Материалы к программным проектам

Дима Трушин

Содержание

Аннотация

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

1 Полезные ссылки

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

2 Код на плюсах

2.1 Модель компиляции C++

В C++ используется так называемая двух ступенчатая модель компиляции (реальных этапов компиляции там дофига, про все стадии можно почитать тут):

1. Этап компиляции в бинарные файлы.
2. Этап линковки бинарных файлов в исполняемый файл программы.

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

```
// function.h

void function();

// function.cpp

#include "function.h"

void function() {
}

// main.cpp

#include "function.h"

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

Теперь надо провести компиляцию и линковку кода командами[1]

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

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

```
// TU for function.cpp

void function();

void function() {
}
```

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

```
// function.bin

function:
  return;
```

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

```
// TU for main.cpp

void function();

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

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

Теперь эта единица трансляции компилируется в машинный код.

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

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

Теперь у нас есть два бинарных файла

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

И нам нужно вызвать линковку командой

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

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

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

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

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

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

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

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

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

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

2.2 Шаблоны и модель компиляции плюсов

А теперь добро пожаловать в мир интересных интересностей – шаблоны.

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

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

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

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

```
1  template<class T>
2  void f(T) {}
3
```

```
4  template<class T>
5  int x;
6
7  template<class T>
8  class A {};
```

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

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

```
1  // file.h
2
3  template<class T>
4  T f(T t) {
5    return t;
6  }
```

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

```
1  // file.h
2
3  template<class T>
4  T f(T t) {
5    return t;
6  }
7
8  // main.cpp
9
10 #include "file.h"
11
12 int main() {
13   int x = f(2);
14   return 0;
15 }
```

Если мы теперь запустим компиляцию compile `main.cpp` to `main.bin`, то произойдет следующее. Компилятор дойдет до строчки 13 и увидит, что используется имя `f`, которое является шаблоном функции. По-хорошему, мы должны были указать `f<int>(2)`, чтобы сказать с каким параметром мы строим функцию. Однако, компилятор для функции умеет определять тип параметров. Здесь передается константа 2, которая по умолчанию имеет тип `int`, а значит компилятор определит тип `T = int`. В этот момент, когда компилятор понял, что ему надо использовать `f<int>`, он налету создаст код для этой функции поместит его в бинарник:[4]

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

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

```
1  // file.h
2
3  template<class T>
4  T f(T t);
5
6  // file.cpp
7
```

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

```cpp
#include "file.h"

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

// main.cpp

#include "file.h"

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

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

```cpp
// TU for file.cpp

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

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

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

```cpp
// TU for main.cpp

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

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

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

```cpp
// main.bin

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

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

```cpp
// file.bin

// main.bin

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

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

Шаблоны и время компиляции О да, шаблоны любят поднасрать в длительность компиляции вашего проекта. Ведь вы же каждый раз как встретили шаблон, должны сгенерировать для него код. Можно надеяться, что компиляторы будут кэшировать информацию о встреченных шаблонах, но по-хорошему, компилятор видит только одну единицу трансляции за раз. Потому без дополнительных костылей разные единицы трансляции должны заново генерировать код для всех встреченных шаблонов. «Это не дело!» решили программисты из Microsoft и придумали precompiled headers. Идея их вот в чем. Создается единый файл *.pch, в котором компилятор хранит какое-то внутреннее представление для шаблонов для их быстрой генерации. Получается такой не маленький файл. Обычно туда отправляют файлы из STL или других внешних библиотек, которые вы не будете менять ибо при любых изменениях pch файл надо перекомпилировать, а это долго. Но если вы его один раз создали, то можете потом быстро генерировать шаблоны, которые в нем учтены. Более того, таких pch файлов можно делать несколько и подключать к сборке только нужные из них в нужный момент. Вот такая вот технология. Я знаю, что gcc тоже умеет делать что-то подобное, не удивлюсь, что и clang умеет, но врать не буду.

2.3 Правила организации кода и пространства имен

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

```
// Project.h

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

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

// Project.cpp

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

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

// main.cpp

#include "Project.h"

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

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

2.4 Требования к оформлению кода

Выберете для себя какой-нибудь style-guide. Например, есть хорошие готовые google или LLVM. Можете создать свой на основе одного из них.

Чтобы не мучить ни себя ни других рекомендую прикрутить проверку стиля и автоматический формат кода. Есть несколько разных программ для этого. Из элементарных в настройке и установки – AStyle, но качество его работы – так себе. С шаблонами и лямбдами он справляется очень не очень. По-хорошему, стандарт

де факто сейчас clang format.[5] Настройте автоматическое форматирование вашего кода при сохранении файла. Да, вам придется помучиться с тем, чтобы пройтись по всем настройкам форматирования clang format. Но это надо будет сделать один раз. Зато потом вы не будете мучиться с неконсистентностью кода. Хочу отметить, что LLVM входит в последние сборки Visual Studio 2019, потому можно ненапрягаясь пользоваться clang format из под нее. Но и установить LLVM не должно составить труда.

2.5 Мои рекомендации по оформлению кода

2.5.1 Оформление классов

Когда оформляете класс, запомните, в ООП самое важное – это открытый интерфейс класса, потому что именно этим будет пользоваться тот, кто будет пользоваться вашим классом, а имплементация – это самое неважное. Вы можете сказать: «Как же, не важно, я сейчас вам наговнокодю сортировку за n^2 вместо $n \log n$, будет вам не важно.» Обратите внимание на то, что вы не правильно понимаете слово интерфейс. Интерфейс – это не только имя функций, доступных из класса, но и контракт – сложность исполнения функции. К сожалению этот контракт нельзя закодить в плюсах, но тем не менее, сложность – это НЕ деталь имплементации – это часть вашего интерфейса. И если вы меняете имплементацию, в которой меняется сложность, вы не меняете интерфейс с точки зрения языка, но меняете логический интерфейс. И самое плохое в этом, что вы не можете переложить на язык помощь с проверкой сложности исполнения.[6] Потому рекомендуемый порядок при определении класса следующий:

```
1  class A {
2    using Type0 = double;
3  public:
4    using Type1 = int;
5
6    A();
7    A(Type1);
8    A(const A&);
9    A(A&&) noexcept;
10   A& operator=(const A&);
11   A& operator=(A&&) noexcept;
12   ~A();
13
14   void make_something() const;
15
16   static  void do_something();
17
18   static constexpr const Type1 default_value = 0;
19
20 protected:
21   void do_protected_stuff();
22 private:
23   void do_private_stuff();
24
25   static Type0 private_static_data_;
26
27   Type1 private_data_ = default_value;
28 };
```

Давайте я прокомментирую пример выше. В начале у вас идут псевдонимы для типов закрытые и открытые. Закрытые можно определять ниже, там где они нужны.[7] После псевдонимов идут генерируемые автоматом методы: дефолтный конструктор, конструктор, копи и мув конструкторы, копи и мув присваиватели, деструктор. Опять же, эти операторы нужно писать только если они необходимы. Правила о том, как именно их надо определять можно глянуть тут и тут, но основное правило такое, если компилятор сгенерирует по умолчанию то, что надо, то определять самому руками НЕ надо. Далее идут публичные методы, публичные статичные методы, публичные статичные переменные (обычно константы, ибо менять переменные рекомендуется только через методы, тогда вы сможете отслеживать обращения, добавлять в метод счетчик, чтобы проверять количество обращений, валить обращение при специальных условиях для дебага и так далее).

[5] Вот тут можно глянуть настройки опций форматирования.

[6] Где-то я видел про подобную фичу, вот только не помню для плюсов или для D. Но это все вилами по воде писано.

[7] Старайтесь не пользоваться голыми типами, а использовать псевдонима. Они лучше говорят про код и в случае любых изменений, не надо будет ползать по коду и исправлять миллион мест своего и чужого еще не написанного кода.

Потом идет защищенная зона методы, статические методы и данные. Хотя данные тоже не рекомендуется открывать детям напрямую. Потом идут приватные методы, приватные статические методы, приватные статические данные и приватные данные.

2.5.2 Названия

1. Названия функций должны начинаться с глагола. Хорошие функции содержат один и только один глагол в названии (существуют исключения, когда вам надо писать методы для мьютексов, но поверьте, это с вероятностью 99% не ваш случай).

2. Названия переменных должны быть существительными или словосочетаниями означающими объект, предмет (эквивалент существительного). Название должно отражать назначение и намерение в использовании.

3. Названия локальных переменных класса приватных и защищенных рекомендуется снабжать дополнительным обозначением. Есть несколько вариантов `data_` или `m_data`. Это важно в плюсах вот по какой причине. Когда вы вызываете метод класса, то переменная класса является для вашей функции глобальным данном в рамках данной инстанциации объекта. А глобальные данные – это чревато. Потому вам нужен механизм, чтобы подчеркнуть, что данная переменная не является локальной для метода, а является переменной класса. Статические переменные тоже хорошо бы выделять префиксом например `s_data`.

4. Эти указания не касаются публичных переменных. Например в структуре

```
struct Pair {
  int first;
  int second;
};
```

   Нет никакого смысла навешивать дополнительные суффиксы, вы и так понимаете, что у вас просто набор данных, к которым вы хотите достучаться по именам, например `my_pair.first`. У вас нет методов у этого класса.

5. Другое исключение – публичные константные статические данные класса. Сразу скажу, если у вас класс, то только самоубийцы делают публичные данные. Потому у вас публичными могут быть только статические данные. А раз статические данные доступны всем, то только самоубийцы делают их не константными. А если у вас есть константные статические данные, то они должны быть `constexpr`. А тогда к ним удобнее обращаться по красивым именам без всяких префиксов и суффиксов.

6. Я предпочитаю называть классы с большой буквы верблюжьем стилем.

2.5.3 Шаблоны

Так как шаблоны надо размещать целиком в header-е, то я бы рекомендовал писать их определения inline, а не разделять на объявление и определение. Чтобы не быть голословным, давайте сравним следующие два кода. Первый кусок.

```
template<class T1>
struct A {
  using ReturnType = int;
  template<class T2>
  ReturnType function(T2);
};

template<class T1>
template<class T2>
A<T1>::ReturnType A<T1>::function(T2 t) {
  return ReturnType();
}
```

И второй кусок.

```
template<class T1>
struct A {
  using ReturnType = int;
  template<class T2>
  ReturnType function(T2 t) {
    return ReturnType();
  }
};
```

Я все понимаю, что там чистый интерфейс, разделять интерфейс и имплементацию. Но вы представьте на сколько много лишнего кода надо писать для вложенных шаблонов, особенно если у них зависимые типы у возвращаемых значений. У вас отношение кода к шуму будет стремиться к нулю. Кроме того, надо еще догадаться куда там надо пихать все эти ключевые слова с шаблонами, как делать специализации шаблонов и прочую дрянь.

2.6 Политика обработки ошибок

Существует два вида ошибок:

1. **Ошибка программиста.** Проще говоря, я сам дурак, что написал такой мусор вместо программы. Надо исправлять.

2. **Ошибка при обращении к ресурсам среды исполнения.** Проще говоря, я вызвал функцию операционной системы, а операционка не смогла. Ну что я тут могу поделать? Ничего.

Так вот эти ошибки надо обрабатывать по-разному! Для первых придуман `assert`, для вторых – `exception`. В частности это означает, что НЕ надо проверять свои ошибки исключениями, ваши ошибки – это косяки, которые все равно надо исправить, а не залатывать костылями из исключений.

Ошибки программиста Что такое `assert`? Это функция, которая в дебаг режиме проверяет условие, и если оно не выполнено, то валит программу с указанием того, в какой строчке кода какое условие было нарушено, а в релизе[8] не генерирует никакого кода. Так как вы все равно тестите основное время программу в дебаге, то как только встретилась ошибка (допущенная по вашей вине), компилятор вам об этом красочно скажет, а если вы компилите в релизе, то вы не теряете в производительности, никакого лишнего кода `assert` не сгенерирует.

Как этим всем барахлом пользоваться. Обычно, у вас в коде есть некоторые предположения, например, указатель не нулевой, аргументы функции положительны, выполнен такой-то инвариант класса при конструировании и исполнении и т.д. Так вот, все это – ответственность программиста, а значит, надо проверять `assert`. Например

```
void f(int* x) {
  assert(x && "argument must not be nullptr!");
  *x = 42;
}
```

По-хорошему, вы должны были написать `assert(x != nullptr)`. Во-первых, так было бы читаемее, во-вторых, у `assert` нет способа оставить сообщение об ошибке, потому тут приходится пользоваться костылями и добавить адрес статической строки (который не ноль) в виде дополнительного условия через `&&` оператор.

Мои рекомендации по по использованию `assert` кратко выражаются так: чем больше, тем лучше. Чем ближе к багу программа упадет на ошибке, тем быстрее вы его найдете и поправите.

1. Ставьте `assert` в начале каждой функции, чтобы проверить preconditions.

2. Если не уверены в алгоритме, ставьте `assert` в конце каждой мутирующей функции.

3. Все конструкторы класса должны проверять не только preconditions для аргументов, но и инварианты класса. Выделите в классе приватные функции по проверке инвариантов и пихайте их в методы класса.

[8] Это не совсем правда, она не генерирует никакого кода, если определен макрос `NDEBUG`. Компилятор от Microsoft автоматически определяет этот макрос в релизе, а gcc и clang не торопятся этого делать. Бывает полезно проверить условия в релизе, потому что пути оптимизации компилятора неисповедимы. Тогда в msvc вам надо убрать при компиляции в релизе объявление макроса `NDEBUG`.

4. Не пишите слишком длинные и сложные выражения внутри `assert`, вы иначе никогда не поймете, что вы проверяете. Выделите отдельные вспомогательные функции с говорящими названиями.

5. Обязательно вставляйте `assert` перед обращением через указатель, он может быть нулевой. Другие методы могут содержать `assert` внутри себя, но если `operator->()` не перегружен (а для указателей он у вас не перегружен), то вы не можете вставить `assert` внутрь, потому проверки надо делать до вызова функции.

6. НЕ делайте `assert` на ошибки, которые разрешимы на этапе компиляции (например в шаблонной магии), для этого используйте `static_assert`.

7. И конечно же не забывайте сообщить об ошибке текстом. Вам же будет проще.

Ошибки исполнения среды Теперь поговорим о том, когда же нужны исключения. Причина такой ошибки – программа хотела, система не смогла. Например, вы пытаетесь запросить у системы ресурс, а она вам его не дает. Обычно приводят такой пример: вы пытаетесь запросить память под вектор и система отказывает, потому что нет памяти. Это не лучший пример вот почему

1. Если вы запросили память и ее больше в системе нет, то что вы собираетесь делать? Скорее всего у вашей системы проблемы покруче, чем невозможность продолжить работать вашей программе. Смысла тут делать что-то нет.

2. Память запрашивает чуть ли не каждый кусок кода в любой библиотеке. Проверять на исключение каждое обращение к операционной системе? Вам что, делать нечего? В таком случае программа только и будет делать, что проверять исключения и ничего больше.

Давайте я постараюсь привести другой, более удачный пример обработки исключения. Скажем, вы хотите работать в программе с клавиатурой через какой-нибудь интерфейс операционной системы. Делаете запрос, на доступ к этому интерфейсу, а операционная система говорит, что нельзя и не дает доступ. То есть у системной функции, которую вы зовете есть возможность сфейлиться и вы не можете даже начать выполнение программы в таком случае. Вот такую ситуацию как раз надо ловить исключением. И политика тут такая, если все получилось, то хорошо, если не удалось получить доступ, то сообщаем пользователю, что операционка нас не любит и умираем.

Не злоупотреблять исключениями Тут я бы хотел сформулировать несколько соображений о том, как я вижу обработку исключений.

1. По-хорошему у вас должна быть одна точка по обработке исключений (или несколько, но не много). Чаще всего – это event loop в вашей программе, где вы можете сообщить об ошибке и умереть.

2. Исключений никогда не должно быть в критической для производительности части. Более того, рекомендуется помечать код `noexcept` (даже если он кидает исключения, например на запрос памяти у операционки), чтобы компилятору было дозволено больше оптимизаций.

3. Исключения не должны ни в каком виде контролировать workflow вашего кода. За это должна отвечать логика программы.

Идеология RAII «Resource Acquisition is Initialization» – это волшебная фраза, которая говорит о том, как заворачивать ресурсы в обертки в системе с поддержкой исключений. А нафига вообще что-то надо? Смотрите, основная проблема исключений следующая: внутри любой функции исключение может возникнуть в любой строчке кода. То есть если вы выполнили одну строчку кода, то у вас нет гарантии, что выполнится следующая. Например

```
1  void g() {
2    throw std::runtime_error("Ups!");
3  }
4
5  void f() {
6    int* x = new(0);
7    g();
8    delete x;
9  }
```

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

1. В строчке 6 происходит сразу несколько вещей.
   (a) Выделение памяти на стеке для указателя `x`.
   (b) Вызов оператора `new` для выделения памяти под переменную типа `int`, помещаем туда значение `0`.
   (c) Пишем адрес памяти в переменную `x`.
2. Вызываем функцию `g`.
3. Функция `g` бросает исключение. В этом месте мы вываливаемся из функции `g` в строчку 7 из функции `f`.
4. В строчке 7 после исключения мы должны выйти из функции `f` и будем так выходить до тех пор, пока не встретим `catch`, который поймает исключение. Но чтобы выйти мы должны почистить стек от функции `f`. Это называется stack unwinding. Надо удалить все локальные переменные и аргументы функции `f`. Для этого после строчки 7 вызывается деструктор `x`, а так как это встроенный тип, то никакого деструктора нет и мы просто уменьшаем стек и выкидываем переменную `x` со стека.
5. Почистив стек мы выходим в больший scope из которого вызвана `f` и продолжаем вываливаться дальше.

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

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

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

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

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

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

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

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

```
1  struct IntPtrPair {
2    IntPtrPair(int x, int y) : ptr1_(x), ptr2_(y) {}
3
4    IntPtr ptr1_;
5    IntPtr ptr2_;
6  };
7
8  IntPtrPair x(1, 2);
```

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

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

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

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

1. Я везде пользуюсь wrapper-ами для указателей и мне вообще не надо задумываться про деструкторы.
2. Обратите внимание, что нигде не пользуюсь `new` явно, вместо этого пользуюсь `std::make_unique` функцией.
3. Если честно, то `std::unique_ptr` не самая высоко оптимизированная вещь.[9] Но для большинства ситуаций этой обертки хватает на ура. Правда деревья я бы на ее основе не собирал, как минимум убьете стэк в деструкторе или получите неописуемые тормоза.

Аналогично тому, что я написал про память используется для выделения других ресурсов. Например для мьютексов используются `std::lock_guard` или похожие механизмы. Любой запрос к операционной системе, который требует освобождения своих ресурсов, должен выполняться через wrapper, где в конструкторе вы запрашиваете ресурс, а в деструкторе освобождаете. Вот тут я могу привести пример того, как я оборачивал Windows нативный thread pool в exception safe обертку. Это уже не искусственный пример, где много разных взаимозависимых ресурсов требуют своего последовательного выделения или освобождения. Наследование позволяет это сделать удобным и контролируемым. Но надо пописать какой-то код. Обратите внимание, что в имплементации методов я даже позволяю себе бросать исключения в конструкторах, при наличии ошибки. Тогда все выделенные ресурсы в базовой части автоматом почистятся и не надо переживать на эту тему.

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

Напоследок хочу сказать еще одно замечание про реализацию выделения памяти под зависимые данные. Есть другая альтернатива

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

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

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

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

2.7 Наследование

Наследование в языках достаточно забавная штука. Одна из основных особенностей наследования в том, что есть разные виды наследования и они служат в качестве разных инструментов для решения разных задач. Но очень часто про наследование говорят не различая эти виды, смешивая все в одну кучу, и потому нельзя в точности понять, чем и как надо пользоваться. Я бы для начала выделил два «чистых» вида наследования

1. Наследование без виртуальных функций. Этот вид наследования является «горизонтальной» композицией, а НЕ абстракцией, и является основой для policy based design.

2. Наследование с чисто виртуальными функциями или интерфейсы. Этот вид наследования используется для достижения полиморфизма. Однако, надо понимать, что с появлением variadic templates у нас есть стирающие типы, которые являются более адекватной заменой этому виду наследования.

Наследование без виртуальных функций Давайте рассмотрим следующие две ситуации:

```
1  struct A {
2    void f() {}
3  };
4
5  struct B {
6    void g() {}
7    A a;
8  };
9
```

```
struct C : A {
  void g() {}
};

int main() {
  B b;
  b.g();
  b.a.f();

  C c;
  c.g();
  c.f();
  return 0;
}
```

Давайте сравним объекты `B` и `C`. С точки зрения пользователя в случае `B` интерфейс объекта имеет иерархию, часть методов доступна напрямую, а часть сгруппирована внутри подъобъекта `a`. С другой стороны, в случае объекта `C` интерфейсы `A` и `C` слиты воедино «горизонтально» и не имеют иерархии. В обоих случаях методы `A` видят только методы `A`, а методы `B` (или `C`) видят как свои методы, так и методы `A`. В этом смысле они не отличаются. На самом деле оба метода посвящены объединению объектов в одно целое с наличием или отсутствием иерархии. И оба этих метода являются композицией. В случае `B` это «вертикальная» композиция, где вы вводите иерархию и выделяете интерфейс подобъекта отдельно. В случае `C` это «горизонтальная» композиция, где вы сливаете два интерфейс подобъекта с интерфейсом класса.

Идейные отличия двух видов композиции мы обсудили. Давайте перейдем к техническим отличиям. Вы можете контролировать видимость интерфейса объекта `A` в обоих случаях с помощью ключевых слов `public`, `protected` и `private`. И тут появляется еще одно отличие. Если больший объект хочет получить доступ к `protected` методам объекта `A`, то тут есть только один способ – наследование. При этом если вы хотите закрыть интерфейс `A`, то вы можете это сделать приватным или защищенным наследованием. А открыть часть методов можно с помощью `using` директивы:

```
class A {
public:
  void f() {}
  void g() {}
protected:
  void h();
};

class B : private A {
public:
  using A::f;
  void h1() {
    h();
  }
};
```

В этом случае вы открыли только интерфейс `B` и закрыли интерфейс `A` от всех даже наследников. При этом директива `using` открывает функцию из класса `A`.

Важно отметить, что наследование без виртуальных функций относится к миру value semantics и safe by default. Такой подход бывает полезен, когда вы хотите собрать объект с разными настройками. Например, предположим, вы написали класс дерева с корнем и хотите сделать для него итератор. Какие свойства итератора можно контролировать? Например: константный он или нет, в каком порядке он обходит дерево, что он возвращает при разыменовании – данные в вершине или саму вершину и т.д. Тогда этот объект можно собрать из разных кусочков (код ниже весьма условный, тут надо навесить кучу шаблонов на самом деле, но я хочу продемонстрировать идею):

```
class ConstData {...};
class NonconstData {...};

class NodeDereference {...};
class DataDereference {...};

class Preorder {...};
class PostOrder {...};

ConstPreorderIterator = Iterator<ConstData, NodeDereference, Preorder>;
```

```
11  PreorderIterator = Iterator<NonconstData, NodeDereference, Preorder>;
```

Этот подход собирает объекты как из блоков лего. Он называется policy based design и очень яро продвигался Andrei Alexandrescu в его книге Modern C++ Design. Очень рекомендую почитать ее.

Интерфейсы Это совсем другой вид наследования, его задача была в том, чтобы достигнуть полиморфное поведение. Давайте вкратце скажу, что это. Язык C++ является статически типизированным, то есть у вас типы объектов определены заранее и вы не можете хранить в переменной одного типа переменную другого типа. Это один из механизмов безопасности языка. Но он дает и ограничения. Предположим у нас есть два вида объектов с одинаковым интерфейсом и мы бы хотели хранить их в одном векторе, например так:

```
1  struct A {
2    void f() const {}
3  };
4
5  struct B {
6    void f() const {}
7  };
8
9  std::vector<?> data;
10 data.push_back(A());
11 data.push_back(B());
12 for (const auto& x : data)
13   x.f();
```

И весь вопрос в том, что нужно подставить вместо `?` чтобы этот код заработал. Забегая вперед скажу, что правильное решение – стирающие типы. Грубо говоря, стирающий тип, это такой объект, который в данном случае умеет в себе сохранять любой объект, который в интерфейсе имеет метод `f`, а так же умеет вызывать этот метод `f`, когда объект в нем хранится. Классическим примером стирающих типов является `std::function`, еще есть типы `std::any`, `std::variant`.

Однако, стирающие типы в языке C++ стали доступны лишь с появлением variadic templates начиная с C++11. А до этого момента нужно было придумывать какой-то другой механизм. И этот механизм был с помощью виртуальных функций. Вот как бы этот пример выглядел 20 лет назад.[10]

```
1  struct I {
2    virtual ~I() = default;
3    virtual void f() const = 0;
4  };
5
6  struct A : I {
7    void f() const override {}
8  };
9
10 struct B : I {
11   void f() const override {}
12 };
13
14 std::vector<I*> data;
15 data.push_back(new A());
16 data.push_back(new B());
17 for (auto p : data)
18   p->f();
```

То есть мы создаем единый интерфейс, от него наследуем все классы, которые мы хотим поддерживать и после этого создаем объекты на куче, а храним вместо указателей на сами объекты, указатели на родительский класс. И через механизм виртуальных функций, у нас вызывается метод нужного класса. Основная проблема этого подхода в том, что он меняет value semantics на reference semantics и вам надо самому беспокоиться о менеджменте памяти. Это плохая тема.

Собственно стирающие типы были придуманы, как замена этому механизму. И стирающие типы внутри себя используют ровно этот же самый механизм. Только благодаря шаблонам можно налету создавать любые наследники и подключать их в момент, когда пользователю кода они становятся необходимы. На эту тему опять же стоит обратиться к докладу Sean Parent на тему run-time polymorphism.

[10] Не удивляйтесь даже если вы из будущего и видите такой код прямо сейчас. Я никогда не верил в человечество.

К сожалению, в стандартной библиотеке C++ сейчас нет какого-то внятного настраиваемого шаблона со стирающим типом. Вам придется под каждый интерфейс писать свой стирающий тип (спасибо тебе Sean Parent, что ты рассказал, как это делать), либо писать свою библиотеку со стирающим типом. Вот тут можно посмотреть мою вариацию на тему стирающих типов. Этот типо поддерживает только move семантику и позволяет настроить любой интерфейс, который вы хотите.

Когда и как пользоваться наследованием Если кратко, то наследование без виртуальных функций – это хорошо и применяется часто, а наследование с виртуальными функциями – это опасно и применяется очень редко только для написания библиотечного кода, про который вы потом можете доказать, что снаружи ваша имплементация безопасна. Давайте пройдемся по случаям когда стоит использовать наследование.

- Самая популярная тема – policy based design. Я уже приводил пример с итератором к дереву, а конкретный код можно глянуть тут. Опять же, это написание библиотечного настраиваемого кода.
- Наследование может помочь с организацией RAII в сложных случаях, когда надо инициализировать ресурсы в определенном порядке и они зависят друг от друга и на любом этапе может что-то сломаться. В этом случае вы обычно не хотите делать иерархию на ресурсах, так как они все одно уровня, но зависимы. Примером может являться доступ к thread pool-у в Windows. Соответствующий пример можно посмотреть тут. (Он не написан не лучшим образом, но тем не менее дает понять, что имеется в виду). Другой пример – доступ к видеокарте, где надо последовательно инициализировать ресурсы. Соответствующий пример можно глянуть тут.
- Наследование может помочь организовать компоненты приложения. Например, если вы разрабатываете интерактивное приложение, то у вас есть три компоненты: model, view, controller. При этом вы хотите гарантировать, что model не видит никого, view тоже не видит никого, а вот controller видит и model и view. Тогда можено сделать класс `AppKernel` где живут все models, класс `AppGUI` где живут все views, и унаследовать от них обоих класс `AppImpl`. Тогда в ядре будут жить все модели, в GUI части будут жить все view-хи, а контроллеры будут жить в `AppImpl` и в этом же месте вы соединяете все компоненты вместе (например через observer pattern или просто мастером). Пример можно глянуть тут.
- Если вы используете виртуальные функции, то это признак того, что вы имплементируете какую-то библиотечную парадигму, структуру или шаблон. В этом случае вы должны позаботиться о том, чтобы снаружи ваш код имел value semantics и соответствовал всем уровням безопасности, которые вы требуете от кода. Например, если вы имплементируете свои стирающие типы.

Что касается того, когда не надо использовать наследование.

- Если вы используете виртуальные функции и не пишите библиотечный код – вы в большой беде. Так делать не надо. Если вы мешаете чисто виртуальные функции с данными, то скорее всего вы вообще не понимаете, что вы делаете. Так что тут все просто, виртуальные функции – главный указатель на проблему. Надо помнить, что наследование без виртуальных функций – это конструкция из мира value semantics, а интерфейсы – это конструкция из мира reference semantics. Потому, если вы мешаете эти две парадигмы, вы преобретаете все проблемы и недостатки обоих миров без плюсов этих самых миров. Так иногда приходится делать внутри сложных имплементаций библиотечных типов, но это очень редкие ситуации и в этих случаях вы можете доказать, что все безопасно снаружи.

2.8 Производительность

2.8.1 Общие соображения

Производительность – это хорошо, но корректно работающий код – еще лучше. Очень легко сделать программу быстрой и не верной. Корректность мы все таки ценим больше скорости.[11] Потому никогда не надо заранее оптимизировать, особенно если вы до конца не понимаете структуру вашей программы.

Я бы глобально думал про код так. У вас есть глобальная архитектура. На этом уровне у вас все медленное (условно). Тут оптимизации должны сводиться к правильной архитектуре, правильным компонентам, правильному взаимодействию между ними, организация потока данных и т.д. На этом уровне вы вряд ли почувствуете огромную пользу от оптимизаций и говнокода ради нее, но очень будете страдать от нечитаемости и неподдерживаемости. Я бы хотел тут добавить ссылку на доклад по SOLID principles.

[11] Слова, кстати, не мои, это Scott Meyers сказал. Хотя, я думаю, что более или менее все другие серьезные люди повторяют эту фразу на разные лады.

В рамках большой архитектуры у вас всегда будут критические куски, где производительность важна. Как понять, какой кусок критичный? Ответ просто – протестируйте! Если вы не сделали ни одного теста на производительность, то нет смысла и оптимизировать. Есть ситуации, когда очевидно, что какое-то изменение будет полезным, но их все таки не много. В сложной системе найти бутылочное горшылко – та еще задача.

2.8.2 Узкое место

Вот вы нашли бутылочное горлышко и пора оптимизировать. Я бы выделил два вида кода, где это понадобится:

1. Хитрые алгоритмы с данными, где важна асимптотика, но не критично время обращения к памяти.
2. Математические вычисления с большими объемами данных. Тут важна не только асимптотика, но и время обращения к памяти.

В большинстве случаев вам можно не париться о времени доступа к памяти. Вы просто пользуетесь `std::map` или `std::list` и радуетесь. Часто достаточно иметь хорошую асимптотику по количеству операций и все будет работать в допустимом диапазоне. И тут рекомендации такие. Если вы, продумывая данный модуль, понимаете как решать проблему, какие там бывают асимптотики, какие есть решения и знаете, как можно быстро собрать оптимальное решение, то делайте. Если же вы сомневаетесь или не знаете как реализовать наилучшее решение, то лучше сделайте в начале как-то, как-нибудь так, как вы знаете и можете. Код будет работающим, читаемым, поддерживаемым, его можно будет тестировать и замерять производительность. Может оказаться, что и не надо больше танцев с бубном. А если и понадобится оптимизировать, то вы точно будете знать, что, где и зачем. И вот тогда надо будет тратить время на изучение нужного алгоритма и всю эту возьню.[12]

Когда вы разбираетесь с математическими вычислениями на больших данных, то у вас возникают проблемы другого сорта – доступ к оперативной памяти. Дело в том, что процессоры сейчас умеют выполнять огромное количество действий благодаря конвеерной системе, SIMD, большому количеству ядер. И главная проблема – обеспечить эти радости данными для вычислений.[13] Проблема не в том, что процессор не может быстро посчитать, проблема в том, что он все время ждет, когда же ему дадут данные. Эта проблема очень часто выстреливает именно в задачах с большим количеством математических вычислений.[14] В таком случае вам нужно беспокоиться не только об ассимптотике, но и о правильной организации данных в памяти. Дело в том, что современные процессоры обращаются в память через систему кэшей. Можно себе это представлять приблизительно так: CPU, cache L1, cache L2, cache L3, Ram. Для понимания картины стоит сказать, что запрос данных из L1 кэша происходит в 40[15] (а то и более) раз быстрее, чем из Ram.

Как происходит чтение данных из памяти. Когда CPU затребовал данные из переменной, которая где-то лежит в памяти, то читается не только сама переменная, но целая кэш линия (обычно 64[16] байта) и помещается в L3 кэш. Потом из этой кэш линии кусок с переменной помещается в L2 кэш, потом из нее отрезается еще кусочек с переменной и помещается в L1 кэш и только после этого данные о переменной помещаются в регистр процессора. Предположим, что вы попытались сложить переменные x и y. Если переменные лежали рядом в памяти, то в результате этой процедуры они обе лежат в L1 кэше и процессор быстро помещает обе переменные в регистры и складывает.[17] Однако, если переменные x и y лежали далеко, то после обращения к x переменная y может не оказаться даже в L3 кэше и придется запрашивать ее из Ram (а это на несколько порядков дольше). Потому важно данные используемые вместе хранить рядом. По этой причине `std::vector` является самой лучшей структурой.

Кроме сказанного выше надо учесть еще и выравнивание. Это дело важно процессору, когда он читает данные. Выровненные данные читаются и пишутся быстрее. Так уж устроено железо. Потому выравнивайте ваши контейнеры данных при хранении. Это особенно пригодится при работе с SIMD.

[12] Не забывайте о таком ресурсе как время. У вас всегда стоит выбор между сделать как-нибудь как вы понимаете сейчас быстро и сделать супер-пупер хорошо неведомым смыслом неведомо в какие сроки. Второй путь плох тем, что вы можете даже не дойти до результата, а если и дойдете, то он может того не стоить. Лучше за два часа написать медленный работающий код, чем 2 месяца потратить на ненужную оптимизацию. А с опытом, решения, которые вам будут приходить в голову, все равно будут разумными по скорости.

[13] Слышали про технологию Hyper Threading? Так вот, оказывается, что при всех наших стараниях, мы все равно загружаем ядра процессора только на половину. Потому на уровне операционной системы на каждое ядро мы видим несколько логических ядер (обычно 2) и операционная система по-очереди подключает логические ядра к физическим. И за счет таких танцев с бубном получается донагрузить простаивающие ядра.

[14] Привет физические симуляции, графические движки, градиентный спуск и прочие радости.

[15] Привет читатель из будущего, я знаю, что у вас эти цифры совсем другие, но нам в прошлом нужны были ориентиры.

[16] Ох уж эти читатели из будущего, да знаю я, что у вас во по-другому.

[17] Тут еще надо запариться по поводу того, куда писать результат, но давайте мы не будем так напрягаться.

- Если же вам нужно воспользоваться деревом или списком, но надо локализовать данные в памяти, то приходится идти одним из двух путей. Первый – flattening, вы переписываете свою структуру на основе вектора. Второй – использование кастомного аллокатора. К счастью сейчас есть библиотека `std::pmr`, которая помогает в более или менее простых ситуациях собрать нужный аллокатор и этого с лихвой хватает. Глянуть стоит сюда и сюда. Кроме того, есть два видео от Jason Turner-a

  1. Вот это кратко с примерами
  2. А вот это продолжение более подробное

- Вот тут можно почитать неплохой текст про кэш и его работу. Так же там есть хорошая ссылка на доклад Scott Meyers-a.

- Если вам нужны SIMD, то я бы не стал пользоваться ассемблером или интринзиками напрямую. Вместо этого есть потрясающая библиотека от Agner Fog vector class library. Сайт Agner-a содержит кучу всяких вещей по оптимизации под x86 архитектуру. Он же автор библии оптимизации под нее.

- Если вам нужна многопоточность, то существуют два подхода: lock free подход и прямой контроль доступа к ресурсам. Для lock free подхода есть более или менее 3 серьезные библиотеки (по качеству они почти не отличаются друг от друга):

  1. Intel TBB тут Доступна на всех популярных операционках: Windows, Linux, Mac.
  2. Microsoft PPL тут Доступна только под Windows и более того, является частью Windows Run-Time Library под Visual Studio. Не уверен, что вырезается оттуда и, что ее можно использовать с другими компиляторами.
  3. Apple LibDispatch тут Доступна на Linux и Mac. Среди всех трех она чуть пошустрее (если верить Sean Parent) но не значительно.

  Очень рекомендую доклад Sean Parent объясняющий, как это все работает.

- Еще есть замечательная библиотека по сбору асинхронных пайплайнов. Вот тут Sean Parent рассказывает о ней. Ссылку на github можно найти по первой ссылке вместе с описаниями и инструкциями.

Существуют и другие решения по гетерогенным вычислениям, по автоматической векторизации кода и куча всего. Я постараюсь дополнять этот раздел по мере необходимости.

2.8.3 Языковые тонкости

Семантика языка В начале я хочу поговорить об обще языковых вещах. В плюсах самое важное – это механизмы построения, копирования и разрушения объектов. Для этого в языке предусмотрены следующие механизмы:

1. Constructor (дефолтный и не дефолтные)
2. Copy constructor
3. Move constructor
4. Copy assignment
5. Move assignment
6. Destructor

У любого класса эти шесть функций (для конструктора речь про дефолтный) автоматически генерируются компилятором, потому что они обязаны быть. Они могут быть удалены, но быть должны. Почему? Потому что сущности языка должны уметь конструироваться, копироваться и разрушаться. Если сущность этого не умеет, то она не может существовать в рамках парадигмы языка.

Плюсы относятся к так называемым value semantics языкам. Что это значит? рассмотрим следующий код

```
1 int x = 0;
2 int y = 1;
3 x = y;
```

Третья строчка означает «возьми значение из y и скопируй его значение в x». Есть еще и другие опции. Следующий вариант был бы «возьми значение из y и перемести его в x». Это тоже парадигма value semantics, потому что мы трактуем переменные как стаканы с данными и она встречается в rust-е. Еще есть третий подход «начни называть именем x объект, который называется именем y». Это уже reference semantics. Он в плюсах встречается, когда определяется ссылка, но java, c#, python используют этот подход для всех переменных. В рамках такого подхода переменная – это имя или бирка и присваивание – это перевешивание бирок.

Причем тут производительность? Большую часть времени ваш код будет строить, копировать или уничтожать объекты. И программа все это будет делать там, где вы этого даже не будете замечать. В любом месте, где есть операторы присваивания, или вызов функции у вас автоматом идет конструирование объектов (создание ссылки например или преобразование типов) или копирование (copy или move это другой разговор). Потому надо понимать, что из этого долго, что быстро и на сколько это долго или быстро, чтобы понимать, надо ли с этим бороться.

Локальная часть объекта Давайте рассмотрим следующий код.

```
struct A{
  A(long long x, int y) : x_(x), y_(new int(y)){
  }
  ~A() {
    delete y_;
  }
  long long x_;
  int* y_;
};
```

Если мы на 64 битной машине, то размер `long long` – 8 байт и размер адреса `int*` тоже 8 байт.[18] Структура `A` будет уложена в памяти и займет место в 16 байт. Эта часть объекта называется локальной. То есть та часть, которая лежит непосредственно по адресу объекта. С другой стороны, мы видим, что при конструировании и разрушении у нас еще выделяется память по указателю `y_`. Сам адрес хранимый в `y_` лежит в локальной части, но данные по указателю уже к локальной части не относятся.

То есть надо понимать, что данные в локальной части – это данные, которые лежат по адресу объекта (это может быть как стек, так и куча, а может быть и статическая область). А не локальные данные обязательно выделяются где-то еще. Это где-то еще может быть разным. За это самое где-то еще отвечают аллокаторы. И по дефолту самое простое – это обращение к `new` и выделение памяти на куче. Но можно выделять память более умной стратегией. Обратите внимание, что сейчас я имею в виду конкретный процесс – построение и разрушение объекта! Вот какие тут особенности:

1. Вам надо запросить память у операционной системы (или удалить ее). Это медленно не потому, что куча – это какая-то особенная память, а потому, что для поддержания ее структуры, ОС должна выполнить некоторые дополнительные операции для ее менеджмента.

2. Сегментированность. Ваш объект лежит тут, а данные по указателю – где-то там. Машина считывает данные и быстро перемещает в регистры то, что лежит рядом.

Теперь еще более важное замечание: нелокальная часть объекта – это НЕ хорошо и НЕ плохо. Под разные задачи требуются разные подходы. Например, как я писал выше, в серьезных математических вычислениях требуется локализовать все данные, но есть и другие ситуации (даже с математикой), и очень сложно напрямую замерить импакт того или иного решения. Все надо тестировать и еще важнее – понимать, какие у вас есть альтернативы и на что они влияют.

Copy и Move То о чем я буду говорить тут – это механизм по облегчению копирования данных или точнее избежание копирования. Про это есть несколько хороших докладов.

1. Прежде всего вот этот доклад про move семантику.

2. Два доклада от Nicolai Josuttis про move семантику: первый и второй. Во втором докладе больше рассказывается про всякие мало известные тонкости.

[18] Я не хочу обсуждать экзотику, даже если она есть.

3. Этот доклад Sean Parent-а не посвящен напрямую copy и move семантике. Однако, он строит полиморфные типы в плюсах, которые позволяют хранить объекты с требуемым «интерфейсом» в широком смысле слова. И в рамках этого разговора затрагиваются всякие аспекты языка и вопросы копирования и перемещения данных.

В плюсах есть встроенные базовые типы и типы, которые вы строите классами (или структурами, это то же самое). Данные у встроенных типов всегда лежат по адресу переменных у них нет нелокальной части. У классов по разному. Например, `std::vector` имеет очень легкую локальную часть, обычно это три указателя. А вот `std::array` наоборот, имеет очень тяжелую локальную часть, все данные лежат непосредственно по адресу объекта. Если вы конструируете, копируете или разрушаете объект, то вам всегда надо сконструировать, скопировать или разрушить его локальную часть. Без этого вы никогда не обойдетесь. А вот с нелокальной частью можно жульничать. На этом строится идея copy и move в плюсах.

Что такое move и copy. Если ваш объект состоит только из локальной части, то никакой разницы между этими операциями нет, это всегда копирование локальной части.

```
1 std::array<int, 1000000> x;
2 std::array<int, 1000000> y {/*...*/};
3 x = y;
4 x = std::move(y);
```

Третья и четвертая строчки просто копируют данные из y в x и никаких ухищрений тут для облегчения этой процедуры придумать нельзя. Обе эти операции абсолютно одинаковые, потому что вы реально хотите создать копию огромного куска данных в памяти и эту операцию сделать придется. Теперь посмотрим на другой код

```
1 std::vector<int> x;
2 std::vector<int> y(1000000);
3 x = y;
4 x = std::move(y);
```

В строчке 3 происходит полное копирование данных, после этого в x лежит точная копия того, что было в y. А вот в строчке 4 происходит так называемое перемещение данных, а именно мы копируем локальную часть y в x, а в y его локальную часть обнуляем. В итоге x начинает ссылаться на те же самые данные, которые раньше хранились физически в y. После этой операции в x по сути лежат те данные, что были в y, а состояние y неизвестно, но валидно. Для вектора это поведение даже более определено, y просто станет пустым вектором. То есть этот механизм реализует вариант «возьми значение из y и переложи его в x».[19] Таким образом, если у вас локальная часть объекта легкая, а сам объект тяжелый, то его очень дешево перемещать. Опять же, дешево или дорого это для конкретно вашей ситуации надо проверять. Но это хороший ориентир.

Я бы сформулировал следующие наблюдения по рекомендациям, как передавать данные.

1. Маленькие данные хранящие все в локальной части надо передавать by value. К таким данным относятся все встроенные типы. Потому, скажем, `int` или `double` надо передавать только by value.

   Самый простой подход ничего больше не передавать by value по умолчанию. Однако, разумно передавать by value все объекты, которые имеют только локальную часть и по размеру не превосходят какой-то границы. Можно выбрать 8 байт в качестве такой границы (как размер `double` или указателей на 64 битной системе). Данные меньше этого размера не имеет смысла передавать по ссылке, ибо ссылка – это указатель на данные и вы все равно должны будете скопировать этот указатель, а потом еще по нему обратиться. Проще уж сразу вместо указателя положить сами данные. Можно выбрать границу побольше, например в размер кэш линии 64 байта. Большее нее точно делать не стоит.[20] А где найти разумную границу? Ну, тесты вас спасут.

2. Большие данные, состоящие из большой локальной части надо передавать по константной ссылке или по указателю. Например очень плохая идея передавать `std::array` по копии, особенно если его размер в миллионы элементов. И тут есть два варианта. Если вы не хотите менять данные, то передавайте по ссылке, если хотите, то я бы рекомендовал использовать указатель, а не неконстантную ссылку. Причина вот какая. Сравните два кода

[19] Ну, вообще говоря с оговорками. Например для встроенных типов оба варианта равносильны и не обнуляют ни в каком смысле данные справа.

[20] Хотя признаюсь честно, даю эту рекомендацию глядя на круги на воде оставленные вилами моей фантазии, относитесь к этому как «звучит разумно».

```
// first option

void f(int&);
int x;
f(x);

// second option

void g(int*);
int y;
g(&y);
```

В первом примере, вызывающий функцию `f`, не видит по вызову функции, что она меняет саму переменную `x`. Во втором случае, мы вынуждены брать адрес от переменной и это превращается в дополнительный механизм языка, подсказывающий нам, что `y` может измениться внутри функции `g`.

3. Если ваш объект имеет маленькую локальную часть и большую нелокальную часть, то можно воспользоваться move вместо константной ссылки. Если локальная часть действительно маленькая, то вы не увидите разницу между move и обращением по ссылке.[21] Однако, принципиальная разница в том, что вы гарантируете работу с объектами by value, а не by reference. A value semantics safe by default. Например, если вам надо передать данные между двумя thread-ами, то никогда ни при каких условиях не передавайте их по ссылке, замучаетесь бороться с race condition. Вместо этого делайте move данных. Тогда в каждый момент времени только один thread владеет данными и у вас нет race condition.[22]

Copy elision, rvo, nrvo Как вы уже поняли, копирование – это плохо, это выполнение кучи ненужной работы по перемещению данных, а не по их обработке. Кроме copy и move есть еще такая вещь как rvo или copy elision. Вот тут можно почитать как оно работает (это актульно для until c++17, но помогает понять, что происходит). Еще хорошее объяснение есть вот тут и тут.

Если не вдаваться в детали, то предположим у вас есть такой код

```
1 class A {...};
2
3 A func() {
4   return A(...);
5 }
6
7 A a = func();
```

Обратите внимание, что в строчке 7 при вызове функции, мы внутри нее должны создать временный объект, а потом скопировать его содержимое в `a`. Оказывается, что вместо того, чтобы выполнять это копирование, мы можем сразу создать временный объект по адресу `a` и избежать лишнего временного объекта и копирования. Это называется copy elision и эта фигня теперь в стандарте плюсов (начиная с 17-го), хоть и поддерживалась кучей компиляторов де факто. А точнее называется это return value optimization или rvo. Обратите внимание, что никакие сайд эффекты copy constructor не будут выполнены, потому что он не будет вызван![23] Есть и другой вариант, например

```
1 A operator+(const A& x, const A& y) {
2   A result = x;
3   result += y;
4   return result;
5 }
6
7 A a = b + c;
```

В этом случае сработает nrvo (named return value optimization). А именно, объект `b+c` сконструируется сразу по адресу объекта `a`. Nrvo от rvo отличается лишь тем, что вы можете возвращаемый объект хранить во временной переменной. Но есть всякие ограничения, когда оно сработает, а когда нет. И лучше себя на всякий случай проверять, действительно ли у вас нет лишних копий.

[21] Тесты могут прояснить этот вопрос.

[22] Для данных с тяжелой локальной частью надо делать wrapper, перемещающий локальную часть в нелокальную. Идеальный вариант `std::shared_ptr` на константные данные, потому что он имеет value semantics.

[23] Это легко проверяется дописыванием `std::cout` во все конструкторы.

Есть много тонких моментов, когда rvo выполняется, а когда нет. Подробно можно почитать в ссылках выше. Однако тут я хочу сказать пару вещей, которые полезно держать в голове. Вам надо, чтоб ваш локальный объект, не являлся аргументом функции и тип возвращаемого объекта в точности совпадает с типом возвращаемым функцией. В частности, если мы сделаем `return std::move(result)`, это помешает nrvo. Так же nrvo может не сработать при наличии нескольких retrun-ов. Компилятор может на этапе компиляции не понять, какой объект вы пытаетесь вернуть и не сделает nrvo. Про это хорошо написано по ссылкам выше.

2.9 Тестирование

Прежде всего давайте я скажу о том, что есть такая штука как Google Test. Очень рекомендую почитать их мануал на тему того, как пользоваться их системой и заодно просветиться тем, как в целом можно писать тесты. А сейчас я поговорю о том, как я вижу механизм написания тестов руками.

Тесты своими руками Под каждый модуль создаем отдельный тестовый файл

```
testA.h

void testA();

testA.cpp

void testA() {
  // implement your test here
}
```

Потом все тесты пихаем в один файл

```
tests.h

void run_all_tests();

tests.cpp

#include "testA.h"
...

void run_all_tests() {
  testA();
  ... // include all test functions here
}
```

Далее в `main.cpp` вызываем функцию запускающую все тесты

```
main.cpp

#include "tests.h"

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

Такая ручная система собирает единую тестовую функцию, которая прогоняет все тесты. Сообщать о результатах тестов лучше в `std::cout`. Ибо тогда вы сможете перенаправлять результаты тестов в файл из командной строки.

Какие тесты и как писать Писать тесты – это отдельная задача, которая может оказаться сложнее задачи по написанию самой программы. Я рекомендую почитать документацию Google Test на эту тему. А теперь пара слов от меня. Есть тесты двух видов:

1. Тесты на стабильность – стресс тесты.
2. Тесты на корректность – тесты на проверку условий.

Давайте поговорим отдельно по каждому виду тестов.

1. **Стресс тесты.** Задача тестов – проверить на случайных данных, что программа не падает и работает (может быть не корректно, но работает). В таких тестах надо проверять всякие corner case-ы (особые ситуации, которые надо было рассматривать в логике программы отдельно). Обычно это случай пустого контейнера или одного элемента в нем, случай очень большого количества элементов, случай равенства или не равенства каких-нибудь параметров и т.д.

   Для таких тестов достаточно генерировать случайные данные и гонять на них. Однако, **ВАЖНО** ваши случайные тесты должны быть детерминированы. Это значит, что при каждом запуске программы тесты идут с одними и теми же значениями и выдают один и тот же результат. То есть, если вы используете случайный генератор, то обязательно зафиксируйте `seed` для генератора, можно прямо захардкодить или выбрать детерминированную функцию по выбору `seed`, например использовать номер теста в качестве него и т.д. Это нужно, чтобы вы могли повторить тесты. Вот представьте, вы запустили тест и он упал. Вы пытаетесь отловить ошибку, и не можете повторить тест, потому что не знаете с какими параметрами он прошел, какие случайные данные использовались. Потому никогда НЕ используйте в качестве `seed` текущее время или что-то такое непостоянное. Лучше сделайте перебор по разным `seed` в цикле.

   По хорошему надо бы тестировать все функции класса на таких тестах и желательно в режимах релиза и дебага с включенными `assert`-ами.

2. **Тесты на корректность.** Задача тестов – проверить, что программа исполняется корректно. Проблема таких тестов, для них надо генерировать не только входные данные но и ответы, чтобы можно было с чем-то проверять. Это не всегда возможно. Однако, для математических алгоритмов, обычно бывает можно сгенерировать тесты от ответа. Когда вы знаете ответ, портите его и потом запускаете свой алгоритм.

   Например, для программы по диагонализации матриц, можно взять диагональную матрицу D, взять случайную обратимую матрицу C и перейти к матрице $A = CDC^{-1}$. Тогда ваш алгоритм по диагонализации должен получить D с точностью до перестановки элементов на диагонали. Потому можно отсортировать диагональ и сравнивать результат. Так же бывает возможным генерировать правильный ответ, если у вас есть библиотечная или сторонняя функция, решающая ту же или похожую задачу.

   Бывает, что сгенерировать входные данные и ответ не получается, но вы можете прийти к ответу разными путями. Скажем внутри алгоритма можно в разном порядке реализовать промежуточные шаги или есть разные вариации алгоритма или просто разные алгоритмы, которые должны приводить к нужному ответу. Тогда вы можете просто сравнить результаты этих вариаций. Это не гарантирует верность работы, но может позволить вам найти ошибку, если результаты алгоритмов расходятся.

Обратите внимание, что `assert`-ы и тесты – это ортогональные вещи. `assert`-ы проверяют код "изнутри а тесты "снаружи". Обычно тест не может проверить внутренний инвариант класса, если только вы не хотите специально нарушить инкапсюляцию данных (так можно делать, сделать тесты friend классами и обернуть все это в `#ifdef` директивы, чтобы лишние друзья пропадали при компиляции с нужными флагами). Кроме того, `assert`-ы отключаются макросом `NDEBUG`. Очень полезно в релизе отключать этот макрос, чтобы `assert`-ы могли прощупать релизную версию. Потому что пути оптимизации неисповедимы и если ваш код не падал в дебаге, это не гарантия, что `assert`-ы не сработают в релизе. Однако, не забудьте вернуть макрос на место, когда будете релизить ваш код, чтобы не плакала производительность.

3 Система поддержки версий

3.1 Про git

Не удивляйтесь, других систем поддержки версий не существует. Точнее существует, но мы предпочитаем в это не верить. Если вы хотите разобраться с git в целом, то я бы рекомендовал пройти по следующим шагам:

1. Прежде всего есть потрясающий текст объясняющий устройство git в целом. Можно читать на любом языке, который вам нравится.

   Тут объясняется ментальная модель, которую надо держать в голове, когда работаете с git. Всего есть 5 компонент у git (но тут затронуты только первые 3):

   (a) История (history) представляет из себя ациклический направленный граф, его терминальные вершины — это ветки.

(b) Рабочая папка (workspace).

(c) Буфер между ними (stage).

(d) upstream repository это и есть точка подключения удаленного репозитория.

(e) stash — это кладовка, в которую можно сваливать файлы, если тебе их надо срочно сохранить, чтобы сделать что-то другое.

В этом тексте все показывается наглядно и объясняются все механизмы взаимодействия между первыми тремя компонентами git. Это самое лучшее введение в git, что я когда-либо находил.

2. Если вы хотите чуть лучше изучить все 5 компонент, то следующий шаг – это посмотреть серию видео вот тут. По стилю изложения очень напоминает Visual Git выше, но затрагивает более или менее все 5 компонент git.[24]

3. Если вы хотите хороший референс по всем командам с пониманием, какие компоненты git и как затронуты, то я бы рекомендовал вот этот cheat sheet.

4. Из классических ссылок – книга pro git. Это не плохой туториал, но его беда – она не формирует правильную ментальную модель. К сожалению, при всех достоинствах книги[25], она сфокусирована на отдельных командах и не создает картины в целом. Если у вас уже есть ментальная модель, то книга будет читаться на ура, а если нет, то она ее не сформирует. Потому я бы не рекомендовал читать ее первой.

5. Еще есть вот такой базовый виртуальный учебник git. Он совсем примитивный, но помогает визуально увидеть эффект всех твоих команд в истории. К сожалению, он не дает представления об устройстве git и не формирует правильную ментальную модель. Как и все другие обучалки, они сосредоточены только на 1-й компоненте (истории) и совсем не объясняют, как это все сопрягается с тем, что вы имеете на компьютере в рабочей папке. Так же сюда не входят удаленные команды, для работы с удаленным репозиторием. Но может быть это будет полезным.

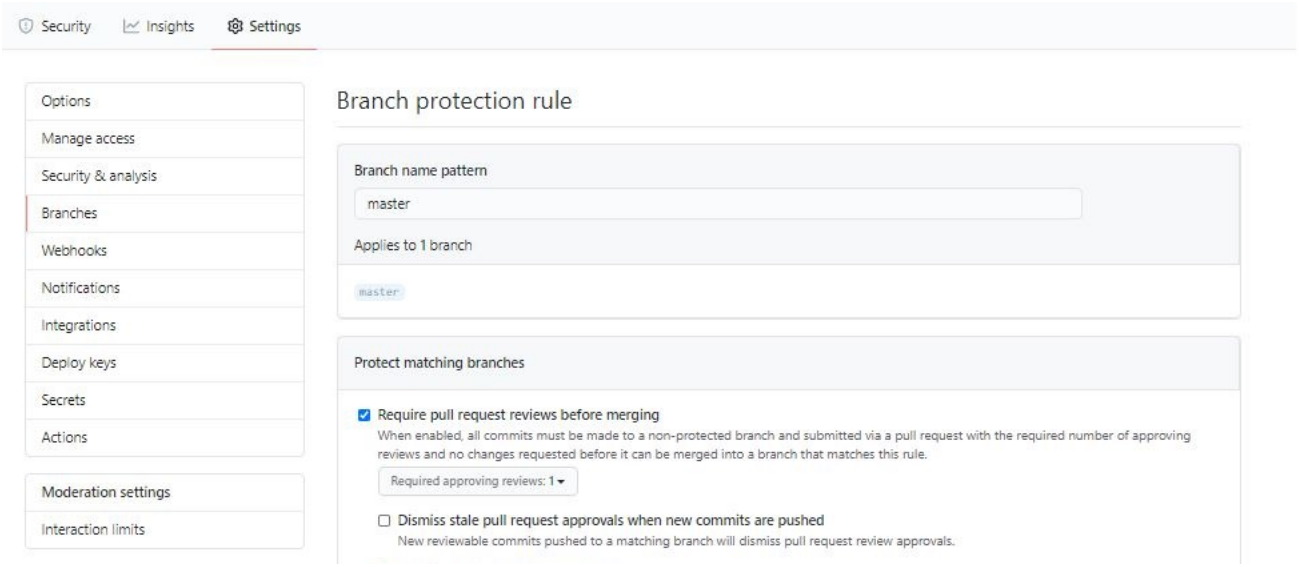
3.2 Рекомендации для работы с git

Работа с git – это локальная работа, то есть git позволяет вам для себя следить за разными состояниями проекта. Если вы хотите работать в команде или под руководством кого-то, вам придется выходить на github. И тут я бы рекомендовал несколько шагов.

1. Создаете пустой публичный репозиторй на github.

2. В настройках делаете ветку master защищенной и в настройках ставите галочку «Require pull request reviews before merging». Теперь в ветку мастер нельзя будет ничего смерджить без разрешения ревьювера.

[24] И мы все еще ждем последнюю завершающую часть про github.
[25] А она хороша.



3. Далее создаете себе ветку (или ветки) для работы над проектом и его фичами, например, dev. И как только вы что-то хотите залить в master, то будет сформирован pull request, в котором надо будет указать меня в качестве reviewer-a.

   При таком подходе, я смогу проследить за всеми изменениями и ничто не пройдет мимо моего взгляда. Для вас ветка dev будет аналогом master, в которую вы можете лить все, что угодно. А вот master будет копией dev, в которую лить можно только после моего одобрения и проверки. У себя на машине вы можете иметь миллион локальных веток для работы и организации фичей.

4. При инициализации проекта я бы рекомендовал первым делом не писать код, а настроить следующее:

   (a) Файл `.gitignore`. У самого github есть куча разных шаблонов `.gitignore` для разных проектов и для плюсов и для LaTeXи для много чего еще. Можно у них своровать. Я обычно делаю закрытый временный репозиторий, инициализирую его с `.gitignore` для нужного шаблона, а потом из него копирую себе github-овский `.gitignore`. Может быть можно как-то проще их оттуда достать, но так работает.

   (b) Написание readme файла с указанием в нем названия проекта, а так же в дальнейшем можно в него помещать описание сборки проекта и инструкций к действию.

   (c) Для плюсовых проектов настроить файл `.clang-format`.

5. Если вам нужны какие-то сторонние библиотеки, то часто они уже есть на github и их можно использовать с помощью git submodules. Хороший текст для понимания вот тут, а референс по командам тут.

A Примеры вставок кода

Этот кусочек посвящен примерам вставки кода в LaTeX. Я для этого собрал свое собственное окружение на основе listings package.

Вставка из файла Можно добавить код из файла, указав с какой строчки по какую вы добавляете.

```
1   CAnyMovable() = default;
2
3   template<class T>
4   CAnyMovable(T&& Object)
5     : pIObject_(std::make_unique<CObjectStored<CObjType<T>>>(std::forward<T>(Object))) {
6   }
7
8   template<class T, class... TArgs>
9   CAnyMovable(std::in_place_type_t<T>,TArgs&& ... args)
10    : pIObject_(std::make_unique<CObjectStored<T>>(std::forward<TArgs>(args)...)) {}
```

Если надо выдрать не непрерывно идущий кусок из строк, то можно указать семейство диапазонов.

```
template< template<class>class TInterface,
          template<class, class>class TImplementation>
class CAnyMovable {
};
```

Набор кода вручную Здесь пример того, как можно написать свой собственный кусок кода прям в LaTeX. В нем демонстрируется вся поддерживаемая подсветка синтаксиса.

```
#include <math.h>

char * str = "for if then some string! i = 0 ";

/*
Comment
*/

class Complex {
public:
  Complex(double re, double im)
    : _re(re), _im(im) {
  }
  double modulus() const {
    return sqrt(_re * _re + _im * _im);
  }
private:
  double _re;
  double _im;
};

void bar(int i) {
  static int counter = 0;
  counter += i;
}

namespace Foo {
namespace Bar {
void foo(int a, int b) {
  for (int i = 0; i < a; i++) {
    if (i < b)
      bar(i);
    else {
      bar(i);
      bar(b);
    }
  }
}
} // namespace Bar
} // namespace Foo
```

Стиле ключевых слов Для удобства ниже приведены стили для ключевых слов. По сути сейчас из множества ключевых слов выделены типы переменных, которые отображаются голубым, остальные – оранжевым. Все это можно перенастроить в файле code.sty. Даже можно добавить свои новые классы ключевых слов, как вам вздумается и выбрать для них свой способ отображения.

```
alignas,
alignof,
and,
and_eq,
asm,
atomic_cancel,
atomic_commit,
atomic_noexcept,
auto,
bitand,
bitor,
bool,
```

```
13  break,
14  case,
15  catch,
16  char,
17  char8_t,
18  char16_t,
19  char32_t,
20  class,
21  compl,
22  concept,
23  const,
24  consteval,
25  constexpr,
26  constinit,
27  const_cast,
28  continue,
29  co_await,
30  co_return,
31  co_yield,
32  decltype,
33  default,
34  delete,
35  do,
36  double,
37  dynamic_cast,
38  else,
39  enum,
40  explicit,
41  export,
42  extern,
43  false,
44  float,
45  for,
46  friend,
47  goto,
48  if,
49  inline,
50  int,
51  long,
52  mutable,
53  namespace,
54  new,
55  noexcept,
56  not,
57  not_eq,
58  nullptr,
59  operator,
60  or,
61  or_eq,
62  private,
63  protected,
64  public,
65  reflexpr,
66  register,
67  reinterpret_cast,
68  requires,
69  return,
70  short,
71  signed,
72  sizeof,
73  static,
74  static_assert,
75  static_cast,
76  struct,
77  switch,
78  synchronized,
79  template,
80  this,
81  thread_local,
82  throw,
83  true,
```

```
try,
typedef,
typeid,
typename,
union,
unsigned,
using,
virtual,
void,
volatile,
wchar_t,
while,
xor,
xor_eq,
```