MASTER OF
SOFTWARE
ENGINEERING

Правила сдачи домашних заданий

17 сентября 2021 г.

В процессе семестрового курса студенты сдают несколько (обычно три) домашних задач. Задачи проверяются как автоматическими тестами, так и вычитыванием кода. Это означает, что необходимо внимательно относится к тому, чтобы код был легко читаемым.

1 Процесс сдачи задач

1. Для каждой домашней работы должна быть создана отдельная папка в Subversion-репозитории. Точное имя директории, а также список и названия исходных файлов, которые необходимо использовать, будут указаны в формулировке задачи.

   Не кладите в репозиторий ненужные файлы (файлы проектов, настройки IDE, бинарники и т. д.).

2. После реализации задачи и ее коммита в репозиторий в системе Trac (https://trac.compscicenter.ru/itmo/) должна быть создана задача (task) проверяющему преподавателю *makarov*.

3. Этот таск должен иметь тип «Ожидается проверка» и приоритет «Проверка». В ответ на этот таск преподаватель проверит вашу работу и изменит тип на «Ожидаются исправления» с описанием проблем. Об этом вам должна прийти нотификация на почту (если вы еще не настроили ее в trac'е, сделайте это как можно скорее). После исправлений вы снова изменяете тип таска. Всего допускается 3 попытки сдачи задачи. Также преподаватель может создать таск с приоритетом «Пожелания», который вы можете принять к сведению, но можете не исправлять.

4. Результат сдачи задачи также будет указан в таске (либо зачтена, либо нет с указанием причин).

5. Ниже приводятся рекомендации и пожелания по выполнению задач. При систематическом или грубом невыполнении рекомендаций проверяющий преподаватель оставляет за собой право не проверять задачу с её последующим незачтением. Уважайте, пожалуйста, проверяющего преподавателя.

Дополнительная информация о порядке сдачи задач указана на главной странице Trac и в тексте домашнего задания.

2 Рекомендации

Список рекомендаций будет пополняться.

Общие

1. Используйте единый стиль кодирования (см. ниже).

2. Старайтесь писать легко читаемый код. Не скупитесь на отделение логических блоков строками, используйте вертикальное выравнивание и т. д.

3. Программа должна собираться с вашим `Makefile`/`CMakeLists.txt` без ошибок.

4. Предупреждения (warnings) при сборке программы недопустимы (уровень `-Wall` для `g++`, `Level 4` для `MSVC`).

5. Используйте стражи включения в хидерах.

6. Используйте правила «одна функция – одна задача», «один объект – одна задача». Не стоит функцию, которая, например, считает интеграл, заставлять выводить на экран результат. Или, например, калькулятор стоит отделить от лексического анализатора.

7. Выделяйте отдельные логические блоки в разные единицы трансляции (а общие объявления выносите в заголовочные файлы). Это упростит разбор кода и ускорит перекомпиляцию. Ищите разумный баланс, 10 cpp-шников по 10 строк — возможно, не лучший выбор, как и файл в 1000 строк кода.

8. Реализация в заголовочном файле функции допустима только в случае указания директивы встраивания (inline) или в случае реализации функций-членов внутри определения класса (что эквивалентно директиве inline). Иначе сборка чревата ошибкой линковки.

9. Используйте динамическую память только там, где это необходимо.

10. Обязательно проверяйте утечки памяти (например, с помощью valgrind) — перед выходом из программы вся выделенная память должна быть освобождена.

11. Входные параметры передавайте в функцию по константной ссылке; по ссылке — чтобы они лишний раз не копировались, по константной — чтобы вы не могли их случайно изменить. Не забывайте про const! Выходные параметры передавайте в функции по указателю — чтобы вы могли их изменить; по указателю, а не по ссылке, — чтобы вы могли в месте вызова отличить входные параметры от выходных по амперсанду перед именем переменной. Размещайте входные параметры перед выходными в списке параметров функции или метода.

    Аргументы примитивных типов следует передавать в функции по-другому. Входные параметры типов `int`, `char`, `bool`, `double` передавайте по значению. Они будут копироваться, но это так же почти бесплатно, как и в случае ссылок или указателей. При этом вы не сможете их изменить изнутри функции, что и нужно, т. к. это входные параметры. Если вам нужны эти типы как выходные параметры функции, лучше передавайте их по ссылке, т. к. иначе легко внутри функции перепутать указатель на переменную с самой переменной, и сделать совсем не то, что вы собирались.

12. В большинстве случаев нельзя сравнивать числа типа `float` и `double` просто операторами `<`, `>`, `<=`, `>=`, `==`: при вычислениях в вещественных типах накапливается погрешность, вследствие чего равные по сути числа, вычисленные с помощью разной последовательности действий, могут получить различные значения в типах `float` и `double`, и даже `a < b` может измениться на `b < a`. Погрешность вычислений можно оценить, используя точные знания о том, как именно выполняются арифметические операции, а также как происходят вычисления в используемых вами функциях. Обычно делать этого точно не нужно, т. к. точность типа `double` позволяет хранить 15–16 знаков, а требуемая в задачах точность обычно порядка 10^{-6} или 10^{-9}, но не меньше. Однако для того, чтобы корректно сравнивать числа, следует использовать порог сравнения.

13. Объявляйте переменные как можно ближе к месту их первого использования. Старайтесь сразу же инициализировать переменные. Если переменная используется только внутри функции, она должна быть локальной для функции. Если только внутри цикла, она должна быть локальной для цикла.

    Никогда не делайте глобальных переменных. Локальные переменные блока предпочтительнее по сравнению с локальными переменными функции, локальные переменные функции — по сравнению с переменными-членами класса, а последние — по сравнению с глобальными переменными. Стремитесь сократить “время жизни” каждой

переменной: чем меньше время жизни переменных, тем меньше переменных приходится одновременно держать в голове при чтении и написании кода. Исследования показывают, что человек может эффективно держать в памяти не более 5-7 переменных одновременно. Большее количество неизбежно приводит к ошибкам.

14. Пишите комментарии только по делу. В идеальном случае лучше обходиться вообще без них — ваш код прокомментирует сам себя. Конечно, так редко удается, поэтому комментарии к классам и функциям бывают полезными.

    Не нужно оправдывать плохое имя (см. следующий раздел) подробным комментарием. Если у вас встречается объявление вида

    ```
    int n; // number of balls in the bucket
    ```

    то нужно заменить его на `int number_of_balls;` или `int numBallsInBucket;` в зависимости от принятого стиля, от того, бывают ли шары не в корзине, и от контекста.

    Писать комментарий следует над тем, к чему он относится. Комментарии в конце строки значительно удлиняют ее, поэтому ухудшают читаемость. При этом желательно, чтобы строка влезала в 80 символов, а зачастую бывает жесткое ограничение по длине строки. Если вы все же пользуетесь комментарием в конце строки, то отделяйте его двумя пробелами от кода.

    Комментарии к функции должны быть написаны рядом с интерфейсом, а не с реализацией, если они разделены: пользователь будет в первую очередь смотреть на интерфейс, к тому же реализация сторонних библиотек может быть вовсе недоступной. То же самое относится и к классам: комментарии к классу и к его методам должны быть в интерфейсе класса, а не в реализации.

    Если вы решили снабдить свой код подробными комментариями, указывайте в них то, что будет интересно читающему. Для класса это описание того, для чего класс нужен, как им пользоваться. Для функции и метода — что они делают, что возвращают, что принимают на вход, какие исключения могут бросать.

15. Не оптимизируйте преждевременно. Не нужно оптимизировать с целью ускорить программу в константу раз, если это хоть сколько-нибудь усложняет код. Старайтесь сделать свое изначальное решение максимально простым. Оптимизировать нужно только после того, как вы четко замерили время работы программы, убедились, что оно слишком большое, определили, какая именно функция создает узкое место. Даже суперпрофессионалы не берутся заранее предсказывать узкие места системы: в наше время, когда компиляторы умеют делать сумасшедшие оптимизации, это практически невозможно предугадать. Поэтому профессионалы и не пытаются делать это заранее и оптимизировать что-либо заранее. Сначала измерьте, найдите узкое место, а потом уже пытайтесь его оптимизировать. Напишите максимально простое решение, добейтесь правильной его работы, и если вдруг после этого оно окажется слишком медленным — только тогда оптимизируйте. Ваша задача в программах, которые вы пишете на этом курсе, — написать наиболее простой, понятный, читаемый и гибкий код, среди тех, которые проходят в ограничения по времени и памяти. Помните об этом и не оптимизируйте, жертвуя простотой и удобством.

16. Не допускается использование платформо-зависимого кода. Ваш код должен соответствовать стандарту C++03, C++11 или C++14 и компилироваться в удовлетворяющем стандарт C++ компиляторе. Код должен корректно работать как на 32-битных, так и 64-битных платформах. Сборка будет производиться `g++` версии не ниже 4.8, а также опционально другими компиляторами (`clang, MSVS`).

Классы

1. Минимизируйте явный интерфейс класса. Не вносите лишние функции в интерфейс. Чаще всего открытые функции-члены класса, которые могут быть реализованы через другие открытые функции-члены класса, должны быть вынесены из класса.

2. Функции, не меняющие состояния класса, должны быть объявлены как константные.

3. Уберите все лишние `this->` из кода (кроме синтаксически необходимых). Это портит читаемость кода.

4. Объявление класса выносите в заголовок (если, конечно, класс не предназначен для использования исключительно внутри некоторой единицы трансляции — в этом случае стоит обернуть объявление и определение класса в анонимный namespace).

5. Открытые и защищенные члены данных в структурах с нетривиальным интерфейсом недопустимы. Не должно быть так, чтобы пользователь мог нарушать инвариант класса.

6. Инициализация данных класса по возможности должна проходить в списке инициализации конструктора.

7. Вспомогательные функции-члены, служащие для реализации функций интерфейса, но сами к интерфейсу не относящиеся, должны быть закрыты.

Пример стиля кодирования.

1. Для именования имен переменных, функций и методов используются маленькие буквы с разделением знаком _ между словами (under-score style). Для именования классов используются идентификаторы, начинающиеся с заглавной буквы (CamelCase style).

2. У каждой создаваемой сущности в коде есть имя. Сперва автор, а впоследствии и все читающие код ассоциируют имена с сущностями, которые они обозначают. Чтобы в каждый момент точно понимать, что в переменной хранится, чтобы быть уверенным в том, что вызов функция вернет ожидаемое значение, имена нужно давать осмысленные и грамотно определенные.

   Имена переменных должны быть длинными и понятными. Каждый раз, когда вы пишете одно-двух-буквенное название переменной или используете что-то вроде `cur`, должно возникать неприятное чувство. Единственное место, где можно позволить себе однобуквенные переменные, — в качестве счетчика в очень коротком `for`'е без вложенных циклов. И то, у вас должны быть серьезные опасения, когда вы это делаете, вы должны делать это осознанно. Иначе можно легко допустить ошибку с индексами, например перепутать `i` с `j`, что происходит постоянно, если называть так переменные. Искать такую ошибку вы будете несколько часов или дней. Даже если в описании задачи у меня есть названия `R` и `L`, это не значит, что в программе нужно их так называть. Стиль математического текста очень сильно отличается от стиля кода программы. В математическом тексте есть очень много слов, описывающих формулы и то, что в них происходит. В самих формулах ценится краткость. В коде же наоборот, слов, описывающих происходящее, практически нет. Код должен описывать сам себя, названиями переменных, методов и классов. Поэтому названия должны быть очень прозрачными. Не должно быть нужно возвращаться и смотреть вверх в объявление переменной или смотреть на ее инициализацию, чтобы понять, что она в себе содержит. Никогда не называйте переменные `something1` и `something2`, так как очень легко ошибиться и попасть по соседней клавише, тем самым очень легко сделать баг, а искать его будет тяжело. Используйте `something_first` и `something_second` или что-нибудь еще.

   Все, что относится к именам переменных, относится и к именам функций, классов и методов. Кроме того, в названиях методов (функций) обязательно должен быть глагол, описывающий действие, которое выполняет метод. Это действие должно быть одно. У каждой функции должна быть одна ясная цель. Если вы понимаете, что не можете придумать название функции без слова And (например ReadFromFileAndSort), значит функция выполняет две разные цели, и, скорее всего, ее нужно разбить на несколько меньших функций (ReadFromFile и Sort), и из внешней вызывать подряд внутренние.

   Не сокращайте слова в названиях. Это ухудшает читаемость кода, а также делает невозможным поиск по нему. Не нужно сокращать `index` до `ind` или `idx`, `current` — до `cur` и т. д. Единственное исключение — общепринятые сокращения типа Http и т. д.

3. Выделяйте названия приватных членов классов, это позволяет отличить их от аргументов методов. Наиболее распространенными способами являются подчеркивание в конце: `name_`, — или префикс `m_`: `m_name`. Начинать имя переменной с подчеркивания не принято; следует помнить о том, что имена, начинающиеся на два подчеркивания или подчеркивание и заглавную букву, зарезервированы стандартом, и использовать их нельзя.

4. Никогда не используйте “магические константы” в коде. Если у вас где-то в коде встречаются, например, ’a’ и ’z’, означающие минимальный и максимальный символ алфавита, то их надо заменить на именованные константы.

5. Фигурные скобки всегда начинаются и заканчиваются на отдельных строках (Allman style, см. http://en.wikipedia.org/wiki/Indent_style#Allman_style).

6. Рекомендуется ставить знаки препинания, руководствуясь правилами естественного языка (если специально не сказано обратного). Например, перед запятой не ставится пробел, после — ставится; перед точкой с запятой не ставится пробел и т. д.

7. После операторов ставится пробел:

```
if(statement)  // неверно
if (statement) // верно
```

8. Не жадничайте с пустыми строками. Вставляйте всегда пустые строки между определениями глобальных функций, классов, констант, `typedef`’ов, `#include`’ов, между объявлениями методов и функций, между реализациями функций, между объявлениями классов и реализациями функций и т. д.

9. Вставляйте пустые строки в код реализации функций, чтобы подчеркнуть разделение логических частей кода.

10. Не размещайте `if`, `else`, `for`, `while` и др. на одной строке со своим statement вот так:

```
if (condition) statement;
else statement;
...
for (...) statement;
```

    Это, во-первых, ухудшает читаемость кода. Вы можете вообще один из statement’ов не заметить или ошибочно решить, что он относится к `if`’у:

```
if (number % 2 == 0) std::cout << "Even\n"; even = true;
```

    А во-вторых, при отладке debugger’ом невозможно понять, выполнив команду «Step Over», выполнилось или не выполнилось условие (или сколько итераций цикла прошло).

11. Обрамляйте в фигурные скобки тело `if`, `else`, `for`, `while`:

```
for (int index = 0; index < array.size(); ++index)
{
    statement1;
    statement2;
    ...
}
```

    даже если внутри только один statement.

```
if (number % 2 == 0)
{
    std::cout << "Even\n";
}
```

Это более читаемо и безопасно. В варианте без скобок легко ошибиться, например, вот так:

```
if (number % 2 == 0)
    std::cout << "Even\n";
    even = true;
```

Легко подумать, что код even = true; — тоже находится под `if`-ом.

12. После окончания `namespace`'а ставьте комментарий с его именем:

```
namespace my_lib
{

namespace details
{
  // content
  // ...
} // details
} // my_lib
```

```
namespace my_lib::my_module
{
  // content
  // ...
} // namespace my_lib::my_module
```

13. Табы должны заменяться на пробелы (4 пробела), кроме `Makefile`'ов, там табы — элемент синтаксиса.

14. Для форматирования отступов и выравниваний можно использовать инструменты автоматического форматирования кода. Например, `clang-format`.

15. Оформление списка наследования и списка инициализации:

```
struct my_struct
    : public base1
    , private base2
{
    //...
};

some_struct::some_struct(/*...*/)
    : member1(/*...*/)
    , member2(/*...*/)
{
    // ...
}
```

16. Используйте исключения для обработки критических ошибок. Т. е. тогда, когда нарушаются предусловия выполнения функции. При этом:

    (a) всегда наследуйте свои исключения от `std::exception`;

    (b) не забывайте экспортировать типы исключений, если они могут быть проброшены через границы модуля (хотя таких ситуаций лучше избегать);

    (c) принимайте исключения по `const&`;

    (d) если используется ромбовидное наследование (весьма вероятно, т. к. все усключения унаследованы от `std::exception`) обязательно делайте его виртуальным — иначе по catch (`std::exception const&`) не поймаете.

17. Не используйте ввод-вывод в стиле C через функции `scanf`, `printf` — используйте вместо них операторы `>>` и `<<` у `std::cin` и `std::cout` соответственно

18. Старайтесь использовать по возможности классы RAII для управления памятью и ресурсами.

```
std::ofstream file(...);
// вместо
FILE* file = fopen();
//...
fclose(file);
```

19. Если используется значение типа истина/ложь, то используйте тип `bool`, а не `int`.

20. Не используйте тип `long`. Более стандартный тип — `int`, к нему у всех уже привыкли глаза, и `long` с теми же намерениями — просто смотрится странно. На 32-битных машинах оба типа являются 32-битными и ничем не отличаются, поэтому используйте `int` вместо `long`. Если вам нужен 64-битный тип, придется воспользоваться типом `long long` — отличайте его от просто `long`.

21. При прочих равных, используйте преинкремент `++i`, а не постинкремент `i++`. Это полезная привычка. В случае `int`'ов это все равно, но если у вас будет в коде сложный итератор, то в процессе постинкремента создается его копия в памяти, что может создать вам неожиданные тормоза и повышенное использование памяти, а догадаться о том, что вся проблема — в коротком выражении `it++` — будет сложно.

22. Вставляйте слово const везде, где только это возможно по смыслу. Если какая-то переменная по сути меняться в функции не должна, она должна быть `const`. Если метод класса не меняет при вызове содержимое класса, он должен быть `const`-методом. Таким образом вы обезопасите себя от многих глупых ошибок: они отловятся еще на этапе компиляции.

    Если у вас из-за того, что вы где-то поставили в правильном месте `const`, не компилируется код, то `const` выполнил свою главную задачу. Тогда надо не его убирать, а найти и исправить проблему в другом месте: вы где-то еще забыли поставить `const` или изменяете переменную, которую не собирались изменять. Надо в этом разобраться, доставить `const` туда, где он еще нужен, а не удалять там, где он вам мешает

23. Предпочитайте делегирование и агрегацию наследованию.

24. У базового класса либо должен быть виртуальный деструктор.

25. Используйте везде в программе индексацию с нуля. Если какие-то входные или выходные данные в задаче используют индексацию с единицы, лучше в функции ввода, соответственно вывода, переведите индексацию из одной системы в другую, а везде внутри программы, помимо функций ввода и вывода пользуйтесь индексацией с нуля. Весь язык C++ так спроектирован, что индексация с нуля гораздо удобнее, а как только вы начинаете использовать индексацию с единицы, становится неудобно, появляются вычитания единицы из переменных по всему коду и т. д.

26. Задумывайтесь о переполнениях типов. Если у вас есть две переменные типа `int`, значение каждой равно миллиону, и вы их перемножаете, то тип переполнится (максимальное значение — $2^{31} - 1$), и вы получите неправильный результат. Необходимо перед перемножением привести обе переменные к 64-битному типу `long long`. Если у вас есть две `int` переменные со значением два миллиарда и вы их складываете, — тоже произойдет переполнение, тоже нужно предварительно приводить к `long long`.

27. Не пользуйтесь макросами для определения констант (и функций!). Макросы — это очень опасная и неудобная вещь. Их раскрывает специальный препроцессор, который начинает работать еще до компилятора C++, и он ничего не знает о самом языке. Все конструкции раскрываются буквально. В связи с этим есть множество возможных неочевидных побочных эффектов, а у компилятора нет возможности выполнить проверку типов, константность и т. д.

28. Иногда удобно разбить объявления методов на несколько таких групп. Это улучшает читаемость кода. Выполняйте это правило в разумных пределах.

```
struct Student
{
    virtual std::string const & name() const noexcept = 0;
    virtual void set_name(const std::string & name) = 0;

    virtual size_t age() const = 0;

    virtual float average_score () const = 0;
};
```

29. Закрытые поля классов именуйте с подчеркиванием в конце. Это не относится к функциям.

30. Если у класса есть открытые поля, то все ее поля должны быть открытыми. В таких классах из методов допускается только конструкторы и операторы.

31. Используйте современные средства языка. Например, auto для объявления переменных — код сразу станет меньше. Очень полезны анонимные функции.

32. Если используете структуру в модулях, которые используют ее совместно, а компилируются отдельно — следите внимательно за выравниванием полей в этой структуре. Хорошим решением может оказаться обернуть ее директивой выравнивания полей (`#pragma pack`).

33. Аргументы в параметрах функции не отделяются пробелами от скобок

```
        void do_smth( double d, int i ) // неверно
        void do_smth(double d, int i)   // верно
```

34. Все бинарные операторы должны окружаться пробелами. Исключением являются операторы `.`, `->`, `::`.