    }
7    int* ptr1_;
8    int* ptr2_;
9  };
10
11 IntPtrPair x(1, 2);
```

Беда этого подхода вот в чем. Когда вы в строчке 11 вызываете конструктор, то сначала выделяется память под `ptr1_`, а потом под `ptr2_`. И если при первом вызове память выделится, а во втором нет и будет брошено исключение, то конструктор не завершит работу, а значит объект не будет считаться построенным, а значит и не вызовется деструктор для него во время stack unwinding. А это значит, что память, выделенная первым вызовом не освободится. Есть много костылей для этой проблемы, но правильное решение – обернуть каждый указатель в свой wrapper. Например так

```
1  struct IntPtrPair {
2    IntPtrPair(int x, int y) : ptr1_(std::make_unique<int>(x)),
3                               ptr2_(std::make_unique<int>(y)) {}
4
5    std::unique_ptr<int> ptr1_;
6    std::unique_ptr<int> ptr2_;
7  };
8
9  IntPtrPair x(1, 2);
```

И теперь вам вообще не надо беспокоиться о деструкторах и прочих радостях, потому что у вас везде value semantics. Все это за вас для каждого ресурса делает wrapper для работы с одним указателем.

Зависимые ресурсы и исключения Существует еще одна ситуация, о которой стоит рассказать. А что если у нас данные зависимы? Например, я должен сначала выделить память под `x`, а потом выделяя память под `y`, я должен туда положить информацию про `x`? То есть у вас есть естественная зависимость порядка конструирования данных. Мы уже обсуждали, что для конструирования по частям как раз и создано наследование в разделе 6.3.16. Оно собирает объект в определенном порядке. Давайте приведу такой пример

```
1  struct A {
2    A(int x) : ptr_(std::make_unique<int>(x)) {}
3
4    std::unique_ptr<int> ptr_;
5  };
6
7  struct B : A {
8    A(int x) : A(x), ptr_(std::make_unique<int*>(A::ptr_.get()) {}
9
10    std::unique_ptr<int*> ptr_;
11 };
12
13 B x(1);
```

В таком случае у вас в строчке 13 сначала создается базовая часть объекта, то есть выделяется память под `x` и кладется адрес в `A::ptr_`. А потом вы выделяете память под `B::ptr_` и кладете туда адрес `A::ptr_` из базовой части. Понятно, что это бессмысленная деятельность, но демонстрирует зависимость данных. Тут прошу обратить внимание на несколько вещей:

1. Я везде пользуюсь wrapper-ами для указателей и мне вообще не надо задумываться про деструкторы.

2. Обратите внимание, что нигде не пользуюсь `new` явно, вместо этого пользуюсь `std::make_unique` функцией.

3. Если в конструкторе `B` при выделении памяти под `B::ptr_` будет кинуто исключение, то так как базовая часть была построена и ее конструктор завершил работу, то вызовется деструктор базовой части `~A()`. Это позволяет безболезненно кидать исключения при построении объекта по частям.

Вот еще один пример по сборке Thread Pool через Windows Api. Для инициализации у нас есть несколько ресурсов:

1. `PTP_POOL pool`
2. `TP_CALLBACK_ENVIRON environment`
3. `PTP_CLEANUP_GROUP cleanup_group`

При этом логика инициализации следующая:

1. Создать `pool` так

```
pool = CreateThreadpool(nullptr);
```

   Эта операция может завершиться ошибкой. В этом случае `pool` будет `nullptr`

2. Создать `environment` так

```
InitializeThreadpoolEnvironment(&environment);
```

3. Соединить `pool` и `environment` так:

```
SetThreadpoolCallbackPool(&environment, pool);
```

4. Создать `cleanup_group` так

```
cleanup_group = CreateThreadpoolCleanupGroup();
```

   Эта операция может завершиться ошибкой. В этом случае `cleanup_group` будет `nullptr`.

5. Соединить `environment` и `cleanup_group` так

```
SetThreadpoolCallbackCleanupGroup(&environment, cleanup_group, nullptr);
```

Чтобы инициализировать целиком все данные можно сделать следующее. Под каждый ресурс заводим класс, который будет за него отвечать. После чего наследуем их друг от друга, чтобы инициализировать их в правильном порядке.

```
class Pool {
public:
  Pool();
  ~Pool();
  /*other methods*/
private:
  PTP_POOL pool_;
};
```

```
Pool::Pool() : pool_(CreateThreadpool(nullptr)) {
  if (pool_ == nullptr)
    throw std::runtime_error("Fail!\n");
}

Pool::~Pool() {
  CloseThreadpool(pool_);
}
```

```
class Environment : protected Pool {
public:
  Environment();
  ~Environment();
  /*other methods*/
private:
  TP_CALLBACK_ENVIRON environment_;
};
```

```
Environment::Environment() {
  InitializeThreadpoolEnvironment(&environment_);
  SetThreadpoolCallbackPool(&environment_, pool_);
}

Environment::~Environment() {
  DestroyThreadpoolEnvironment(&environment_);
}
```

```cpp
class CleanupGroup : protected Environment {
public:
  CleanupGroup();
  ~CleanupGroup();
  /*other methods*/


private:

  PTP_CLEANUP_GROUP cleanup_;
};
```

```cpp
CleanupGroup::CleanupGroup()
  : cleanup_(CreateThreadpoolCleanupGroup()) {
  if (cleanup_ == nullptr)
    throw std::exception("Fail!\n");
  SetThreadpoolCallbackCleanupGroup(
      &environment_, cleanup_, nullptr);
}
CleanupGroup::~CleanupGroup() {
  assert(cleanup_ != nullptr);
  CloseThreadpoolCleanupGroup(cleanup_);
}
```

```cpp
class ThreadPool : protected CleanupGroup {
public:
  ThreadPool();
  ~ThreadPool();
  /*other methods*/
};
```

```cpp
ThreadPool::ThreadPool() = default;

ThreadPool::~ThreadPool() {
  CloseThreadpoolCleanupGroupMembers(
      cleanup_, false, nullptr);
}
```

В такой схеме можно спокойно кидать исключение в любом конструкторе и вся часть класса, построенная на предыдущих этапах очистится корректно.

Другие виды ресурсов Аналогично тому, что я написал про память используется для выделения других ресурсов. Например для мьютексов используются `std::lock_guard` или похожие механизмы. Любой запрос к операционной системе, который требует освобождения своих ресурсов, должен выполняться через wrapper, где в конструкторе вы запрашиваете ресурс, а в деструкторе освобождаете.