Материалы к программным проектам

Дима Трушин

Содержание

1 Полезные ссылки **1**

2 Код на плюсах **1**
2.1 Модель компиляции C++ . . . 1
2.2 Шаблоны и модель компиляции плюсов . . . 4
2.3 Правила организации кода и пространства имен . . . 7
2.4 Требования к оформлению кода . . . 7
2.5 Мои рекомендации по оформлению кода . . . 8
2.5.1 Оформление классов . . . 8
2.5.2 Названия . . . 8
2.5.3 Шаблоны . . . 9
2.6 Политика обработки ошибок . . . 10
2.7 Производительность . . . 11

3 Система поддержки версий **13**
3.1 Про git . . . 13
3.2 Рекомендации для работы с git . . . 14

A Примеры вставок кода **15**

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;
}
```

Теперь эта единица трансляции компилируется в машинный код.

```
// main.bin

main:
  call 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.

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

```
1  void f(); // declaration
2
3  void f() {} // definition
4
5  static int x; // declaration
6
7  static int x = 1; // definition
8
```

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

```
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
8  #include "file.h"
9
10 template<class T>
11 T f(T t) {
12   return t;
13 }
14
15 // main.cpp
16
17 #include "file.h"
18
```

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

```
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:

```
// 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

```
// 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

```
// 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

```
// 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.

[5] Вот тут можно глянуть настройки опций форматирования.

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:

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

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

  void make_something() const;

  static  void do_something();

  static constexpr const Type1 default_value = 0;

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

  static Type0 private_static_data_;

  Type1 private_data_ = default_value;
};
```

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). 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).

[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.

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

```
1 void f(int* x) {
2   assert(x && "argument must not be nullptr!");
3   *x = 42;
4 }
```

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.

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.

[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`.

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.

2.7 Производительность

Производительность – это хорошо, но корректно работающий код – еще лучше. Очень легко сделать программу быстрой и не верной. Корректность мы все таки ценим больше скорости.[9] 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.

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.

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.

[9] 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 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 возню.[10]

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.[11] 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.[12] 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[13] (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[14] 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.[15] 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.

- 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.

[10] 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.

[11] 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.

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

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

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

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

- 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-а 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.

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.[16]

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[17], 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.

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

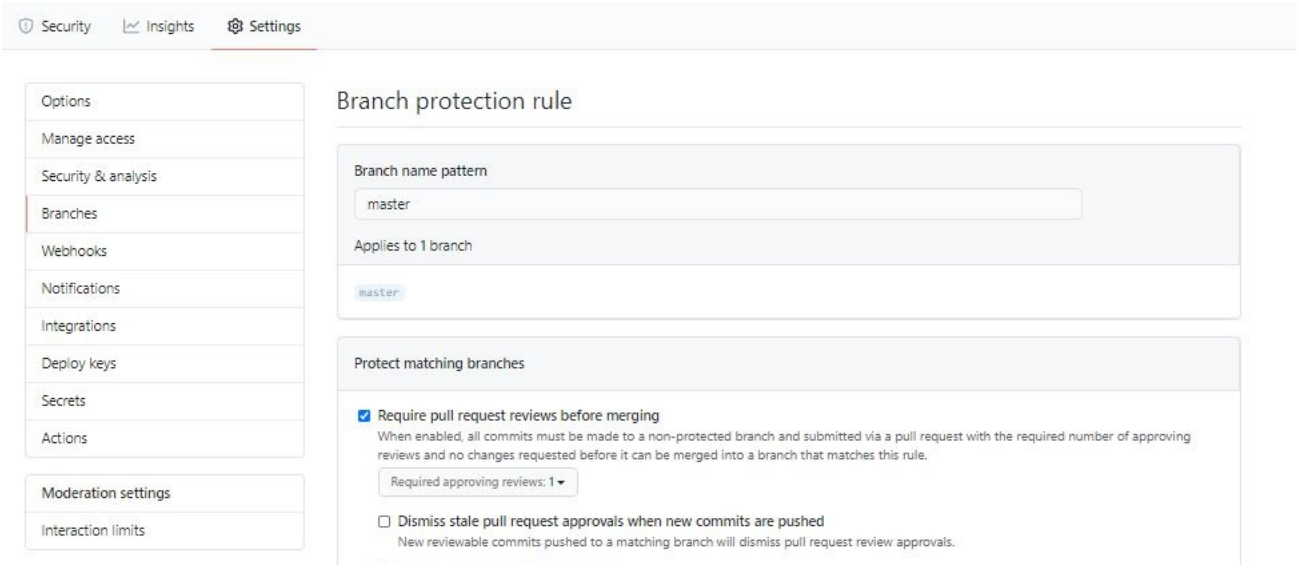
5. Еще есть вот такой базовый виртуальный учебник git. Он совсем примитивный, но помогает визуально увидеть эффект всех твоих команд в истории. К сожалению, он не дает представления об устройстве git и не формирует правильную ментальную модель. Как и все другие обучалки, они сосредоточены только на 1-й 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.



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.

```
  CAnyMovable() = default;

  template<class T>
  CAnyMovable(T&& Object)
    : pIObject_(std::make_unique<CObjectStored<CObjType<T>>>(std::forward<T>(Object))) {
  }

  template<class T, class... TArgs>
  CAnyMovable(std::in_place_type_t<T>,TArgs&& ... args)
    : 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,
break,
case,
catch,
char,
char8_t,
char16_t,
char32_t,
class,
compl,
concept,
const,
consteval,
constexpr,
constinit,
const_cast,
continue,
co_await,
co_return,
co_yield,
decltype,
default,
delete,
do,
double,
dynamic_cast,
else,
enum,
explicit,
export,
extern,
false,
float,
for,
friend,
goto,
if,
inline,
int,
long,
mutable,
namespace,
new,
noexcept,
not,
not_eq,
nullptr,
```

```
operator,
or,
or_eq,
private,
protected,
public,
reflexpr,
register,
reinterpret_cast,
requires,
return,
short,
signed,
sizeof,
static,
static_assert,
static_cast,
struct,
switch,
synchronized,
template,
this,
thread_local,
throw,
true,
try,
typedef,
typeid,
typename,
union,
unsigned,
using,
virtual,
void,
volatile,
wchar_t,
while,
xor,
xor_eq,
```