Содержание

1 Вступление про устройство компьютера.

Стандартно до непосредственно C++ рассказывают немного Архитектуры ЭВМ и немного ассемблера. Разумеется, только базовое представление. Зачем? Когда мы будем говорить про языковые конструкции, мы не будем говорить только то, как они работают, но ещё и почему так, как эффективно ими пользоваться и т.п. Почему, например, лямбды без списка захвата приводятся к указателю на функцию, а со списком — нет? Без изучения основ архитектуры компьютера, мы не сможем ответить на этот вопрос. А если мы изучим базу, вопрос будет не такой, а «как может быть иначе». Есть более простые и практические вещи: если вы что-то профилируете и желаете ускорить, то высокоуровневые оптимизации (убрать ненужное действие, не считать что-то дважды) — это легко, а более низкоуровневые вообще нельзя сделать без знаний того, как внутри компьютер работает. Скаков даже приводил пример с подсчётом суммы положительных чисел на массиве (если его предварительно отсортировать, будет сильно быстрее). Существует множество историй о том, как программисты не могли разобраться в вопросе, почему их код работает не с такой скоростью, как они рассчитывают, пока им это не пояснила Intel. Ещё основы архитектуры компьютера пригодятся при отладке. Если программа работает не так, то для понимания, где же именно ошибка, нам требуется понимать, что внутри процессора происходит. И наконец есть более философская полезность: если вы чем-то пользуетесь, то чем лучше вы разбираетесь, как оно работает, тем грамотнее можно этим пользоваться. То же самое с библиотеками, например (если библиотека что-то умеет, а вы не знаете об этом, вы начинаете городить костыли). Например, в x86 есть специальная инструкция с длинной арифметикой, и лучше бы писать такой код, чтобы компилятор догадался пользоваться ей.

Работа с памятью. В нашем поверхностном знакомстве мы построим упрощённую модель, а потом уточним её там, где надо. В жизни всё, разумеется, работает очень сложно, и мы будем сталкиваться (не часто, но будем) с тем, что не сможем объяснить. Что с точки зрения нашей простой модели есть память? Пронумерованный набор ячеек со значениями от 0 до 255. К одному биту у нас обратиться способа нет, как мы знаем из ЭВМ (нам придётся считать байт и работать битовыми операциями с ним). Память поддерживает две операции: чтение и запись. Первое по номеру ячейки даёт число (данные в нужной ячейке), второе — по числу и номеру — записывает данные. Номер называется адресом памяти (то есть адрес — это число, ничего более умного).
Как процессор берёт команды из памяти? У него есть регистр (то есть маленький кусочек памяти), который называется IP (instruction pointer) — адрес следующей инструкции, которую он будет исполнять. И вот процессор миллиарды раз в секунду читает IP, исполняет записанную по этому адресу инструкцию, после чего переходит к следующей (увеличивая IP).

Ассемблер и возможности процессора. Команды пишутся байтами. Это неудобно читать, неудобно писать и, самое главное, неудобно запоминать, какая комбинация байтов что значит. Поэтому, чтобы не запоминать, люди придумали набор человеко-читаемых мнемоник и назвали его «ассемблер». Важно понимать, что набор мнемоник отличается не только от архитектуры к архитектуре, но и от прибора, которым вы ассемблируете код, к прибору.
Что процессор умеет? Кроме IP у процессора (на x86) есть 8 регистров общего назначения (`AX`, `BX`, `CX`, `DX`, `SP`, `BP`, `SI`, `DI`). Нам сейчас не очень важно, что они все значат. Каждый из них имеет 16 бит. Самая простая команда процессора — `MOV dst, src`. Это простое присваивание вида `dst = src`. В качестве её аргументов можно пихать константы или регистры. Также при помощи `MOV` можно читать и писать память. Для этого используется обозначение вида `[10]`: когда мы так пишем, мы имеем ввиду не число 10, а значение в десятой ячейке памяти. `MOV` может оперировать память двумя путями: `MOV [AX], BX` и `MOV AX, [BX]`, но не может `MOV [AX], [BX]` (для этого физически в кодировке команд нет возможности). А какие возможности есть? А вот такие:

- `MOV reg, reg`
- `MOV reg, imm`
- `MOV reg, [reg]`
- `MOV reg, [imm]`
- `MOV [reg], reg`
- `MOV [imm], reg`
- `MOV [reg], imm`
- `MOV [imm], imm`

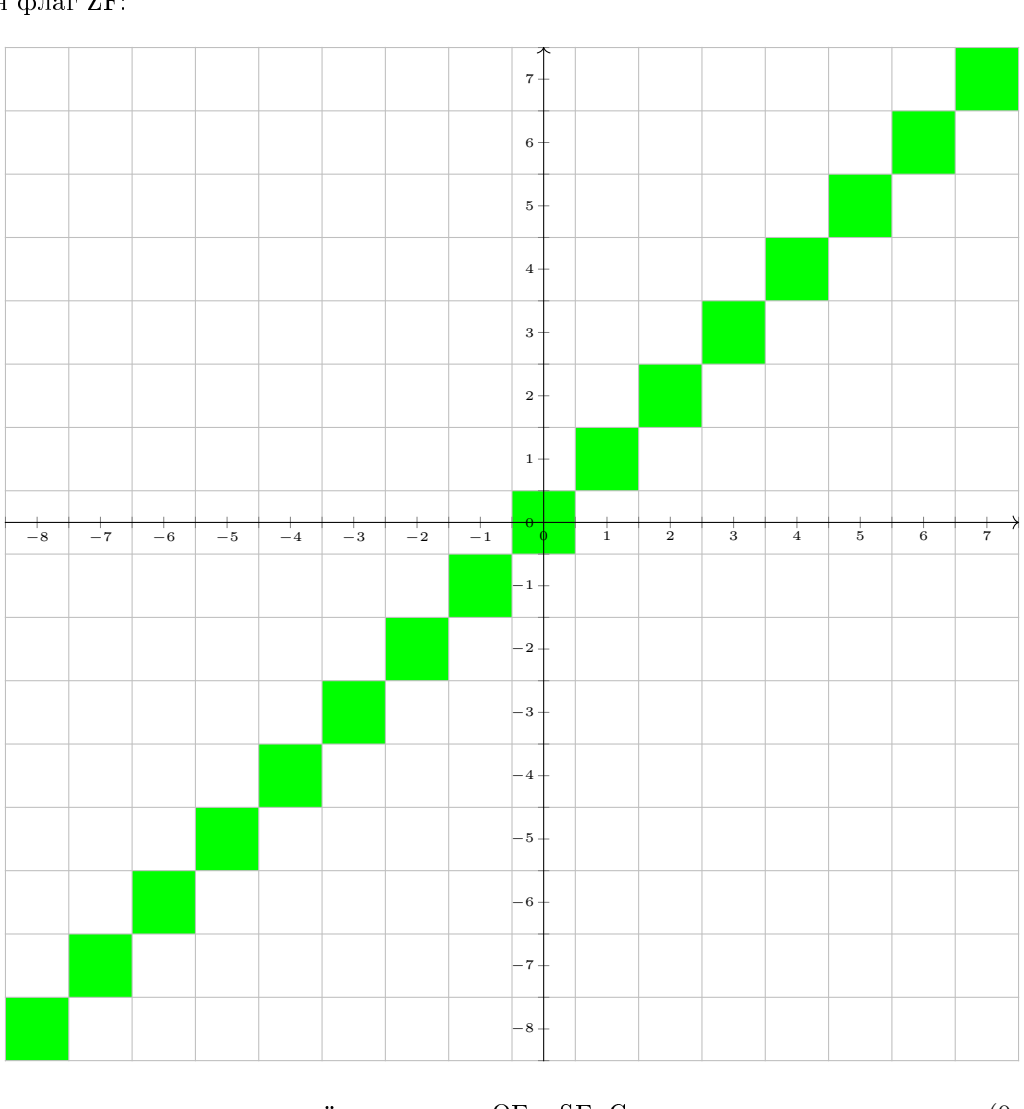
Хорошо, теперь давайте подумаем вот о чём. Регистр `AX` шестнадцатибитный, а значит при выполнении команды `MOV` нужно откуда-то эти 16 бит читать. Откуда? А вот откуда: читается не только тот адрес, который запросили, но ещё и следующий. Но возникает вопрос: какой из прочитанных байт старший, а какой — младший? Оба варианта возможны. Если байты с младшими адресами являются младшими байтами, то это little-endian, если старшие — то big-endian. Пример: если у нас в ячейке с номером 100 записан байт `0x6C`, а с номером 101 — `0xAA`, то в LE при чтении ячейки `[100]` мы получим число `0xAA6C`, а в BE — `0x6CAA`. В x86 используется исключительно первый. Комментарий: если у нас просто есть число, то некорректно говорить о его endianess (как говорить и endianess регистров), а endianess возникает, когда мы пишем либо читаем что-то в байтовом представлении.
Хорошо, есть ли возможность читать 8 бит, а не 16? Да, есть, для этого есть регистры `AL`, `AH`, `BL`, `BH`, `CL`, `CH`, `DL`, `DH`. При этом `AL` — младшие 8 бит `AX`, а `AH` — старшие, и аналогично с `BX`, `CX`, и `DX` (в остальных 4 регистрах такого разделения нет).

Арифметика. Процессор умеет выполнять все базовые арифметические инструкции (`ADD`, `SUB`, `AND`, `OR`, `XOR`). Работают они как `dst += src`, `dst -= src`, `dst &= src`, `dst = src|`, `dst ^= src`. Все они поддерживают те же комбинации `reg`/`imm`/`[reg]`/`[imm]`, что и `MOV` (см. выше). Также как и в `MOV`, в них нельзя смешивать 8 бит и 16 бит, даже в C++ если совершать операции с типами разного размера, компилятор расширит их до общего.
Ещё есть унарные операции: `INC` и `DEC`. Они увеличивают/уменьшают значение на единицу. При этом, если передать унарным операциям регистр, то будет ясно, что они сделают, а в случае с памятью не вполне понятно, они оперируют 8 бит или 16. Поэтому в случае с памятью пишут `INC byte [reg]` или `INC word [imm]`. Аналогично работают унарные операции `NEG` и `NOT` (`dst = -dst` и `dst = ~dst`).
Теперь разберёмся с умножением (`MUL`). Оно в процессоре x86 унарное и выполняется таким образом: аргумент всегда умножается на регистр `AX`. При этом если брать 16-битное умножение, то младшие 16 бит результата пишутся в `AX`, а старшие — в `DX`, а если 8-битное — то аргумент умножается на `AL`, а результат пишется в `AX`. Но это ещё не все, потому что в отличие от сложения, вычитания и прочих уже рассмотренных операция умножение разное в беззнаковых и в знаковых типах данных. Поэтому помимо `MUL` (который выполняет умножение для `unsigned`), есть ещё `IMUL` — для `signed`. Аналогично им есть есть `DIV` и `IDIV`. Они делят также принимают только один аргумент. Эти операции выполняют деление 32-битного числа на 16-битное. При этом 32-битное число берётся из регистров `DX` (старшие 16 бит) и `AX` (младшие). Результат деления считается 16-битным и записывается в `AX` частное. При этом также `DIV` и `IDIV` считают остаток, который пишется в `DX`. Поскольку мы делим 32 бита на 16, возникает вопрос, как поделить 16 на 16. Для беззнаковых чисел всё кристально ясно: надо обнулить старшие 16 бит, то есть обнулить `DX`. Обнулять `DX` рекомендуется при помощи `XOR DX, DX` (она занимает на один байт меньше, чем `MOV DX, 0`). А что со знаковыми числами? Тут в дополнении до 2 надо старшие 16 би заполнить битом знака (то есть старшим битом `AX`). Для этого есть специальная команда, — `CWD`. Теперь давайте узнаем, что будет, если мы делим на 0? Происходит системное прерывание. Причём не в таком случае, а ещё если мы делим 32 бита на 16, а результат не помещается в 16. Но есть ещё один способ получить деление на ноль (не деля при этом на ноль) для этого нужно поделить поделить минимальное целое число на -1; это происходит потому, что -2^{15} влезает в 16 бит, а 2^{15} — нет. Все эти ситуации, как уже было сказано, приводят к системным прерываниям. Системные прерывания работают следующим образом: у процессора есть IDTR — массив, содержащий обработчики прерываний (для каждого типа), который ОС подготавливает при запуске.

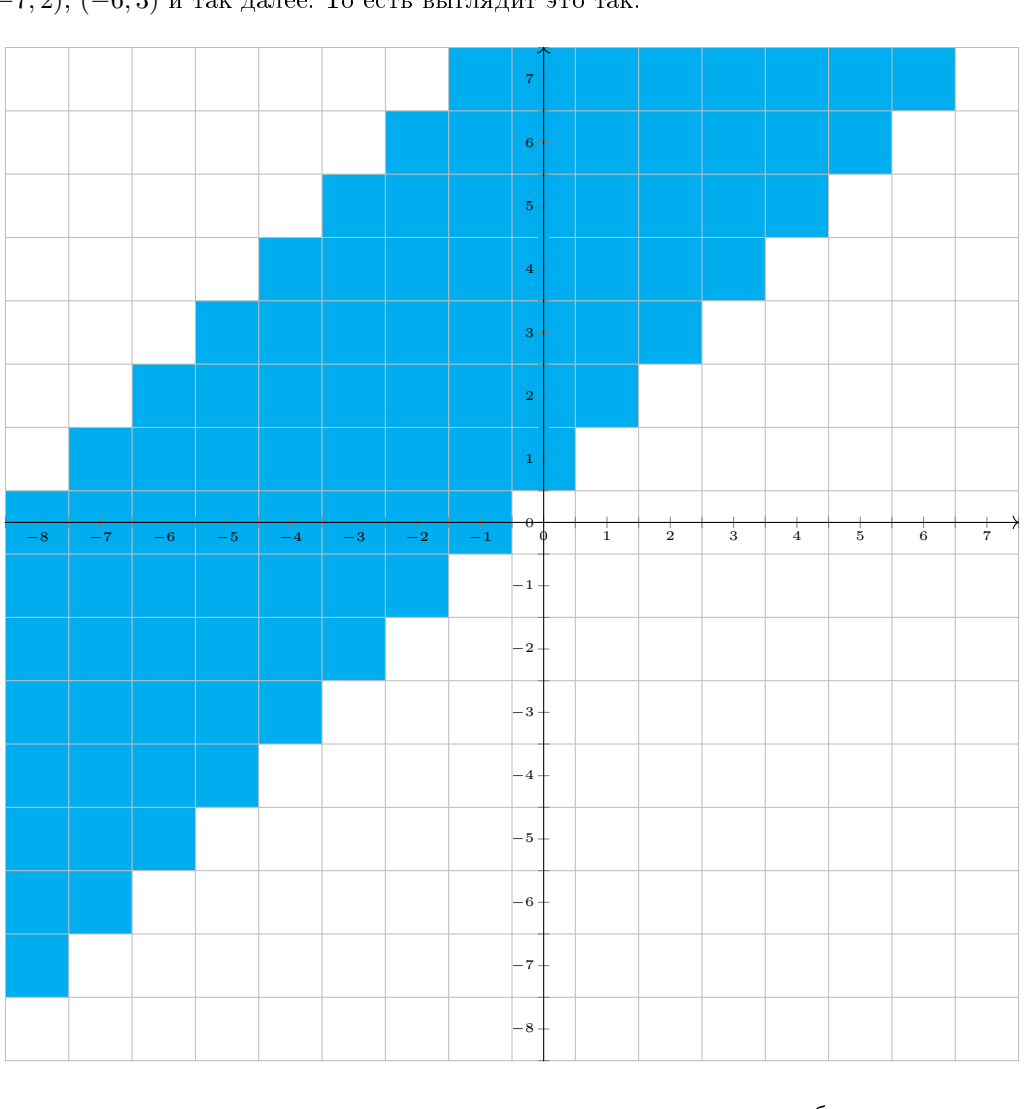
Control-flow. Мы уже понимаем, что компилятор делает с арифметическими выражениями, а значит линейный код мы уже можем перевести на ассемблер руками. А вот ветвления и циклы — пока нет нет. Как они идейно работают? У процессора есть команды, которые называются branch'ами. Самая простая из них — `JMP` (своего рода `goto`). Её аргументом является число, которое нужно прибавить к IP, чтобы перейти на адрес следующей инструкции. В ASM, однако, это вместо этого пишется метка, по которой и осуществляется переход.
Помимо `JMP` есть conditional branch (то есть переход, если выполнено какое-то условие). Осуществляются они комбинацией команды `CMP` и какой-то из команд `JE`, `JG`, `JL` или подобной. Первая команда (пока непонятным нам образом) сравнивает два регистра, а потом вторая получает результат этого сравнения и совершает переход только в определённом случае. В таблице ниже перечислены условия для каждой из команд:

Команда	Эквивалент	Расшифровка
`JE`	`left == right`	jump if equal
`JG`	`(signed)left > (signed)right`	jump if greater
`JL`	`(signed)left < (signed)right`	jump if less
`JA`	`(unsigned)left > (unsigned)right`	jump if above
`JB`	`(unsigned)left < (unsigned)right`	jump if below
`JNE`	`left != right`	jump if not equal
`JNG`	`(signed)left <= (signed)right`	jump if not greater
`JNL`	`(signed)left >= (signed)right`	jump if not less
`JNA`	`(unsigned)left <= (unsigned)right`	jump if not above
`JNB`	`(unsigned)left >= (unsigned)right`	jump if not below

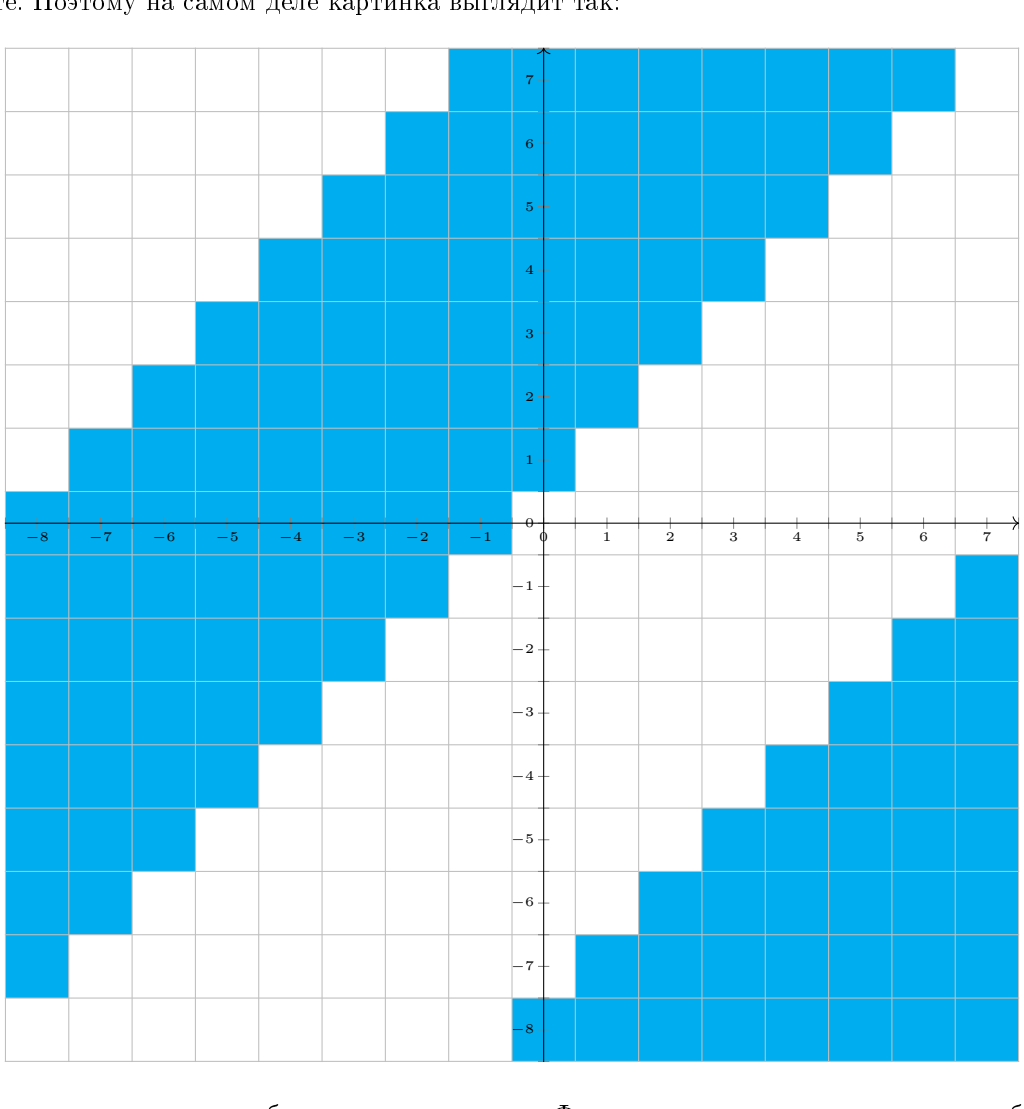
Теперь давайте всё же поговорим, как работает `CMP`? Для этого нам нужно поговорить о такой штуке как регистр флагов. Он содержит, собственно, битовые флаги. Из их большого набора нас интересуют CF — carry flag, ZF — zero flag, SF — sign flag и OF — overflow flag. В процессе своей работы разные инструкции устанавливают различные флаги (хотя какие-то на определённые флаги не влияют, у каких-то инструкций эти флаги значат свой, поэтому читайте документацию). А мы рассмотрим, как с ними работают инструкции, в частности арифметические. `ADD` и `SUB` работают так: они выставляют ZF — если результат равен 0, SF — если результат отрицательный, CF — если произошёл перенос в сложении/заимствование в отрицании беззнаковых чисел, OF — если знаковая операция вызвала переполнение. Так вот, `CMP` устанавливает флаги также как `SUB`. То есть она по сути делает вычитание, но отбрасывает результат. Поэтому, например, `JE` просто совершает переход, когда ZF — единица. Кстати, в контексте флагов было бы не лишним упомянуть, что существуют специальные инструкции условного перехода, которые совершают либо не совершают переход в зависимости от флагов. К таким относятся `JZ`, `JC`, `JS`, `JO`, `JNZ`, `JNC`, `JNS` и `JNO`. Их их названия понятно, что они делают. Кстати, несмотря на то, что в ассемблере есть `JE` и `JZ`, по сути они делают одно и то же и даже одинаковым набором байт обозначаются (поэтому если вы будете ассемблировать-дизассемблировать код, учтите, что в коде ASM одна из этих инструкций может замениться на другую). Теперь давайте определим, при каких комбинациях флагов совершают переходы оставшиеся инструкции. `JB` — понятно, если CF — истина. Также понятно `JA` — если CF и ZF оба ложны. А на какие биты смотрят `JG` и `JL` — хороший вопрос. Чтобы это понять, давайте сделаем график. Оси будут обозначать левый и правый аргумент, а клеточку мы будем закрашивать, если нужный нам флаг будет поставлен при `CMP`. График будет нарисован для трёхбитных чисел, чтобы на нём было всё лучше видно. Например, вот так выглядит область, где поставлен флаг ZF:



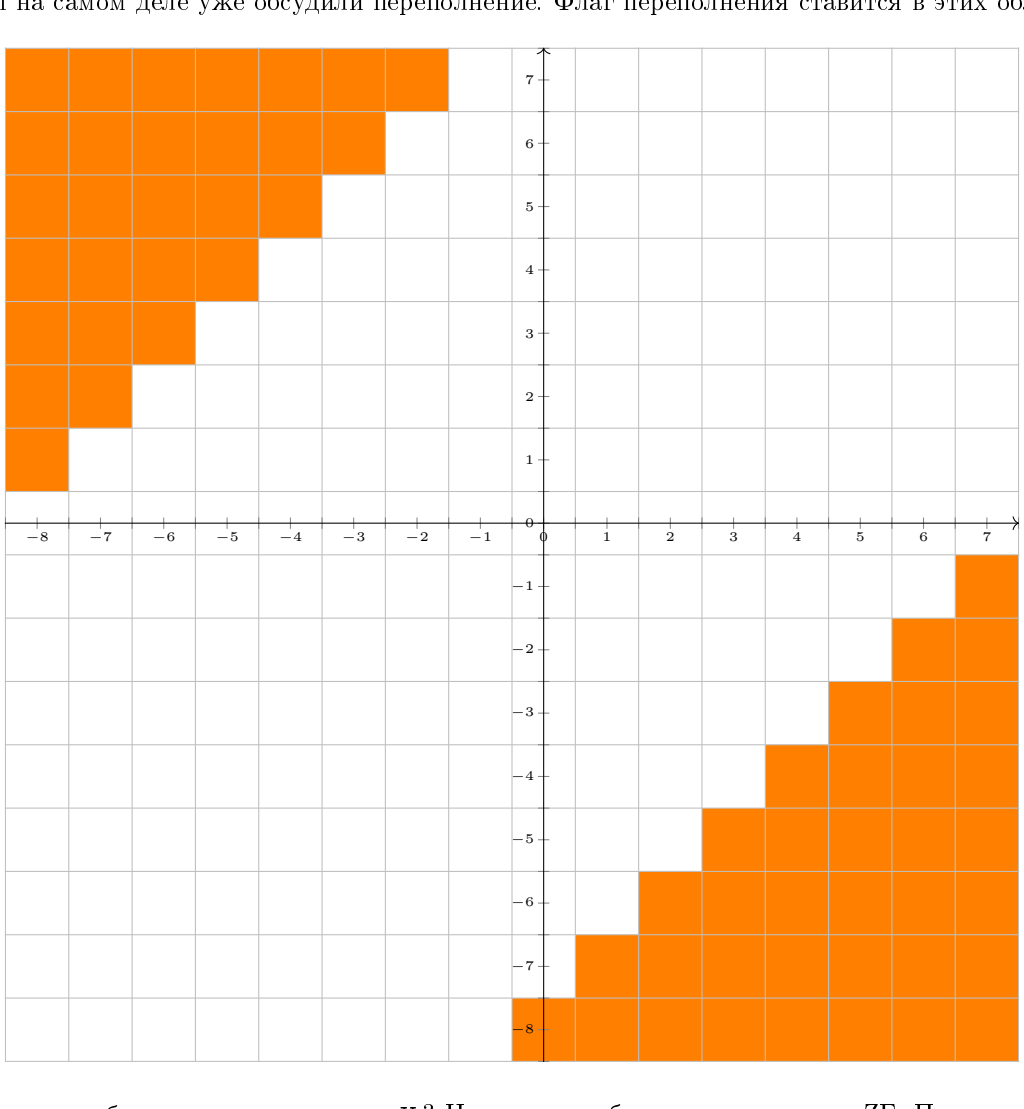
Но он нам не очень интересен, давайте нарисуем OF и SF. Сначала посмотрим на клетку $(0;1)$. Стойт ли там SF? Ну, да, ведь $0-1<0$. А стойт ли он выше? Ну, по-хорошему, тоже да, но в определённый момент мы станем из очень большого отрицательного числа вычитать что-то, и у нас будет происходить переполнение (то есть результат будет положительным). В каком месте это будет происходить? Ну, $(-8;1)$, $(-7;2)$, $(-6;3)$ и так далее. То есть выглядит это так:



Но на самом деле эта картинка неверна, потому что мы можем вычесть из большого положительного числа большое отрицательное и получить переполнение в другую сторону и отрицательное число в результате. Поэтому на самом деле картинка выглядит так:



А ещё мы на самом деле уже обсудили переполнение. Флаг переполнения ставится в этих областях:



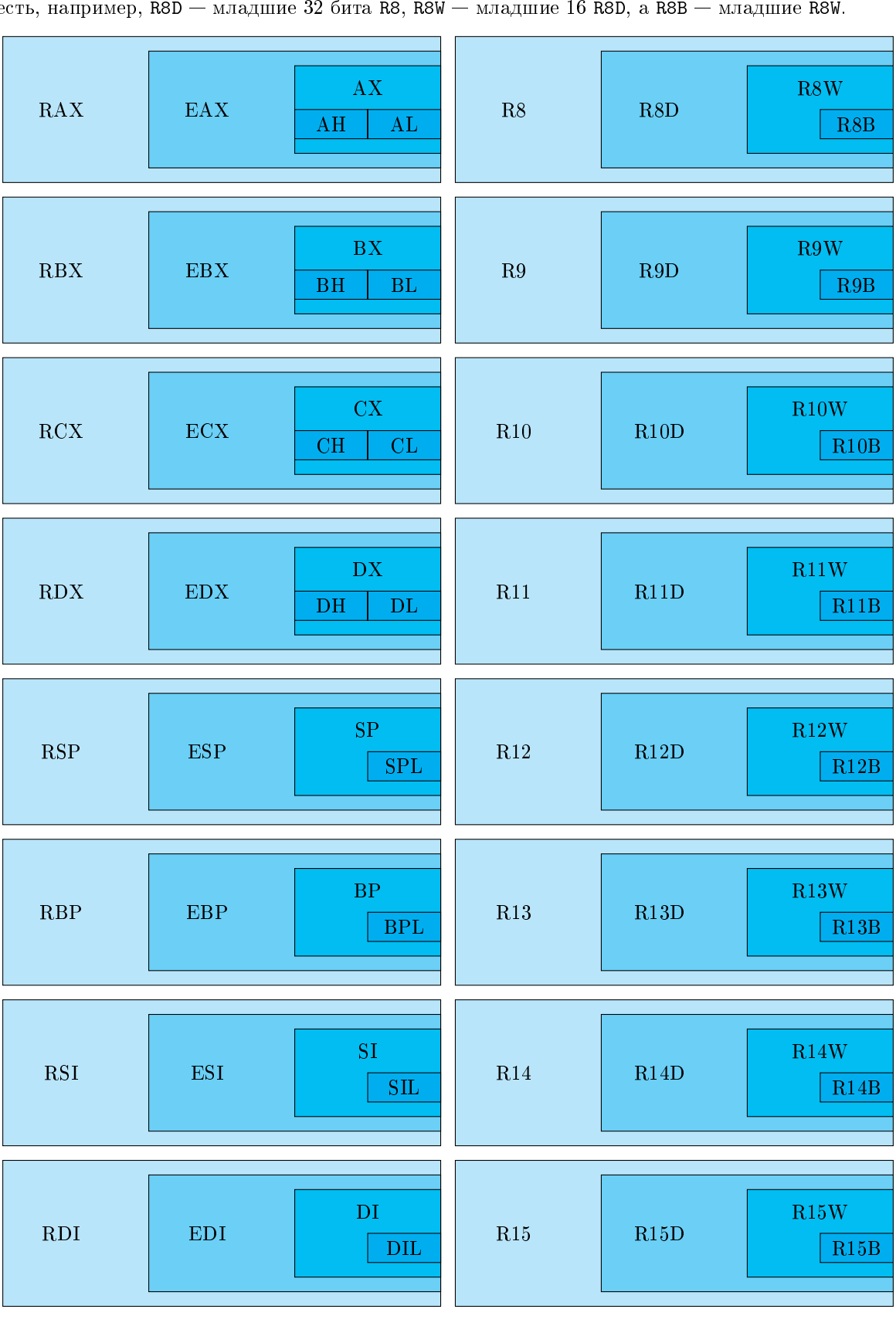
При этом какая область нам нужна для `JL`? Нам нужна область строго выше ZF. Посмотрев внимательно на графики, получим что это SF xor OF либо же SF != OF. А что же нам нужно для `JG`? Ну, очевидно, SF == OF & !ZF. Теперь мы можем составить полную таблицу условий, которые проверяют conditional-branch'и:

Инструкция	Флаги
`JE`	`ZF`
`JG`	`SF == OF & !ZF`
`JL`	`SF != OF`
`JA`	`!CF & !ZF`
`JB`	`CF`

Как определяются условия для остальных команд, либо очевидно (для `JZ`, `JC` и подобных), либо следует отсюда (для `JNE`, `JNG` и т.д.)
остаётся только заметить, что нам не всегда нужно делать `CMP` до условного перехода, ведь иногда у нас флаги уже после каких-то операций стоят как нам хочется. Например, это можно пронаблюдать в таком примере:

```
loop:   MOV DX, AX
        ADD AX, BX
        MOV BX, DX
        DEC CX
        JNZ loop
```

Регистры большей битности. Переносимся из 1978 года в 1985, когда Intel выпускают 32-битный процессор. Тогда появились регистры `EAX`, `EBX`, `ECX`, `EDX`, `ESP`, `EBP`, `ESI`, `EDI`, младшими 16-ю битами которых являются старый набор регистров. А в 2003 годы появилось 64-битное расширение, которое создало `RAX`, `RBX`, `RCX`, `RDX`, `RSP`, `RBP`, `RSI`, `RDI`. Но помимо к этому количество регистров удвоилось; для новых, правда, не стали придумывать названий, они называются по номерам: `R8`–`R15`. Их версии меньшей битности называются `R8D`–`R15D`, `R8W`–`R15W` и `R8B`–`R15B` (от слов dword, word и byte соответственно). То есть, например, `R8D` — младшие 32 бита `R8`, `R8W` — младшие 16 `R8D`, а `R8B` — младшие `R8W`.



Инструменты. Чтобы увидеть, во что ваш компилятор превращает ваш код, пишите *g++ -S -masm=intel ...*, после чего вам создадут *.s* файл с ассемблерными командами. Или можно использовать `https://godbolt.org`, куда можно запихнуть разные компиляторы, параметры оптимизации и прочее, и сразу увидеть результат (причём понятнее, чем в *g++*). Ещё, кстати, на `https://godbolt.org` можно навести на команду и узнать информацию по ней. Если нужно больше информации — лезьте на `https://felixcloutier.com/x86/`.

Оптимизации компиляторов. Если вы скомпилируете код вида `return a + b`, вы не увидите команду `ADD`, а увидите `LEA`. Эта инструкция расшифровывается как load effective address и делает следующее: `LEA` принимает на вход `LEA AX, [BX]`, но не читает адрес памяти с номером `BX`, а просто записывает значение `BX` в `AX`. И в x86 можно записать как адрес что-то такое: `[reg+1/2/4/8*reg+const]`, что позволяет написать `LEA AX [BX+CX]`, что будет быстрее, чем `ADD`+`MOV`.
Ещё в x86 есть `SHL`/`SAL` — сдвиг влево, `SHR` — логический сдвиг вправо (оставляет на пустых местах нули), `SAR` — арифметический сдвиг справа (на пустые места пихает знаковый бит). В C++ для беззнаковых используется `SHL` и `SHR`, а для знаковых — `SAL` и `SAR`. Сдвиг используются также для быстрого умножения и деления числа на степени двойки. Но даже для деления на прочие числа не всегда используется `DIV`. Например, результат компиляции `x / 3` содержит магический хак такого рода: вместо $\frac{a}{3}$ делают $\frac{a\cdot\frac{2^n}{3}}{2^n}$, и там подбирается n, чтобы работало. Много таких хаков можно почитать в книге Hacker's Delight.

Стек. Зачем в процессоре стек? А вот зачем. Для глобальных переменных можно назначить фиксированный адрес памяти. Ну, правильно, а хуле. А с локальными переменными (и параметрами) такое не работает, потому что в программе может присутствовать рекурсия. К счастью, мы заходим и выходим в функции «в порядке стека» (то есть если мы зашли в первую функцию до второй, то выйдем сначала из второй, а потом — из первой), поэтому стек и используется. И процессор предоставляет вам интерфейс для работы с этим самым стеком. Самые простые команды — `PUSH` и `POP`. У нас есть регистр `RSP` (stack pointer), который показывает на вершину стека. Инструкция `PUSH` уменьшает этот адрес (вершина стека имеет наименьший адрес), после чего записывает по нему переданные данные, а `POP` — наоборот — сохраняет текущие данные, сдвигает `RSP` обратно и записывает эти данные по переданному ей адресу. Ещё есть инструкции `CALL` и `RET` — вызов и возврат из функции соответственно. Первая записывает адрес следующей инструкции после вызова в стек и делает `JMP` по адресу вызываемой функции, а вторая — берёт со стека адрес того, куда вернуться, после чего переходит по нему.
Хорошо, вызывать функции мы научились, теперь нам хочется передавать в них параметры. Быстрее всего положить их в регистры (например, `EDI` и `ESI`). Но регистров у нас конечное число, поэтому если параметров много, то может не хватить. В таком случае параметры можно передавать через стек. И чтобы получить их, мы делаем `MOV RAX, [RSP+8]` или `MOV RAX [RSP+16]`. Но после возвращения из функции, у нас параметры всё ещё на стеке, надо исправить. Мы можем сделать две вещи. Можем сделать так, чтобы вызывающая функция почистила стек, прибавляя константу к `RSP` каждый раз. Но не лучше ли запихнуть это в вызываемую функцию? Ну, идейно да, но надо куда-то сохранить возвращаемое значение... В x86 вообще есть `RET n`, который берёт значение, сдвигает `RSP` на `n` и возвращается. Это называется «вызываемая функция чистит стек». Осталось понять, что делать с локальными переменными? Ну, их можно положить выше на стек после адреса возврата.
Хорошо, а кто определяет, как передавать параметры. В каких регистрах, в каком порядке на стеке, как возвращается результат, кто чистит стек и т.д. Процессору чисто поебать. А вот разработчикам компилятора надо бы договориться, как жить. Все эти соглашения — «соглашения вызова функции». Исходно это определяет ОС — ОС компилируется неким компилятором, он что-то использует, а значит теперь ОС работает именно так. Windows в данном случае является исключением, потому что там есть много разных соглашений (cdecl, stdcall, thiscall, fastcall...). Чтобы полностью все их понять, открываете доку и читаете. Итак, кто чаще чистит стек? Вызывающая функция. Почему так, это же, вроде как, неудобно? Потому что variadic-функции. Когда вы делаете `printf`, компилятор просто пихает параметры на стек, а вызываемая в душе не ебёт, сколько их. Поэтому чистит вызывающая.
Ещё есть такая штука как «стековый фрейм». Вокруг тела функции пихаются команды `PUSH RBP`, `MOV RBP, RSP`, ..., `POP RBP`. Зачем это? Что делает этот код? Он сохраняет старое значение `RBP` на стек. А при вызове рекурсивной функции в `RBP` будет `RSP`. То есть во внутренней функции мы запихаем на стек адрес первой. И так далее. То есть на стеке есть односвязный список значений `RBP` от нового к старому. Поэтому пойдя по `RBP`, можем напечатать весь стек. Буквально так очень давно работали отладчики. В наше время этот режим менее актуален, поэтому генерировать эти инструкции — избыточно. Вместо этого компиляторы вместе с кодом генерируют отладочную информацию как раз о том, какие команды каким строкам кода соответствуют, как раскручивать стек и подобное. *gcc* генерирует стековый фрейм только с ключом *-g*. MSVC генерирует их по умолчанию. Возможно, это потому, что рандомная опубликованная exe'шка может в отладочных данных содержать секретики. Их вы публиковать не хотите, поэтому только стековые фреймы. К чему это всё для нас? Если вы пользуетесь какими-то инструментами, которые снимают стек (отладчик или профилировщик), убедитесь, что у вас есть отладочные символы, и дебажите вы тоже согласно им. Либо и то, и другое наоборот.
А знаете ли вы, что будет, если засрать стек или целенаправленно сдвинуть `RSP` куда не надо (это же регистр общего назначения, никто же не мешает нам его изменить). Обычно после стека находится немного данных, которые помечены как те, в которые нельзя писать и которые читать. Если вы обратитесь туда, вам дадут прерывание, управление перейдёт ОС, ваша программа умрёт. А вот если вы уйдёте совсем далеко от стека, то вы пойдёте гулять по памяти, и вам никто не поможет.

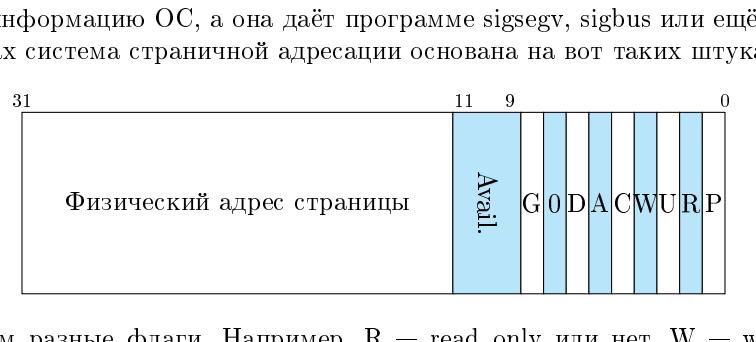
Взаимодействие разных программ. В прошлый раз мы научились исполнять одну программу, но не научились делать абсолютно ничего, связанного с взаимодействием разных программ. И вообще совершенно не узнали ничего о том, как множество программ на вашем компе в принципе уживаются вместе.
Пусть у вас есть программа:

```
int main()
{
    for (;;)
        ;
}
```

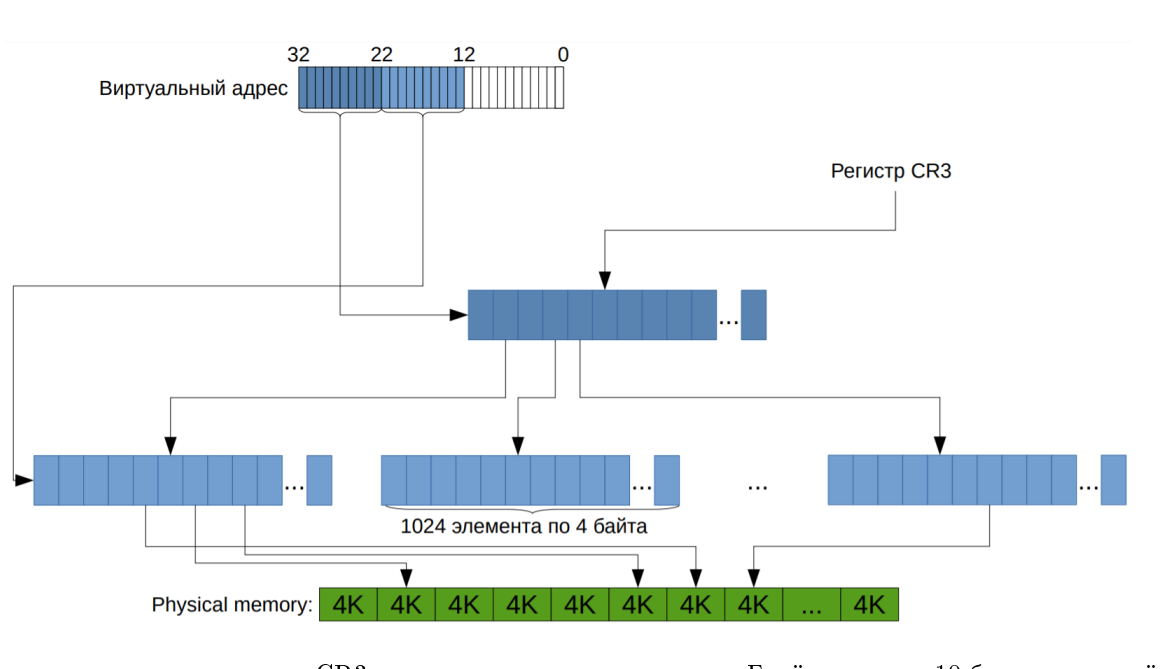
Если мы посмотрим как это выглядит в ASM, то совершенно понятно, как метка и переход по ней же. В нашей предыдущей модели такая программа повесила бы процессор навсегда вообще. Но когда мы это запустим, такого не произойдёт, а значит есть что-то, что позволяет жить в таком случае.

Кратко о прерываниях. Это что-то — прерываний в процессоре. Изначально они были созданы для того, чтобы устройства, которые, например, читают данные, сами оповещали процессор о том, что они дочитали, вместо того, чтобы он постоянно их сам спрашивал. С точки зрения программы это выглядит так: ОС её прерывает, выполняет что-то, после чего возобновляет выполнение там, где прервала.
Если вам хочется посмотреть на прерывания, вам нужен файл `/proc/interrupts`. Там вы можете посмотреть, все типу прерываний с процессорами и описаниями. Например, при нажатии на кнопку и отжатии кнопки на клавиатуре посылается специальное прерывание, с тачпадом то же самое и т.п. Сейчас нас интересует local time interrupts — прерывания, которые можно запрограммировать по таймеру. При загрузке ОС она взводит их, чтобы мочь получать из программы управление назад. При этом она сохраняет куда-то состояние регистров остановленной программы, а при запуске её обратно — восстанавливает.
При этом ждать, пока будет прерывание, Вы не можете (если вы не ядро; если ядро, у вам есть команда `HLT`, которая это и делает). Ядро ОС умеет усыплять ядра процессора, а изнутри программы мы это сделать не может — максимум вызвать что-то из ОС, что потом вас прервёт. Например, читать из stdin, где нет данных или делать sleep.

Работа с памятью (на самом деле). Теперь давайте поговорим про второй вид ресурсов — память. Что будет, если мы возьмём рандомное число, кастанём его в `char*`, после чего запишем по нему букву. Будет SEGFAULT. Что это такое вообще? Чтобы ответить на этот вопрос, надо понять, как в одной памяти живут программы. В процессоре есть механизм, который позволяет делать две вещи. Первая — виртуализация (для программы создаётся иллюзия, что вся память — её память). Вторая — изоляция программ друг от друга. То есть одна программа без разрешения не трогает данные другой. ОС использует эти функции, чтобы испортить чужие данные вы не могли.
Хорошо, как конкретно это работает? В нашей прошлой модели память — пронумерованные ячейки. Мы будем называть *физической памятью* то, что у нас в жизни в оперативке. Этот уровень недоступен даже ядру ОС, максимум можно догадываться, что там. Вместо этого когда вы обращаетесь к памяти, процессор этот адрес как-то преобразует (*транслирует*) в физический адрес, по которому и обращается. У вас нет способов этого избежать, максимум (если вы в ядре) — слабо повлиять. И суть в том, что пересчёт разный для разных программ, поэтому в разных программах одно и то же число адреса — разные ячейки физ. памяти.
Набор адресов физической памяти — *физическое адресное пространство*, а то, что видит программа — *виртуальная память*. На базовом уровне механизм в процессоре делает вот что: разделим физическую и виртуальную память на кусочки по 4 КБ, которые называют страницами. Процессор может настроить механизм так, чтобы произвольная страница виртуальной памяти транслировалась в произвольную физической. Этот механизм — *механизм страничной адресации*. В принципе, ОС ничего не мешает сделать две страницы виртуальной памяти и направить их в одну страницу физической. Работать это будет так, как вы думаете. Если делать это из разных программ, они смогут общаться. Ещё мы можем запрещать какие-то страницы (помечая их как отсутствующая). То есть считается, что эта страница никуда не транслируется. Именно поэтому и происходят SEGFAULT'ы. Соответственно, процессор даёт эту информацию ОС, а она даёт программе sigsegv, sigbus или ещё чего.
В 32-битных системах система страничной адресации основана на вот таких штуках:



Сначала адрес, затем разные флаги. Например, R — read only или нет. W — write through — про кеширование, прочие флаги содержат другие данные для ОС. В 64-битных то же самое, но адрес побольше, и сама структура занимает больше бит. Такие структуры хранятся в памяти в сложной структуре в виде дерева. Как конкретно? Если подойти наивно, то получится набор страниц, для каждой храним 32-битное число. То есть имеем массив таких структурок. Это работает, но имеет одну проблему: в 32-битном режиме имеем 4ГБ памяти, то есть нужно 4МБ памяти на процесс. Это до жопы, особенно учитывая то, что в древности 4МБ — это типовой размер был. А в наше время в 64-битном режиме на одну программу понадобится 32ПБ. Поэтому заметили вот что. Большая часть программ используют меньше 4ГБ памяти. А значит бо́льшая часть страниц недоступна. Поэтому давайте вместо 1048576 элементов хранить 1024*1024 (*таблица страниц*). И мы можем просто в изначальном массиве на 1024 элемента поставить недоступность, чтобы пометить недоступным сразу много страниц. Хорошо, а как корень дерева найти? Для этого есть специальный регистр CR3, который и переключается при смене исполняемой программы.



То есть процессор смотрит на CR3, получает массив структурок. Берёт старшие 10 бит адреса, идёт в этот массив и берёт нужную оттуда. Там есть адрес, по которому можно получить следующий массив. Со следующими 10 битами идёт в нужный элемент второго массива, где берёт требуемую страницу. Остаток адреса означает, какой байт в этой странице надо взять.
А что в 64-битном режиме? Там виртуальный адрес сильно больше (64 бита, собственно). Ещё изменение — больше физической памяти. Размер структурки также удваивается, а значит структурок в одном массиве теперь 512, а не 1024. То есть адрес мы уже делим на кусочки по 9 бит, а не по 10. Но даже в 3 уровня останется куча лишних битов. Тогда «лишние» биты должны совпадать со старшим не-лишним. Имеем по сути 39-битный виртуальный адрес. Современные процессоры уже и 4 уровня поддерживают, что даёт нам 48 бит. А самые новые intel'овские процессоры дают нам 5 уровней и 57 битов (127 ПБ можно адресовать).
Как это используется ОС? Чтобы раздавать память — понятно, но из интересного есть memory-mapped файлы (дай мне в память файл, и теперь когда вы туда обращаетесь, вам дают иллюзию, что вы тупа обращаетесь в этот файл). Преимущество этого — ОС читает только те страницы, которые вам нужны. Второе преимущество — если несколько программ так сделали, вы можете переиспользовать физическую память (то есть всем давать один и тот же кусочек памяти, совсем одинаковый). Так, файл в 2 ГБ будет исполняться мгновенно, потому что ОС mapp'ит файл в память, а подчитывает всё нужное потом. Ещё с memory-mapped связано то, что вы можете mapp'ить то, что больше памяти. Тогда ОС просто будет читать с диска заново то, что не помещается.
Ещё из интересного есть уже обсуждённая разделяемая память. Это самая низкоуровневая вещь для взаимодействия программ. А вообще есть, например, PIPE'ы, реализованные на базе разделяемой памяти.
Интересный вопрос: можете ли вы зажрать больше памяти, чем есть? Как ни странно, да, тогда ОС создаст файл подкачки (swap) на жёстком диске, где и будет хранить память. Процессор об этом ничего не знает, если что.
Ещё есть механизм copy-on-write — это вот что. Обычно недостаточно просто дать программе управление и всё. А ещё бывает нужно исправить то, что мы загрузили. То есть формально файл не совсем такой, как прочитано. Тогда при записи мы делаем копию памяти, а то, что не менялось — переиспользуем.

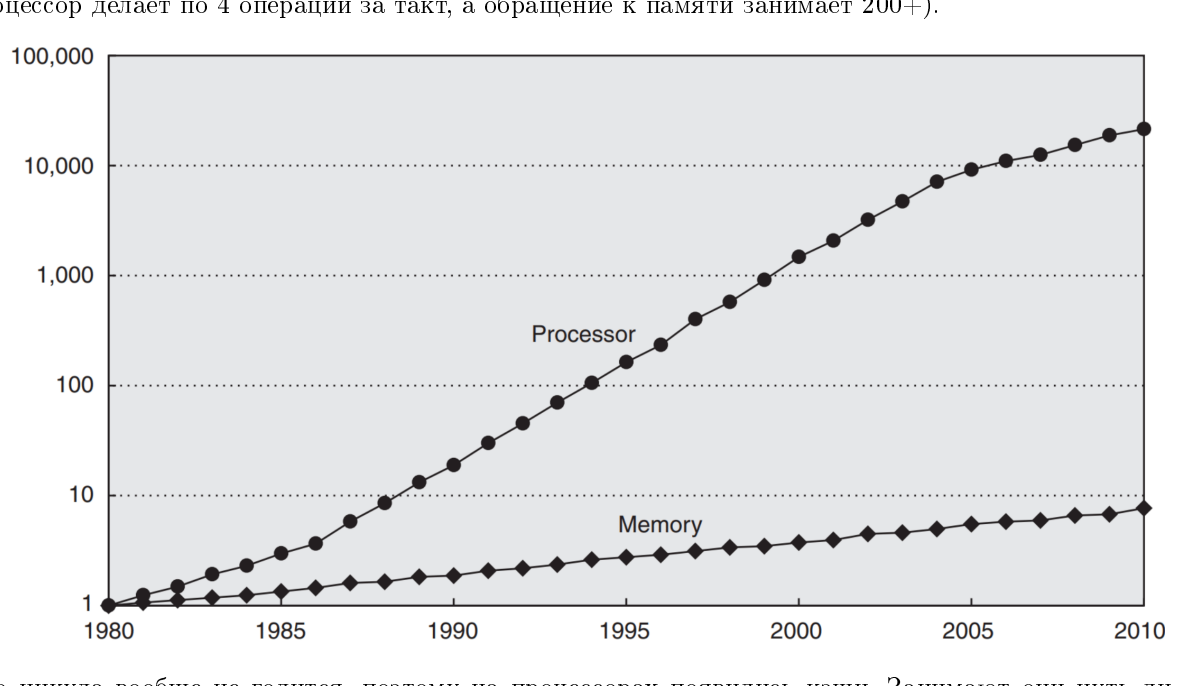
Уточнение модели процессора. Прошлые разы поговорили об одной программе, о нескольких программах, одну программу мы даже написали, а сегодня поговорим о том, какие операции процессора дорогие, а какие — дешёвые. Потому что нам надо знать, как оптимизировать программы. Скорее всего, наш код, если переписать его в плюсы, а потом скомпилировать, то результат будет оптимальнее. У нас и не было цели сделать быстро, но всё же. И сегодня мы поговорим о различных интересных аспектах процессора.
Если мы занимаемся оптимизацией, то какие-нибудь высокоуровневые оптимизации (когда что-то считается много раз, например) мы можем сделать легко, а на более низком уровне придётся знать подноготную процессоров, потому что там всё работает не так уж интуитивно.
Мы рассмотрим несколько примеров, которые в нашу текущую модель не укладываются вообще, разберёмся в них и уточним модель. При этом мы не стремимся получить максимальную детализацию модели. Во-первых, если у нас есть эффект, дающий нам 0.01%, он нам не очень критичен. Во-вторых, чтобы анализировать и делать выводы, нам придётся нашу модель сделать проще. Физики вон, упрощают реальный мир, чтобы делать свои предсказания.
Сначала разберёмся, о каких процессорах мы будем говорить. Ответ — в каком-то смысле про все современные (последние 17 лет) процессоры. Причём вне зависимости от архитектуры. И это не заговор разработчиков, а в течение 40 лет все думали, как процессоры сделать быстрее и энергоэффективнее, и то, что оптимизации поддавалось, то оптимизировали. А то, что поддавалось оптимизации сложно, то оставалось медленным. Например, операция деления занимает много больше времени, чем, скажем, сложение. И это не потому, что разработчики процессоров дураки или лентяи, а потому что есть внутренняя сложность в операции деления.

Кэш. Начнём с совсем простой модели: процессор делает инструкции одну за другой, время работы пропорциональна их количеству. Это, на самом деле, далеко не самая плохая модель, но она имеет ошибку на таком примере:

```
for (size_t i = 0; i < N; ++i)
    for (size_t j = 0; j < N; ++j)
        a[i][j] = 0;
```

```
for (size_t i = 0; i < N; ++i)
    for (size_t j = 0; j < N; ++j)
        a[j][i] = 0;
```

Разница в этих примерах будет очень большая (второй медленнее), из-за причин, обсужденных в курсе АрхЭВМ. Если посчитать зависимость среднего времени одного присваивания в зависимости от N. Это **не** объясняется специальными инструкциями процессора, который обращается в несколько ячеек последовательно за раз, потому что мы можем написать то же самое на ASM, результат будет тот же. А дело, как мы знаем, в процессорном кэше.
Зачем существует кэш вообще? Как мы знаем, согласно закону Мура, мощность процессоров от времени зависит экспоненциально. Так вот памяти — тоже. Но скорость памяти растёт сильно медленнее (сейчас процессор делает по 4 операции за такт, а обращение к памяти занимает 200+).



Это никуда вообще не годится, поэтому на процессорах появились кэши. Занимают они чуть ли не половину кристалла, и так было что в пентиумах, что во всех современных процессорах. Типовое время обращения к памяти — 4, 12, 36 и 230 тактов для L1, L2, L3 и RAM соответственно. Поэтому сначала надо оптимизировать данные, а уже потом код.

Префетчинг. Префетчинг — это метод, при помощи которого заранее определяется, к каким областям памяти в будущем пойдут обращения. Осуществляется это, например, при помощи улавливания последовательных обращений к памяти (если вы сначала обращаетесь к первой ячейке, а потом — ко второй, то можно заранее начать обращение к третьей, вдруг вы туда обратитесь). В современных процессорах есть аж целых 4 разных префетчера, два для L1 и по одному для L2 и L3. Идейно префетчинг позволяет тратить ресурсы процессора на то, чтобы ускорять память, что полезно всё по той же причине — память медленнее.

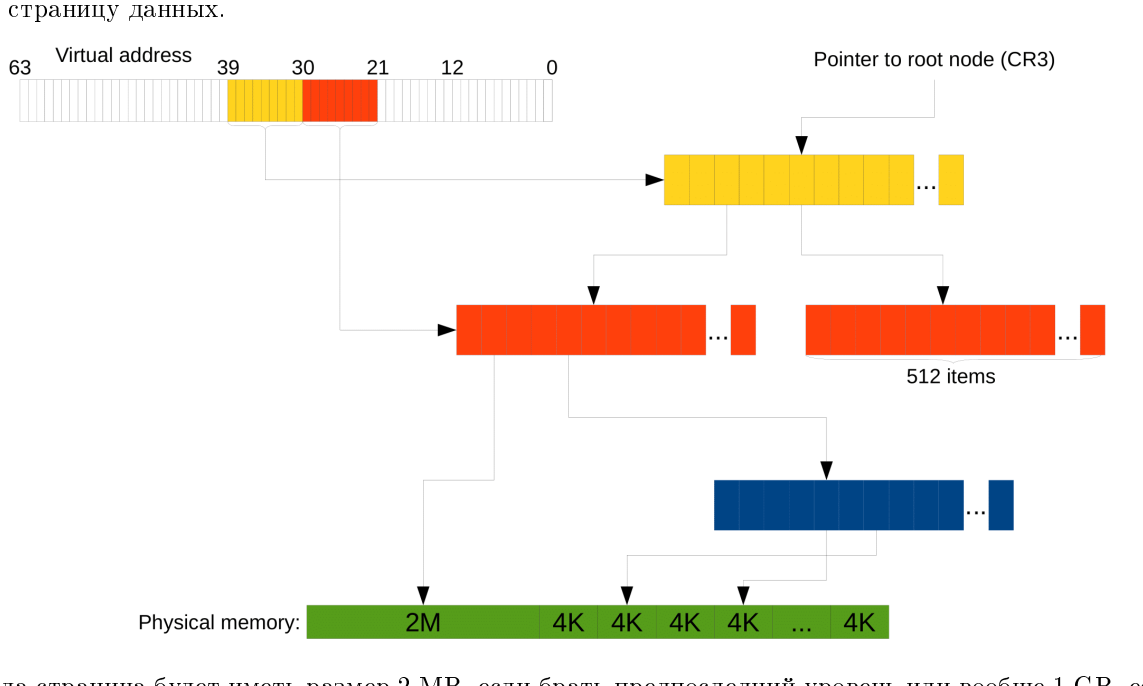
Промежуточные выводы касательно памяти. Касательно памяти есть следующие практики:

- Иногда выгоднее будет что-то пересчитать, чем хранить всё в памяти (процессор быстрее же).
- Если вы знаете, в каком порядке вы будете обращаться к данным, важно правильно их положить (тоже последовательно).
- Хранить «горячие» и «холодные» данные нужно в разных местах. «Горячие» — это которые вам часто нужны, «холодные» — которые не очень. Почему в разных? Потому что иначе у вас в кэше будут лежать и «горячие», и «холодные» данные, вместо того, чтобы забить его «горячими» полностью.

Как, например, правильно реализовать хэш-таблицу с открытой адресацией? Мы можем хранить всё парами из ключа и значения, а можно — сначала все ключи, а потом все значения. А вот что лучше, зависит от hit rate'а нашей таблицы. Если мы чаще находим значения, мы смотрим на ключ, и берём значение, которое рядом с ним, удобно. А если мы редко их находим, то мы ведь долго бежим чисто по ключам, а значит нам выгоднее будет сохранить их вместе.
Ещё сто́ит избегать непрямых обращений к памяти, потому что обращение по рандомному указателю — это, здравствуйте, промах по кэшу. Вместо этого можно просто заменить этот указатель на сами данные. С другой же стороны, если у вас указатель на «холодные» данные есть в «горячих», то тупо вставлять эти данные внутрь не очень выгодно — как уже было сказано «горячие» и «холодные» надо бы хранить в разных местах.

Отступление про указатели. Что ещё можно сказать про указатели? А то что ваша программа процентов на 80 состоит из них. Так, `std::vector` — это три указателя, `std::list` — два указателя и одно число размером с указатель, `std::set` и `std::map` — один указатель и одно число. При этом указатели на x86-64 здоровые, 64 бита занимают. А ведь вам обычно не нужны 64-битные указатели, вам хватит и 32 бит, а значит памяти они занимают вдвое больше, чем могли бы. Поэтому Intel сделали ABI x32. Для ОС это выглядит как 64-битная программа, но внутри она имеет 32-битные указатели. Во всём остальном она ~~верблюд~~ x86-64. Конвенции вызова там точно такие же, вещественная арифметика — тоже, количество регистров — 15 целочисленных и 16 вещественных. К сожалению, этот ABI не очень прижился, что довольно грустно. Но что интересно, на целочисленных тестах x32 было на 5–8% быстрее, чем x86-64 (просто из-за того, что данных меньше), на дробных данных разницы нет, а на одном специфичном бенчмарке (там просто куча указателей) ускорение достигало 40%. А в сравнении с x86, ускорение было соответственно в 7–10% и 5–11% для целых и дробных тестах, а специфичный бенчмарк был с 64-битной арифметикой (и тоже имел 40% ускорение).

Ускорение трансляции адресов. Давайте вспомним про виртуальную память. Имея нашу древовидную структуру, на каждое обращение к памяти в программе нам придётся делать 4 (или 5, если уровней 4) обращения в память внутри процессора (чтобы пройтись по дереву). Это треш, поэтому результат трансляции виртуальной памяти в физическую также кэшируется. Буфер, в который кэшируется, называется TLB (translation lookaside buffer). А значит это всё, что если наша программа обращается к реально большому объёму памяти, то мы платим не только за этот факт сам по себе, но и за трансляцию.
Что ещё может улучшить сию неприятную ситуацию с трансляцией? Страницы по 4 KB достаточно маленькие, и заниматься администрированием их довольно накладно. Поэтому в процессоре есть механизм Huge Pages, который позволяет нам на промежуточном уровне дерева установить специальный бит, который говорит, что этот промежуточный элемент хранит не ссылку на следующий, а сразу страницу данных.



И тогда страница будет иметь размер 2 MB, если брать предпоследний уровень или вообще 1 GB, если третий снизу. Это экономит и время трансляции, и место в TLB.

Выделение памяти ядром. Как мы помним, выделение памяти из ядра осуществляется при помощи `mmap`. Можно заметить интересный эффект: если мы просто делаем выделение памяти кучу раз, это будет относительно быстро, а когда мы начнём к выделенной памяти обращаться, ты мы зависнем надолго. Почему так? Точнее на вопрос «почему» ответ простой — потому что ОС даёт нам память только тогда, когда мы её используем, а вот «зачем» — интересный вопрос. Дело в том, что обычно у нас нет свободной оперативной памяти, она вся либо отдана другим программам, либо используется как дисковый кэш. Поэтому когда кто-то просит память, вам придётся сбрасывать дисковый кэш, либо выгружать кого-то в swap-файлы. А если вдруг ваша программа будет использовать выделенную память как-то потом (или не будет вообще), то давать ей память сразу невыгодно, поэтому ОС даёт её при обращении и маленькими кусочками (ведь используете вы маленькими кусочками, а `mmap` выделяет сразу кучу памяти).
Кстати, в `mmap` есть специальный флаг (`MAP_POPULATE`), который заставляет выделять память сразу. При этом, если сделать так, то работать это будет быстрее, чем если выделить память обычным образом, а потом к ней пообращаться, потому что с флагом не будет происходить множество переходов между userspace'ом и ОС при каждом обращении.
Хорошо, а что в `mmap` занимает больше всего времени? А обнуление выделенной страницы. Зачем? А вдруг там кто-то криптографию оставил. Кто-то, кого вы только что убили, например. Но вообще под это тоже есть специальный флажок `MAP_UNINITIALIZED` (но только для анонимных страниц, правда), который был создан специально для миниатюрных устройств. И работает он только в том случае, если ОС была собрана специальным образом. Если, кстати, делать `mmap` без флага `MAP_POPULATE`, то ещё больше времени, чем на зануление, будет тратиться на передачу обращения между ОС и userspace'ом.

Продолжаем уточнять модель процессора. Теперь мы поговорим о чуть менее затратных вещах (которые дают накладные расходы меньше 200 тактов).

Branch prediction. Мы уже давно знаем о том, что эффективность условного перехода зависит от того, насколько хорошо он предсказывается. Это даёт расходы в 14–19 тактов. Отсюда хочется писать код без условных переходов. Для этого рассмотрим пример с делением на 2. Деление на 2 нельзя сделать арифметическим сдвигом, потому что округление идёт не туда. И вообще, можно было это заifать, но условие — это вроде как плохо, вместо этого компилятор хитрит и делает несколько арифметических операций, которые по сути делают то же самое, что делал бы условный переход.
Касательно предсказаний, процессор умеет предсказывать много различных паттернов, поэтому искусственно составить что-то, что будет сложно предсказываться, довольно проблемно. Но это не единственное, что процессор умеет. Давайте рассмотрим генератор случайных битов, который называется линейный регистр сдвига с обратной связью. (Раз биты случайные, то предсказываться должно просто отвратительно.) Он работает примерно так:

```
unsigned lfsr = 0xace1;
for (unsigned i = 0; i != 1000000000; i++)
{
    unsigned lsb = lfsr & 1;
    lfsr >>= 1;
    if (lsb)
        lfsr ^= 0xb400;
}
```

Его можно написать без условия двумя способами:

```
unsigned lfsr = 0xace1;
for (unsigned i = 0; i != 1000000000; i++)
{
    unsigned lsb = lfsr & 1;
    lfsr >>= 1;
    lfsr ^= 0xb400 * lsb;
}
```

```
unsigned lfsr = 0xace1;
for (unsigned i = 0; i != 1000000000; i++)
{
    unsigned lsb = lfsr & 1;
    lfsr >>= 1;
    lfsr ^= 0xb400 & -lsb;
}
```

Если сравнить время работы этих программ, то они будут отличаться так, как ожидается (первая самая долгая, третья — самая быстрая), но ненамного. А ещё, если проанализировать программу какими-нибудь прошаренными средствами, то можно заметить, что неверные предсказания дают нам 0.01% от всех предсказаний. А дело в том, что компилятор породил инструкцию `CMOVNE`, которая является условным переходом без условного перехода. Компилятор просто посчитал обе ветки и решил, что нам нужна вот эта специальная инструкция, которая делает `MOV`, но только если не установлен Z флаг. Очень удобно, что процессор умеет вот в такие штуки.
Хорошо, но всё же что будет, если правда сделать условный переход? Для этого компилятору надо дать ключи *-fno-if-conversion -fno-if-conversion2*, которые заставят его не использовать эту ебейшую инструкцию, и вот тогда уже всё замедлится намного сильнее.
Что ещё нужно знать? Ещё нужно знать, почему такой код

```
for (size_t c = 0; c < size; c++)
    if (data[c] >= 128)
        sum += data[c]
```

работает быстрее, если предварительно отсортировать массив. Ну, тоже потому что предсказание перехода. Если данные случайны, то переход случайно происходит, его не предскажешь. А в отсортированных данных вы сначала всегда не переходите, а потом всегда переходите.

Спекулятивное взаимодействие с памятью. У нас есть чтения и записи. По-хорошему, нельзя читать до того, как закончат выполняться записи, потому что в случае, если мы читаем то, что писали, хочется прочитать то, что записано, а не что-то левое. Но процессор выполняет операции спекулятивно, в надежде на отсутствие коллизий. Поэтому код

```
uint32_t count[256] = {};

for (size_t i = 0; i < size; i++)
    ++count[src[i]];
```

Будет работать тем быстрее, чем более разные данные в `src` (если данные одинаковые, процессор спекулятивно запускает несколько инкрементов за раз, потом понимает, что они инкрементят одно и то же и откатывает все, кроме одного; в результате не только простаивает superscalar, но и происходят откаты туда-обратно). Чтобы это пофиксить, придётся делать так:

```
uint32_t count[8][256] = {};

size = size / 8 * 8;
for (size_t i = 0; i < size;)
{
    ++count[0][src[i++]];
    ++count[1][src[i++]];
    ++count[2][src[i++]];
    ++count[3][src[i++]];
    ++count[4][src[i++]];
    ++count[5][src[i++]];
    ++count[6][src[i++]];
    ++count[7][src[i++]];
}
```

Ссылки. Подробнее про это все хаки и финты можно почитать в двух книжках: J. Shen, M. Lispasti — Modern Processor Design: Fundamentals of Superscalar Processors и J. Hennessy, D. Patterson — Compiuter Architecture: A Quantitative Approach. Они сильно выходят за рамки нашего курса, но тем не менее интересны.

2 Введение в C.

Раньше мы разговаривали про какие-то ассемблерные штуки и познакомились с тем, как (упрощённо) работает компьютер. Это раздел кончился, и мы начнём говорить про C++. И начнём мы с базовых вещей, которые появились в C. И вообще текущая тема может быть также названа введением в C++. Однако C и C++ — это разные языки: во-первых, их разрабатывают разные люди с разными целями. Во-вторых, они имеют разные компиляторы, несмотря на то, что обычно компании, имеющие компиляторы C++ также имеют компилятор C. Эти компиляторы имеют общий код, но они всё же не одинаковы — этот самый общий код также используется и для Go, и для D, и для Ada... Правда, clang является исключением из этого правила, там просто `if`'ами различаются C и C++. Правда, там ещё и Objective-C и нечто ещё. В-третьих, стили программирования на C и C++ кардинально отличаются, если вы пишете на них одинаково, вы дурачок. А в-четвёртых, C не является подмножеством C++, случайная программа на C вообще не факт что будет корректна в C++. Правда, обычно придётся менять её не очень сильно. Примером такого отличия является код вида `a ? b : c = 42`. В C — это `(a ? b : c) = 42`, а в C++ — `a ? b : (c = 42)`.
Так вот, у нас всё будет обсуждаться в терминах C++: в местах отличий мы не будем обсуждать оба языка. Поправка по курсу. Считается, что мы знаем, что такое переменная, как вообще всё живётся, а обсуждать будет то, о чём либо редко говорят, либо о том, что специфично для C/C++.

Типы данных.

Целочисленные типы. Объявление целочисленного типа — знаковость, размер и слово `int`. При этом всё из этого можно опустить, по умолчанию считается обычный размер и `signed`. Знаковость — `signed` и `unsigned`, размер — `short`, размер по-умолчанию, `long` и `long long`.
Ещё есть символы. `char`, `unsigned char` и `signed char`. И это три разных типа. Да, это правда, что `char` по диапазону значений совпадает с одним из остальных двух, но типы это разные.
Кстати, как проверить, что два типа — это одно и то же? Можно определить перегрузку функции `void f(int)` и `void f(signed int)`, и тут будет ошибка компиляции, потому что `int` и `signed int` — одно и то же. А вот если написать

```
void f(char)
{}

void f(signed char)
{}

void f(unsigned char)
{}
```

То это будет исправно компилироваться.
Хорошо, мы уже поговорили, что есть 4 разных размера. А какого они размера в байтах? А вот хрен знает. Вообще в стандарте не специфицировано, там лишь написано, какой минимальный размер они имеют и кто меньше кого. На практике же можно встретить не так много вариантов. На 32-битных системах размеры — 1/2/4/4/8 байт, на 64-битном Windows — тоже, а на 64-битном Linux — 1/2/4/8/8. Разумеется, вы можете найти экзотическую архитектуру, где значения будут другими, поэтому не надо никак опираться на данные, которые вы тут видите. Если вам чётко 2 байта, вам придётся брать заголовочный файл `<cstdint>` и брать оттуда типы `int8_t`, `uint8_t`, `int16_t`, `uint16_t` и подобные прочие. Разумеется, сам этот заголовок делает `typedef`'ы на базовые типы, но на какие именно — вы никогда не знаете.
Хорошо, а как жить в таком коде?

```
for (??? i = 0; i < 100; i++)
    arr[i] = 42;
```

Какого типа должно быть `i`? Для этого есть специальный тип — `size_t`, который необходим и достаточен для обращения в память. Если вы будете брать тип фиксированной длины больше, чем `size_t` — будет немного медленнее, а если меньше — то может не хватить для адресного пространства. К тому же компилятор может немного хуже оптимизировать код, если вы используете битность меньше `size_t`. У `size_t` есть знаковый аналог — `ptrdiff_t` — это результат разности двух указателей.
Как выбрать тип для переменной? Если вы получаете значение как возврат функции, то не выёживайтесь и пишите то, что написано в библиотеке, вне зависимости от того, что там, даже если какой-нибудь `DWORD_PTR`. В противном случае проверьте, не считаете ли вы объекты в памяти, или не является ли у вас ваша переменная индексом массива. Если да — то берёте `size_t` или `ptrdiff_t`. Если же это тоже не так, то берёте тип фиксированного размера в зависимости от предметной области. Это алгоритм не полон, и мы ещё поговорим о нём. Например, что делать, если вам нужно перечислить все типы (например, вы пишете свой `std::numeric_limits`)? Тут уж придётся перечислять все 11 типов языка (а не 8 фиксированной битности).

Перечисляемый тип. Самым простым составным типом является перечисляемый тип. В C++ мы будем использовать только строгий их вариант — `enum class`. Это кто? Это перечисление набора вариантов. Зачем это используется? Ну, вместо этого можно использовать константы, но `enum class` безопаснее (в него сложно присвоить то, чего в нём нет), понятнее (если в функцию передаётся три разных `int`'а, то вы легко их перепутаете, а перепутать три разных перечисляемых типа вам не дадут). А ещё когда пишутся константы, не видно, что они как-то связаны, а в `enum class`'е явно видно, что «вот эти штуки» — они вместе. Изнутри `enum class` — это просто `int`, причём вы можете ещё и явно указать, какой конкретно `int` при помощи такой конструкции:

```
enum class color : uint16_t
{
    red,
    green,
    blue
};
```

Но арифметика с перечисляемыми типами (строгими) не работает.

Структура. Следующая вещь — структура. Это способ сгруппировать набор данных в одну сущность. Этим прям надо пользоваться, не надо таскать в разные места несгруппированные данные. Иначе ваш код раздувается до невероятных размеров и вообще обретает форму спагетти. А ещё вы увеличиваете вероятность ошибиться.
Поговорим ещё о том, как структуры лежат в памяти. Если мы сделаем структуру

```
struct point
{
    int x, y, z;
};
```

То у вас эта штука занимает 12 байт, первые 4 — это `x`, следующие — `y`, последние — `z`. А сколько памяти занимает

```
struct mytype
{
    char a;
    int b;
};
```

А вот не 5, а 8. Это связано с тем, что процессоры либо не умеют, либо плохо читают N байт, адрес начала которых не делится на N (это называется невыравненное обращение к памяти). Поэтому компиляторы стараются располагать структуры так, чтобы данные в них были выравнены. В нашем случае в структуре есть один байт под `a`, три пустых байта и ещё 4 байта под `b`. Хорошо, давайте поправим данный пример:

```
struct mytype
{
    int b;
    char a;
};
```

Тут будет 5? Нет, не будет, потому что вы можете сделать массив таких структур, и тогда во всех, кроме первой `b` будет не выравнен. Поэтому всё равно 8.

Объединение. У нас есть тип, которых хранит первое, **И** второе, **И** третье. А что, если мы хотим хранить строго одно из нескольких значений? Специально для этого есть тип `union`, который этим и занимается. Пока `struct` хранит следующее поле по смещению относительно предыдущего, у `union`'а всё хранится по одному смещению. **Важно**: никакой информации о том, что хранит, `union` не знает. Если вас это не устраивает, вас это устраивает, а иначе вам нужно связать `union` с `enum class` в одну структуру (иначе undefined behaviour). ~~И называется она `std::variant`.~~ Пример использования такого — вы хотите у себя сохранить команду процессора на x86. Тогда у вам команда `MOV` может иметь своим вторым аргументом либо число, либо регистр.

Указатели и массивы. Об указателях можно думать как о числах хранящих адрес ячейки памяти. Пишутся они так: `type* p;`. И с ними есть две интересные операции: разыменование и взятие адреса. Взятие адреса — это `&a`, разыменование по указателю дают значение в данной ячейке. Когда мы делаем указатель на структуру и хотим обратиться к полю `y`, мы делаем `(*p).y`, что имеет более короткую форму `p->y`.
Массивы — это `std::array<type, size> arr`. Обращаются к нему по квадратным скобкам (`arr[24] = '0'`). Ещё у него есть возможность взять указатель на первых элемент (`arr.data()`), что используется для арифметики указателей. То есть мы можем делать что-то такое:

```
int* p = arr.data();
p++;
```

И теперь `p` указывает на следующих элемент массива. Вообще с указателями можно делать вот какие операции: их можно складывать с числами, вычитать числа из них (в.т.ч. при помощи инкрементов и декрементов), а ещё можно брать разность указателей. Результат будет иметь тип `ptrdiff_t` и равен количеству элементов между указателями. Ещё по указателям можно обращаться по квадратным скобкам и тогда выражение `p[ind]` равносильно `*(p + ind)`.
Ещё про массивы можно сказать, что обычно нам сто́ит использовать `std::array`, а не встроенные массивы из C (которые `type arr[size]`), ведь те во всех планах хуже:

1. Массивы неявно конвертируются в указатели (что вызывает путаницу с тем, являются ли указатели и массивы одним и тем же или нет).
2. Встроенные массивы нельзя копировать (поэтому нельзя их в функцию передавать, например).
3. А если вы напишете массив в параметре функции, то он тоже неявно конвертируется в указатель.

Ещё в массивах из C есть вопрос с тем, что `int* a[10]` — это кто такой? Это массив указателей или указатель на массив? Первое. Второе — это `int (*a)[10]`. В общем случае суффиксные деклараторы имеют бо́льший приоритет, чем префиксные (т.е. это в первую очередь массив *чего-то*, а вовторую — *что-то* это указатели). Ровно также работает их использование: если вы пишете выражение `x = *a[1];`, то у вас сначала будет обращение к первому элементу, а потом его разыменование. Но вообще люди обычно не пишут все эти скобки, а пишут что-то такое:

```
typedef int type[10];
type* a;
```

Вместо `int (*a)[10];`. Хорошо, если мы хотим завести себе `typedef`, который будет являться типом переменной `int ***(***a[10][20][30])[40][50][60];`. Как это сделать? Да элементарно, `typedef int ***(***type[10][20][30])[40][50][60];`. То есть никакой разницы, переменную вы объявляете или `typedef` делаете.
Ещё хороший составной тип в данном контексте — указатель на функцию. В ASM можно делать что-то такое:

```
mov rbx, func
...
call rbx
```

То есть вы где-то запомнили адрес функции, а потом этот адрес вызываете. В C и C++ это делается так:

```
void func(int)
{}

void main()
{
    void (*a)(int) = &func;
    (*p)(42);
}
```

Это можно использовать для полиморфного поведения. Ещё сто́ит сказать, что указатель на функцию можно вызвать. И более того, функции неявно преобразуются в указатели на себя. Поэтому точно такой же код можно написать так:

```
void func(int)
{}

void main()
{
    void (*a)(int) = func;
    p(42);
}
```

И вот указатели на функции подчиняются тем же правилам приоритета, что массивы и указатели. При этом считаются суффиксными.
Следующий ужас, который мы можем увидеть — функция, возвращающая указатель на функцию. Это выглядит так: `void (*get_function())(int)`. То есть это как объявление указателя на функцию, но с круглыми скобками после имени (это же не сам указатель, а функция, возвращающая его). То есть возвращаемое значение пишется не слева от функции, а вокруг. Хотя на практике с таким не встречаются, а делают `typedef`.

Rvalue, lvalue и их друзья. Понятно, что мы не можем написать что-то типа `2 + 2 = 7`, хотя и слева, и справа — `int`. Но всё же, почему конкретно, как это в языке работает? А вот так. У нас глобально есть rvalue и lvalue. Первое может стоять справа от оператора присваивания, второе — может слева. Ещё обычно у lvalue можно взять адрес, а у rvalue (обычно) нельзя. Пример этого — `a++` и `++a`. Первый — rvalue, второй — lvalue. Поэтому написать `++++a` можно, а `a++++` — нет. А вот `++a++` интерпретируется как `++(a++)`, поэтому тоже некорректно (суффиксные операторы имеют бо́льший приоритет, напомню). Хорошо, дальше интереснее. Что такое `+++a`? Ну, сначала надо сказать, что это `++(+a)`, потому что лексер разбивает на токены жадно (поэтому `a++++b` не сработает, несмотря на то, что существует способ распарсить это корректно. Ещё проговорим про такое: `a = 5 = 6`. Это некорректно, потому что равенство правоассоциативно (т.е. такая штука парсится в `x = (5 = 6)`), в отличие от остальных операторов. Но если мы напишем `(x = 5) = 6`, то будет бессмысленно, но корректно. Потому что `x = 5` — lvalue (присваивание возвращает левый операнд). Чуть менее бессмысленно это выглядит вот в таком варианте `(x += 5) += 6`. Давайте дальше в примеры lvalue и rvalue. Что вот тут: `& &a? &a` — это rvalue, увы.

Детали работы с числами.

Суффиксы констант. Какой тип имеет `42`? `int`. А если мы хотим другой? `42U`, `42L`, `42UL`, `42LL`, `42ULL`, `42.0f`, `42.0L`. Это соответственно `long`, `unsigned long`, `long long`, `unsigned long long`, `float` и `long double`.

Конверсии. Что будет, если складывать числа разного типа? Например, `unsigned` и `long`. А вот что. Упорядочим числа так: `int`, потом `unsigned`, `long`, `unsigned long`, `long long`, `unsigned long long`. Тогда из двух типов выбирается тот, кто больше, оба аргумента приводятся к нему и результат будет того же типа. Нюансы проявляются в том, что будет, если складывать два `char`'а, например. Будет `int`. А всё потому, что все арифметические операции с числами меньше `int`'а выполняются в типе `int`. Более того, вам даже операции делать не надо, при вызове функции это преобразование также происходит. Поэтому, кстати, если вы хотите принимать в функцию все типы (чтобы они сохраняли численное значение), то вам хватит `int` и больше. Реализовать только `long long` и `unsigned long long` вы не можете, потому что `long`, например, не знает, куда ему конвертиться. А вообще это работает так. В языке есть 3 типа конверсий (exact match, promotion и convertion), каждый следующий хуже всех предыдущих, и если у вас есть два одинаково хороших варианта, то ошибка компиляции. Так вот конверсия из `short`'а в `int` — promotion, а `long` в `long long` и `unsigned long long` оба являются convertion'ами. Про всё это подробно можно почитать на https://eel.is/c++draft/over.best.ics.

Мем.

```
switch (a)
{
case 0:
    std::cout << "0";
    if (false)
case 1:
    std::cout << "1";
    if (false)
case 2:
    std::cout << "2";
    if (false)
default:
    std::cout << "x";
}
```

Во-первых, это компилируется. Во-вторых, это выдаёт `0` для `0`, `1` — для `1`, `2` — для `2` и `x` для остальных. Как это понять? `case` — это метки. И `switch` делает `goto` по ним. И если не опускать фигурные скобки в данной записи, то получится что-то такое:

```
switch (a)
{
case 0:
    std::cout << "0";
    if (false)
    {
    case 1:
        std::cout << "1";
    }
    if (false)
    {
    case 2:
        std::cout << "2";
    }
    if (false)
    {
    default:
        std::cout << "x";
    }
}
```

И теперь в целом понятно, что происходит, мы прыгаем внутрь `if (false)`. Так писать ни в коем случае не надо, но с точки зрения языка возможно.

3 Компиляция и всё с ней связанное.

Зачем нам нужно это изучать? Во-первых, у студентов возникают с этим проблемы — когда компилятор пишет ошибку, а человек не понимает, что ему говорят. Во-вторых — если вы делаете ошибку в организации программы, причём такую ошибку, которая сразу к проблеме не приводит, то бывает такое, что при компиляции чуть-чуть по-другому всё сломается. Причём даже в крупных компаниях такое случается. Самое интересное, что ни в одной литературе про компиляцию не рассказывается (в совсем базовой считается что это сложно, а в продвинутой — что вы всё знаете), а все кто это знает, говорят, что пришло с опытом.

Этапы компиляции. Как мы компилируем программу обычно? *g++ program.cpp*. Пока что ничего магического нет. А вот с чем мы не знакомы — так это с тем, что *g++* не делает всю работу самостоятельно, а вызывает другие команды, которые выполняют компиляцию по частям. И если посмотреть, что там, то там происходят *cc1plus*, потом *as*, в конце *collect2*, который вызывает *ld*. Давайте попытаемся это повторить. Дальше будет перечисление стадий с указанием двух моментов: как их можно выполнить руками и какое расширение обычно имеет результат этой стадии.

1. Препроцессирование. Выполняется при помощи *g++ -E* (если дополнительно передать ключ *-P*, то вывод будет чуть короче), выходной файл обычно имеет расширение *.i*.
На файл с расширением *.i* можно и глазами посмотреть — в нём будет куча текста вместо `#include`, а потом наш код. Собственно, `#include` — директива препроцессора, которая тупо вставляет указанный файл в то место, где написана. Также препроцессор занимается макросами (`#define`).
2. Трансляция. Выполняется при помощи *g++ -S*, выходной файл обычно имеет расширение *.s*.
Трансляция — это (с английского) «перевод». Кого и куда переводим? Наш язык в ассемблер. Кстати, в этом ассемблере можно заметить интересную вещь, которую мы ещё не обсудили. Скомпилировав hello world, на x86-64 мы получим что-то вида

```
SUB     RSP, 8
MOV     EDI, OFFSET FLAT:LC0
CALL    puts
XOR     EAX, EAX
ADD     RSP, 8
RET
```

`LC0` — это метка, где хранится строка `"Hello, world"`, а вот зачем менять `RSP`, не ясно. Когда мы пишем на ассемблере, смысла в этом никакого. Но есть ABI — набор правил, как транслировать программу (какие регистры используются для возврата значения, например). И вот одно из правил — при входе в функцию стек должен быть выровнян по 16 байт.
3. Ассемблирование. Выполняется специальной утилитой *as*, выходной файл обычно имеет расширение *.o* (и называется объектным файлом).
Файлы *.o* бесполезно смотреть глазами, они бинарные, однако для этого есть специальные утилиты, например, *objdump*. Про него будет рассказано чуть позже. А вообще в *.o* файл попадают инструкции, которые были в ассемблерном файле, но в уже в бинарной форме.
4. Линковка. Выполняется простым вызовом *g++* от объектного файла. На выходе даёт исполняемый файл.

Хорошо, а что если файлов несколько? Да то же самое, честно говоря. Мы запускаем препроцессор независимо для каждого файла, потом независимо их всех транслируем и ассемблируем, а объединяются они только на этапе линковки. Независимые файлы *.cpp* файлы называют *единицами трансляции*. Стоит сказать, что это до появления модулей в C++20, где можно доставать данные одного файла для другого. То есть там появляется зависимость друг от друга, а значит компилировать их надо в определённом порядке.

Объявление и определение. Продолжим про компиляцию нескольких файлов. Пусть у нас есть два файла:

```
// a.cpp
int main()
{
    f(42);
}
```

```
// b.cpp
#include <cstdio>

void f(int x)
{
    printf("%i\n", x);
}
```

Мы получаем ошибку, что в файле *a.cpp* `f` не определена в области видимости. Дело понятно в чём — в том что ошибка происходит на этапе трансляции, когда файлы ещё независимы. Всё потому, что для того чтобы вызвать функцию, надо что-то про неё знать. Например, если мы передаём в функцию `int` — это одно ассемблерный код, а если `double` — то совершенно другой (потому что разные calling convention'ы могут быть). Поэтому на этапе трансляции нужно знать сигнатуру функции. Это исправляется вот так:

```
// a.cpp
void f(int);

int main()
{
    f(42);
}
```

```
// b.cpp
#include <cstdio>

void f(int x)
{
    printf("%i\n", x);
}
```

Когда мы пишем функцию и точку с запятой — это объявление/декларация (declaration). Это значит, что где-то в программе такая функция есть. А когда мы пишем тело функции в фигурных скобках — это определение (definition).
Кстати, написать объявление бывает нужно даже если у нас один файл. Например, в таком файле:

```
#include <cstdio>

int main()
{
    f();
}

void f()
{
    printf("Hello, world\n");
}
```

Тут дело в том, что компилятор смотрит файл сверху вниз, и когда доходит до вызова функции `f` внутри `main`, он ещё не дошёл до её определения. Тут можно переставить функции местами, да, но если у нас есть взаимнорекурсивные функции, то там переставить их не получится — только написать декларацию.

```cpp
struct complex
{
private:
    double re;
    double im;

public:
    complex()
    {
        re = im = 0;
    }
    complex(double re, double im)
    {
        this->re = re;
        this->im = im;
    }
};

void main()
{
    complex a1;
    complex a2();
    complex a3 = complex();
    complex a4{};

    complex b1(1, 2);
    complex b2 = complex(1, 2);
    complex b3{1, 2};
}
```

Первые 4 варианта и последние три эквивалентны. Кстати, выражение вида `complex(1, 2)` может как оно есть в функцию передаваться — тогда создастся временный объект и передастся. Этот временный объект, кстати, является rvalue. Теперь давайте внимательнее посмотрим на `a3` и `b2`. Там мы вроде как сначала создаём объект, а потом присваиваем его куда-то. Так вот да, но нет. У компилятора есть такое понятие как избегание копирования: если правый аргумент — rvalue, то он ничего не копирует, а просто вызывает конструктор на `a3`/`b2`.
Ещё сто́ит сказать про неявные конструкторы:

```cpp
struct complex
{
private:
    double re;
    double im;

public:
    complex() {...}
    complex(double re, double im) {...}
    complex(double re)
    {
        this->re = re;
        this->im = 0;
    }
};

void foo(complex) {...}

void main()
{
    foo(42.0);

    complex a = 42;
}
```

Такой код неявно преобразует `42.0` в `complex` и вызывает от него функцию. В случае с `complex` это оправдано, но если у вас контейнер инициализируется количеством элементов, то так неявно делать странно. Поэтому если вы такого не хотите, напишите перед конструктором слово `explicit`. Тогда вы запретите ещё `complex a = 42`, можно только `complex a(42)`.

Деструкторы. Ну, хорошо, вы написали свой контейнер. Он выделяет память при создании. А когда вы её освободите-то? А вот для конструкторов есть парные функции — деструкторы, которые автоматически вызываются, когда объект уничтожается. Он пишется как конструктор по умолчанию, но с тильдой в начале. Когда происходит уничтожение? Обычные переменные умирают когда наступает фигурная скобка блока, где вы объявили переменную. Совершенно не важно, каким образом вы покидаете блок, `return` у вас, `break`, `throw` или даже `goto`. Только если `longjmp` вы используете, тогда вы не знаете, вызовутся деструкторы или нет. Мораль — не используйте `longjmp`, потому что он всё равно корректно работает только вверх по стеку, а вверх по стеку можно заменить на `throw`-`catch`. Временные объекты умирают по концу выражения, где они созданы. Для глобальных переменных конструктор вызывается до `main`, а деструктор — после него. Для полей конструктор вызывается во время конструктора внешнего класса, а деструктор — после его деструктора. При этом деструкторы объектов одного блока вызываются в порядке, обратном порядку конструкторов.

Перегрузка операторов. Для класса `complex` очень хочется иметь арифметические операции. Чтобы так можно было, в C++ есть ключевое слово `operator`. Они пишутся как обычные функции, только называются как `operator+`, `operator-` и тому подобное. Кстати, надо сразу рассказать, как перегружать `++` и `--`, ведь у вас два таких. Тут синтаксический костыль — постфиксные операторы принимают второй аргумент `int`, который не используется.
Также как и обычные функции операторы могут быть внешними или внутренними:

```cpp
complex operator+(complex a, complex b)
{
    return complex(a.real() + b.real(), a.imag() + b.imag());
}
```

Или

```cpp
class complex
{
    // ...
    complex operator+(complex other) const
    {
        return complex(re + other.re, im + other.im);
    }
};
```

Работают они как совершенно обычные функции, поэтому сказать про них можно немногое.

Оператор ->. Особо нужно посмотреть на `->`. Его обычно перегружают, когда пишут какие-то свои указатели. И выглядят это вот так:

```cpp
struct my_ptr
{
    // Что-то.

    complex* operator->()
    {
        return /* Что-то */;
    }
};
```

Можно было бы подумать, что `->` — это бинарный оператор (у него есть то, у чего мы берём поле/метод и имя этого самого поля/метода). Но правая штука — это не выражение. В C++ нет рефлексии. Поэтому `->` — это унарный оператор. Если вы возьмёте `my_ptr x` и вызовете `x->im`, то это преобразуется в `(x.operator->())->im`. Поскольку `operator->` возвращает `complex*`, к нему нормально можно применить `->`. А ещё можно из оператора `->` вернуть что-то другое, к чему применим оператор `->`. И тогда они будут вызываться по цепочке, пока не дойдём до обычного указателя.

Ссылки. Вопрос — что делать с `+=` и им подобными? Уж совершенно точно не это

```cpp
void operator+=(complex a, complex b)
{
    a.set_real(a.real() + b.real());
    a.set_imag(a.imag() + b.imag());
}
```

Потому что `a` копируется. Есть идеологически неверное, но всё же рабочее решение:

```cpp
void operator+=(complex* a, complex b)
{
    a->set_real(a->real() + b.real());
    a->set_imag(a->imag() + b.imag());
}

int main()
{
    complex x, y;
    &x += y;
}
```

Но это выглядит странно и некрасиво. Специально для этой вещи в C++ введены ссылки. В первом приближении о них можно думать как об указателях, но со специальным синтаксисом. Вот что можно написать для ссылок:

```cpp
T a;

T& r = a;
foo(r);
r.bar;
```

И это будет «примерно равносильно» вот такому на указателях:

```cpp
T a;

T* const r = &a;
foo(*r);
r->bar;
```

При этом ссылка не бывает `nullptr` (ну, правда, зачем вам константный указатель на `nullptr`). И со ссылками можно писать вот такое:

```cpp
void operator+=(complex& a, complex b)
{
    a.set_real(a.real() + b.real());
    a.set_imag(a.imag() + b.imag());
}
```

И, кстати, по канону (то есть чтобы было как во встроенных типах) `+=` возвращает lvalue — левый аргумент, а значит `complex&`. Если бы мы возвращали просто `complex`, то во-первых, получили бы лишнее копирование, а во-вторых, мы не смогли бы написать `(a += b) += c`.

Немного best practices (until C++23). Сто́ит посмотреть, что делать, если вы реализуете свою строку. Вам хочется оператор `[]`. По-хорошему он выглядит так:

```cpp
struct string
{
    char* data;
    size_t size;
    size_t capacity;

    // ...

    char& operator[](size_t index)
    {
        return data[index];
    }
};
```

Но на самом деле вы хотите вызывать этот оператор на неизменяемой строке тоже, а от неё указанный оператор не вызывается (нельзя кастовать `const string* this` в просто `string* this`). Поэтому вам придётся написать ещё один вариант этого же оператора:

```cpp
const char& operator[](size_t index) const
{
    return data[index];
}
```

Почему `const char&`, а не `char`? Чтобы от константности строки не зависело, lvalue у вас или rvalue. А `const char&` — это lvalue, у него есть адрес.

Проблемы с внутренними операторами. Теперь про то, где писать операторы, внутри или вовне? Можно было бы подумать, что правила те же, что и с обычными функциями, но это не правда. Посмотрим вот на что:

```cpp
struct complex
{
    // ...

    /* implicit */ complex(double re)
    {
        // ...
    }
};

complex operator+(complex a, complex b)
{
    return complex(a.real() + b.real(), a.imag() + b.imag());
}

void main()
{
    complex a;
    a + 42; // 1
    42 + a; // 2
}
```

Оба варианта хорошо работают. А вот в таком написании:

```cpp
struct complex
{
    // ...

    /* implicit */ complex(double re)
    {
        // ...
    }

    complex operator+(complex b) const
    {
        return complex(re + b.real(), im + b.imag());
    }
};

void main()
{
    complex a;
    a + 42; // 1
    42 + a; // 2
}
```

Тут уже последняя строка `main`'а не компилируется, потому что компилятор преобразует её в `42.operator+(a)`, что не верно, ведь левый аргумент точки не преобразуется никуда.
Другой момент вот какой:

```cpp
struct mytype
{};

mytype& operator++(mytype& a)
{
    return a;
}

mytype foo()
{
    // ...
}

void main()
{
    ++foo();
}
```

Не компилируется, потому что `foo` возвращает rvalue, а мы хотим lvalue. И это хорошо, а с внутренним же оператором всё работает, несмотря на бессмысленность.
Поэтому всегда пишите операторы вовне. Если вы очень хотите иметь доступ к приватным полям/методам, то можно написать вот такую штуку:

```cpp
struct complex
{
    // ...

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

То есть тут по сути внешняя функция, но пишется внутри класса и имеет доступ ко всему.
Вообще, в C++11 вторую проблему особым образом пофиксили:

```cpp
struct mytype
{
    mytype& operator++() &
    {
        return *this;
    }
}

mytype foo()
{
    // ...
}

void main()
{
    ++foo();
}
```

Это гарантирует вам, что вызывать метод можно только на lvalue. Проблему с конверсиями же не исправить никак.

Копирование и присваивание.

```cpp
int main()
{
    string s = "Hello";
    string t = s;
}
```

Что это за странная вторая строка? А тут вот что. Компилятор умеет копировать все данные по умолчанию — всё что можно копируется покомпонентно. В случае с `complex` всё хорошо работает, а тут мы скопируем указатель на данные, который потом освободим два раза. Ошибка, понятно, в том, что каждая строка должна иметь свой указатель на данные, раз уж мы всегда их освобождаем. Потому нам нужно какое-то кастомное копирование — копирующий конструктор. Он пишется вот так:

```cpp
struct string
{
    char* data;
    size_t size;
    size_t capacity;

    // ...

    string(const string& other)
    {
        size = other.size;
        capacity = other.capacity;
        data = (char*)malloc(capacity);

        memcpy(data, other.data, size);
    }
};
```

Но это не единственная проблема. Ведь мы можем написать вот что:

```cpp
int main()
{
    string s = "Hello";
    string t;

    t = s;
}
```

Всё изменилось! (Нет.) Тут надо прописать оператор присваивания, он выглядит почти также, только надо ещё освободить те данные, которые были до присваивания.

```cpp
struct string
{
    char* data;
    size_t size;
    size_t capacity;

    // ...

    string& operator=(const string& other)
    {
        free(data);

        size = other.size;
        capacity = other.capacity;
        data = (char*)malloc(capacity);

        memcpy(data, other.data, size);
    }
};
```

Одна лишь проблема — можно написать `t = t`, но это исправляется при помощи `if (this == &other)`.

Специальные функции-члены класса. Итого мы имеем четыре специальных метода, которые компилятор может генерировать за программиста: конструктор по-умолчанию, конструктор копирования, оператор присваивания и деструктор. Последние три генерируются, если вы их явно не написали, а конструктор по-умолчанию генерируется, если не написали ни один другой.

Списки инициализации. Посмотрим вот на что:

```cpp
struct string
{
    // ...

    string()
    {
        data = strdup("");
        size = capacity = 0;
    }

    string(const char* str)
    {
        // ...
    }

    string(const string& other)
    {
        // ...
    }

    string& operator=(const string& other)
    {
        // ...
    }

    ~string()
    {
        free(data);
    }
};

struct person
{
    string name;
    string surname;

    person()
    {
        name = "Eric";
        surname = "Adams";
    }
};

int main()
{
    person p;
}
```

Сколько тут будет аллокаций и деаллокаций памяти? В смысле сколько, `malloc`'ов и `free`? А вот 6. Почему? Давайте аккуратно считать.

0. Мы вызываем конструктор класса `person`.
2. У двух `string`'ов вызывается конструктор по умолчанию, каждый из которых выделяет память.
4. Неявно вызываются конструкторы `string("Eric")` и `string("Adams")`, которые тоже выделяют память.
6. Два раза выполняется присваивание, которые также выделяют память.

Можно ли это оптимизировать? Компилятор теоретически может, конечно, но срабатывают эти оптимизации настолько редко и ненадёжно, что надеятся на них нельзя. Как это оптимизировать программисту? При помощи списков инициализации — хочется указать, что мы сразу вызываем конструктор `name` и `surname` от строки, а не по-умолчанию. Это делается при помощи такого синтаксиса:

```cpp
struct person
{
    string name;
    string surname;

    person() : name("Eric"), surname("Adams")
    {}
};

int main()
{
    person p;
}
```

И тут аллокаций будет 2, как и ожидается. Кстати, если есть объект, который не имеет конструктора по-умолчанию (потому что логически не существует состояния по-умолчанию), то без списков инициализации просто невозможно.
Что сто́ит сказать про это? А то что списки инициализации — это не только (и не столько) оптимизация, сколько по смыслу не то же самое, что мы написали сначала. Только для встроенных типов разницы особой нет, но вообще списки инициализации обычно и для них используются, потому что это гораздо более идиоматический код. А ещё сто́ит сказать, что если в списке инициализации не написано ничего для некоторого поля, то для него используется конструктор по-умолчанию.
Поскольку деструктор у нас один, разрушаются поля в определённом порядке. А в конструкторе вы, в вроде как, можете в разном написать. Так вот нет, потому что список инициализации всегда вызывает конструкторы в порядке расположения полей структуры. И лучше бы вам писать список инициализации в этом же порядке, чтобы не путать людей. И, кстати, в инициализации некоторого поля можно использовать то, что было создано ранее. Аналогично с использованием `this` в списке инициализации — можно, но осторожно, надо понимать, что конструктор `this` ещё недовыполнился. Аналогично осторожным быть надо в деструкторе по той же причине.

`= delete` и `= default`. Помните «Специальные функции-члены класса»? А что если копирование не имеет смысла? То есть копировать либо нельзя, либо слишком трудоёмко. И в языке есть специальный синтаксис для этого:

```cpp
string(const string& other) = delete;
```

Начиная с некоторого C++, кстати, автогенерация конструктора копирования является deprecated. А если вы хотите явно прописать, что вам нужен стандартный сгенерированный конструктор, то вы можете написать

```cpp
complex(const complex& other) = default;
```

Для деструктора, конструктора копирования и оператора присваивания эта строка ничего не изменит, но вы можете захотеть явно сгенерировать конструктор по-умолчанию (напомню, он не генерируется, если есть другие конструкторы), то `= default;` — ваш бро.

`new` и `delete`. Итак, мы знаем, что объекты создаются при заходе в блок, а удаляются при выходе. А если Вы хотите вызывать их в произвольный момент? Тогда есть `new` и `delete`. Вы пишете `Type* p = new Type(...);` — это конструирует вам объект в куче, а `delete p;` разрушает вам этот объект. Разница между ними и `malloc`—`free` (упрощённо) в том, что второй вариант пришёл к нам из C, где конструкторов и деструкторов не бывает, поэтому их он и не вызывает. Выделение на куче, кстати, немного дороже, чем на стеке.

Ещё немного best practices. Давайте вот ещё на что посмотрим. Наш класс `string` вызывает в конструкторе аллокацию памяти. Так вот, вообще считается, что это кринж, так делать не надо. А ещё при присваивании C-строки в нашу строку мы сначала конструируем, а потом присваивает — давайте вместо этого явно пропишем `operator=(const char*)`, и будет также 2 аллокации.

5 Исключения.

Вообще обычно сначала рассказывают про наследование и прочий кал, а потом про исключения. Это логично, потому что исключения требуют некоторых фактов из наследования. Но тогда исключения будут очень поздно, а домашки давать надо. Поэтому мы обсудим исключения, не вдаваясь в глубокие подробности о наследовании, а уже потом в конце дополним наш рассказ.

Мотивация. В коде ядра Linux и связанного с ним можно найти несколько функций, которые возвращают `int` и выглядят так:

```cpp
int err;

err = function(...);
if (err)
    return err;
```

И так много раз — вызвать тысячу функций, после каждой прочекать ошибку и вернуть, если есть. *Очень красиво выглядит.* А если у нас есть что-то более сложное, чем последовательное выполнение, то было бы ещё более некрасиво и совсем непонятно. Вообще больше мотивации по исключениям можно посмотреть в лекциях по Java. И специально, чтобы такой херни не было, и придумали исключения. Пишется это так:

```cpp
try
{
    // Делаем что-то, что может завершиться с ошибкой.
    ...
    ...
    // Завершиться с ошибкой~--- это вот так:
    throw something;
    ...
    ...
}
// Что делать, если ошибка.
catch (const error_type& e)
{
    // Обрабатываем ошибку.
}
catch (const another_error_type& e)
{
    // Обрабатываем другую ошибку.
}
```

Тогда всё до происхождения ошибки будет выполнено, а всё что после — нет. Как это взаимодействует с деструкторами? Ведь мы могли инициализировать локальные переменные до или после `throw`, как с ними жить? Ну, то что проинициализировали — разрушаем, то что нет — не трогаем. Что, собственно, логично, точно то же самое делает `return`, `break` и всё остальное. Тут, кстати, как в Java есть наследование исключений, и если вы кидаете исключение наследуемого типа, вы можете поймать его как базового. И в стандартной библиотеке есть дерево наследования, наследующееся от `std::exception`. Хотя вообще кидать вы можете всё что угодно — любой копируемый тип.

`catch (...)` и `throw;`. Ещё есть специальный тип `catch`'а — `catch (...)`, который ловит вообще всё что угодно. Но, как в известном анекдоте, есть один нюанс: `catch (...)` ловит не обязательно исключения C++. В Windows, например, исключения C++ — это частный случай исключений, встроенных в Windows. И эти встроенные исключения также ловятся при помощи `catch (...)`, что вы хотите явно не всегда. Системные исключения вообще не надо ловить, если взять POSIX'ивое исключение `abi::__forced_unwind` и съесть его, то может быть всё очень плохо. Ещё более интересный момент есть в GCC. Там исключения поддерживаются не только плюсовые, а и другие разные, которые работают не так. Куча языков реализуют исключения так, что `catch` может либо прервать, либо продолжить работу. В C++ такого нет, но ABI такое поддерживает. И такое уж точно вы не хотите ловить. Но вообще `catch (...)` имеет идиоматическое использование:

```cpp
catch (...)
{
    // Что-то почистим.
    throw;
}
```

`throw;` просто берёт то, что поймал и кидает его же. Без копирования. Поэтому подобный синтаксис даже в обычных `catch` бывает полезен, так как он лучше, чем

```cpp
catch (const type& e)
{
    throw e;
}
```

В варианте выше есть лишнее копирование с slicing (сейчас не важно, что это). А `throw;` не имеет копирования.

Best practices и worst practices. Совет — всегда ловите исключение по константной ссылке. Если Вы, конечно, не хотите исключение менять, тогда просто по ссылке (может, даже rvalue ссылке...).
Ещё советы. Не кидайте указатель на исключение (то есть не пишите `throw new error_type(...);`). Если вы так сделаете, ловить вам надо будет `catch (error_type*)`, и тогда надо не забыть сделать `delete`. И понятно, что нельзя делать `catch (...)`, потому что тут вы никак не сделаете `delete`.

Цена исключений. Если открыть что-то в интернете, вам скажут, что исключения — это бесконечно дорого. Чтобы говорить об этом серьёзно, то сначала надо понять, какую операцию мы измеряем по времени. Как работает `try`? Есть две стратегии реализации.

1. Поддерживать связный список на стеке, по которому в случае чего мы и идём.
   Проблема тут в том, что каждый вход в `try` (даже если исключения не происходит) жрёт время и память. А исключения у вас должны происходить редко. Поэтому сейчас эту реализацию не используют.
2. Zero-cost исключения. Компилятор для каждой инструкции записывает, куда идти в случае исключения.
   Тогда вход в блок `try` не генерирует абсолютно никакой код. И тут у нас при `throw` никакого оверхеда от исключений почти нет. Почему «почти»? Потому что в общем случае код может быть не совсем одним и тем же. Если мы делаем какие-то операции с памятью, например. Мы должны видеть все эффекты до `throw` и не видеть ни одного после. Поэтому компилятор может меньше переупорядочивать код в таком случае. И если написать бенчмарк для тестировки этого, то результат будет в том, что замедление будет небольшим.

Есть заморочки с конкретными реализациями (в Itanium ABI, как уже обсуждалось, есть мем с продолжением выполнения после `throw`). Поэтому Itanium ABI раскручивает стек два раза — чтобы проверить, надо ли прерываться и чтобы потом вызвать деструкторы.

`noexcept`. `std::terminate`. Что будет, если кинуть исключение из деструктора при обработке исключения? То есть мы получили исключение, вызываем деструкторы, и получаем исключение там. По стандарту, это ошибка в программе, считается, что при двух исключениях происходит что-то непоправимо ужасное, и в программе вызывается `std::terminate` — экстренное завершение программы. Более простой способ получить `std::terminate` — выкинуть исключение в `main`. Кстати, поскольку деструкторы не должны бросать исключение, такие функции как `free` явно в языке помечены как неспособные бросать исключения.
Ещё один нюанс при бросании исключения — функции можно пометить как `noexcept` (после круглых скобок), чтобы указать, что функция не бросает исключений. Бросание исключений из таких функций также приводит к `std::terminate`. Ещё помимо `noexcept` можно указать `noexcept(/* булевое выражение */)`. Это нужно, если вы шаблонный код пишете, и у вас функция может в зависимости от шаблона либо быть `noexcept`, либо нет. И вот что важно знать — деструкторы по-умолчанию `noexcept`. Если вам нужно это исправить — `noexcept(false)` вам в помощь. Тогда при обычном бросании исключения в деструкторе всё будет хорошо, а при обработке исключений — `std::terminate`.

RAII. Вот на что посмотрим:

```cpp
int* p = new int(1);
int* q = new int(2);

// ...

delete p;
delete q;
```

Тут с исключениями есть проблемы. Если у нас второй конструктор кинет исключение (или память не выделится), то мы не освободим первый объект. Или если у нас что-то после этого бросит исключение, то мы вообще ничего не освободим. Поэтому долгое время ходило мнение, что исключения нельзя использовать без сборщика мусора. Но потом было показано, что это вообще не только про память, но и про открытие файлов тоже, например. Если написать так:

```cpp
FILE* my_fopen(const char* name, const char* mode)
{
    FILE* result = fopen(name, mode);
    if (result == nullptr)
        throw std::runtime_error("No such file.");
    return result;
}

int main()
{
    FILE* p = my_fopen("1.txt", "r");
    FILE* q = my_fopen("2.txt", "r");

    // ...
}
```

Проблема будет такой же. Но, например, в такой ситуации никаких проблем нет

```cpp
int main()
{
    std::string p = "Hello";
    std::string q = "World";

    // ...
}
```

Почему тут нет проблем? Потому что у строк есть деструктор, которых их освободит. Про классы, которые в конструкторе что-то выделяют, а в деструкторе — освобождают говорят, что они удовлетворяют идиоме RAII (resource acquisition is initialization). То есть на конструкторе мы захватываем ресурс, а на деструкторе освобождаем его. И в стандартной библиотеке всё так и есть. Хотя вообще данная идиома не очень хорошо названа, потому что выделять-то мы не обязаны в конструкторе, мы можем вызвать `std::istream::open`, чтобы не при конструкторе файл открыть, а потом. Но в деструкторе мы всё равно закроем файл, если открывали.

Умные указатели. Так вот, в стандартной библиотеке есть набор классов, которые называют «умными указателями», которые по сути и являются указателями, но по RAII. Самый простой из них — `std::unique_ptr<T>`. Давайте напишем его сами:

```cpp
template <class T> // Пока не смотрите на эту строку слишком внимательно.
class unique_ptr
{
private:
    T* ptr;

public:
    unique_ptr() : ptr(nullptr)
    {}

    unique_ptr(T* ptr) : ptr(ptr)
    {}

    unique_ptr(const unique_ptr&) = delete;
    unique_ptr& operator=(const unique_ptr&) = delete;

    ~unique_ptr()
    {
        delete ptr;
    }

    T& operator*() const
    {
        return *ptr;
    }

    T* operator->() const
    {
        return ptr;
    }

    T* get() const
    {
        return ptr;
    }

    void reset(T* new_ptr = nullptr)
    {
        delete ptr;
        ptr = new_ptr;
    }

    T* release()
    {
        T* result = ptr;
        ptr = nullptr;
        return result
    }
};
```

Вообще функции типа `release` использовать опасно, потому что они убирают RAII'шную обёртку над указателем, и надо следить, чтобы в момент исключения он не оказался сырым.
Кстати, подобные RAII'шные классы полезны, даже если мы не пользуемся исключениями. Если мы в C открываем три файла, то писать придётся так:

```
    FILE* f1 = fopen("1.txt", "r");
    if (f1 == NULL)
        return -1;

    FILE* f2 = fopen("2.txt", "r");
    if (f2 == NULL)
    {
        fclose(f1);
        return -1;
    }

    FILE* f3 = fopen("3.txt", "r");
    if (f3 == NULL)
    {
        fclose(f2);
        fclose(f1);
        return -1;
    }

    // ...

    fclose(f3);
    fclose(f2);
    fclose(f1);

    return 0;
```

Это выглядит некрасиво, и забыть что-то можно очень легко. Как это делают в C, где нет деструкторов? Так:

```
    res = 0;

    FILE* f1 = fopen("1.txt", "r");
    if (f1 == NULL)
        return -1;

    FILE* f2 = fopen("2.txt", "r");
    if (f2 == NULL)
    {
        res = -1;
        goto close_f1;
    }

    FILE* f3 = fopen("3.txt", "r");
    if (f3 == NULL)
    {
        res = -1;
        goto close_f2_and_f1;
    }

    // ...

    fclose(f3);
close_f2_and_f1:
    fclose(f2);
close_f1:
    fclose(f1);

    return res;
```

operator new и operator delete. Давайте теперь подумаем, что делать с нашим классом string. Он весь некорректный, ведь мы вызываем malloc, где память нам могут не дать. Поэтому надо взять некий аналог malloc, который бросает исключение в случае нехватки памяти. И, о чудо, он такой есть в стандартной библиотеке. Он называется operator new. И ещё есть operator delete как парный к нему. Это не то же самое, что и new—delete, это именно что своя штука. И operator new отличается от malloc только тем, что исключение кидает. Но это всё равно не позволяет вам выделять память при помощи operator new, а освобождать при помощи free.
Теперь найдите ошибку тут:

```
struct string
{
    // ...

    string& operator=(const string&)
    {
        if (this == &other)
            return;

        operator delete(data);

        size = other.size;
        capacity = other.capacity;
        data = (char*)operator new(size + 1);

        memcpy(data, other.data, size + 1);

        return *this;
    }
};
```

Exception safety. Тут есть большая проблема — соглашения об исключениях. Есть у нас обломается operator new, то вы останетесь в некорректном состоянии (ненулевые size и capacity, но неверный data). Более того, эту строку даже удалить будет нельзя, потому что у неё data указывает на некоторые данные, которые мы уже удалили, и будет double-free. Поэтому правильно будет написать всё как-то так:

```
string& operator=(const string&)
{
    if (this == &other)
       return;

    char* tmp;
    if (other.size != 0)
    {
        tmp = (char*)operator new(size + 1);
        memcpy(tmp, other.data, size + 1);
    }
    else
        tmp = nullptr;

    operator delete(data);

    size = other.size;
    capacity = other.capacity;
    data = tmp;

    return *this;
}
```

Мораль — если у вас есть исключения, наивно писать код нельзя. А вообще есть научное название для этого — exception-safe код. Выделяют различные гарантии в случае происхождения исключения.

1. Nothrow. Исключения не бросаются.
2. Strong. Если в процессе выполнения операции происходит исключение, объект возвращается в состояние до операции. (Именно так работает присваивание выше.)
3. Basic. У наших данных сохраняются все инварианты, но в каком конкретно состоянии мы останемся — неизвестно. (Используется, если нельзя предоставить строгую гарантию. Если мы уже отправили ракету на Марс, вернуть её обратно мы не можем так легко.)

Какие-то более слабые гарантии — не exception-safe, это ошибка в программе. Общее правило, как жить с исключениями, выглядит так: для каждой строки надо задать вопрос, что будет, если в ней произойдёт исключение. Если так проверить всё, то функция OK.
Давайте немного пофантазируем на тему std::vector. Какие его методы каким гарантиям удовлетворяют? Понятно, какие-нибудь size, capacity, empty или begin и end не бросают исключений, с чего бы им. А вот что можно сказать про pop_back? Ну, вроде как кидать там нечего, но ведь деструктор элемента может теоретически и бросать... А вот нет — все стандартные контейнеры накладывают некоторые условия на принимаемые ими типы. И абсолютно все из них требуют noexcept-деструктора. Хорошо, то есть pop_back — это nothrow? А что будет, если мы удалим элемент из пустого вектора? A undefined behaviour. То есть мы можем спокойно вернуть *nullptr и ничего не бросать в таком случае. Поэтому если нам хочется, мы вполне можем написать vector, который имеет noexcept-pop_back. А какие функции имеют базовые гарантии? insert и erase. Вообще их можно сделать так, чтобы гарантии были строгими, но они базовые, чтобы можно было их выполнять через swap'ы. Обычно swap'ы не бросают исключение, но вообще могут делать это. И вообще мы по сути получаем тут, что есть swap не бросает, то гарантии у нас строгие, а иначе — базовые.

Swap-trick. Рассмотрим std::vector с erase, у которого базовые гарантии. А теперь следите за руками:

```
void erase_strong(vector& v, iterator where)
{
    vector copy = v;
    copy.erase(where);
    copy.swap(v);
}
```

И лёгким движением руки базовая гарантия превращается в строгую. Это называется swap-trick. При помощи него же можно и внутри своего кода давать строгие гарантии там, где вы не хотите думать. Например, мы писали сложный оператор присваивания для наших строк, который ещё и на конструктор копирования очень сильно был похож. Оп:

```
string& operator=(const string& other) &
{
    string copy = other;
    std::swap(copy, *this);
    return *this;
}
```

И код становится элегантнее. Можно даже ещё проще:

```
string& operator=(string other) &
{
    std::swap(other, *this);
    return *this;
}
```

Теперь вам даже явно руками копию делать не надо.

RAII вокруг объекта, кидающего исключение при закрытии. Давайте теперь рассмотрим POSIX, в котором мы файлы открываем.

```
struct file_descriptor
{
public:
    file_descriptor(const char* filename)
        : fd(open(filename, O_RDONLY))
    {}

    ~file_descriptor()
    {
        close(fd);
    }
private:
    int fd;
};
```

В чём тут проблема? В том, что на *man* написано, что close имеет возвращаемое значение, в нём может произойти ошибка. И с close был полный треш. В POSIX долгие операции могли быть прерваны, и close был таковым. Причём никто не знает, почему. Единственное объяснение — при работе с магнитными лентами он перематывал их на начало.
Так вот close выполняет две операции — дозаписывает данные до конца, чтобы закрыть файл и возвращает файловый дескриптор операционной системе. Поэтому в идеальном мире мы бы разбили его на две — какой-нибудь flush, который дозаписывает данные и именно что close. И примерно такого поведения мы и хотим, а кидать исключение в случае провала close не хотим, потому что двойное исключение. Давайте напишем всё вот так:

```
struct file_descriptor
{
public:
    file_descriptor(const char* filename)
    : fd(open(filename, O_RDONLY))
    {}

    void flush()
    {
        int result = close(fd);
        if (result != 0)
            throw std::runtime_exception("File closing failed.");
        fd = -1;
    }

    ~file_descriptor()
    {
        if (fd != -1)
            close(fd);
    }
private:
    int fd;
};
```

Теперь из деструктора мы ничего не бросаем. Но теперь мы обязаны явно извне вызывать flush, если заинтересованы в получении ошибок при закрытии.

6 Лирические отступление про оптимизации.

Это скорее практика, а не теория, потому что в следующей домашке надо нужны будут некоторые оптимизации. И без предварительного пояснения люди не справляются с их написанием. В добавок к этому существует некоторый контекст — надо понимать, зачем мы делаем то что делаем в этом самом ДЗ.

Оптимизация аллокаций. Мы уже обсуждали, что самое низкоуровневое API для выделения и освобождения памяти — mmap и munmap. Нам же никто не мешает самим написать обёртку над ними, чтобы у нас были свои malloc и free? Возьмём наивнейшую реализацию — malloc просто вызывает mmap (с некоторыми проверками). Если провести эксперимент, можно увидеть, что такая реализация будет капец медленной. Более того, она и памяти будет есть в 15 раз больше, чем стандартные malloc и free. У нас ведь были тесты mmap'а (где мы обращались к памяти после выделения), если также протестировать malloc, то получим, что за одну секунду мы можем сделать 45000000 malloc'ов (стандартных, не наших новых) и free, но 200000 mmap'ов и munmap'ов.
Следовательно стандартные аллокаторы делают что-то умное. Если посмотреть на то, что именно, то выяснится, что у них разные правила для больших и маленьких объектов. Причём с маленькими объектами все взаимодействуют одинаково, а с большими — по-разному. Почему? Потому что 99% аллокаций — аллокации маленьких кусочков памяти. И для таких ситуаций уже придумали, как оптимально жить. Давайте простоты ради ограничимся объектами размера 32 байта. У нас есть большой кусок данных и нам надо где-то хранить, какие его части заняты, а какие — свободны. И уметь быстро доставать случайный свободный и быстро освобождать. Это можно сделать односвязными списком — список хранит незанятые данные, причём именно в этих данных и хранится указатель на следующий свободный кусок. И это в целом рабочая идея. Именно что идея, потому что нужно уметь возвращать данные операционной системе, а со списком это сложно. Как легко поправить стратегию, когда объекты разного размера? Сделать несколько блоков под разные размеры, и округлять вверх до существующего размера.

Copy-on-write. Мы имели нашу строку. Вопрос — а зачем мы всегда копируем строку при копировании? Можем ли мы сказать, что новая строка ссылается на те же данные? Ну, можем, просто иногда придётся копировать при изменении. Всё что для этого нужно — сделать счётчик ссылок на данный объект, если он 1 — то просто меняем, иначе копируем и меняем. Это называется copy-on-write. При этом если строка пустая, то она является ссылкой на одну «the пустую строку».

Small-object. Одна лишь проблема — copy-on-write оптимизирует нам большие строки, а нам хочется как-то оптимизировать маленькие, маленькие чаще встречаются. Для этого есть следующий трюк: У нас есть size, capacity и data. Это 24 байта. Так а давайте для маленьких строк вместо data и capacity хранить маленькую строку. То есть теперь наши строки выглядят как

```
struct string
{
    size_t size;
    union
    {
        dynamic_buffer dynamic_storage;
        char static_storage[sizeof(dynamic_buffer)];
    }
};
```

И если у нас size <= sizeof(dynamic_buffer), то мы использует static_storage, а иначе dynamic_storage. Это small-object оптимизация. Конкретно в такой реализации она работает не очень хорошо, но можно оптимизировать:

data	size	capacity	1

S	m	a	l	l		s	t	\0			3	0
S	m	a	l	l		s	t	r	\0		2	0
S	m	a	l	l		s	t	r	i	\0	1	0
S	m	a	l	l		s	t	r	i	n	\0	0

Давайте отводить старший бит последнего байта структуры под флаг, короткая строка или длинная. При этом сам последний байт (для короткого объекта) хранит количество оставшихся символов. При записывании нового символа в конец мы уменьшаем это число, и когда память в маленьком объекте кончается совсем, там получается 0, а 0 — это то же самое что '\0'. Примерно так работает строка в реализации Facebook'а.
Теперь вопрос, в каком порядке применять copy-on-write и small-object? Ну, объективно вовне надо пихать small-object, а внутрь — copy-on-write. Почему? Ну, посмотрим что будет, если наоборот. У нас будет указатель, по которому ref_count и наш union. То есть для маленьких строк мы всё равно будем хранить ref_counter (который нам не сдался) и всё равно будет indirection. Это строго хуже даже глупой реализации.

7 Undefined behaviour.

Если просто дать определение этой штуки, возникнет вопрос, на кой она нужна. Поэтому начнём мы с истории из жизни компиляторов.

Aliasing. restrict. Напишем вот что:

```
void sum(float* res, float const* in, size_t n)
{
    for (size_t i = 0; i < n; i++)
        *res += in[i];
}
```

Компилятор же умеет оптимизировать код, почему бы ему не завести регистр, где он будет накапливать чиселки, после чего запишет их в *res. И он действительно заведёт регистр XMM0. Но почему-то он будет записывать результат в *res на каждой итерации цикла, а не после него. Почему? Потому что может так оказаться, что указатели in и res ссылаются в одно и то же место. И тогда нам бы надо прибавить корректное значение. У компилятора нигде не написано, что такого быть не может, хотя ни один адекватный человек так делать не будет. Научным языком ситуация с двумя ссылками на одно место называется aliasing.
Ещё один пример на ту же тему:

```
void memcopy(char* dst, char* src, size_t count)
{
    for (size_t i = 0; i < n; i++)
        dst[i] = src[i];
}
```

Тут проблема та же самая — это должно быть копирование, но если его вызвать от пересекающихся массивов, будет не оно. Так что для компилятора наличие aliasing'а — это причина, почему он не может применять многие оптимизации. Поэтому разработчики компиляторов пытались найти какие-нибудь правила, которым помогли бы им в оптимизации. И нашли они вот что. Мы говорили о том, что нельзя в общем случае предполагать размеры int'а и short'а. Поэтому разработчики компиляторов нагло этим воспользовались:

```
int foo(int* a, int* b)
{
    *a = 1;
    *b = 2;
    return *a;
}
```

Тут нельзя оптимизировать return *a в return 1. Зато тут:

```
int foo(int* a, short* b)
{
    *a = 1;
    *b = 2;
    return *a;
}
```

Вполне можно, ведь некорректно одним указателем ссылаться на int и short. Это strict aliasing rule — вы не можете ссылаться на одни и те же данные разными типами. У него есть два исключения. Первое: int и unsigned (или любые другие данные, отличающиеся только знаковостью). Второе: char, unsigned char и std::byte, которые могут alias'ить всё что угодно. В C signed char тоже может, в C++ (по стандарту) — нет. Это может привести к интересным эффектам. Если мы попытаемся заполнить std::vector<char> нулями, то компилятор будет перечитывать data и size, а то вдруг данные вашего вектора alias'ятся с его полями. Такую же проблему, но в меньших масштабах, имеем с std::vector<size_t>, поскольку у нас size — это size_t.
Такая проблема приводила к тому, что всякие численные методы не писались на C, а писались на Fortran'е, в силу специфики которого компилятор имеет право предполагать, что его аргументы на alias'ятся. Чтобы как-то жить с alias'ингом в C в стандарте C99 даже добавили ключевое слово restrict, которое показывает, что данный указатель не alias'ится ни с одним другим. В C++ его нет, но многими компиляторами поддерживается как расширение и пишется как __restrict. И если написать

```
void sum(float* __restrict res, float const* in, size_t n)
{
    for (size_t i = 0; i < n; i++)
    *res += in[i];
}
```

То компилятор не только вытащит запись в *res вне цикла, но начнёт ещё и векторизацию использовать, что ускорит код в 4 раза.
Хорошо, мы написали функцию, на которую мы повесили __restrict. Вопрос — что произойдёт, если мы-таки передадим туда что-то, что alias'ится? Да хрен его знает, честно говоря. В зависимости от компилятора код может быть оптимизирован, а может быть не. Вы не можете заставить компилятор оптимизировать код по стандарту, а значит не можете заставить сделать что-то конкретное — произойдёт то, что произойдёт. Тут вы по сути вступаете в контракт с компилятором — Вы не передаёте какой-то кал, а компилятор включает оптимизации. Если вы нарушите свои условия, ваш код может сделать всё что угодно. Только тут пример показателен, потому что вы сами кровью подписали контракт, написав __restrict.

Сдвиги. А вот если вы сделаете

```
int shift(int a, int b)
{
    return a << b;
}
```

То непонятно, что произойдёт, если вы передадите b или отрицательное, или большее 32. На разных платформах сдвиги работают по разному. На x86 вот, вообще 32- и 64-битные числа сдвигаются по модулю битности, а 16-битные — не по модулю 16 бит, а тоже 32. Поэтому конкретно прописать, что произойдёт, нельзя, на каких-то платформах нужно будет if'ать. Поэтому это тоже UB. Но в данном примере ничего кровью не подписывали. Впрочем, данный пример имеет ещё одну особенность — можно определить, что под каждую конкретную платформу имеем конкретное поведение. А есть случай, когда мы и так делать не можем.

Разыменование нулевого указателя и обращение вне массива. Почему нельзя его разыменовывать, ведь согласно страничной адресации какое-то значение мы получим. Ну, во-первых, страничная адресация есть не везде, увы, вы проиграли. Во-вторых, даже если у вас страничная адресация, то разные ОС сообщают о обращении к nullptr по-разному, и это сложно стандартизировать. Хотя на самом деле есть более важный аргумент. Ну, хорошо, пусть у вас есть какой-то обработчик обращения к nullptr. Тогда компилятор не может удалить обращение к неиспользуемой переменной (вдруг там nullptr, это же надо обработать), не может переупорядочивать обращения к памяти (в обработчике надо видеть все обращения, произошедшие до, но не видеть все после). В ту же степь идут обращения за край массива: если это проверять и обрабатывать, нельзя будет переупорядочить обращения к памяти.

Переполнение типов.

```
bool is_maximal_int(int a)
{
    return a + 1 < a;
}
```

Этот код будет оптимизирован компилятором в return false. Почему? Ну, смотрите, если a — это не максимальный int, то возвращаем false, иначе по стандарту UB, значит можно вернуть что угодно, например тоже false. По стандарту в **знаковых** числах (и только в них) если результат операции +, - или * не представим в вашем типе, то это UB.

const. Ещё весёлая ситуация есть с const'ами.

```
int const PI = 3;
int length(int r)
{
    return 2 * PI * r;
}
```

Так вот, это оптимизируется в умножение на 6. Несмотря на то, что технически никто не мешает вам снять const при помощи const_cast и записать что-то другое в PI. Потому что когда вы снимаете const, а потом что-то записываете — UB.

Намеренное неопределённое поведение. В GCC есть прекрасная функция, которая называется __builtin_unreachable. Всё что она делает — делает UB. Нахуя, а главное зачем? Чтобы компилятор мог как ему вздумается оптимизировать то, что вы и ваши коллеги считаете недостижимым. Например, если у вас есть switch, где вы перебрали все случаи, то в default можно написать эту функцию, и тогда компилятор не будет проверять, что вы попадёте в один из case'ов.

Best practices. Хорошо, а как делать надо?

- Снимать const всегда плохо, просто так не делайте.
- Если вам нужна операция с переполнением, в GCC есть __builtin_add_overflow, который точно и без UB делает то, что вам нужно.
- Если вы хотите кастить указатели друг в друга, то вам нужен memcpy. Причём memcpy компилятор нормально оптимизирует, получается как каст, только по стандарту. В случае memcpy вы получите всё так, как на текущей архитектуре работает (т.е. LE либо BE), а если вы хотите конкретный порядок байт вне зависимости от системы, напишите что-то вида

  ```
  ptr[3] | ptr[2] << 8 | ptr[1] << 16 | ptr[0] << 24
  ```

  Это даже медленно не будет, потому что компилятор просекает такие паттерны и оптимизирует их.

Способы неправильно воспринимать UB. Люди иногда считают, что компилятор может портить при оптимизации программы, содержащие UB. На самом деле это некорректно, потому что компилятор портит не программу целиком, а только на тех входных данных, которые к UB приводят. Пока вы подаёте программе те данные, на которых нет UB, она обязана работать корректно, даже если в каком-то месте, в каком-то другом случае может быть UB. Например,

```
struct tun_struct *tun = ...;
struct sock *sk = tun->sk;
if (!tun)
    return POLLER;
```

Традиционно это код из ядра Linux, содержащий UB. Но sk не использовалось, поэтому к ошибкам не приводило. А когда GCC начал всё оптимизировать сильнее, он нафиг убрал переменную sk. Но поскольку вы обращаетесь к tun, можно нафиг убрать и проверку его на NULL. Тем не менее, ваша программа останется корректной при любом не'NULL'овом значении tun.
Другое неправильное понимание UB — вот:

```
bool test(int a)
{
    return a + 1 < a;
}
```

И вопрос в том, почему это переводится в return false. Но человек смотрит на то, что в процессоре это совершенно чётко работают, и не понимает, почему в C++ не работает. А дело в том, что опасно при изучении того, как работает код, смотреть на инструкции процессора. Нельзя так, сложение в процессоре и сложение в языке могут различаться. Похожая очень грустная ситуация с многопоточностью, где инструкции есть у процессора, есть команды в C++, и они совсем разные.
А ещё неправильно обращаться к разработчикам компиляторов, которые в новой версии сломали ваш код. Раньше компиляторы оптимизировали всё очень слабо, и исполняли всё так, как написано (соответственно, UB почти не портилось). А потом новые оптимизации, которые всё сильнее и сильнее ломают ваш код, содержащий UB.

8 Валидация программы

Отчасти источником проблем с UB является стандарт, который написан на бумаге, в связи с чем программист не знает, то что он вообще пишет, это UB или нет. Впрочем, на текущий момент с этой ситуацией живётся намного проще.

Ключ компиляции -fsanitize. У компиляторов есть ключи, которые вставляют дополнительные проверки. Например, *-fsanitize=undefined*. Если его поставить, то при запуске на этапе вызова test(std::numeric_limits<int>::max()) вам в stderr напишут, что у вас *integer overflow*. Почему это происходит на этапе исполнения? Потому что все предупреждения стоят определённую цену из времени компиляции. Скорее всего, у GCC есть ключ, который включает статический анализатор, но тогда программа будет компилироваться бесконечно долго.
Другой хороший пример:

```
int* p = new int(42);
delete p;
*p = 43;
```

Если вы скомпилируете это с *-fsanitize=address*, то при исполнении вам выдадут, что в третьей строке вы пишете туда, куда писать нельзя. А именно, что во второй строке вы это освобождали, а в первой — выделяли. Т.е. компилятор вставил в программу необходимые проверки, чтобы это ловить. Работает ключ *-fsanitize=address* следующим образом: создаётся табличка, в которой указано, какие байты где выделены, когда освобождены и т.п. При этом, когда вы обращаетесь к памяти, в эту табличку смотрят и проверяют, легально ли вы сделали. Это табличка называется shadow-байтами. Более того, когда вы что-то освободили, в после этого снова выделяете, обычно аллокатор даёт вам то, что вы только что освободили (пока оно в кэше). А при *-fsanitize=address* эта память находится в карантине и только потом переиспользуется, чтобы отлавливать ситуации вида

```
int* p = new int(42);
delete p;
int* q = new int(43); // Standard allocator will put *q at the place *p was.
*p = 44;
```

Вдобавок вокруг аллоцированных блоков памяти вставляется красная зона, чтобы если вы **чуть-чуть** промахнулись по адресу, вы не попали куда-то, где могут быть корректные данные, а попали чётко туда, куда нельзя. Ещё *-fsanitize=address* включает в себя проверку на утечки памяти — когда вы что-то выделяете и не освобождаете.
-fsanitize=address и *-fsanitize=undefined* можно использовать совершенно свободно. Ещё есть memory- и thread-sanitizer'ы. Первый находит неинициализированную память, второй — состояние гонки. Их использовать гораздо труднее. Почему? В случае address-sanitizer'ов можно скомпилировать часть программы с ним, а часть — без. В случае с memory же, у вас могут быть ложные срабатывания, ведь в shadow-памяти не будет записей из той части программы, которая скомпилирована без sanitizer'а. А этой частью может быть, ну, например, стандартная библиотека. И аналогично с thread-sanitizer'ами — они также требуют записи и чтения.

valgrind. Ещё можно использовать утилиту *valgrind*, который проверяет программы пост-фактум. Она умеет находить почти те же самые ошибки (address, memory), и при этом не требует компилировать программу со специальными ключами. За счёт чего она работает? А она содержит JIT-декомпилятор x86. Понятно, что работает это капец медленно (20 раз против 2 для ключей), но зато нет никаких проблем с memory-sanitizer'ом. Только не забудьте компилировать вашу программу с ключом *-g*, чтобы вы видели, на каких строках кода ошибки. При этом если вы не хотите давать пользователю дебажные символы, просто отправьте их в отдельный файл. И на сервер залейте. Подробнее об этом — ниже.
valgrind, к сожалению, немного меньше ошибок видит просто потому, что, например, не может добавить песочка между выделенными данными. И поэтому же *valgrind* не может проверять undefined и thread. В случае undefined он, например, не знает, из какого языка взялась инструкция ADD, и обладает ли она неопределённым поведением в нём, а в случае thread — не знает, многопоточная ли та инструкция MOV, которую он видит, или нет.

Проверка предусловий стандартной библиотеки. *valgrind* и sanitizer'ы — это не единственное, что вы можете. И, что более важно, они не всегда вам помогут. Рассмотрим вот что:

```
std::list<int> lst;
lst.push_back(1);
auto it = lst.end();
it++;
std::cout << *it << std::endl;
```

На большинстве реализаций стандартной библиотеки тут вы получите *1*, потому что в большинстве реализаций std::list — кольцевой список. При этом с указателями всё хорошо, вы обращаетесь к корректной памяти, а значит sanitizer'ы не найдут ошибок. Но видно же, что программа некорректна. Чтобы такое тоже проверять, есть флаг *-D_GLIBCXX_DEBUG*, который как раз проверяет инвалидацию итераторов, разыменование и инкремент past-to-end и все остальные нарушения контрактов. Понятно, что это даже асимптотически очень медленно — например, бинарный поиск работает за линейное время (потому что проверяет, является ли переданный ему массив отсортированным) — однако очень полезно.

assert. Мы уже можем проверять корректность на двух уровнях — уровне указателей и уровне контрактов. А есть уровень ещё выше. Например, мы пишем свой целочисленный квадратный корень. И мы хотим, чтобы пользователь не полагался на какое-то конкретное поведение, если передать отрицательное значение. Поэтому есть assert. Это такая функция из заголовка <cassert>, которая в зависимости от настроек может делать ничего или писать вам ошибку в программе в случае невыполнения какого-то условия. Очень советуется её писать **везде**, где вы полагаетесь на какие-то условия у себя в голове. Если вы полагаетесь на то, что next->prev == this, напишите на это assert. Если вы полагаетесь на m >= n, напишите assert. Понятно, особого фанатизма не требуется, но тем не менее делайте это, иначе вы где-то можете получить мусор из-за бага, передать этот мусор в функцию, получить мусор там, в результате чего потратить на отладку 3 дня вместо нескольких минут.
Неправильные практики использования assert:

```
FILE* f = fopen("1.txt", "r");
assert(f != NULL);
```

Это вполне штатная ситуация, если файл не открылся, не надо проверять при помощи assert что-то штатное.
Второе:

```
assert(fclose(f) == 0);
```

Тут ещё хуже — когда вы отключите assert'ы, они отключатся полностью, и fclose не выполнится. То, во что раскрывается assert, зависит от макроса NDEBUG. Если он задан, то assert'ы исчезают, иначе — проверяются. Поэтому, если вам нужно написать проверку чего-то сложного (скажем, вы хотите проверить, что значение, посчитанное разными способами, одинаковое), то напишите *#ifndef NDEBUG* и ехайте проверять всё, что вашей душе угодно.
Стóит ли давать программу с assert'ами пользователю? Если вы задались этим вопросом, скорее всего у вас недостаточно assert'ов в программе, ведь когда их достаточно, программа замедляется раза в 2.

9 Статические и динамические библиотеки.

Начнём с повторения того, что мы уже знаем. Каждый исходный файл транслируется в объектный файл, после чего все объектные файлы линкуются в программу. Но иногда бывает кусок кода, который хочется переиспользовать. Мы могли бы оттранслировать объектные файлы этого куска один раз, после чего сразу с ними компилировать. Но так оказалось, что уже существует механизм сгруппировать объектные файлы вместе, после чего отдать их линковщику.

Статические библиотеки. Пусть у нас есть

```
// sum.cpp
int sum(int x, int y)
{
    return x + y;
}

// main.cpp
#include <iostream>

int sum(int x, int y);

int main()
{
    std::cout << sum(2, 2) << std::endl;
}
```

И мы зачем-то пытаемся вычленить *sum.cpp* как библиотеку. Тогда нам нужно написать *ar rcs libsum.a sum.o*. *ar* — это сокращение от «archive», *rcs* — это некоторая магия (читайте *man*), *libsum.a* — название библиотеки. Тогда для компиляции с этой библиотекой надо дописать *-L. -lsum* в опции компилятора. *-L.* говорит, что нужно искать библиотеки в текущем каталоге, *-lsum* говорит, что нужно к строке «*sum*» приписать спереди «*lib*», а сзади — «*.a*», после чего искать библиотеку с полученным именем. Эта библиотека называется статической и по сути не очень отличается от объектного файла.

Динамические библиотеки. Статические библиотеки достаточно хороши, но у них есть следующий недочёт: пусть у вас есть библиотека, которая используется везде вообще. Например, libc. И тогда реально в каждой программе есть функции из неё. А значит в каждой программе они занимают место и на диске, и в памяти. И чтобы переиспользовать и то, и другое, придумали динамические библиотеки. Идея динамических библиотек состоит в следующем: мы ссылаемся как-то на внешнюю библиотечку, а потом **на этапе исполнения** грузим по надобности её части. Как в нашем же примере сделать динамическую библиотеку вместо статической? Нужно скомпилировать *sum.cpp* как *gcc -c -fpic sum.cpp*, потом превратить его в разделяемую библиотеку при помощи *g++ -shared -o libsum.so sum.o*, а *main.cpp* скомпилировать также, как и в случае со статической библиотекой. Но при попытке это запустить, мы получим ошибку — не могу найти *libsum.so*. Потому что в текущем каталоге (под Linux) библиотеки не ищутся. Чтобы это проверить мы можем запустить *ldd ./a.out*, и он покажет, какие библиотеки ему нужны. И в частности покажет, что ему нужна *libsum.so*, которую он не нашёл. Есть два способа это исправить. Первый — *LD_LIBRARY_PATH=. ./a.out*. Мы тупо руками прописали, где ищутся библиотеки. А второй вот как. В самой программе есть место, в котором написано, где она ищет библиотеки (можно посмотреть при помощи *objdump* в секции *Dynamic Section:*). Давайте туда допишем что надо. Делается это так: *gcc -Wl,-rpath=<путь_до_библиотеки> main.cpp -L. -lsum*. *-Wl* говорит, что опцию (а данном случае опцию *-rpath*) надо передать линковщику, а линковщик, увидев эту опцию, запишет в список путей то, что вы захотели. Если вам нужно сразу несколько путей, кстати, то разделяйте их двоеточиями. Но есть одна проблема — в *-rpath* вы так просто не можете написать относительные пути (ведь откуда вы знаете путь до исполняемого файла). Но на самом деле знаете — есть специальный псевдо-путь $ORIGIN, который и является путём до исполняемого файла. Используя его Вы можете свободно написать что-нибудь по типу *-rpath='$ORIGIN/../lib/'*.
В Windows, кстати, это работает немного по-другому. Во-первых, DLL-ки сразу же ищутся в текущем каталоге. Во-вторых, чтобы понять, что вы ссылаетесь на динамическую библиотеку, в Linux вы пишете *-L. -lsum*. В Windows же компиляция DLL создаёт вам специальный *.lib*-файл, который называется import-библиотекой. С ним вы компилируете вашу программу, чтобы она сама поняла, откуда какие функции брать.

Причины иной компиляции динамических библиотек. Хорошо, а что значат все эти магические слова (*-fpic* и *-shared*)? Зачем на как-то особенно компилировать динамические библиотеки? А дело вот в чём — при запуске программы, она первая загружается в адресное пространство, и она сама может выбрать, куда её хочется. Динамические библиотеки такого же по понятным причинам позволить себе не могут. Возникает вопрос — и что теперь? А то, что при наличии глобальных переменных, мы не можем прописать им адреса. Надо куда-то обращаться. Есть путь, которым пошли разработчики архитектуры PowerPC: адреса динамической библиотеки считаются относительно некоторого регистра. И тут жить можно, разве что вам нужно правильно задавать этот регистр, когда обращаетесь к разным библиотекам, и не менять его, если обращаетесь к библиотечной функции из другой функции той же библиотеки. Сложно, но жить можно. Самый простой способ жить с динамическими библиотеками была у Microsoft на 32-битной Windows. У каждой библиотеки был base-address — то куда библиотеке хочется загрузиться. Если там свободно — туда она и загружается, а если нет, то библиотеку загружают туда, где есть место, а в специальной отдельной секции (.reloc) хранится список адресов, которые надо исправить. Разумеется, в случае релокаций умирает переиспользование библиотеки, но Windows вам же полностью предоставляют, там можно расположить системные библиотеки так, как хочется, поэтому в проприетарных системах всё будет хорошо. В Linux же это реализовано по-другому. Смотрите как можем:

```
        CALL next
next:
        POP ABX
        LEA EAX, [EBX + (myvar - next)]
```

Тут идея в том, что мы записываем адрес текущей команды на стек, потом берём его со стека, а дальше вместо того, чтобы писать абсолютный адрес переменной myvar, пишем относительный. Относительный адрес позволяет нам обращаться к ней, если мы не знаем, в какое место памяти будет загружена библиотека, что нам и надо. Вообще мы не очень хорошо написали (процессор не любит непарные CALL и POP), поэтому обычно это выглядит так:

```
get_pc:
        MOV EBX, [ESP]
        RET

get_variable:
        CALL get_pc
next:
        LEA EAX, [EBX + (myvar - next)]
```

Этот код называется position-independent code, и ключ *-fpic* именно генерацией такого кода и занимается. Вопрос — почему для этого не сделали специальную инструкцию? А вот сделали, но в 64-битном режиме. Всё что с квадратными скобками стало уметь обращаться в память начиная со смещения текущей инструкции. И называется это RIP-relative positioning.

GOT/IAT. PLT. В итоге мы имеем кусок данных, который можно загрузить и исполнять. Но на самом деле библиотека — это же не сферический код в вакууме, она хочет вызывать какие-то функции. Например, библиотека для работы с JSON хочет делать fopen. То есть нужно подружить библиотеки друг с другом. Самый простой вариант — когда мы делаем CALL, в файл мы кладём нулевой адрес, а в секцию релокаций записываем, что вместо него нужно положить fopen, после чего при запуске динамический загрузчик всё разложит по местам. То есть то же самое, что с линковщиком. Почему так не делают? Потому что мы от'mmap'или нашу библиотеку, а в ней дырки. И во все места дырок нужно что-то подставить. И опять вы не можете записать библиотеку в память один раз, что вам очень хочется. Поэтому вместо этого просто заводят табличку со смещениями, и теперь все CALL'ы обращаются туда, и туда же динамический загрузчик подставляет истинные адреса функций. Эта таблица в Linux называется global offset table, а в Windows — import address table.
Но на самом деле есть ещё проблема. Давайте посмотрим, что происходит, когда мы делаем

```
void foo();

void test()
{
    foo();
}
```

Как мы обсуждали, тут будет CALL пустой адрес (до линковки пустой). А что будет, если foo — это внешняя функция из какой-то библиотеки? Тогда надо бы вместо простого CALL'а сначала tmp = MEM[got_foo], а потом уже вызов по tmp (то есть там будет написано CALL qword [got_foo]). Но есть проблема — мы узнаём, откуда эта функция только на этапе линковки, а компилировать надо раньше. Поэтому компилятор генерирует первый вариант (CALL foo), а потом, если это было неправильно, просто создаёт свою функцию foo, которая является прослойкой для JMP qword [got_foo]. Такие заглушки, которые просто совершают безусловный переход по глобальной таблице смещений имеют название. В Linux их называют PLT (procedure linkage table), а в Windows как-то по-другому. Но в Linux PLT используется ещё для одной цели. Рассмотрим, скажем, LibreOffice, в котором сотни динамических библиотек с тысячами функций в каждой. Поэтому заполнение GOT — это долго. И нам не хочется смотреть, где лежит каждая функция, после чего записывать её в таблицу. Поэтому эту операцию сделали ленивой: GOT заполняется специальными заглушками, которые динамически ищут в хэш-таблице настоящий адрес функции, после чего записывают его в GOT вместо себя, и вызывают эту функцию, чтобы она отработала. В Microsoft по-умолчанию отложенная загрузка не используется, но его можно включить (delayed DLL loading или как-то так называется). Это фича загрузчика, а не самой Windows, и делает эта фича примерно то же самое. Однако есть разница. В Linux отсутствие библиотеки не позволяет запустить программу. в Windows же библиотека подгружается при первом вызове функции оттуда, что, по их словам, сделано чтобы вы могли за'if'ать ситуацию, когда библиотеки нет. Почему эта фича не включена по-умолчанию? Потому что в Windows в библиотеках существенно меньше функций (ещё обсудим, почему именно), поэтому там проблема с GOT'ом (точнее, IAT'ом) так остро не стоит.

Офф-топ на тему «Как страшно жить». Теперь поговорим про изменение *so*-файлов. Давайте возьмём и во время работы программы поменяем библиотечку на диске, втупую вписав туда другой текст. Результат поразителен —- работа программы также изменится. Почему? Мы же, вроде как, исполняем библиотеку из оперативки, а не с диска. А дело в том, как работает copy-on-write в операционных системах. Когда вы пишете в некоторую страницу, вам копируют её. Но когда кто-то извне её пишет, вам не дают копию старых данных. С исполняемым файлом такое не прокатывает, кстати. Это потому, что вашу программу загружает ядро, и оно может запретить изменять бинарники, а библиотеку загружает ваша программа, которая такого механизма не имеет. Кстати, изменение *so* и перекомпиляция *so* — разные вещи. И если вы во время работы программы перекомпилируете библиотеку, она не обновится. Связано это с тем, что перекомпилированная библиотека — это новый файл, а не старый. По сути вы удалили старую библиотеку и создали новую, вместо того, чтобы библиотеку изменить. А в Linux пока кто-то имеет доступ к файлу, файл не удаляется до конца. И поэтому в вашей программе всё ещё есть та самая библиотека, которую вы загружали (а не новая).

Детали работы со статическими библиотеками в Windows. Никто не удивиться, что набор из `CALL foo@PLT` и `foo@PLT: JMP qword [got_foo]` не очень эффективен (три обращения в память вместо одного). Поэтому в Windows есть спецификатор `__declspec(dllimport)`, который сразу вместо `CALL foo@PLT` вставляет `CALL qword [got_foo]`.
Ещё в Windows есть такая штука как *.def*-файл — линковщик экспортирует из вашей DLL-ки только то, что нужно, и в *.def*-файле указывается, что именно. Это хорошо работает в C, где имена символов и имена функций совпадают, но не очень хорошо в C++, где вам придётся писать сложные декорируемые имена. Поэтому есть второй вариант — написать на самой функции `__declspec(dllexport)`. И вроде бы всё хорошо, мы метите функции, которые экспортируете как `__declspec(dllexport)`, которые импортируете — как `__declspec(dllimport)` и всё классно работает. Но есть проблема. Вы и в библиотеке, и в коде, который её использует подключаете один заголовочный файл, где объявлена функция. И непонятно, что там писать: `__declspec(dllexport)` или `__declspec(dllimport)`. Для этого заводится специальный макрос под каждую библиотеку, которым отличают, DLL вы компилируете или нет.
Есть ещё одна проблема. Пусть вы берёте адрес функции. Если это ваша функция, то вы просто берёте IP-адресацию. Если это функция библиотеки, то вам нужно лезть в GOT. Правда, оттуда вы заглушку возьмёте, а не саму функцию, если не напишете `__declspec(dllimport)`. Впрочем, это может быть и не такая ужасная проблема, не так часто вы что-то делаете с адресами функций. Однако если вдруг, то `__declspec(dllimport)` — ваш бро.
А ещё непонятно, что делать с глобальными переменными. Там проблема ещё более страшная: вы сначала читаете адрес переменной из GOT, а потом по полученному адресу обращаетесь. Тут уже никакую функцию-прослойку не написать, увы. Поэтому если вы не пометите глобальную переменную как `__declspec(dllimport)`, тот тут вы уже точно совсем проиграете, у вас линковка не получится.
А ещё реализация DLL в Windows нарушает правила языка, если вы напишете `inline`-функцию в заголовочном файле. Она просто откопируется в каждую библиотеку, где вы этот заголовок подключился. Поэтому тут вы просто проиграли.
Что ещё есть в Windows. Компилятор вашей программы лет на пять новее Windows. Поэтому вам нужно куда-то деть новую версию стандартной библиотеки. И тут вы либо статически линкуетесь с ней, либо динамически. И в своё время было модно первое, но это лютейший кринж, потому что в каждой библиотеке свой `printf`, например. Что хуже, там свой `errno` и даже свой аллокатор памяти. Поэтому, например, вы не можете передать из одной библиотеки в другую `std::string` (точнее, не можете безопасно его изменить), ведь при перевыделении памяти вы освободите то, что не выделяли.

Детали работы со статическими библиотеками в Linux. Помните мы делали свой `malloc` и `free`, делали библиотеку и подменяли стандартные аллокаторы памяти при помощи *LD_PRELOAD*. То что мы сделали, называется interposition и имеет кучу преколов. Посмотрим на

```
int sum(int a, int b)
{
    return a + b;
}
int test(int x, int y)
{
    return sum(x, y) - x;
}
```

Тут при обычной компиляции вторая функция просто вернёт свой второй аргумент. А при компиляции с *-fpic*, то вы же можете подменить `sum`, а значит оптимизации не будет. Чтобы это пофиксить, можно пометить `sum` как `static` (тогда эта функция будет у вас только внутри файла, а значит его не поменять извне) или как `inline` (потому что `inline` полагается на ODR, а значит функция должна быть везде одинаковой). Но есть ещё способ. Linux по-умолчанию считает, что все функции торчат наружу (т.е. как `__declspec(dllexport)` в Windows). А можно их пометить, как не торчащие наружу, а нужные только для текущей компилируемой программы/библиотеки: `__attribute__((visibility("hidden")))`.
На самом деле атрибут `visibility` может принимать несколько различных значений (`"default"`, `"hidden"`, `"internal"`, `"protected"`), где пользоваться сто́ит только вторым, потому что первый и так по-умолчанию, третий заставляет ехать все адреса, а четвёртый добавляет дополнительные аллокации. Впрочем, `hidden` тоже пользоваться опасно. При этом также есть различные ключи компиляции (типа *-B symbolic*), которые тем или иным образом немного меняют поведение, и пояснить разницу между ними всеми вам могут только избранные. И каждый из них может поменять вам поведение так, что вы можете легко выстрелить себе в ногу. То есть глобально в Linux поведение по умолчанию делаем вам хорошо, но, возможно, немного неоптимизированно, а когда вы начинаете использовать опции, вы погружаетесь в такую бездну, что ускорение заставляет вас очень много думать. Причём замедление от динамических библиотек может быть достаточно сильным: если взять компилятор clang-LLVM и компилировать при помощи его ядро Linux'а, то в зависимости от того, сложен ли clang-LLVM в один большой файл или разбит по библиотечкам, время компиляции отличается на треть. Поэтому ключи использовать придётся. Один из самых безопасных из них — *-fno-semantic-interposition*. Это не то же самое, что и *-fno-interposition* потому, что бинарнику всё равно можно дать interposition, однако в нашем случае функция `test` будет оптимизирована. Ещё один полезный ключ — *-fno-plt*. Он по сути вешает оптимизацию такую же, как `__declspec(dllimport)`, но на весь файл, поэтому функции, написанные в нём же, замедляются. Чтобы не замедлялись — `visibility("hidden")`. Вообще всё это детально и подробно рассказано не будет, если вам интересно, то, во-первых, вы — извращенец, во-вторых, гуглите и читайте/смотрите по теме. Впрочем, все *-fno-plt* и прочие штуки нужны нам тогда и только тогда, когда мы не включили linking-time оптимизации. В GCC все наборы ключей нафиг не нужны, если включить *-flto*. Так что в перспективе *-flto* и *-fno-semantic-interposition* — это единственное, что вам может быть нужно. Но только в перспективе.

Выбор между статическими и динамическими библиотеками. Хорошо, но что же всё-таки использовать: статические или динамические библиотеки?

Преимущества статических библиотек.	Преимущества динамических библиотек.
Обычно занимают больше памяти. Впрочем, не всегда — если из стандартной библиотеки вы используете только `printf`, то вам только его и дают. И люди, сидящие на операционной система Plan 9, вообще отрицают, что статические библиотеки занимают больше памяти.	Обычно дольше работают.
Статическая компиляция с библиотеками даёт вам возможность очень просто перенести вашу программу куда угодно. Если вы хотите поднять ваш бинарник на другом сервере, то со статическими библиотеками вам не нужно заниматься сексом с версиями. В частности, если у вас есть программа времён Windows 98, если она скомпилирована статически, то вы сразу можете её запустить, иначе вы проиграли.	Дистрибутивы Linux почти всегда пользуются динамическими библиотеками, потому что когда вы делаете маленький багфикс, вы не хотите перекомпилировать вообще все зависимости, а хотите только поставить пользователю именно этот багфикс.
Ещё одно преимущество статических библиотек — не нужно бесконечное количество времени их загружать в память. Поэтому консольные утилиты лучше поставлять статическими библиотеками, чтобы скрипты на Shell работали быстро (в скриптах на Shell каждая строка — вызов новой программы).	Динамические библиотеки (только на Linux, правда) поддерживают interposition, что бывает полезно, если вы хотите подсунуть свой аллокатор памяти, потестить что-то или ещё чего.
	При помощи статических библиотек не реализовать кастомные плагины, а при помощи динамических — как нефиг.

10 Введение в шаблоны.

Зачем это нужно, что это такое? Ну, мотивация проста. Мы делаем контейнер стандартной библиотеки, который хранит всё что захочется пользователю. И тогда мы пишем: «хочется вектор целых чисел»: `std::vector<int>`. Это основная мотивация появления шаблонов в языке.

Способы жить без шаблонов. Прежде чем говорить, почему шаблоны такие, какие есть, надо понять, как жить без них. Например, как жить в C.

`void*`. В C живут с `void*` — тип указателя, который (в C, не C++) неявно преобразуется куда угодно и откуда угодно. И тогда всё выглядело бы так:

```
struct vector
{
    void push_back(void*);
    void*& operator[](size_t index);
    const void*& operator[](size_t index) const;
}

int main()
{
    point* p;
    vector v;
    v.push_back(p);
    static_cast<point*>(v[0]);
}
```

Это не выглядит типобезопасно. Мы можем достать из вектора не то, что туда положили. И компилятор ничего не скажет. А если мы можем выявить ошибку на этапе компиляции, сто́ит это делать.
Вторая проблема — количество аллокаций. Если мы хотим хранить целые числа, а не указатели, например, то в `std::vector<int>` мы тупо выделяем большой блок памяти, а в нашем `vector`'е мы сначала выделяем большой блок под указатели, а потом выделяем память под каждый. К тому же подобную штуку не получится prefetch'ить, потому что память под каждый объект выделена в разных местах, а значит лишний indirection.
А ещё есть совсем плохая проблема:

```
vector u;
u.push_back(new point());
```

Этот код не exception-safe, потому что вы никак не освободите память, если произойдёт ошибка при `push_back`.

Макросы. Много можно было бы рассуждать о том, насколько отвратительно решение через `void*`, но нам уже этого достаточно. А пока посмотрим на второй способ жизни без шаблонов: макросы!

```
#define DEFINE_VECTOR(type)                          \
    struct vector_##type                             \
    {                                                \
        void push_back(type const&);                 \
        type& operator[](size_t index);              \
        type const& operator[](size_t index) const;  \
    };
```

Тут уже явно лучше, можно написать это типобезопасно, без проблем с памятью, но тоже имеются проблемы. Например, вот:

```
DEFINE_VECTOR(int);
DEFINE_VECTOR(int32_t);

int main()
{
    vector_int v;
    vector_int32_t u;
}
```

И теперь мы имеем две одинаковые структуры, а хотелось бы одну.
Второе — когда мы имеем `DEFINE_VECTOR(int)` в двух разных местах. Потому что вам и какому-то человеку из Австралии понадобилось одно и то же. А потом кто-то подключает и то, и другое, и он проиграл. Ещё одна проблема — `#DEFINE_VECTOR(unsigned short)`. Компилятор видит это как `struct vector_unsigned short`, и не компилирует.

Шаблоны наконец-то. Хорошо, теперь как это делать нормально? Мы пишем `template <typename T>` либо `template <class T>`, и после этого везде пишем, чего же мы хотим, в уже описанном синтаксисе (`vector<int>`). И тут нет ни одной из описанных проблем. Причём `template` можно навесить на что угодно. На функцию можно:

```
template <typename T>
void swap(T& x, T& y)
{
    T tmp = x;
    x = y;
    y = tmp;
}
```

При этом для функций вы не обязаны писать `swap<int>(a, b)`, а можете написать просто `swap(a, b)`, если переменные `a` и `b` уже имеют тип `int`. При этом если вы подставите в эту функцию `long long` и `int`, вам явно напишут, что нельзя так. Более того:

```
template <typename Dst, typename Src>
Dst my_cast(Src s)
{
    return static_cast<Dst>(s);
}

int main()
{
    int x = 42;
    my_cast(x);         // Непонятно, чему равно Dst, ошибка.
    my_cast<float>(x); // Dst указан явно, Src можно вывести, зная тип x.
}
```

Специализация. У нас есть `std::vector<T>`. Но мы хотим сделать `std::vector<bool>` немного другим. И тогда мы можем явно сказать, что у нас есть `vector<T>` для всех классов, а потом написать

```
template <>
struct vector<bool>
{
    void push_back(bool);
    // ...
};
```

И у вас для всех типов, кроме `bool` будет то, что вы написали изначально, а для `bool` — специализация. При этом когда вы пишете специализацию, вы **целиком** пишете новый класс.
`std::vector<T>` называется primary template, `std::vector<bool>` — explicit specialization. А ещё есть partial specialization — специализировать не обязательно все параметры. И ещё можно особым образом специализировать. Например, мы решили, что мы можем каким-то особым образом хранить указатели. И тогда мы бы написали

```
template <typename U>
struct vector<U*>
{
    // ...
};
```

То есть partial specialization — это специализация, сама являющаяся шаблоном. Теперь, когда вы напишете `vector<int*>`, вам дадут специализацию `vector<U*>`. Вот ещё пример:

```
template <typename A, template B>
struct my_type {}; // 1

template <typename A>
struct my_type<A, int> {}; // 2

template <typename B>
struct my_type<int, B> {}; // 3

int main()
{
    my_type<float, double> fd; // Выбирается 1.
    my_type<float, int> fi;    // Выбирается 2.
    my_type<int, double> id;   // Выбирается 3.
    my_type<int, int> ii;      // Компилятор не может выбрать между 2 и 3. Не компилируется.
}
```

Хорошо, а как определяется, какая специализация лучше? Давайте вот какой пример:

```
template <typename T>
struct bar {}; // Произвольный тип.

template <typename U>
struct bar<U*> {}; // Указатель на что-то.

template <typename R, typename A, typename B>
struct my_type<R (*)(A, B)> {}; // Указатель на функцию.
```

Здесь есть «указатель на что-то» и «указатель на функцию». Кажется, что второе более специализированно. Но как бы это формализовать? Да легко! Является ли произвольный указатель на функцию указателем? Да. А является ли произвольный указатель указателем на функцию? Нет. То есть если мы **всегда** можем **корректно** подставить одну специализацию в другую, но не наоборот, то первая более специализированна. Вопрос: что делать, если шаблон от нескольких параметров зависит? Тут первая специализация более специализированна, чем вторая, если хотя бы по одному параметру она строго более специализированна, а по остальным — не менее.
Примечание: в шаблоны вы можете передавать всё что угодно (хоть `void`, хоть `int(int, int)` (функцию, даже не указатель на неё), хоть `char[]`). Но не любой класс обязан корректно работать с любым классом. Если в хотите в своих классах что-то явно ограничить — подождите, и будет вам счастье.

Специализация функций. Со специализацией функций есть детали. Во-первых, у функций просто нет partial специализаций. Вообще. Во-вторых, с explicit специализациями есть детали, поскольку существуют перегрузки: вместо `template<>`
`void foo<int>(int)` пишут `foo(int)`. Но вообще есть разница. Итак:

```
    template <typename T>
    void baz(T*) {}

#if ENABLE_TEMPLATE
    template <>
    void baz<int>(int*) {}
#else
    void baz(int*) {}
#endif
```

В приведённом примере если включить шаблонную версию `baz`, то код `baz(nullptr)` не скомпилируется, а если не-шаблонную, то скомпилируется вполне. Почему, хочется спросить? А вот почему.
Есть перегрузки функции. Мы их проходили, и одну видим тут: `void baz(int*)`. Так вот, шаблон (весь целиком) считается ещё одной перегрузкой. При этом, когда вы вызываете функцию вам сначала пытаются подставить не-шаблонные специализации, а если не получится — шаблонную, для которой после этого выбирается специализация по уже обсуждённым правилам.
В данном случае посмотрим на `baz(nullptr)`. В случае включённого шаблона имеем всего одну перегрузку — `baz<T>` (одну перегрузку, но со специализацией). Можем ли мы подставить в неё `nullptr`? Объективно, нет, `nullptr` не является указателем, нельзя понять, чему равно `T`. А значит не компилируется. В случае с выключенным `ENABLE_TEMPLATE` имеем две перегрузки: шаблонную и обычную. Можем ли мы подставить `nullptr` в `void baz(int*)`? Ну, да, кто нам мешает, `nullptr_t` неявно преобразуется в любой указатель, значит в `int*` тоже преобразуется. Шаблон даже проверять не надо, но он, как мы видим, не подойдёт.
Кстати, можно немного изменить работу с перегрузками. Можно вызывать функции не как `foo(...)`, а как `foo<>(...)`. В таком случае вы явно отбросите всё, что не является шаблоном, а значит выбирать сможете только из специализаций.

Non-type template parameter. Помимо типов в шаблоны можно пихать чиселки. В стандартной библиотеке есть `std::array`, который имеет два параметра — сколько хранить и что хранить. И вот сколько хранить — это `size_t`. Выглядит это как-то так:

```
template <typename T, size_t N>
array
{
    // ...
};
```

Их всё так же можно специализировать (например, `std::array` имеет специализацию от нулевой длины, потому что в C++ нельзя написать `int[0]`). Точно то же самое можно написать и для функций:

```
template <typename T, size_t N>
size_t size(T (&)[N])
{
    return N;
}
```

Что ещё можно сказать про non-type template параметры? То что это самое число `N` обязано быть известно на этапе компиляции. Потому что только на этапе компиляции существуют типы, а значит только на этапе компиляции существуют шаблонные типы.

Template template parameter. Хочется обёртку над контейнером. Зачем-то.

```
template </* container */ V>
struct container_wrapper
{
    V<int> container;
}

container_wrapper<vector> wrapper;
```

Это пишется вот так:

```
template <template <typename> class V>
struct container_wrapper
{
    V<int> container;
}

container_wrapper<vector> wrapper;
```

Используется, понятно, очень редко. Правила те же самые, что и обычно.

Зависимые имена. Начнём немного издалека: если вы видели шаблонный код, то вам может показаться, что в случайных местах по нему раскиданы `typename` и `template`. Например, вот в таких примерах:

```
typename std::vector<T>::iterator it;
// Вместо std::vector<T>::iterator it;
typename foo<T>::template bar<int> y;
// Что это за хрень вообще?
// foo<T>::bar<int> y, если интересно.
```

Так вот зачем это. Пусть мы где-то в пустоте увидели строку кода `(a)-b`. Что это? Разность `a` и `b` либо каст значения `-b` к типу `a`. Или увидели `int b(a)`. Это либо вызов конструктора `int`'а, либо функция `b`, которая принимает на вход значение типа `a` и возвращает `int`. Ещё страшнее: `a < b && c > d`. Это либо логическое выражение, либо `a` — это шаблон с non-type параметром типа `bool`, `b` и `c` — какие-то логические выражения, а `d` — имя. И тогда это `a<b && c> d` — создание переменной `d` типа `a<b && c>`. Как компилятор читает подобные двоякие примеры? Ну, если `a` — это тип, то одно, если не тип — то другое. И обычно компилятор это знает. Проблема в том, что в шаблонах мы можем сделать что-то такое:

```
template <typename T>
void foo(int x)
{
    (T::nested) - x;
}
```

И один хрен вы знаете, что такое `nested` до подстановки. А очень хотите это знать, чтобы отлавливать ошибки раньше, чем подстановка (например, написав `(T::nested) - y`, вы не получили бы ошибку о том, что не существует `y`, сразу). Поэтому вы можете явно указать, что происходит.

```
template <typename T>
void foo(int x)
{
    (T::nested) - x;         // Вычитание.
    (typename T::nested)-x; // Каст.
}
```

Аналогично

```
template <typename T>
void foo(int x)
{
    int b(T::nested);           // Конструктор переменной.
    int b(typename T::nested); // Объявление функции.
}
```

И

```
template <typename T>
void foo(int x)
{
    T::nested < b && c > d;        // Логическое выражение.
    T::template nested<b && c> d; // Переменная шаблонного типа.
}
```

При этом когда у вас есть что-то внешнее, что **не зависит от шаблона**, и имеет в себе `nested`, там писать всё это не обязательно, компилятор сам определит. То что зависит от шаблона, называется dependent. И вот в dependent-штуках обязательно писать `typename`'ы и `template`'ы, а independent — нет. MSVC, кстати, долгое время делал не так (а полностью разбирал шаблонную функцию при подстановке), за что его загнобили, и больше он так не делает, а делает как все: разбирает dependent-выражения при подстановке, а independent — сразу. Это называется «two-phase name lookup».

Incomplete type.

```
template <typename D>
struct base
{
    typename D::type x;
};

struct derived
{
    base<derived> y;
    typedef int type;
};
```

Компилироваться это не будет. Почему? Потому что мы в тот момент, когда мы имеем по сути `type y`, у нас `type` ещё не определён. Когда содержимое какого-то класса не полностью определено, говорят, что этот класс — incomplete type. Приведённый выше пример не очень впечатляющий, более впечатляющий нам пока не доступен, но выглядит он так:

```
template <typename D>
struct base
{
    typename D::type x;
};

struct derived : base<derived>
{
    typedef int type;
};
```

Мораль — шаблоны подставляются по порядку, а не сразу везде.
Ещё про incomplete-типы:

```
struct nested; // Объявлен где-то не здесь.

struct my_type
{
    std::unique_ptr<nested> p;
};
```

Это не компилируется, говорят, что у `nested` не видно деструктора, а `unique_ptr` его вызывает. Фиксится это так:

```
struct my_type
{
    ~my_type();

    std::unique_ptr<nested> p;
};
```

При этом деструктор `~my_type` должен находиться там, где структура `nested` полностью определена.

Декларации шаблонных типов.

```
// foo.h
template <typename T>
void foo();

// foo.cpp
#include "foo.h"

template <typename T>
void foo()
{}

// main.cpp
#include "foo.h"

int main()
{
    foo<int>();
}
```

Это не компилируется, потому что *main.cpp* не может сгенерировать `foo<int>`, а *foo.cpp* не знает, что нужно сгенерировать. Поэтому шаблонные функции пишутся в заголовочных файлах, к ним неявно для вас приписывается `inline`, и всё работает как с `inline`.
Но вообще есть ещё и явное инстанцирование: прямая просьба сгенерировать шаблон от некоторого параметра. Пишется так: `template void foo<int>();` Ещё в C++11 есть вот такая штука:
`extern template void foo<int>();`. Это говорит, что инстанцировать `foo<int>` не надо, ведь его уже инстанцирует кто-то другой. Используется это, например, в `std::string`. `std::string` на самом деле является специализацией `std::basic_string<char>`, и если каждый будет его инстанцировать, ничего хорошего не произойдёт. Поэтому он помечен именно как `extern template`, чтобы его один раз инстанцировал кто-то один (а кто — написано где-то, где вас не касается). При этом все прочие инстанции `std::basic_string` спокойно будут подставляться во всех единицах трансляции.

11 Инструменты, которые могут помочь жить.

Мы поговорим о каких-то мелких вещах, которые помогут вам в некоторых ситуациях, если в о них знаете.

11.1 Отладчики.

Первое: *-g*. Иначе вы проиграли. Второе — *-O0*. Если дебажить код с оптимизациями, то вам могут убрать переменные, переупорядочить инструкции и подобное. Ещё в GCC есть ключ *-Og*, который включает некоторые слабые оптимизации, которые не должны сильно влиять на отладку. Но на самом деле это не очень правда, если вы не разрабатываете игры и не имеете таймаутов, то лучше использовать *-O0*. Вообще в GCC есть много ключей, которые тем или иным образом включают/выключают оптимизацию, от которой зависит отладка, но тут ищите их сами.
Ещё: не обязательно запускать программу под отладчиком, можно присоединиться отладчиком к уже работающей программе. С Windows вообще проблем нет, в Linux по-умолчанию так делать нельзя (по соображениям безопасности), сами погуглите, как отключить эту опцию. Ещё полезная опция — когда ваша программа аварийно завершается, все Linux'ы умеют сбрасывать на диск память. Куда конкретно — смотрите (или изменяйте) файл */proc/sys/kernel/core_pattern*. И потом эту штуку можно в отладчике открыть и посмотреть. Если файл у вас не создаётся — проблемы с *ulimit -c*. Есть ограничения на максимальный размер core dump'а, вот вам надо поставить на unlimited.
Теперь поговорим про сервер дебажных символов. Как мы знаем, дебажные символы заливают в один файл, который отправляют на сервер. И с не так давних времён, любой дистрибутив этот сервер имеет. И вы можете скачать оттуда дебажные символы и исходники (да ещё и нужной версии), чтобы поотлаживать что-то в дистрибутиве.

Продвинутые breakpoint'ы. Представим, что у вас есть программа, которая работает, а потом падает. И вы знаете условие, когда падает (например, когда переменная достигла определённого значения). И отладчики дают вам такую возможность: conditional breakpoint называется. Останавливается в каком-то конкретном случае. В GDB это пишется как *if i == 41* после указания breakpoint'а, например.
Ещё если вы хотите отлаживать `printf`'ами, вам не обязательно их явно писать и перекомпилировать программу. Вы можете написать breakpoint, который не останавливается, а что-то выводит. В GDB это *commands printf "abacaba" end*.
Ещё есть такая штука как watch point'ы. Вы можете поставить слежение за определённой переменной/полем/выражением. Для этого вы (в *gdb*) так и пишете: *watch <expression>*. Ещё watch point можно на адрес поставить при помощи *-l*. За счёт чего это работает? А watch point'ы в процессоре есть. Но вообще это и программно можно эмулировать (помечаем память как то, куда нельзя писать, процессор при записи делает прерывание, которое и передаётся отладчику).

Визуализаторы структур данных. Что отладчик из Visual Studio, что GDB, что LLDB поддерживают возможность красиво напечатать встроенные структуры данных. Чаще всего вы не хотите видеть дерево, когда смотрите на `std::set` или хэш-таблицу в `std::unordered_set`, а хотите видеть лишь элементы внутри. И вам дают такую возможность. При этом три отладчика позволяют написать свои способы красиво что-то напечатать. В отладчике из VS — это XML-ки, в GDB и LLDB — Python'овские скрипты. Причём в VS визуализатор можно в проект добавить, и VS его подхватит.

Reversible-отладчики. Reversible-отладчики — отладчики, которые позволяют ходить не только вперёд, но и назад. Если вы под Linux, то вам нужен UndoDB, имеющий тот же интерфейс, что и GDB. Проблема — он проприетарный и платный (притом недешёвый). Но это в целом круто, работает как магия, но всё ещё дорого. Поэтому у нас есть только инструмент для бедных, созданный людьми из Firefox'а. Но он имеет немного другую специализацию. У них была такая проблема: были тесты, которые спонтанно ломались. И хочется куда-то записать всё что было. Для этого есть *rr* (record-replay). Вы можете куда-то записать, как программа работает (*rr record*), она запишет вам всё, что было, а по *rr replay* вы можете ходить по этой записи. Нужные вам команды — *reverse-finish* (reverse-«выйти из функции»), *reverse-next* (reverse-«следующая строка кода») и всё остальное, что начинается с *reverse-*.

Как работает эта чёрная магия. Вариант «запоминать всё вообще» не подходит (слишком много памяти). Кстати, такое есть и в GDB (он тоже поддерживает обратный ход, но жрёт бесконечность памяти). А какой вариант подходит? Давайте запомним начало программы и будем эмулировать получение данных из внешнего мира. А чтобы не работало бесконечно долго (чтобы при шаге назад не запускалась вся программа полностью с начала), сделаем несколько snapshot'ов в разных местах программы, и будем запускаться от них.
А как остановиться в определённом месте? Ну, у нас есть счётчики в процессоре (см. ниже в разделе про отладчики), только тут мы не профилируем что-то, а задаём счётчик и говорим «исполни N инструкций». Есть проблема с этим, кстати. Поскольку процессор работает спекулятивно, посчитать N инструкций процессора — это не совсем то, что вы хотите. Для сэмплирования разницы никакой, а тут *rr* долгое время с трудом работал на AMD, потому что там не могли найти детерминированный счётчик.
Ещё проблема — недетерминированные инструкции. Скажем «сколько времени прошло с некоторого момента» или генерация случайных чисел. Но вообще на процессорах есть возможность и на этих инструкциях тоже прерваться.
Ещё проблема есть с shared-памятью. Если вы отлаживаете программу, а кто-то извне пишет в её память, вы проиграли. Чтобы с этим справится, UndoDB делает нечто похожее на механизмы *valgrind*.

11.2 Профилировщики.

Программы, которые помогают вам ответить на вопрос, где больше всего времени проводит программа. Понятно, что вам нужны дебажные символы (*-g*), иначе вам максимум машинные инструкции покажут, а не строки программы. Из самых известных — *perf*, встроенный в ядро Linux, в Visual Studio есть профилировщик, использующий встроенные в Windows механизмы. Самый навороченный и крутой — Intel VTune Amplifier. Долгое время он был платный (причём достаточно дорогой), сейчас, кажется, есть какие-то варианты, но это всё сами включите VPN и посмотрите. При этом всё, что мы расскажем, в основном относится с CPU-профилированию (C++, хуле), но вообще *perf*, например, может и записи на диск профилировать, работу с сетью да и вообще кучу разных событий, которую можно сэмплировать.

Базовые возможности `perf`. В профилировщике Visual Studio разберётесь сами, а мы проговорим о том, что умеет *perf*. Он запускается командой *perf* и дальше имеет кучу опций. Самая простая — *perf stat*, с этой опцией показывается не только время, но и всякая другая фигня (branch-miss'ы, cpu-миграции, разные другие штуки). Что-то из этого поставляется ОС, что-то — процессором. Если вы хотите посмотреть список доступных событий — *perf list*. Чем дальше вниз прокручивать этот список — тем более редко используемые параметры там будут. Окей, это всё хорошо, но хочется понять, на какую функцию грешить, если долго работает. Это делается при помощи *perf record*+*perf report*. Первый собирает статистику о функциях, второй — показывает её вам. Причём статистика — не обязательно время. Можно написать *perf record -e cache-misses <программа>*, и тогда вам будут давать статистику по промахам в кэше. Чтобы самому узнать какие-то опции инструкций — *perf help <инструкция>*.
Каким образом работает *perf*? Зависит от процессора, на самом деле. Процессор предоставляет возможность, скажем, каждый тысячный заход в команду давать прерывание (по которому *perf* будет что-то считать). Кстати, есть проблема: процессор имеет ограниченное количество счётчиков, и в таком случае *perf* будет сначала сэмплировать один набор, потом другой, затем третий, чтобы собрать статистику.

General exploration. Хорошо, вот нашли вы долго работающее место, но совершенно не знаете, в чём там проблема. Тут уже *perf* вам не поможет, но может помочь VTune. У него есть режим «general exploration». Он базируется на упрощённой модели, что инструкция сначала передаётся в какую-то front-end-часть процессора, а потом — в back-end. И потом в зависимости от того, поступают ли инструкции из front'а в back и простаивают ли front и back, вам говорят, где проблема (собственно, во front'е ли, в back'е ли, восстанавливается ли вы от branch miss'а или что). И вот в VTune эту идею довели до ума, в связи с чем вам говорят не только тормозите ли вы на front'е или back'е, но и упираетесь ли вы в память, а если да, то в L1, L2, L3 или RAM. Так что VTune сразу говорит вам, в каких функциях какие проблемы. В *perf*'е это только в зачаточном состоянии есть.

Сбор стека вызовов. Следующая возможность *perf*'а. Вот выяснили вы, что долго работает. А кто вызывает то, что долго работает? *perf record -g --call-graph dwarf*. Тогда вам будут показывать стеки вызовов, что довольно полезно. Опция *-g* их сборкой и занимается, а опция *--call-graph* задаёт то, каком образом они собираются. Чтобы не сильно замедляться при сборке стека, *perf* работает в ядре. Но чтобы ядро очень долго не работало, собираются верхушки стеков, а *perf report* пытается их раскрутить. Это и есть режим *--call-graph dwarf*. Ещё есть режим *--call-graph fp*, который собирает стек вызовов по frame-pointer'ам, что делается быстро и честно, но тогда совершенно всю программу надо собрать с ключом *-fno-omit-frame-pointer*, а стандартная библиотека с ним не собрана.
Ещё *perf* можно натравливать на уже работающие программы при помощи *perf -p <pid>*. Более того, *perf* может профилировать больше одной программы. Хоть вообще всё (*sudo perf record -a*).

Альтернативные методы профилирования. Сто́ит сказать, что профилировщики (*perf*, VS profiler, VTune) полагаются на процессор. А ещё есть tracing-профилировщики, работающие на базе компиляторов. Они вставляют инструкции на моменте входа и выхода из функции. Примером таковых является gprof. Они имеют преимущества и недостатки. Преимущество — могут показать число вызовов функции (честно, а не по количество сэмплов), что бывает полезно, чтобы увидеть, что вы могли посадить где-то квадратичную сложность. Недостаток — tracing-профилировщики врут, увеличивая время работы маленьких функций. В связи с этим они используются очень редко.
Что ещё сто́ит упомянуть — *valgrind* имеет встроенный профилировщик. Он в целом адекватный, но у него просмотрщик не очень. Так вот, *valgrind* — это набор инструментов. Если вы не пишете ничего, запускается инструмент *memcheck*. А когда вы используете инструмент *cachegrind*, то *valgrind* эмулирует работу процессора (сколько времени он бы исполнял заданную инструкцию). С виду вообще бред какой-то (все системные вызовы, например, занимают 0 времени, что это вообще), но и тут есть свои ништяки. Есть история про библиотеку SQLite для работы с базами данных. В один момент её решили оптимизировать и микрооптимизациями ускорили её на 61%. В качестве профилировщика они использовали *cachegrind*, который даёт вам воспроизводимые данные без шума. И если вы сделали микрооптимизацию, вы увидите ускорение на 0.1%, и его не съест шум.

11.3 Проверка покрытия.

Если тесты не заходят в некоторую функцию или на определённую строку надо, значит на тех местах может быть написан полный бред. Поэтому следует проверить, что тесты проходят в вашей программе вообще везде. Это называется coverage, и запускать его так. Сначала надо скомпилировать программу с ключом *--coverage*. Тогда при запуске создадутся файлы расширения *.gcda* и *.gcno*. Чтобы их посмотреть, вам нужна программа *gcov* с ключиками, это сами разберётесь.

11.4 Полезные ключи компиляции.

LTO. Linking-time optimizations — *-flto* (ключ передаётся компилятору и линковщику). Это мы уже обсуждали. Не обсуждали, как работает. А работает так: компилятор откладывает генерацию кода до этапа линковки, а в файл записывает что-то промежуточное. Это промежуточное представление даже глазами читать можно, это просто какой-то императивный код. В Clang'е это представление — LLVM-IR, в GCC — gimple. Второй вообще похож на C. Какие плюсы и минусы у LTO? С одной стороны, бенчмарки говорят, что ускорение на несколько процентов. С другой, на бенчмарках и так всё сильно оптимизировано, даже то, что LTO мог бы ускорить. Ещё LTO сильно уменьшает размер программы. Там, параметры в функцию передаются всегда одинаковые, или `if` всегда в одну сторону. Это позволяет компилятору (если он видит программу целиком), уменьшить размер. Ещё LTO хорошо с шаблонами взаимодействует. Поскольку шаблоны подставляются в каждой единице трансляции, если вы сгенерировали что-то здоровенное, то без LTO оптимизироваться это будет везде, а с LTO — только один раз. И самое интересное — LTO может показать вам ODR, что в C++ очень круто. Недостатки — LTO убивает возможность пересобрать один файл, самое долгое время занимает линковка, вне зависимости от того, один файл вы изменили или все.

PGO. Profile-guided optimizations. Идея в следующем. Иногда изменение компилятора даёт эффект только в некотором случае. Если у нас есть цикл, мы можем за'`if`'ать случай aliasing'а, использовать SIMD, сделать кучу ещё разных интересных вещей. Но есть всё это мы будем делать с каждым циклом, мы проиграем, потому что нужно это отнюдь не всегда, а если мы всё применим, увеличится размер программы, а значит будет хуже работать программный кэш, что может даже к замедлению привести. Или с `if`'ами процессору хочется знать, какая ветка более вероятна. Так вот, вы можете запустить компиляцию с ключом *--profile-generate*, запустить программу, после чего заново скомпилировать с *--profile-use*. Это также даёт ускорение, а из минусов — вам нужно иметь репрезентативный набор тестов. Впрочем, иметь набор тестов и так очень полезно, о каком ускорении можно говорить, если тестов нет.
Кстати, в случае с GCC то, что не попало в профиль, считается «холодным кодом» и оптимизируется на размер, а не на скорость. Поэтому в GCC, если у вас не репрезентативный профиль, вы можете проиграть. В Clang такого нет.

BOLT. BOLT — это не ключ компилятора, это инструмент, которому вы даёте уже скомпилированный бинарник и профиль, после чего оптимизирует. Есть статистика, согласно которой, BOLT работает лучше чем PGO на 15%. Дело в том, что он группирует горячие данные вместе, тем самым эффективно используя кэш для инструкций. Почему это не используется в PGO — непонятно. LLVM пытались создать свой аналог (названный LLVM-propeller), но он **не взлетел**. Поэтому теперь сам BOLT есть внутри LLVM четырнадцатой версии.

11.5 Статические анализаторы.

Это компьютерный сеньор, пользующийся методом пристального взгляда.

```
char* s = static_cast<char*>(malloc(N));
// ...
delete[] s;
```

Статический анализатор от Microsoft скажет вам, что вы дурак. Почему так не делают все компиляторы? Потому что чисто по математическим причинам нельзя в Тьюринг-полном языке проверить, достижима ли строка или нет, а значит тем более нельзя гарантированно найти все ошибки (не имея ложных срабатываний). Впрочем, иногда ошибки статического анализатора переезжали в предупреждения компилятора. Например, код

```
printf("%p\n", 42);
```

В VS2017 давал ошибку статического анализатора, VS2019 — предупреждение.

GCC PR18501.

```
void f()
{
    bool first;

    for (; !finish;)
    {
        if (!first)
        something();
        first = false;
    }
}
```

```
void g()
{
    int value;
    if (flag)
        value = 42;

    something();

    if (flag)
        consume(value);
}
```

Пользователям хочется, чтобы первый пример был некорректным, второй — корректным. Но есть проблема. Статический анализатор можно запускать сразу, а можно после некоторых оптимизаций. Если мы статически анализируем в самом начале, мы не можем во втором примере узнать, что `if (flag)` два раза — это одно и то же, нужно узнать, что `something` его не меняет. Если оптимизировать, то после оптимизации мы не увидим ошибки во втором случае, но не увидим и в первом, так как утратим информацию о том, что `first` не был проинициализирован. Поэтому GCC пытается минимизировать ложные срабатывания, а Clang максимизировать истинные. А чтобы сделать всё хорошо, нужно иметь другую логику, нежели имеют компиляторы.

SAL-аннотации. Хорошо, статические анализаторы GCC и Clang междпроцедурные и даже могут смотреть сквозь единицы трансляции. В Visual Studio, увы, не так, поэтому вот такой код:

```
void f(int* val)
{
    printf("%d", *val);
}
```

Непонятно, корректный ли. Хочется знать, какой у функции контракт. Можно ли передавать туда указатель на неинициализированную память, можно ли `NULL` и т.д. Поэтому есть SAL-аннотации. Например — `_Out_` говорит, что вы передаёте туда выделенную, но не обязательно проинициализированную память. Или `_Out_writes_bytes_(size * sizeof(int))` говорит, что вы передали указатель, в котором можно записать `size * sizeof(int)` байт.
У Clang есть то же самое (например, `int a __attribute__((guarded_by(m)));`). В этом примере при попытке записать в переменную не заблокировав `m`, вам дадут предупреждение.
Несмотря на свою привлекательность, статические анализаторы не очень популярны из-за того, что баланс между ложными срабатываниями и ложными не-срабатываниями не такой, как люди хотят видеть.

12 Пространства имён.

А что там вообще говорить? Да, но что-то проговорить надо.
Что это такое и на кой оно надо? Есть библиотека cairo для векторной графики. И если полазить по её страницам, то можно заметить, что все функции называются как-то вида `cairo_mesh_pattern_begin_patch`, `cairo_mesh_pattern_move_to`, `cairo_mesh_pattern_curve_to` и подобные бесконечно длинные названия? Чтобы не было коллизий. Вот назовёте вы функцию `move_to`, а кто-то другой из другой библиотеки назовёт также свою функцию с совершенно другой семантикой. Проблемы. Поэтому в C все типы, функции и вообще всё имеет префикс по названию библиотеки, названию чего-нибудь ещё, и вот так появляются бесконечно длинные названия. В GNOME таких префиксов штуки по 3. В C++ чтобы подобных префиксов не было, были созданы пространства имён, где вы пишете что-то такое:

```
namespace cairo
{
    namespace mesh
    {
        namespace pattern
        {
            void curve_to();
        }
    }
}
// ...
cairo::mesh::pattern::curve_to();
```

Что мы выигрываем от этого? А то что находясь внутри пространства имён, мы можем вызывать свои функции без этих длинных префиксов:

```
namespace cairo
{
    namespace mesh
    {
        namespace pattern
        {
            void curve_to();

            void test()
            {
                curve_to();
            }
        }

        void test()
        {
            pattern::curve_to();
        }
    }

    void test()
    {
        mesh::pattern::curve_to();
    }
}
```

Кстати, если вам интересно, чем отличаются функции `test` с точки зрения линковщика, то в имена декорированных символов просто вписываются особым образом эти самые пространства имён.
Всё что мы пишем вне любых пространств имён, считается лежащим в «глобальном пространстве имён». Чтобы обратиться явно к чему-то в нём, напишите в начале имени просто двойное двоеточие.

Способы не писать длинные названия извне. Первый — namespace alias. Есть в стандартной библиотеке пространство имён `std::filesystem`. Если мы не хотим писать долгое имя класса `std::filesystem::path`, мы пишем `namespace fs = std::filesystem`, и теперь можем писать `fs::path`.
Второй способ — using-declaration: `using std::filesystem::path`. Теперь вы можете писать просто `path`, и это будет синонимом `std::filesystem::path`. Так можно делать с любыми сущностями, кроме других пространств имён. Надо понимать, что `using`'и объявляют вам тип именно там, где он написан. Если вы напишете `using` внутри другого пространства имён, то там он и останется. То есть

```
#include <filesystem>

namespace f
{
    using std::filesystem::path;
}

path p; // Некорректно.
f::path p; // Корректно.
```

Нечто похожее, но немного другое — using-directive: да возликует Рябчун, `using namespace std`. Можно было бы подумать, что она делает `using` всего, что в пространстве имён есть. Но это не так:

```
namespace n1
{
    class mytype {};
    void foo();
}

namespace n2
{
    class mytype {};
    void bar();
}

// ...
using n1::mytype;
using n2::mytype; // Ошибка.

// ...
using namespace n1;
using namespace n2; // Нет ошибки.
mytype a; // Тут есть ошибка "mytype is ambiguous".
```

`using namespace` не декларирует всё, что есть, а просто помечает, что в текущем пространстве имён используется другое. И компилятор просто берёт, и всегда когда ищет что-то в одном пространстве имён, также ищем это и во втором. Это даёт такого рода эффекты:

```
namespace n
{}

using namespace n;

namespace n
{
    class mytype {};
}

// ...
mytype a;
```

Такая штука вполне компилируется и делает то, что вы предполагаете. Понятно, что на той же строке вместо `using namespace n` написать `using n::mytype` нельзя.

Приоритеты пространств имён. Обычно если мы идём по уровням вверх в глобальное пространство, если в каком-то месте нашли имя, то оно нам и нужно. Если нашли два, то ambiguous. Если нашли два на разных уровнях, выбирается то, что ближе. Но вопрос — как со всем этим взаимодействуют `using`'и.

```
namespace n1
{
    int const foo = 1;
}

namespace n2
{
    int const foo = 2;

    namespace n2_nested
    {
        using n1::foo;

        int test()
        {
            return foo;
        }
    }
}
```

Так программа выводит единицу. Мы не нашли имя в `test`, зато нашли в `n2_nested`, притом одно. Смысл искать выше?

```
namespace n1
{
    int const foo = 1;
}

namespace n2
{
    int const foo = 2;

    namespace n2_nested
    {
        using namespace n1;

        int test()
        {
            return foo;
        }
    }
}
```

Так программа выводит двойку. Дело в том, что имя `foo` видно для нас как будто оно находится в наименьшем общем предке того, где мы находимся (`test`) и `n1`. Поэтому мы найдём `n2::foo` на уровень ближе, чем `n1::foo`, и используем его. И из этого вот такой пример:

```
namespace n1
{
    int const foo = 1;
}

int const foo = 100;

namespace n2
{
    namespace n2_nested
    {
        using namespace n1;

        int test()
        {
            return foo;
        }
    }
}
```

Не компилируется, потому что мы видим `n1::foo` и `::foo` на одном и том же уровне.

Argument-dependent lookup.

```
namespace mylib
{
    class big_integer
    {};

    big_integer operator+(big_integer const& a, big_integer const& b);
    void swap(big_integer& a, big_integer& b);
}

int main()
{
    mylib::big_integer a, b;
    a + b; // Мы не видим оператор + ни в main, ни в глобальном пространстве имён.
           // Но почему-то работает.
}
```

ADL (argument-dependent lookup). Когда мы вызываем оператор, он ищется не как описано выше, а учитывает типы параметров. Точнее, смотрит в то пространство имён, где написаны аргументы оператора. Для каждого аргумента ищет в его пространстве имён. Причём только в его пространстве имён, не выше.
То же самое работает вообще для любых функций, не только для операторов, кстати. Для `swap` тоже. Со `swap` вообще интересная тема. Мы хотим для каких-то шаблонных классов `swap`'нуть их. Как? Вот так:

```
template <class T>
void foo(T a, T b)
{
    // ...

    using std::swap;
    swap(a, b);

    // ...
}
```

Теперь что получается? У нас получается шаблонный `std::swap` и, возможно, не-шаблонный ADL. Если у нас есть ADL, выбирается он, потому что из шаблонного и не-шаблонного выбирается второй. Если нет ADL, то есть только `std::swap`, и он вызывается. Если не сделать `using std::swap`, то функция не будет работать для, скажем, `int`'ов. В контексте шаблонов надо сказать, что ADL работает на этапе подстановки шаблона, в то время как поиск имени по дереву вверх — на этапе парсинга.

Безымянные пространства имён. Есть вот такая штука:

```
namespace
{
    // ...
}
```

Она по определению эквивалентна

```
namespace some_unique_identifier
{
    // ...
}

using namespace some_unique_identifier;
```

На кой это? Вот зачем. В C было ключевое слово `static` под переменные, функции, но не типы. Потому что типы не генерируют код, они и так локальные на единицу трансляции. Но в C++ они (из-за наличия специальных функций-членов класса) его генерируют. Если у нас есть два нетривиально-разрушаемых типа `mytype` в разных единицах трансляции, у него будет конфликт деструкторов. Более того, тут есть ещё более интересный пример:

```
// a.cpp
struct my_type
{
    int a;
};

void foo()
{
    std::vector<mytype> v;
    v.push_back(/*...*/);
}

// b.cpp
struct my_type
{
    int a, b;
};

void bar()
{
    std::vector<mytype> v;
    v.push_back(/*...*/);
}
```

Тут сами классы тривиально делают вообще всё (создаются, копируются и разрушаются), значит с ними нет нарушения. Но есть нарушение ODR в `std::vector<mytype>::push_back`, он делает разные вещи для разных `mytype`.
И тут приходит прекрасный фикс проблемы `static` на типы: анонимные пространства имён, которые (как несложно заметить), делают именно то, что вам нужно. Более того, теперь можно вообще нигде не писать `static`, а писать анонимные пространства имён (собственно, поэтому `static` на самом деле deprecated). И анонимные пространства имён даже лучше:

```
template <int*>
struct foo
{};

int x, y;

int main()
{
    foo<&x> a;
    foo<&y> a;
}
```

С `foo<&x>` мы, вроде, понимаем, что на этапе компиляции мы знаем адрес переменной, вот и хорошо. А когда мы напишем

```
template <int*>
struct foo
{};

static int x, y;

int main()
{
    foo<&x> a;
    foo<&y> b;
}
```

В C++03 возникнет проблема, потому что теперь `x` — больше не уникальное имя. А это проблема, поскольку `foo<&x>` — это декорированное имя `foo`, в который встроили адрес переменной `x`. А когда мы имеем `static`, из-за не уникальности строки `&x` в разных единицах трансляции, уникально задекорировать `foo<&x>` не получится.
Итак, `static` сделали deprecated в C++03, но в C++11 сказали, что если человек пишет «`static`», он имеет ввиду безымянное пространство имён.

13 Type-based dispatch.

В языке существует и иногда необходимо узнавать какое-то свойство у типа. Если мы пишем обобщённое возведение в степень, нужно спрашивать, что считается единицей. Или есть мы пишем операции с числами, хочется взять максимум данного типа. Очень много из такого делает стандартная библиотека: например, есть `std::advance` — функция, которая делает итератору `+=`, даже если он так не умеет, а умеет только `++`. И тут мы либо делаем `+=`, либо `++` много раз, в зависимости от типа. Надо спросить, умеет ли он так.
Часть запросов к типу предоставляет стандартная библиотека, например, в заголовочном файле `type_traits`, где есть бесконечное количество шаблонных констант `is_trivially_destructible_v`, `is_empty_v`, и прочих других. Какие-то из встроены в компилятор, какие-то вы можете реализовать сами (`is_signed_v`, например, можете запросто). Или можете сами реализовать полностью `std::numeric_limits` — класс с миллионом статических полей, который для целочисленных типов и типов с плавающей точкой даёт информацию о минимуме, максимуме или чём-то ещё.
Как работают штуки из `type_traits`? И почему оканчиваются на `_v`? Дело в том, что до C++14 у вас не было шаблонных переменных (а по сути `is_empty_v` — шаблонная переменная и есть). Поэтому создали шаблонный класс `is_empty` со статическим полем `value`, в котором то, что вам нужно. А когда в C++14 такое появилось, вы смогли писать `is_empty_v`, и это уже реальная `bool`'евая константа, которую можно использовать.

Наивный способ диспатчить что-то исходя из типа. *Constexpr if statement.* Пример использования `type_traits`: хотим мы вызвать деструкторы всех элементов на отрезке:

```
template <class T>
void destroy(T* first, T* last)
{
    for (T* p = first, p != last; p++)
        p->~T();
}
```

Для некоторых классов это можно реализовать иначе. Например, для типов, которые тривиально разрушаются, можно ничего не делать:

```
#include <type_traits>

template <class T>
void destroy(T* first, T* last)
{
    if (!std::is_trivially_destructible_v<T>)
        for (T* p = first, p != last; p++)
            p->~T();
}
```

Но тут есть проблема. Давайте аналогичным образом напишем свой `std::advance`.

```
#include <type_traits>
#include <iterator_traits>

template <class It>
void advance(It& it, ptrdiff_t n)
{
    using category = typename std::iterator_traits<It>::iterator_category;
    if (std::is_base_of_v<std::random_access_iterator_tag, category>)
    {
        it += n;
    }
    else
    {
        while (n > 0)
        {
            --n;
            ++it;
        }
        while (n < 0)
        {
            ++n;
            --it;
        }
    }
}
```

Тут когда вы запустите это от `std::list<T>::iterator`, ваш код не скомпилируется. Потому что компилируются всё равно обе ветки, в частности компилируется `+=`, а для итератора списка оно не компилируется. В C++17 есть простое решение этой проблемы: `if constexpr` — работает как `if`, но только с compile-time константами, и при этом компилируется только нужная ветка. Но так сделать у вас есть возможность не всегда.

Iterator dispatch. В нашем случае можно переписать так:

```
#include <iterator_traits>

template <class It>
void advance_impl(It& it, ptrdiff_t n, std::random_access_iterator_tag)
{
    it += n;
}

template <class It>
void advance_impl(It& it, ptrdiff_t n, std::input_iterator_tag)
{
    while (n > 0)
    {
        --n;
        ++it;
    }
    while (n < 0)
    {
        ++n;
        --it;
    }
}

template <class It>
void advance(It& it, ptrdiff_t n)
{
    using category = typename std::iterator_traits<It>::iterator_category;
    advance_impl(it, n, category());
}
```

Это называется iterator dispatch, и работает также хорошо, как и `if constexpr`, несмотря на передачу лишнего параметра (этот параметр — пустая структура, его на самом деле никто не передаёт).

`std::conditional`. Ну, хорошо, но тут у нас уже есть теги. А если их нет? Например, если мы хотим таким же образом переписать `destroy`? Тогда создадим эти теги сами.

```
struct trivially_destructible_tag
{};

struct not_trivially_destructible_tag
{};

template <class T>
void destroy_impl(T* first, T* last, trivially_destructible_tag)
{}

template <class T>
void destroy_impl(T* first, T* last, not_trivially_destructible_tag)
{
    if (!std::is_trivially_destructible_v<T>)
        for (T* p = first; p != last; p++)
            p->~T();
}

template <class T>
void destroy(T* first, T* last)
{
    // Хочется как-то выбрать одну структуру из двух на этапе компиляции.
}
```

Хочется как-то выбрать одну структуру из двух на этапе компиляции. Как? Вот так:

```
template <bool Cond, typename IfTrue, typename IfFalse>
struct conditional
{
    using type = IfFalse;
};

template <typename IfTrue, typename IfFalse>
struct conditional<true, IfTrue, IfFalse>
{
    using type = IfTrue;
};

template <class T>
void destroy(T* first, T* last)
{
    using tag = typename conditional<is_trivially_destructible_v<T>,
                                     trivially_destructible_tag,
                                     not_trivially_destructible_tag>::type;
    destroy_impl(first, last, tag());
}
```

Барабанная дробь, такое уже есть, и называется `std::conditional`. А `typename std::conditional</*...*/>::type` также сокращается до `std::conditional_t`. Итого наш пример выглядит так:

```
#include <type_traits>

struct trivially_destructible_tag
{};

struct not_trivially_destructible_tag
{};

template <class T>
void destroy_impl(T* first, T* last, trivially_destructible_tag)
{}

template <class T>
void destroy_impl(T* first, T* last, not_trivially_destructible_tag)
{
    if (!std::is_trivially_destructible_v<T>)
        for (T* p = first; p != last; p++)
            p->~T();
}

template <class T>
void destroy(T* first, T* last)
{
    using tag = std::conditional_t<is_trivially_destructible_v<T>,
                                   trivially_destructible_tag,
                                   not_trivially_destructible_tag>;
    destroy_impl(first, last, tag());
}
```

Это кайф, но у этого есть проблема. Когда у нас функции были как перегрузки, мы могли свободно добавлять в список перегрузок новые классы с новыми свойствами. А когда мы делаем это `if`'ами (хоть `if constexpr`, хоть `std::conditional_t`), новые классы с новыми свойствами не добавить.

SFINAE. И есть способ. Напишем вот что:

```
template <class C>
void erase(C& cont, typename C::iterator pos)
{}

template <class T, size_t N>
void erase(T (&cont)[N], T* pos)
{}
```

Когда мы вызываем это от чего-то, у нас сначала вывод параметров, потом подстановка. Что будет при вызове от `int arr[100]`? Вывод даст нам `C = int[100]` в первом случае, `T = int`, `N = 100` во втором. Но когда мы подставим тип в аргументы, во втором случае всё получится, а в первом мы увидим `int[100]::iterator`. Так нельзя, это ошибка компиляции! Но тогда как-то совсем было бы грустно жить с таким кодом. К тому же зачем, если у нас есть корректная подстановка. Поэтому считается, что если ошибка подстановки, то вариант просто не рассматривается. Это имеет название: substitution failure is not an error (SFINAE). Провал вывода также не является ошибкой компиляции, если что.

`std::enable_if`. Теперь, вооружившись SFINAE, сделаем так, чтобы наш `destroy` работал без `if`'ов:

```
template <bool>
struct enable_if
{};

template <>
struct enable_if<true>
{
    using type = void;
};

template <class T>
typename enable_if<std::is_trivially_destructible_v<T>>::type // Это возвращаемое значение.
destroy(T* first, T* last)
{}

template <class T>
typename enable_if<!std::is_trivially_destructible_v<T>>::type
destroy(T* first, T* last)
{
    if (!std::is_trivially_destructible_v<T>)
        for (T* p = first, p != last; p++)
            p->~T();
}
```

Опять же, подобная штука в стандартной библиотеке есть, и называется `std::enable_if`. Для `typename enable_if</*.` также создана короткая версия: `std::enable_if_t`. На практике SFINAE применимо где-нибудь в таком месте:

```
template <class T>
struct vector
{
    void assign(size_t count, T const& value);

    template <class InputIt>
    void assign(InputIt first, InputIt last);
};

int main()
{
    vector<size_t> v;

    v.assign(10, 0); // Выбирается шаблонная перегрузка, потому что подходит идеально.
}
```

Исправляется вот так:

```
#include <iterator_traits>
#include <type_traits>

template <class T>
struct vector
{
    void assign(size_t count, T const& value);

    template <class InputIt>
    std::enable_if_t<
       std::is_base_of_v<
            std::input_iterator_tag,
            std::iterator_traits<InputIt>::category
         >
    > assign(InputIt first, InputIt last);
};

int main()
{
    vector<size_t> v;

    v.assign(10, 0);
}
```

Это долго и неудобно. Зато работает. Но есть вам очень не нравится, в C++20 есть концепты. Пример выше с их использованием вообще пишется на ура:

```
#include <iterator>

template <class T>
struct vector
{
    void assign(size_t count, T const& value);

    template <std::input_iterator InputIt>
    void assign(InputIt first, InputIt last);

    /* Также можно вот так:
    template <class InputIt>
        requires std::input_iterator<InputIt>
    void assign(InputIt first, InputIt last);

    После requires можно и что-то более сложное писать. */
};

int main()
{
    vector<size_t> v;

    v.assign(10, 0);
}
```

У концептов есть ещё одно преимущество, помимо размера. Когда мы пользуемся SFINAE, нам необходимо перебрать все случаи перегрузок. Если вы в `destroy` написали перегрузку под `std::trivially_destructible`, напишите под `!std::trivially_destructible`. А если вы хотите расширять, будьте добры изменить предикаты. А концепты умеют понимать, что один концепт расширяет другой, как с шаблонами. И выбирать наиболее специализированный вариант.

14 Наследование.

Предполагается, что что-то о наследовании вы знаете и примерно представляете, что это такое (хотя бы на уровне концепта). Если нет — жаль. А мы не будем обсуждать всё в мельчайших деталях, потому что в книжках оно обсасывается очень подробно.
Про наследование сложно говорить в том же ключе, в котором мы говорили об исключениях/шаблонах и т.п. Говорили мы о том, что есть проблема, и вот как она решается. Нет конкретной задачи, где необходимо наследование. Поэтому на эту тему таким способом смотреть не будем, а будем иначе: у нас сначала будет механизм, и потом мы будем его применять.

Немного введения. Итак, откуда идёт наследование? Пусть мы решаем моделирование дорожной сети, хотим понять, как переключать светофоры, где построить дорогу и т.п. И в этой области у нас есть какие-то объекты. И вот объектам предметной области сопоставим объекты нашей программы. И у нас очень естественно получается полиморфизм — у нас есть произвольные транспортные средства, которые очень похожи, а есть автобусы, трамваи и подобное, то есть более специализированные штуки. И есть мы можем что-то сделать с транспортным средством в общем, давайте так и будем делать. Это не столько способ организации программы, сколько образ мысли. Причём полезный: мы можем сразу начать декомпозировать задачу (даже не зная её решения), можем сразу объяснить, что где происходит и т.д. И, кстати, совсем необязательно объекту реальности сопоставлять объект программы, это может быть неэффективно — если мы решаем задачу о минимизации чего-то (за минимальные деньги перестроить дорогу так что...), то совершенно необязательно у вас будут такие же объекты. А ещё бывает ситуация, когда вы придумываете движок регулярных выражений — никакие структуры из внешнего мира не приходят, вы делаете что-то, не имеющее отношения к реальности. И реальные программы где-то посередине: часть из реального мира, часть к нему не имеет отношения.

Итак, наследование. Итак, как делать наследование в C++.

```
struct vehicle
{
    size_t registration_number;
};

struct bus : vehicle
{};

struct truck : vehicle
{};
```

(Надо понимать, что вам не нужно создавать класс `bus`, если с автобусом вы не хотите как-то особо взаимодействовать.)
Так вот, что тут происходит? Тут у классов `bus` и `truck` также есть регистрационный номер. Правильный способ думать об этом — как будто у них есть особое поле типа `vehicle`, и если вы обращаетесь в чему-то в вашем классе, чего в нём нет, то вы ищете это в базовом классе. Что будет, кстати, есть в базовом и производном классе есть переменная с одним и тем же именем? Во-первых, это разрешено компилятором — у вас будет две разных переменные с одним именем (потому что вы можете не знать своих родителей полностью, или когда вы добавляли поле себе, у родителя его не было). Если вы хотите обратиться к переменной базового класса, то делается это вот как:

```
struct base
{
    int xyz; // 1
};

struct derived : base
{
    int xyz; // 2
};

int main()
{
    derived d;
    d.xyz = 123; // Изменяется 2.
    ((base&)d).xyz = 123; // Изменяется 1.
    d.base::xyz = 123; // Специальный синтаксис для изменения 1.
}
```

Второе, что позволяет делать наследование — приведение указателей и ссылок наследуемого класса к указателям и ссылкам базового, как тут в предпоследней строке. Те же самые правила применимы к методам.

Worst practices. Во-первых, не надо наследоваться, если вам нужно только расширить класс. Вот хотите вы добавить новый функционал в std::string, не нужно от него наследоваться. Потому что вы и ваш коллега захотят расширить строки по-разному, получите два новых типа, замените std::string на свой, а потом не сможете вызывать функции друг друга. Не надо так, создайте обычную функцию. Обычные функции — это хорошо, не надо писать всё классами от того, что вы научились.
Во-вторых, не надо создавать отдельный класс под одну операцию.

```
struct string_printer
{
private:
    std::string msg;

public:
    string_printer(const std::string& msg)
        : msg(msg)
    {}

    void print()
    {
        std::cout << msg;
    }
};
```

Это просто шиза, этим и является, но это пока пример простой выглядит идиотски. К тому же, у этого есть другая проблема — вот сделали вы string_printer("Hello, world").print(). А что если вы не сделаете print, или сделаете дважды? Потому что напечатать-то вы можете и в конструкторе, зачем вам метод? Мораль: не заводите класс, если вам нужно сделать действие. Исключение — какие-нибудь компараторы, которые в общем случае могут быть полноценными классами, но вообще могут являться и обёрткой вокруг функции. И вот тут ничего не поделаешь, std::map принимает класс, а не функцию.

Виртуальные функции. Виртуальные функции — единственное, для чего вам нужно наследование. Если вы не используете виртуальные функции, наследование вам по сути не нужно.

```
struct vehicle
{
    void print()
    {
        std::cout << "vehicle" << std::endl;
    }
};

struct bus : vehicle
{
    void print()
    {
        std::cout << "bus" << std::endl;
    }

};

struct truck : vehicle
{};

int main()
{
    vehicle v;
    v.print(); // vehicle
    bus b;
    b.print(); // bus
    truck t;
    t.print(); // vehicle
}
```

А теперь мы делаем функцию foo:

```
void foo(vehicle& v)
{
    v.print();
}
```

Тут всегда мы смотрим на vehicle и всегда вызываем его функцию print, даже если передадим туда bus. А не хочется. Сначала введём пару определений. Есть статический тип — это то, что видит компилятор (в данном случае vehicle). Но по сути ваш vehicle может быть bus'ом или truck'ом. И вот это называется динамическим типом. Так вот, виртуальные функции позволяют выбирать метод исходя из динамического типа, а не статического:

```
struct vehicle
{
    virtual void print()
    {
        std::cout << "vehicle" << std::endl;
    }
};

struct bus : vehicle
{
    void print()
    {
        std::cout << "bus" << std::endl;
    }

};

struct truck : vehicle
{};

int main()
{
    vehicle v;
    foo(v); // vehicle
    bus b;
    foo(b); // bus
    truck t;
    foo(t); // vehicle
}
```

Где это бывает нужно в жизни? Ну, например, вы можете вывести в поток что-то:

```
std::ostream& operator<<(std::ostream& ostr, /*something*/)
{
    return ostr << /* something */ << /* something else */;
}
```

И теперь, поскольку и std::stringstream, и std::ofstream наследуются от std::ostream, это работает для совершенно любых потоков, учитывая то, что у них по-разному определён оператор << от, например, числа.

Срезка (slicing) наследуемого класса.

```
int main()
{
    bus b;

    vehicle v = b;
    v.print();
}
```

Тут b неявно приводится к vehicle&, после чего вызывается созданный компилятором конструктор копирования vehicle(vehicle const&). И создаётся новый объект, у которого динамический тип vehicle, и у него вызывается его print.
Это называется срезкой и в большинстве случаев — не то что вы хотите. Так что удалите конструктор копирования и оператор присваивания у класса vehicle.

Виртуальные деструкторы.

```
int main()
{
    bus* b = new bus();

    vehicle* v = b;
    v->print();

    delete v;
}
```

Тут вызывается v->~vehicle. А если bus имеет какой-то нетривиальный деструктор, он не вызовется. Поэтому тут вам всё также надо вызывать деструктор в зависимости от динамического типа, а не статического.

```
struct vehicle
{
    virtual ~vehicle() {}
};
```

С точки зрения языка, вы не имеете права делать delete у базового класса, есть создали наследуемый и не пометили деструктор базового как virtual. Если будете так делать — UB. **Даже если все деструкторы тривиальные.** Почему? А вот:

Множественное наследование.

```
struct base1
{
    int x;
};
struct base2
{
    int y;
};
struct derived : base1, base2
{};

int main()
{
    derived* d = new derived;
    base2* b2 = d;

    delete b2;
}
```

Это некорректно, потому что первый базовый класс лежит по тому же адресу, что и оригинальный класс, а второй — со смещением. Поэтому его удалить нельзя, вы освобождаете память не по тому указателю. А виртуальный деструктор вас спасёт.
Ещё про множественное наследование нужно сказать вот что:

```
struct base1
{
    int x;
};
struct base2
{
    unsigned x;
};
struct derived : base1, base2
{};

int main()
{
    derived d;
    std::cout << d.x; // Некорректно, непонятно, какой x имеется ввиду.
    std::cout << d.base1::x; // Корректно, x из base1.
    std::cout << d.base2::x; // Корректно, x из base2.
}
```

Ещё один интересный момент про множественное наследование:

```
struct base2;
struct derived;
base2& to_base2(derived& d)
{
    return (base2&)d;
}

struct base1
{
    int x;
    base1(int x) : x(x) {}
};
struct base2
{
    int y;
    base2(int y) : y(y) {}
};
struct derived : base1, base2
{
    derived(int x, int y)
        : base1(x), base2(y) {} // Кстати, так вызываются конструкторы базовых.
};

int main()
{
    derived d(1, 2);
    std::cout << to_base2(d).y << std::endl; // Выводится 1.
}
```

Почему? А вот почему. Когда мы пишем to_base2, мы ещё не знаем, что один класс наследуется от другого, причём так, что ещё и указатели надо двигать. Он будет их двигать, если написать to_base2 после классов, а так нет. Поэтому в C++ запретили касты из C, а добавили 4 новых.

Касты.

static_cast. Чаще всего вам нужен именно он. Кастует числа друг в друга, ссылки и указатели по иерархии наследования в любую сторону, void* в любой указатель и обратно. При этом, понятно, кастовать void* куда-то корректно, если там изначально было то, куда вы кастуете. Аналогично, вниз по иерархии (от базового к наследуемому) можно кастовать только тогда, когда совпадает динамический тип. Иначе UB.

const_cast. Снимает модификаторы const и volatile. Чаще всего это делать не надо, но иногда бывает нужно всё-таки. В стародавние времена, когда const'ов не было, были функции, принимавшие указатель. Неконстантный. Хотя не меняли его содержимое. И вот в таком случае вы можете снять const с указателя. Про снятие const'ов мы уже говорили, если изначальный объект был const, снимать с него const ни в коем случае нельзя. Если изначальный объект константным не был, а потом вы сначала навесили const, а потом сняли, то всё хорошо.

reinterpret_cast. Это всё зашкварное из C-style cast'а. Перевод указателей из несвязанных друг с другом типов, указатели в число. Простой и эффективный способ получить UB. В стандарте так и написано, это implementation-defined cast. Обратитесь к поставщику вашего компилятора, чтобы понять, как у вас работает reinterpret_cast.

dynamic_cast. Это немного другое, нежели все остальные касты. dynamic_cast работает только для указателей и ссылок на полиморфные (хотя бы одна виртуальная функция) классы. Кастует по иерархии вниз (это, напомню, от базового к наследуемому), но, в отличие от static_cast'а, может проверять, что преобразование корректно. То есть static_cast по иерархии вниз берёт и кастует. А dynamic_cast кастовать пытается, и если у него не получается, то возвращает nullptr в случае указателей, или кидает std::bad_cast в случае ссылок. Чтобы понять, как это работает, надо понять, как вообще работают виртуальные функции изнутри.

Полиморфизм изнутри. Как бы мы сделали полиморфизм руками, если бы у нас его не было? Ну, через указатели на функции. По-другому не получится, потому что мы не знаем список всех наших наследников:

```
struct base
{
    base();

    void (*foo)(base*);
    void (*bar)(base*, int);
    void (*baz)(base*, double);
};
void foo_base(base* self)
{
    // ...
}
void bar_base(base* self, int x)
{
    // ...
}
void baz_base(base* self, double y)
{
    // ...
}
base::base()
    : foo(foo_base), bar(bar_base), baz(baz_base)
{}

struct derived : base
{
    derived();
};

void foo_derived(base* self)
{
    derived* derived_self = static_cast<derived*>(self);
    // ...
}
void bar_derived(base* self, int x)
{
    derived* derived_self = static_cast<derived*>(self);
    // ...
}
void baz_derived(base* self, double y)
{
    derived* derived_self = static_cast<derived*>(self);
    // ...

}
derived::derived()
: foo(foo_derived), bar(bar_derived), baz(baz_derived)
{}

int main()
{
    derived d;
    d.bar(&d, 42);
}
```

Так в целом можно, но можно и чуть оптимальнее. У нас наши тройки функций не могут комбинироваться как им вздумается, всегда либо все из base, либо все из derived. И к тому же, нам не хочется с каждой новой функцией увеличивать размер структуры. Поэтому есть такая штука как таблица виртуальных функций. Это мы берём наши 3 указателя и выносим их в особый объект, указатель на который помещается в нашу структуру. А таких структур создать под каждый класс глобальными переменными. Это даёт нам ещё один indirection, но сокращает размер структур. И именно так это и работает во всех компиляторах. Во множественном наследовании у класса просто появляются две таблицы, под каждый базовый класс своя.

RTTI. typeid. И как же работает dynamic_cast? И почему требует полиморфных типов? А потому что в таблицах виртуальных функций может хранится что-то другое, не только указатели на функции. В частности, в них хранится такая штука как RTTI — runtime type information. Это какая-то информация, которую компилятор вставляет в таблицу, чтобы понимать динамический тип. И к ней даже можно получить. Для этого есть такая штука как ключевое слово typeid, который вам возвращает std::type_info const&, который по сути и является RTTI.
Кстати, в большинстве компиляторов можно выключать RTTI, чтобы не тратить лишнее место. И в каких-то кодовых базах можно увидеть код без dynamic_cast'ов и typeid.

Чисто виртуальные функции. Вот создали вы, скажем, логгер. И отнаследовались из него, создав логгер, который пишет в файл, логгер, который пишет в консоль и т.д. Возникает вопрос — а что должен делать базовый логгер? Ну, непонятно. Ничего путного. Для этого есть механизм чисто виртуальных (абстрактных) функций — пометить, что этой функции не существует:

```
struct logger
{
    virtual void log_message(std::string const& str) = 0;
};
```

Тогда вы не можете её вызвать. И не можете создать класс, содержащий чисто виртуальную функцию. Но есть проблема:

```
struct base
{
    base()
    {
        foo();
    }
    void foo()
    {
        bar();
    }
    virtual void bar() = 0;
};

struct derived : base
{
    void bar() {};
};
```

Тут, когда вы создадите derived, вам на этапе исполнения скажут, что вы вызываете чисто виртуальную функцию. Почему? Если бы вы вызывали derived::bar, то вы бы вызвали его тогда, когда у вас даже поля derived не проинициализированы. Поэтому считается, что когда вы конструируете объект типа derived, когда вы конструируете base, в тот момент ваш динамический тип равен base. То есть когда вызывается конструктор объекта, он не сразу с тем динамическим типом, который у него есть, а изменяется по чуть-чуть, сначала он базовый, а потом нормальный. Когда иерархия больше, динамический тип меняется большее количество раз.
А ещё этот пример не работает без прослойки вида foo. Почему так? А вот смотрите. У вас есть возможность вызвать виртуальную функцию напрямую (без виртуальных функций), например так: base::foo. Но это не единственный пример. Если вы вызываете функцию в конструкторе и деструкторе, то вы точно знаете, что сейчас ваш динамический тип правильный. Поэтому, написав код

```
struct base
{
    base()
    {
        bar();
    }
    virtual void bar() = 0;
};

struct derived : base
{
    void bar() {};
};
```

Вы получите ошибку компиляции, потому что в base::base происходит вызов base::bar, а не просто bar, а значит мы явно вызываем виртуальную функцию. А у функции base::bar нет тела, её нельзя вызвать.

Парочка полезных ключевых слов.

final. final — нельзя наследовать. Либо нельзя наследовать класс, либо больше нельзя больше изменять функцию. Пишется так:

```
struct inderriveable final
{
    // ...
};

struct base
{
    virtual void foo() {}
};

struct derived : base
{
    void foo() final {}
};
```

override. Явно указать, что вы наследуете виртуальную функцию, а не пишете что-то своё. Очень советуется это писать. Если кто-то изменит базовый класс, вы хотите явно видеть, что все функции поломаются. Пишется в том же месте, где и final.

Наследование против union'а. Мы же можем использовать наследование для той же цели, что и union/std::variant — выбирать из альтернативы. В случае с std::variant мы даже можем проверять корректность обращения. Что же лучше?

Наследование	union
Если альтернативы разного размера, то union жрёт много памяти.	Наследование работает по указателю, а это даёт лишнюю индирекцию.
Можно легко добавить новую альтернативу.	Можно легко добавить новую операцию.

protected. Представим, что мы пишем виджет на основе QT. Там есть базовый виджет, у которого есть операции, что делать в случае нажатия мышки, в случае перемещения колёсика и прочее подобное. Вам всё это нужно переопределить, и всё будет хорошо. И в таком случае используется ключевое слово protected. Оно для похожих случаев и было создано, лол. Это модификатор доступа, дающий доступ только дочерним классам и себе. С ним есть вопрос, как идейно с ним жить. Если метод не ломает инвариант, почему он не public, а если ломает, то хотим ли мы давать доступ дочерним классам. Тем не менее, эти вопросы не риторические, если вы нашли на них ответ — делайте protected.

Best practices. Давайте дополним наш пример с виджетами. Вот есть у нас виджет, который знает, как его красить. И это виртуальная функция. Мы наследуется, меняем функцию, всё хорошо. Но есть второй вариант — создать отдельный класс, который отвечает за покраску, наследовать только его, и передавать этот объект в конструктор виджета. Это может быть очень полезно, если мы хотим, например, одинаково красить разные классы в разных местах. Более того, мы можем собирать наш виджет из кусочков. В QT, например, используется оба подхода. Реакция на мышку, на клавиатуру, перекраска и некоторые другие штуки обычно очень сильно связаны с самим классом, а какая-нибудь стилизация — уже что-то внешнее.
Надо лишь понимать, что комбинируя кусочки, можно зайти так далеко, что вы будете складывать 2+2, получая двойки из какого-то data_provider'а, складывая каким-нибудь классом adder и подобное. Не надо плодить фабрики непонятных классов. Когда вы делаете точку настройки, вы делаете ставку на то, что будете менять. Тут надо сильно думать. Более того, если вы сделали какие-то точки настройки, а расширять надо в другую сторону, то ваши точки настройки будут вам во вред, потому что вам надо будет их с новыми согласовывать.
Мораль: когда вы делайте фабрики/точки настройки/всё остальное, думайте, для чего вы это делаете.

Квадрат и прямоугольник. Посмотрите лекцию по Java. Посмотрели? Класс. Теперь посмотрим, как с этим жить. У нас есть квадрат и прямоугольник. Вопрос в том, кого из их откуда наследовать. Ответ на этот вопрос нельзя дать без вопроса, какие операции нам нужны. Например, вот так:

Квадрат	Прямоугольник	Оба
get_side	set_width set_height	get_width get_height set_side

Тогда ни один из классов нельзя наследовать от другого (согласно принципу подстановки Лисков). Но если у нас внезапно пропадают set_*, то у прямоугольника нет специфичных методов, а значит можно отнаследовать квадрат от него. А ещё вопрос в том, зачем вам наследование, правда ли вы хотите полиморфно использовать один как другой?

Виртуальное наследование.

```
struct A
{
    int x;
};
struct B : A {};
struct C : A {};
struct D : B, C {};

int main()
{
    D d;
    d.x = 7; // Не работает, x is ambiguous.
    d.B::x = 7; // Работает.
    d.C::x = 7; // Работает.
}
```

Если две копии x — это то, что вы хотите, то хорошо. А иначе есть виртуальное наследование:

```
struct A
{
    int x;
};
struct B : virtual A {};
struct C : virtual A {};
struct D : B, C {};

int main()
{
    D d;
    d.x = 7; // Работает.
}
```

По сути у вас в D есть только один подобъект типа A, а не два, в отличие от предыдущего случая. Хорошо, а что будет, если то же самое сделать с функцией?

```
struct A
{
    void foo();
};
struct B : A {};
struct C : A {};
struct D : B, C {};
```

Ведь функция-то всего одна, какая ей разница. Но ведь в эту функцию вы передаёте this, и какой this её передавать, зависит от того, какой подобъект A вы хотите передать. Так что с функциями это работает точно также.
А как это сочетается с виртуальными функциями?

```
struct A
{
    virtual void foo() = 0;
};
struct B : A
{
    void foo() override {}
};
struct C : A {};
struct D : B, C {};

int main()
{
    D d;
}
```

Это даже в таком случае не компилируется, потому что D считается абстрактным классом, у него есть абстрактный подкласс C с абстрактным подклассом A. Понятно, что если мы сделаем так:

```
struct A
{
    virtual void foo() = 0;
};
struct B : A
{
    void foo() override {}
};
struct C : A
{
    void foo() override {}
};
struct D : B, C {};

int main()
{
    D d;
}
```

То класс создать мы вполне сможем (но не сможем вызвать foo). А с виртуальным наследованием это взаимодействует так:

```
struct A
{
    virtual void foo() = 0;
};
struct B : virtual A
{
    void foo() override {}
};
struct C : virtual A {};
struct D : B, C {};
```

Это корректно. Потому что у вас есть только одна виртуальная функция foo, и вы её отнаследовали.

```
struct A
{
    virtual void foo() = 0;
};
struct B : virtual A
{
    void foo() override {}
};
struct C : virtual A
{
    void foo() override {}
};
struct D : B, C {};
```

А это некорректно, потому что в классе D непонятно, что за foo. Разве что вы переопределите её и там. И это, кстати, работает вообще с любыми функцими и данными одного имени, не только с виртуальными функциями. То есть вот так тоже можно:

```
struct A
{
    int x;
};
struct B : virtual A
{
    int x;
};
struct C : virtual A {};
struct D : B, C {};
```

Тут у вас действительно две переменных x, но в D по-умолчанию берётся B::x.

Модификаторы доступа наследования. Как и у полей, можно указывать такие штуки как class A : public B. Причём в случае приватного наследования, вы не только поля из B не будете видеть в A, у вас даже static_cast не сработает. То есть наследование с модификаторами доступа — то, кто знает об этом наследовании.

Применения предыдущих двух параграфов. Где это может пригодится? Ну, например, вы имеете какой-то публичный базовый класс (например, widget_painter), и вы создаёте несколько похожих его наследников. А потом видите, что наследники похожи, их можно обобщить, и получить какую-то такую иерархию:

```
struct widget_painter
{
    virtual void paint() {/* ... */}
};

// ...

struct my_base_painter : widget_painter
{
    void paint() override {/* ... */}
};
struct my_painter1 : my_base_painter
{};
struct my_painter2 : my_base_painter
{};
```

Но мы не хотим, чтобы кто-то приводил наши классы my_painter к my_base_painter, это деталь реализации. Поэтому хочется написать struct my_painter1 : private my_base_painter. Но это не сработает, потому что тогда никто снаружи и наследование от widget_painter видеть не будет. Поэтому вот как надо:

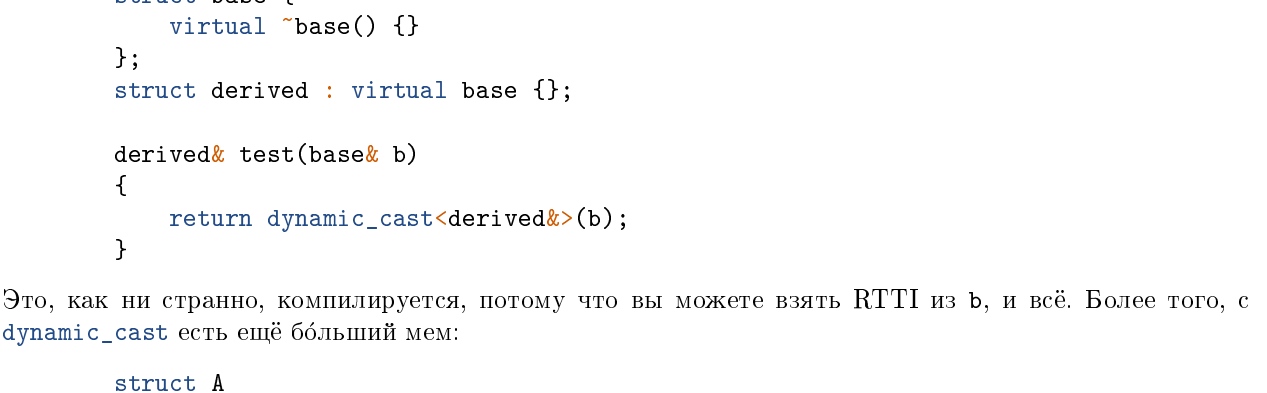
```
struct my_base_painter : virtual widget_painter
{
    void paint() override {/* ... */}
};
struct my_painter1 : private my_base_painter, virtual widget_painter
{};
struct my_painter2 : private my_base_painter, virtual widget_painter
{};
```

Ещё хороший пример — если мы хотим реализовывать интерфейсы по чуть-чуть. У нас есть виртуальная база, которая умеет, там, рендериться, изменяться, что-то ещё делать, и мы её часть в одном классе, часть — в другом. Тогда для сборки этих штук в одну придётся использовать виртуальное наследование:

```
struct game_object
{
    virtual void render() = 0;
    virtual void update() = 0;
};

struct billbord_object : virtual game_object
{
    void render() override {/* ... */};
};

struct static_object : virtual game_object
{
    void update() override {/* ... */};
};

struct static_billboard : billbord_object, static_object
{};
```

Мораль: не надо бояться и пренебрегать виртуальным наследованием.

Виртуальное наследование from the inside. Во что бы мы оттранслировали виртуальное наследование? Давайте вместо того, чтобы внутри объекта B хранить объект A, хранить указатель на него. Но если у нас много виртуальных баз, то хочется табличку. А табличку нельзя, у всех объектов типа D они свои. Давайте хранить не указатель, а смещение до нужной виртуальной базы. И тогда в каждом объекте всё одинаковое, а значит можно объединить в табличку — табличку виртуальных баз.



А теперь пример:

```
struct base {};
struct derived : virtual base {};

derived& test(base& b)
{
    return static_cast<derived&>(b);
}
```

Так вот это не компилируется, потому что мы совершенно не шарим, откуда в объекте b взять смещение, которое хранится в derived. Но зато можно так:

```
struct base {
    virtual ~base() {}
};
struct derived : virtual base {};

derived& test(base& b)
{
    return dynamic_cast<derived&>(b);
}
```

Это, как ни странно, компилируется, потому что вы можете взять RTTI из b, и всё. Более того, с dynamic_cast есть ещё бо́льший мем:

```
struct A
{
    virtual void foo() {}
};
struct B {};
B& test(A& a)
{
    return dynamic_cast<B&>(a);
}
```

Как ни странно, это компилируется, более того, даже может быть корректно, если вы создадите класс C и отнаследуете его и от A, и от B.

Наследование и исключения. На самом деле вы можете кинуть наследуемый тип, а поймать базовый (по ссылке, разумеется). Даже если в них нет RTTI. Как это работает — хз. Нужно понимать, что catch работает тогда, когда можно однозначно определить подобъект. В частности такой код:

```
struct A {};
struct B : A {};
struct C : A {};
struct D : B, C {};

int main()
{
    try
    {
        throw D();
    }
    catch (A const&)
    {}
}
```

не поймает исключение, пока B и C отнаследованы не виртуально.

Инструкция не читер. Делайте это на своей машине тестируйте

```
import Data
    ...
    = ... Void #method ()
    ...
import Decimal : Data
    ...
    = ... Void #method ()
    ...
|| #main()
    ...
    #method()
    #method()
```

Если у вас включена #method() ошибка: что не найдет это решение. А если все хотите, чтобы у вас тогда было 3 одинаковых решения как:

```
import Data
    ...
    = ... Void #method ()
    ...
import Decimal : Data
    ...
    = ... Void #method ()
    ...
|| #main()
    ...
    #method()
    #method()
```

То вы также работаете только у вас множественное наследование:

```
import Data
    ...
    = ... Void #method ()
    ...
import Data2
    ...
    = ... Void #method2 ()
    ...
import Decimal : Data, Data2
    ...
|| #main()
    ...
    #method
    #method2
```

Если вы хотите не наследовать, попробуйте сделать как четыре.
Пример четыре можно применить в интерпретаторе, чтобы составные были или продолжались по мере ввода или нет, в зависимости от ситуации.

Инструкция в XXX. Я надеюсь отделили только вводит новые это от того как вы работаете.