РУКОВОДЯЩИЕ УКАЗАНИЯ

ПО ПРОГРАММИРОВАНИЮ И ОФОРМЛЕНИЮ

ТЕКСТОВ ИСХОДНЫХ КОДОВ

НА ЯЗЫКЕ C++

Листов 68

Инв. № подл.	Подп. и дата	Взам. инв. №	Инв. № дубл.	Подп. и дата

2023

АННОТАЦИЯ

В данном документе приведены требования, предъявляемые к программистам при разработке исходных кодов языке C++ для программ, использующихся на разрабатываемых изделиях.

В разделе «Общие сведения» указано назначение и описание общих характеристик языка, его возможностей, основных областей применения.

В разделе «Элементы языка» указано описание синтаксиса и семантики базовых и составных элементов языка, а также правила форматирования текста и правила программирования.

Раздел «Стиль программирования» содержит рекомендации по стилю программирования. Основное внимание уделяется разработке классов и использованию проверочного кода.

Настоящий документ подготовлен в соответствии с требованиями ГОСТ 19.506-79 [1].

Сознательно сделано отклонение от стандарта ГОСТ 19 в части оформления содержания (межстрочный интервал) и заголовков/подзаголовков (жирный шрифт) для улучшения читаемости.

СОДЕРЖАНИЕ

1. ВВЕДЕНИЕ

Данный документ содержит требования, предъявляемые к программистам при разработке исходных кодов языке C++ для программ, использующихся на разрабатываемых изделиях.

В разделе «Элементы языка» указано описание синтаксиса и семантики базовых и составных элементов языка, а также правила форматирования текста и правила программирования. Правила ограничивают набор разрешенных для использования средств языка программирования. Это позволяет избежать многих серьезных ошибок или облегчает обнаружение таких ошибок. Кроме того, устанавливаются правила форматирования текста программы и выбора имен идентификаторов. Такие правила делают исходный текст программы легко читаемым и понятным для других программистов. Благодаря этому значительно упрощается сопровождение и дальнейшая разработка программы.

Раздел «Стиль программирования» содержит рекомендации по стилю программирования. Основное внимание уделяется разработке классов и использованию проверочного кода.

2. ОБЩИЕ СВЕДЕНИЯ

Язык программирования C++ — компилируемый, статически типизированный язык общего назначения [2], широко применяемый для написания исходных текстов программ.

Данный язык поддерживает такие парадигмы программирования, как процедурное программирование и объектно-ориентированное программирование. Язык имеет богатую стандартную библиотеку, которая включает в себя распространённые контейнеры и алгоритмы, ввод-вывод и многое другое.

Являясь одним из самых популярных языков программирования, C++ имеет широкую область применения от написания операционных систем и драйверов устройств до разнообразных прикладных программ. Процесс стандартизации

языка C++ начался в 1989 году и продолжался до 1998 года, когда вышел стнадарт C++98 (ISO/IEC 17882-1998) — за основу был взят язык в том виде, в котором он был описан его создателем [2, 3].

При разработке в основном следует ориентироваться на компилятор GCC-8, почти полностью поддерживающий стандарт языка C++17. В большинстве задач достаточно использования стандартов C++11 и C++14.

3. ЭЛЕМЕНТЫ ЯЗЫКА

3.1. Средства, запрещенные к использованию

Язык C++ и стандартные библиотеки предоставляют программисту широкий выбор различных средств программирования. Однако непродуманное использование некоторых средств может привести к тяжелым и труднообнаруживаемым ошибкам в программе. Ниже перечисляются средства, использование которых запрещается без разрешения руководителя проекта:

1) Запрещено использовать какие-либо средства выделения памяти, кроме оператора **`new`**. В частности, категорически запрещено использовать функцию `malloc`.

2) Категорически запрещено использовать функции `strcpy` и `strcat`.

3) Запрещено использовать исключения, не выведенные из базового класса исключений.

4) Запрещено создавать новые макросы, за исключением специально оговоренных случаев (см. ниже).

5) Запрещено инициализировать структуры, за исключением массивов структур, списком инициализаторов.

Нежелательно использовать виртуальное наследование (**`virtual`**) в конечном продукте. Использование виртуального наследования может быть оправданно в исследовательских проектах, когда необходимо, например, создать сходные и мало отличающиеся по пользовательскому функционалу объекты.

3.2. Выбор идентификаторов

Основное требование к выбору идентификаторов — понятность. Следует избегать непонятных сокращений, бессмысленных или однобуквенных идентификаторов (кроме переменных цикла `i`, `j`, `k`). Кроме того, на выбор идентификатора налагается ряд формальных и неформальных ограничений.

Желательно, чтобы по идентификатору объекта или метода можно было понять, для чего объект или метод предназначен. Это означает, что по возможности идентификаторы должны быть осмысленными выражениями. Однако не стоит впадать и в другую крайность: слишком длинные идентификаторы загромождают выражения. Обычно идентификатор должен быть не длиннее 15—20 символов. При этом более простым и часто используемым объектам и методам следует давать более короткие идентификаторы, более сложным и редко используемым объектам и методам, а также глобальным объектам – более длинные, но более понятные.

Общее правило образования идентификаторов: идентификатор состоит из отдельных слов или частей слов, написанных маленькими буквами, причем второе и последующие слова всегда начинаются с большой буквы (`prevItem`, `DefaultUserMsg`) (подчеркивание для разделения слов не используется).

Начинать идентификатор с символа подчеркивания разрешается только для устранения конфликта между именем аргумента метода класса и именем поля этого класса. Например, если в классе `CPoint` есть два поля с именами `x` и `y`, то конструктор этого класса может иметь аргументы с именами `_x` и `_y` или `x_` и `y_`.

3.2.1. Обязательные правила образования имен

3.2.1.1. Перечисления

К имени элемента перечисления добавляется приставка, образованная из первых букв слов, составляющих название перечисляемого типа. После приставки ставится символ подчеркивания.

Листинг 1 – Образование имени перечисления

```
1 enum TInputMode {
2     IM_Scaner,
3     IM_Batch,
4     IM_File
5 };
```

3.2.1.2. Константы

Имена для констант пишутся с большой буквы. Иногда допустимо использовать все заглавные буквы в названии константы (если так принято в соответствующей дисциплине/науке/направлении или если название константы — аббревиатура):

```
const int Re = 6371; ///< Радиус Земли, [км]
const int ID = 120; ///< Идентификатор
const int VGC = 2; ///< Очень хорошая константа (very good const)
```

3.2.1.3. Классы, структуры

Имена классов и структур начинаются с приставки «C», слово за которой начинается с большой буквы.

```
class CWorker
struct CWorkerParams
```

3.2.1.4. Интерфейсы

Имена для интерфейсов начинаются с приставки «I», слово за которой начинается с большой буквы.

```
IBlock, ISubsystem
```

3.2.1.5. Пространства имен

Имена для пространств имен начинаются с большой буквы и не имеют какой-либо специальной приставки.

```
1 namespace MyNamespace /// Мое пространство имен
2 {
3 ...
4 }
```

3.2.1.6. Определения типов

Имена typedef-ов начинаются с большой буквы. Имена typedef-ов для структур и классов начинаются с большой буквы «C». Имена других типов, не являющихся классами и структурами, начинаются с большой буквы «T».

```
typedef const BYTE *TBytePtr;
typedef const CMYCLASS *CMyClass;
typedef const CMYSTRUCT *CMyStruct;
typedef const TMYENUM *TMyEnum;
```

3.2.1.7. Переменные

Имена глобальных переменных пишутся с большой буквы. Имена локальных переменных и параметров методов пишутся с маленькой буквы. Запрещено добавлять к именам переменных префикс, зависящий от типа данных (так называемая, «венгерская нотация»).

3.2.1.8. Функции/методы класса

Имена глобальных функций начинаются с большой буквы. Имена статических функций начинаются с маленькой буквы. Регистр буквы, с которой начинаются имена членов класса (полей и методов) зависит от характера их использования:

1) при локальном использовании полей или методов, служащих только для реализации класса или библиотеки, используется маленькая буква;

2) если поле или метод используется вне этого класса, то употребляется большая буква (под использованием здесь подразумевается не только доступ к членам класса, но также и переопределение при наследовании).

Очевидно, что в такой системе **`private`** члены должны начинаться с маленькой буквы, а **`public`** и **`protected`** — обычно с большой буквы. Имена статических полей класса пишутся также, как и имена обычных полей.

Запрещено добавлять к именам полей класса префикс `m_<имя поля>` (данная система именования используется в библиотеке MFC).

3.2.2. Использование сокращений в именах

Следует избегать использования сокращений в качестве части имени. Заведомо не нужно использовать сокращения, если без их использования идентификатор имеет приемлемую длину.

3.2.3. Рекомендации по образованию имен

3.2.3.1. Бинарный атрибут

Для полей классов, которые по своей сути являются бинарными атрибутами, имя в большинстве случаев должно начинаться с приставки «is» или «has». Например, для класса `CStream` можно завести переменную `isOpen`. Методы, которые возвращают бинарную характеристику объекта, также рекомендуется начинать с этой приставки.

3.2.3.2. Количество элементов

Если переменная служит для хранения количества элементов в каком либо списке или массиве, то к названию такой переменной следует добавлять приставку «numberOf» или суффикс «count». Например, переменная, указывающая количество свободных блоков в менеджере памяти, может называться `numberOfFreeBlocks` или `freeBlocksCount`. Вместо приставки «numberOf» допустимо сокращение «n».

3.2.3.3. Размер/длина

Слово «Size» применяется для переменных и методов, задающих размер (число байт). Слово «Len» (или «Length») используется в имени переменных и методов, служащих для задания числа элементов. Замечание: Слово «Size» — это

размер, а «Length» — длина. Не следует размер блока памяти называть blockLen, а длину строки — stringSize.

3.2.3.4. Методы получения бинарного атрибута

Имя метода получения бинарного атрибута должно быть предикатом (вопросом, на который можно ответить Да или Нет). В большинстве случаев имя метода должно начинаться с глаголов «Is» или «Has».

3.2.3.5. Методы получения и установки небинарного атрибута

Обычно метод получения небинарного атрибута называется также как и **`private`** атрибут, только его имя начинается с большой буквы. Допускается добавление к названию метода получения атрибута глагола «Get». Если кроме метода получения атрибута в классе есть также метод установки атрибута, то названия этих методов должны начинаться с глаголов «Get» и «Set» соответственно.

Если для получения возвращаемого значения требуются нетривиальные вычисления, название метода обычно начинается с глагола, описывающего эти вычисления, например «Find». Такой метод обычно не приводит к изменению состояния объекта и должен объявляться как константный. Запрещается начинать имя метода, требующего нетривиальных вычислений, с глагола «Get».

3.2.3.6. Методы, производящие действие

Методам, изменяющим объект либо внешние по отношению к объекту данные, даются имена, начинающиеся с глагола, обозначающего это действие. Например, `FilterDust(), BuildTree()` и т. п. Часто такими глаголами являются «Do, Make, Process».

3.3. Правила форматирования текста

Во всех случаях без исключения после запятой ставится пробел.

Текст выравнивается с помощью табуляции размером 4 пробела, комментарии отделяются пробелом:

```
namespace MyNamespace /// Мое пространство имен
{
class CWorker
{
    int someField;
    void someMethod
    {
        //Do smth...
    }
};// end class CWorker
}// end namespace MyNamespace
```

При особо длинных конструкциях (и отсутствия необходимости их рефакторить для уменьшения длины) рекомендуется добавлять в конце комментарий какая именно конструкция закрывается (см. в листинге выше `//end class CWorker, // end namespace MyNamespace`). Также рекомендуется комментировать это для вложенных циклов:

```
for( int i = 0; i < I; i++ ) {
    for( int j = 0; j < J; j++ ) {
        //
        // Много текста
        //
    }// end for j
    for( int k = 0; k < K; k++ ) {
        //
        // Много текста
        //
    }// end for k
}// end for i
```

3.3.1. Комментарии

Для написания комментариев следует пользоваться правилами разметки, принятыми в система разметки исходного кода doxygen [4].

Система doxygen допускает применение различных стилей многострочных[1)] комментариев [5]:

1) Javadoc стиль:

```
/**
* ... text ...
*/
```

2) Qt стиль:

```
/*!
* ... text ...
*/
```

3) C++ стиль:

```
///
/// ... text ...
///
```

Разрешается использовать для многострочных комментариев вариант №3, то есть три косые черты `///`. Дольшинство IDE (например, QtCreator) поддерживают автосоздание тегов разметки для классов, методов и т.д.: достаточно поставить курсор выше комментируемой строки, набрать `///` и нажать клавишу «Enter».

Однострочный комментарий, идущий строго после текста, который он комментирует, должен оформляться как `///<`, то есть добавлением символа `<` к чертам. Без данного символа комментарий не будет распознан doxygen. Исключение — комментирование пространства имен, где `<` добавлять не следует

```
namespace MyNamespace /// Мое пространство имен
{
    ...
}// end namespace MyNamespace
```

[1)] Такие комментарии начинаются с новой строки (первого столбца строки).

Для комментариев, которые не предполагается извлекать при помощи doxygen, следует пользоваться двойной косой чертой `//`.

Комментарии вида `/*` и `*/` являются нежелательными. Целесообразно временно применять их для быстрого комментирования больших участков кода при отладке. В release-версии таких комментариев не должно быть.

Текст комментария всегда начинается с заглавной буквы, точка в конце не ставится. Если в комментарии несколько предложений, то точка между ними ставится, в конце — нет.

3.3.2. Стандартный заголовок файла

Следует использовать заголовок, размеченный согласно правилам doxygen. Минимально применимый заголовок:

```
1 ///
2 /// \file       myclass.h
3 /// \brief      Класс такой–то для того–то и того–то
4 /// \date       30.10.19
5 /// \author     Иванов И.И.
6 ///
```

Описание обязательно должно быть в `*.h` файлах, но может отсутствовать в соответствующих им `*.cpp` файлах. Заголовок начинается с первой строки любого файла, содержащего текст программы, и отделен от остального текста пустой строкой.

Следует уделять особое внимание описанию содержимого файла в стандартном заголовке. Doxygen предлагает большое количество тегов для разметки, наиболее используемыми являются:

1) \file — название файла;
2) \brief — краткое описание;
3) \data — дата создания файла;
4) \author — автор/авторы через запятую;
5) \param — параметр функции/метода;
6) \attention — сообщение на что обратить особое внимание;
7) \warning — предупреждение.

В описании под тегом `\remarks` или `\details` желательно указано назначение классов данного файла.

3.3.3. Комментирование исходного текста программы

Целью комментирования исходного текста программы является облегчение его понимания. Уместный и адекватный комментарий значительно упрощает сопровождение программы, и его важность несомненна.

Комментарии в разрабатываемых файлах пишутся на русском языке, исключение составляет legacy-код.

Следует помнить, что комментарии в `*.h` файле пишутся для человека, который будет использовать класс, а комментарии в `*.cpp` файле пишутся для человека, который будет поддерживать класс.

3.3.4. Что нужно комментировать?

Обязательно наличие комментария в следующих случаях:

1) Общий комментарий к содержимому файла в стандартном заголовке файла.
2) Объявление каждого класса в заголовочном файле.
3) Объявление каждого поля класса, включая **`private`** и **`protected`** поля.
4) Объявление каждого метода класса, включая **`private`** и **`protected`** методы. Исключением являются стандартные методы, такие как конструкторы копирования, конструкторы по умолчанию, деструкторы и операторы присваивания. Можно не комментировать методы получения/установки значения поля класса.
5) Объявление глобальных и статических переменных и функций.
6) Элементы перечислимых типов (`enums`).
7) Все нетривиальные решения в реализации функций и методов.

Если класс является реализацией какого-либо интерфейса, то можно не писать отдельный комментарий к каждому виртуальному методу этого интерфейса. Достаточно написать общий комментарий перед всей группой этих виртуальных методов.

Комментарии перед шаблонными функциями или классами должны содержать описание аргументов шаблона. Комментарии перед методами классов, операторами или функциями должны содержать описание параметров.

Комментарий должен описывать прежде всего назначение комментируемого класса, поля или метода. Комментарий не должен быть просто переводом на русский язык названия комментируемой сущности.

Категорически запрещается использовать комментарии для удаления кода. Если вы удаляете код, то надо его удалять, а не комментировать. Исключением может быть случай, когда вы намереваетесь использовать этот код, но по тем или иным причинам не можете сделать это немедленно. В таких случаях в начале кода должен идти текст на русском языке, в котором говорится, что это за код и где вы его намерены использовать.

При модификации методов нужно тщательно следить за тем, чтобы комментарии, особенно в заголовочных файлах, правильно отражали их семантику, особенности использования и реализации.

3.3.5. Определение классов

```
1  ///
2  /// \brief Класс моего окна
3  ///
4  class CMyWindow : public CWindow
5  {
6      DECLARE_MESSAGE_MAP() ///< Обявление того-то
7
8  public:
9      CMyWindow(); ///< Конструктор по-умолчанию
10
11     ///
12     /// \brief Параметрический конструктор
13     /// \param param1 – первый параметр
14     /// \param param2 – второй параметр
15     ///
16     CMyWindow( int param1, int param2 );
17
18     ~CMyWindow();
19
20     int MyMethod(); ///< Метод делающий то-то
21     void ConstantMethod() const; ///< Метод делающий сё-то
22
23 private:
24     int value; ///< Значение такое-то
25
26     int myMethod1(); ///< Метод делающий то-то
```

```
27      int myMethod2(); ///< Метод делающий сё-то
28  };
```

Порядок размещения разделов должен быть следующим: **public**, **protected**, **private**. В пределах каждого раздела сначала идут описания полей, потом пустая строка и описания методов. Между двумя разделами всегда стоит пустая строка. Все содержимое фигурных скобок, за исключением слов **public**, **protected**, **private**, выделено на одну табуляцию. Запрещается опускать ключевое слово **private** для классов только с приватными полями.

В описаниях методов сначала должны идти все конструкторы, затем деструктор. После описания деструктора, перед описанием следующего метода должна быть пустая строка.

Ключевое слово **const** в объявлении метода пишется на той же строке и отделяется пробелом от закрывающей круглой скобки.

Макросы типа DECLARE_MESSAGE_MAP и т.п. вставляются в начале блока до ключевого слова **public**.

3.3.6. Определение методов

```
1  int CMyClass::MyMethod()
2  {
3      int i = 0;
4      return i;
5  }
```

Тело метода сдвигается на одну табуляцию (4 пробела). Запрещается ставить открывающую скобку на одной строке с именем метода. Запрещается ставить точку с запятой после закрывающейся фигурной скобки тела метода.

Это правило касается также определения **inline** методов, сделанного вне класса.

Особый случай — определение конструктора:

```
1  CMyClass::CMyClass( int intParam, bool boolParam ) :
2      intField( intParam ),
3      boolField( boolParam )
4  {}
```

После закрывающей скобки через пробел ставится двоеточие и на следующих строках записываются выражения инициализации баз и членов. Сначала записываются выражения инициализации всех баз, а затем членов. Все выражения инициализации сдвинуты на одну табуляцию.

3.3.7. Определение inline методов

При определении **inline** методов возможен один из двух стилей форматирования:

```
1  class CMyClass
2  {
3      int myMethod1() { return myVar; }
4      int myBigMethod( int param );
5  };
```

и

```
1  inline int CMyClass::myBigMethod( int param )
2  {
3      assert( param > 0 );
4      return param * 2;
5  }
```

В первом случае после открывающей и перед закрывающей фигурной скобкой стоит пробел. Запрещается ставить точку с запятой после закрывающейся фигурной скобки тела метода.

При использовании первого метода для определения конструкторов выражения инициализации баз и членов записываются в строку сразу после закрывающей скобки параметров конструктора. Перед и после двоеточия ставится по одному пробелу.

В определении класса следует включать только однострочные **inline** методы. Если метод занимает несколько строк, то лучше определить этот метод в этом же заголовочном файле после определения класса, используя ключевое слово **inline**.

3.3.8. Оформление оператора «if»

```
if( !isOpen ) {
    DoOpen();
}

if( str.Length() > 0 ) {
    // Do smth
} else {
    // Do smth else
}
```

Внутренние блоки сдвинуты на табуляцию. Пробел не ставится между «if» и «(». Пробел ставится между «)» и «», до и после «else». Запрещается писать открывающие фигурные скобки на отдельной строке (кроме случая сложного условия), равно как и разносить «else» и прилегающие фигурные скобки на несколько строк.

Фигурные скобки необходимо обязательно использовать, даже если блок содержит только один простой оператор. Запрещается ставить точку с запятой после закрывающейся фигурной скобки.

```
if( hasObject() ) {
    processObject();
} else {
    skip();
}
```

Допускается написание цепных условий в следующем формате:

```
if( isspace( c ) ) {
    ProcessSpace();
} else if( isdigit( c ) ) {
    ProcessDigit();
} else if( isalpha( c ) ) {
    ProcessLetter();
} else {
    assert( false );
}
```

Этот способ применяется вместо использования оператора **`switch`** только для условий сложного вида, когда **`switch`** использовать невозможно. При этом нужно для всех блоков использовать фигурные скобки.

В случае сложного условия рекомендуется следующий способ записи оператора:

```
if( term1
    && term2
    && term3
    && term4 )
{
    DoSomething();
}
```

3.3.9. Оформление оператора «for»

```
for( int i = 0; i < array.Size(); i++ ) {
    Process( array[i] );
}
```

Внутренний блок сдвинут на табуляцию. Между «for» и «(» пробел не ставится, а между «)» и «» стоит пробел. Запрещается писать открывающую фигурную скобку на отдельной строке, за исключением случая сложного условия. Запрещается ставить точку с запятой после закрывающейся фигурной скобки. Если условие цикла не помещается на одной строке, то вторая и последующая строки условия сдвигаются на табуляцию, а открывающаяся фигурная скобка ставится на отдельной строке:

```
for( int i = 0; i < array.Size() && IsValid( array[i] )
    && CanProcess( array[i] ); i++ )
{
    Process( array[i] );
}
```

Фигурные скобки необходимо обязательно использовать, даже если блок содержит только один простой оператор. В вырожденном случае, когда тело цикла пустое, необходимо также пользоваться фигурными скобками:

```
for( int i = 0; i < array.Size() && IsValid( array[i] ); i++ ) {
    ...
}

for( int i = 0; i < array.Size() && IsValid( array[i] ); i++ ) {
}
```

3.3.10. Оформление оператора «while»

```
while( i > 0 ) {
    cout << i;
    i--;
}
```

Внутренний блок сдвинут на табуляцию. Между «while» и «(» пробел не ставится, а между «)» и «» стоит пробел. Запрещается писать открывающую фигурную скобку на отдельной строке, за исключением случая сложного условия. Запрещается ставить точку с запятой после закрывающейся фигурной скобки. Если условие цикла не помещается на одной строке, то вторая и последующая строки условия сдвигаются на табуляцию, а открывающаяся фигурная скобка ставится на отдельной строке:

```
while( CanProcess( i, object ) && i > 0
    && IsValid( i ) )
{
    Process( i, object );
    i--;
}
```

Фигурные скобки необходимо обязательно использовать, даже если блок содержит только один простой оператор. В вырожденном случае, когда тело цикла пустое, необходимо также пользоваться фигурными скобками:

```
while( ( *ptr1++ = *ptr2++ ) != 0 ) {
}
```

3.3.11. Оформление оператора «do ... while»

```
do {
    ret = Process();
} while( ret > 0 );
```

Внутренний блок сдвинут на табуляцию. Между «do» и «» и между «» и «while» стоит пробел. Фигурные скобки не опускаются, даже если тело цикла состоит из одного оператора. Запрещается писать фигурные скобки на отдельной строке.

3.3.12. Оформление оператора «switch»

```
1  switch( source ) {
2      case S_Scaner:
3          ScanImage();
4          break;
5      case S_Image:
6      case S_File:
7      {
8          CString name = getName();
9          ReadImage( name );
10         break;
11     }
12     default:
13         assert( false );
14 }
```

Метки **case** сдвинуты на табуляцию, а сам код сдвинут на две табуляции. Пробел после метки **case** перед двоеточием не ставится.

В конце секции обязательно необходимо ставить оператор **break**. Единственное исключение — когда несколько меток относятся к одной секции кода. Семантика оператора **switch** не должна зависеть от порядка расположения секций.

Если в какой-либо из секций содержится определение переменной, то данная секция должна быть заключена в фигурные скобки. Открывающаяся фигурная скобка ставится на отдельной строке после метки **case**. Закрывающаяся скобка ставится на отдельной строке после оператора **break**. Открывающаяся и закрывающаяся скобки сдвинуты на табуляцию относительно оператора **switch** и находятся на одном уровне с метками **case**.

Секция **default** всегда идет последней.

Запрещается ставить точку с запятой после закрывающейся фигурной скобки оператора **switch**.

3.3.13. Оформление лямбда-выражений

Лямбда-выражения представляют более краткий компактный синтаксис для определения объектов-функций. Необходимость использования лямбд следует согласовывать с руководителем проекта в каждом конкретном случае. Примеры оформления приведены ниже.

Простой пример:

```
struct CMyStruct
{
    int x, y;
    int operator()( int );
    void f()
    {
        [=]()->int {
            return operator()( this->x + this->y );
        };
    }
};
```

Пример с использованием внутри `std::function`:

```
std::function<arma::vec( const arma::vec &vectorA )> myMethod =
    [&]( const arma::vec &vectorA )->arma::vec {
        arma::vec res;
        // ...
        return res;
    };
```

Пример с использованием внутри `std::transform`:

```
void func( std::vector<double> &v, const double &e )
{
    std::transform( v.begin(), v.end(), v.begin(), [e]( double d )->double {
        if( d < e ) {
            return 0;
        } else {
            return d;
        }
    } );
}
```

Пример с использованием внутри `std::transform` и с переносом длинной строки:

```
1  void func( std::vector<double> &v, const double &epsilon )
2  {
3      std::transform( v.begin(), v.end(), v.begin(), // Перенос длинной строки
4          [epsilon]( double d )->double
5      {
6          if( d < epsilon ) {
7              return 0;
8          } else {
9              return d;
10         }
11     } );
12 }
```

3.3.14. Написание прочих выражений

Изложенное ниже касается также списка параметров метода и объявления переменных. Приведенные правила не охватывают всех аспектов, оставшиеся детали остаются на усмотрение программиста.

1) После запятой пробел ставится всегда, перед запятой — никогда.

2) Если после открывающей круглой скобки стоит пробел, то перед парной закрывающей скобкой тоже должен стоять пробел, и наоборот.

3) После «[» и перед «]» пробелы не ставятся.

4) Бинарные операции (кроме `->` `.` `::` `.*` `->*`) с двух сторон окружаются пробелами.

5) Унарные операции пишутся слитно с операндом.

6) Операция « ? : » пишется с пробелами вокруг «?» и «:»

7) В описании переменных «*» и «&» примыкают к переменной:

```
int *ptr;
int &var;
```

однако если после «*» и «&» должно стоять **`const`**, то это может записываться следующим образом:

```
int *const constPtr;
```

8) После типа стоит всегда один пробел.

3.3.15. Несколько классов в одном файле

Если в одном файле содержится объявления нескольких классов, их следует разделять строкой вида:

```
...
}; // end CSomeClass

//———————————————————————————————————————————————————————————————————————
///
/// \brief Другой класс
///
class CAnotherClass
{
...
```

Этот же разделитель используется, если в одном файле содержится реализация методов для нескольких классов. В этом случае он разделяет группы методов, относящихся к разным классам.

Разрешается использовать данный разделитель также для разделения текста на разные смысловые группы везде, где это уместно.

Длина строки-разделителя принимается равной 80 символов[1)]. Как правило до строки-разделителя оставляется пустая строка.

3.3.16. Порядок методов внутри файла

В начале файла, содержащего реализацию класса, должны располагаться конструкторы класса. Затем должен идти деструктор класса.

Методы, реализующие простые базовые операции с объектами класса, группируются в файле реализации и в объявлении класса по смыслу. Группировка таких методов в объявлении класса и в файле реализации должна быть одинаковой.

Методы, реализующие какие-либо нетривиальные алгоритмы, и вызываемые из этих методов приватные методы должны располагаться в файле реализации в порядке, соответствующем развернутому дереву вызовов.

[1)] 80 символов 12 шрифтом помещаются на одну строку листа формата А4 при полях справа и слева по 2 сантиметра.

Допускается разворачивать дерево вызовов как от методов верхнего уровня к методам более низкого уровня, так и в обратном порядке.

3.3.17. Обработка длинных строк

Строка считается длинной, если она не помещается в окно редактора IDE, развернутое на весь экран.

Длинная строка разбивается на две и более, причем вторая и последующие части сдвинуты на табуляцию.

В случае, если требуется распечатка на бумаге формата A4 файлов исходного кода «как есть», то для сохранения нумерации строк и вида исходного файла, следует ограничить длину строки в 80 символов (также см. 3.3.15).

Стоит отметить, что если распечатка на бумаге формата A4 файлов исходного кода «как есть» не предполагается (что разумно в современном мире), то при выборе длины строки следует ориентироваться на разрешение широкоформатного монитора (разрешение минимум 1440 на 900), при котором допустимы строки вплоть до 120 символов (с учетом бокового браузера файлов в IDE).

Примеры:

```
int ret = CreateDialog( GetApplicationObject()->MainWindow,
    filePath, nameDict, extensionsDict, currentFormat,
    dialogTitle );

for( const CWnd *wnd = GetFirst(); wnd != 0;
    wnd = wnd->GetNext() )
{
    wnd->EnableWindow( TRUE );
    wnd->ShowWindow( SW_SHOW );
}

int CMyClass::Func( int parameter1, int parameter2,
    int parameter3 )
{
}

CImageDialog::CImageDialog( CWnd *parent, int _format,
    const CString &_title ) :
    CDialog( parent, title ),
    format( _format )
{
}
```

3.3.18. Использование пустых строк

Пустая строка обязательно ставится между определениями (телами) методов и классов. Запрещается ставить несколько пустых строк подряд.

В остальных случаях расстановка пустых строк — дело вкуса и здравого смысла программиста. Не нужно разделять пустыми строками все операторы, но и не нужно слитно писать метод на два экрана. Расстановка пустых строк должна делить текст на логически связанные части и, таким образом, улучшать читаемость текста.

3.3.19. Шаблонные методы и классы (template)

```
1 template<class T>
2 class CArray : public CBaseArray<T>
3 {
4 }
5
6 template<class T>
7 void CArray<T>::Add( const T &anElem )
8 {
9 }
```

Внутри угловых скобок пробелы не пишутся. Исключением является случай, когда аргументом шаблонного класса является другой шаблонный класс. В этом случае пробел между закрывающимися угловыми скобками требуется компилятору для разделения лексем. В этом случае пробелы пишутся симметрично:

```
CPointerArray< CArray<CString> > myArray;
```

3.3.20. О декоративном форматировании

Существует распространенная практика, которой следует избегать. Заключается она в том, что тексту стремятся придать «красивый вид» путем выравнивания нескольких подряд идущих строк по вертикали.

Характерный пример:

```
1 CWindow *Func1    ( int    param1 );
2 int      Func2    ( long   param2 );
3 void     Function ( LPCSTR param4 );
```

Применение такого декоративного форматирования ухудшает читаемость программы и создает дополнительные сложности при редактирование текста. Применять такой стиль форматирования текста запрещается.

Одни из немногих случаев, когда в середине строки может стоять более одного пробела подряд — это описание инициализаторов для двумерных таблиц и написание inline ассемблера, где эта практика применяется традиционно. Во всех остальных случаях в середине строки более одного пробела подряд стоять не может.

3.4. Правила программирования

3.4.1. Выбор типа переменной

При выборе типа идентификатора необходимо пользоваться следующими соображениями:

1) если есть стандартный тип языка C++ подходящий для реализации переменной, то нужно им и пользоваться, а не изобретать typedef;

Разрешены типы:

void, **bool**, **char**, **wchar_t**,

short, **int**, **long**,

float, **double**.

Разрешены дополнительные типы:

int8_t, **int16_t**, **int32_t**, **int64_t**,

uint8_t, **uint16_t**, **uint32_t**, **uint64_t**.

2) **short** рекомендуется использовать, например, при желании сэкономить память для размещения переменной. Если при этом нельзя гарантировать, что значение переменной ни при каких условиях не выйдет за границы, допустимые для переменных типа **short**, необходимо менять **short** на **long** или **int**;

3) типы **int8_t**, **int16_t**, **int32_t** используются в случае, когда нужно явно указать количество байт (например при работе с внешним интерфейсом, файлами);

4) нельзя пользоваться беззнаковым целочисленным типом переменных без крайней на то нужды;

5) запрещено использовать тип `size_t`, кроме случаев, обусловленных стандартом языка, например, при определении оператора **new**.

3.4.2. Стандартный интерфейс метода (использование const и void)

Интерфейс метода должен отображать характер работы метода с аргументами.

Если метод модифицирует входной аргумент, переданный по указателю или ссылке, или метод изменяет **this**, то он должен возвращать **void** или **bool** (только для возврата информации об успешном завершении метода).

Если метод не изменяет входной аргумент, переданный по указателю или ссылке, или метод не изменяет **this**, обязательно следует употреблять описатель **const**. Этот описатель показывает, как метод использует аргумент и, таким образом, облегчает чтение. Кроме того, **const** гарантирует от неправильного использования входных данных.

Если метод не **const** и необходимо выбрать: вернуть ли объект или использовать ссылку на модифицируемый объект и вернуть **void**, то необходимо использовать 2-й способ.

Пример 1: `CString CString::MakeUpper()const;`

Пример 2: `void CString::MakeUpper();`

Пример 3: `CString &CString::MakeUpper(); //Ошибка стиля`

Напомним, что описатель **const** также должен использоваться для полей класса, значения которых инициализируются в конструкторе и не меняются за время жизни объекта.

Особые правила относятся к методам контейнера, возвращающим ссылку на объект, находящийся внутри контейнера. Рекомендуется определить два метода с одинаковым именем: константный метод возвращает константную ссылку на объект, а не константный метод возвращает не константную ссылку.

Пример:

```
1  class CMyContainer
2  {
3      CMyObject &GetObject( int index );
4      const CMyObject &GetObject( int index ) const;
5  };
```

Если константный по сути метод тем не менее модифицирует объект, например, вычисляя некоторые данные по требованию и запоминая их для последующего использования, то метод нужно объявлять как **const**, а модифицируемые поля — как **mutable**.

Пример:

```
enum TBloodType
{
    BT_A,
    BT_B,
    BT_AB,
    BT_O,
    BT_Unknown
};

class CPerson
{
public:
    TBloodType GetBloodType() const;

private:
    mutable TBloodType bloodType;
    TBloodType calculateBloodType() const;
};

inline TBloodType CPerson::GetBloodType() const
{
    if( bloodType == BT_Unknown ) {
        bloodType = calculateBloodType();
    }
    assert( BT_A <= bloodType && bloodType <= BT_O );
    return bloodType;
}
```

3.4.3. Инициализация переменных и полей

Следует всегда инициализировать локальные переменные и поля классов тех типов, которые не имеют специально созданных для этой цели конструкторов. Инициализировать локальные переменные нужно сразу при их объявлении, а поля классов во всех конструкторах этого класса.

Пример:

```
struct CStruct
{
    int Field1;
    IObject *Field2;
    CPtr<IObject> Field3;
    LOGFONT Field4;

    CStruct()
    {
        Field1 = 0;
        Field2 = 0;
        memset( &Field4, 0, sizeof( LOGFONT ) );
    }
};
```

За этим необходимо аккуратно следить, т.к. компилятор выдаёт предупреждение об использовании неинициализированных переменных только в самых простых случаях. При этом неинициализированное поле класса может стать причиной ошибки, которую будет нелегко воспроизвести.

3.4.4. Использование структур

Структуры следует использовать в тех случаях, когда для них не определены никакие **private** и **protected** методы и поля, нет виртуальных функций и функций со сложной семантикой и не предполагается наследование. У структур могут быть явные конструкторы, но не должно быть явно описанного деструктора. У структур должны быть автоматически сгенерированные компилятором конструктор копирования и оператор присваивания.

Структуры применяются вместо классов чтобы сообщить программисту, который будет читать программу, что объект имеет простую семантику и его использование не влечёт скрытых накладных расходов.

Пример:

```
struct CMyData
{
    int Field1;
    int Field2;
};
```

Запрещено инициализировать структуры, используя список инициализаторов, без согласования с руководителем проекта.

Пример плохого стиля: `CMyData data = { 1, 2 };`

Ограничение вызвано тем, что тяжело вручную поддерживать контроль соответствия данных и полей структуры. Если у структуры появляется новое поле, инициализация становится ошибочной.

При необходимости рекомендуется создать конструктор (с возможностью контроля данных) или метод инициализации, если есть массивы данных такого типа и по соображениям эффективности конструктор определять нежелательно.

3.4.5. Проверка значений указателей

Язык C++ имеет выделенное значение для указателей: 0. Поэтому если в программе необходимо проверить равенство указателя нулю, это следует делать в виде явной проверки `ptr != 0`. Запрещается использовать константу `NULL` или использовать указатель как булевское значение. Лучше использовать ключевое слово `nullptr`.

Пример правильной проверки:

```
if( ptr != 0 )
```

Примеры неправильных проверок:

```
if( ptr != NULL )
if( !ptr )
```

3.4.6. Использование адресов временных объектов

Согласно стандарту C++ временные объекты живут до конца вычисления выражения:

```
1  // Безопасно
2  MyFunc( ( str1 + str2 ).Ptr() );
3  MyFunc( GetText().Ptr() );
4
5  // Ошибка
6  return( str1 + str2 ).Ptr();
```

3.4.7. Область видимости переменной цикла «for»

Запрещено полагаться на область видимости локальной переменной, определённой в операторе инициализации оператора **for**.

Если всё-таки требуется использовать переменную цикла после оператора **for**, необходимо её определить до оператора цикла:

```
1 int i = 0;
2 for( ; i < a.Size(); i++ ) {
3     ...
4 }
```

3.4.8. Преобразование типа

В языке C++ введён синтаксис для явного преобразования типа при помощи ключевых слов **static_cast**, **const_cast**, **reinterpret_cast**, **dynamic_cast**. Для явного преобразования типа следует всегда пользоваться этими ключевыми словами. Использовать явные преобразования типа в стиле языка C запрещается. Это позволяет избежать следующих ошибок:

1) Снятие константности. Использование в этом случае **const_cast** позволяет избежать ошибочного преобразования к указателю (ссылке) на объект другого типа.

2) Преобразование указателя (ссылки) от базы к потомку. Компилятор не диагностирует ошибочное использование преобразования типа в стиле языка C, когда потомок объявлен, но не определён. В этом случае такое преобразование работает как **reinterpret_cast**, что приводит к ошибке, когда предок является не первой базой потомка. Использование **static_cast** позволяет избежать подобной ошибки.

3.4.9. О длине тела метода

Методы должны содержать операторы, выполняющие однородные действия.

Пример «плохого» кода:

```
1  void CMyClass::f()
2  {
3      for( int i = 0; i < 1000; i++ ) {
4          // Действия по инициализации
5      }
6      while( condition() ) {
7          // Содержательные действия
8      }
9      for( int i = 0; i < 1000; i++ ) {
10         // Действия по очистке
11     }
12 }
```

Пример «хорошего» кода:

```
1  void CMyClass::f()
2  {
3      init();
4      doSomething();
5      cleanUp();
6  }
```

При этом названия методов более низкого уровня должны объяснять их семантику, а если семантика нетривиальна — должны быть исчерпывающие комментарии.

3.4.10. Использование логических переменных

При проверке логического условия не следует сравнивать с **`true`**.

Пример «плохого» кода:

```
1  if( x == true ) {
2      ...
3  }
```

Пример «хорошего» кода:

```
1  if( x ) {
2      ...
3  }
```

При присваивании значения булевской переменной или возврате булевского значения из функции запрещается использовать оператор `?` с вариантами результата **`true`** и **`false`**. Т.е. нужно писать:

```
bool value = x > 0;
```

а не:

```
bool value = x > 0 ? true : false;
```

3.4.11. Деструкторы

Поскольку при исключении происходит свертка стека и вызов деструкторов автоматических объектов, деструкторы не должны генерировать исключения, по крайней мере в release-версии. Поэтому в деструкторах нельзя делать никаких сложных действий, могущих вызвать исключения. Кроме того, деструкторы не должны полагаться на состояние объекта или системы и на порядок вызова других деструкторов. Основная задача деструктора — освободить ресурсы, занятые объектом, и оставить систему в корректном состоянии.

Рекомендации:

1) Отладочные проверки в деструкторах можно делать только макросом `presume` (см. раздел 4.4.1).

2) Объект, владеющий ресурсами, например, памятью или открытыми файлами, должен иметь указания, какими он ресурсами владеет в данный момент. Для этого можно использовать выделенные значения (0 для указателя) либо флаги.

3) При конструировании объекта все указатели должны быть инициализированы адресами существующих объектов либо нулем. При удалении объекта при помощи **`delete`** указателю на объект нужно присвоить нуль.

4) Если в деструкторе приходится производить нетривиальные действия по освобождению ресурсов, которые могут приводить к генерации исключений, нужно поставить перехватчик исключений и (обычно) выдать пользователю сообщение об ошибке.

3.4.12. Битовые поля

В некоторых случаях для уменьшения размера структуры используются битовые поля. Следует помнить, что если битовое поле используется для знакового типа, то один бит отводится под знак. Особое внимание следует обратить на битовое поле целочисленного знакового типа, например **`int`**, **`short`**. Знаковое битовое поле размером один бит может представить только два значения: 0 и -1, и не может представить значение равное 1. Для булевских битовых полей следует использовать стандартный тип **`bool`**.

Запрещается использовать битовые поля для перечисляемых типов.

Вопрос об использовании битовых полей находится в компетенции руководителя проекта.

3.5. Опасные приемы

Опасными называются приемы, приводящие к серьезным ошибкам, разрушающим программу: выходу величины за границы диапазона, обращению к неаллокированной памяти и т.п. Серьезные ошибки подобного рода не всегда легко отлаживаются и совершенно недопустимы в готовой программе. В оправдание «опасных» приемов и против защитного стиля иногда приводится довод о неэффективном коде, порождаемом защитным стилем. Удешевление разработки программ и существенное повышение надежности полностью оправдывают незначительное снижение скорости и умеренный рост объема кода. Наиболее часто встречающиеся опасные приемы перечислены ниже.

3.5.1. Использование буферов в памяти как массивов

В языке С нет другого типа массивов, чем буфера в памяти. При индексации буфера в памяти не делается проверок на выход индекса за границы буфера. Еще опаснее использовать реаллокируемый буфер как динамический массив. Использование функций работы с блоками памяти из стандартной библиотеки делает программу почти не верифицируемой и очень ненадежной. Везде, где это возможно, следует использовать контейнеры стандартной библиотеки std.

3.5.2. Использование макросов

Запрещается создавать новые группы связанных друг с другом макросов. Запрещается создавать макросы вместо inline-функций, шаблонов или констант.

3.5.3. Использование в программе явных числовых значений

Запрещено использовать явные числовые константы кроме 0 и 1. Большинство нужных констант целой арифметики описаны в limits.h. Всем остальным константам в программе должны присваиваться символические имена с помощью **enum** (сам **enum** может быть неименованным) или используя ключевое слово **const**.

Из этого правила есть важное исключения: не нужно присваивать константе символическое имя, если выполняются следующие условия

1) Константа используется ровно в одном месте (константа не связана функциональными зависимостями с другими константами).

2) Константа имеет смысл только в контексте окружающего выражения.

В этом случае смысл константы обязательно должен быть описан комментарием.

3.5.4. Код, маскирующий ошибку

Запрещается вместо проверочного кода использовать код, маскирующий ошибку. Предположим, функция работы со строками в смысле C не должна получать нулевой указатель. Следовательно, одним из предусловий, проверяемых с помощью `assert` (см. ниже), должно быть неравенство этого указателя нулю. Примером маскирования ошибки, будет функция, которая в случае равенства указателя нулю не будет делать ничего. Подобная практика есть грубейшее нарушение производственной дисциплины.

3.5.5. Выход из аварийной ситуации с минимальными потерями

Необходимо учитывать, что реализация `assert` в ряде проектов не прерывает выполнения программы. Следовательно, при срабатывании проверочного условия, после `assert` в ряде случаев должен располагаться код, который бы позволил системе выйти из сложившейся ситуации с минимальными потерями.

Кода обработки аварийных ситуаций должен быть нацелен на поддержание системы в работоспособном состоянии. Например, если сбой произошёл в системе выдачи информации оператору, то достаточно выдать признак, что данные некорректны. Если сбой произошёл в одном из каналов многоканальной системы, то допустимо в крайних случаях вместо поступивших некорректных данных использовать какие-то значения по-умолчанию. Аварийный код должен быть максимально простым, чтобы не усугубить ситуации.

Приведём ряд примеров.

1) Функции, работающие с массивами данных, в случае получения слишком больших объемов данных должны обрабатывать столько данных, сколько могут, или не делать ничего:

```
1  void procData( const void *data, int size )
2  {
3      if( size > MaxSize ) {
4          assert( false );
5          size = MaxSize;
6      }
7      doSomething( data, size );
8  }
```

2) Функции, работающие со строками, в нештатных случаях должны возвращать пустую строку:

```
1  const char *GetStateName( TPrepareState state )
2  {
3      switch( state ) {
4          case PS_NoDevice:
5              return "NoDevice";
6          case PS_BrokenChannel:
7              return "Broken";
8          ...
9          default:
10             assert( false );
11             return "";
12     }
13 }
```

3) Функции, возвращающие ссылку на объект по его идентификатору, в случае получения некорректного идентификатора после `assert` должны возвращать ссылку на специально предусмотренный «мусорный» объект:

```
1  static CTarget target[MaxTarget + 1];
2  CTarget &GetTarget( int iTarget )
3  {
4      if( iTarget >= 0 && iTarget < MaxTarget ) {
5          return target[iTarget];
6      } else {
7          assert( false )
8          return target[MaxTarget];
9      }
10 }
```

В случае ошибок, которые могу быть вызваны только общесистемными проблемами, писать аварийный код не следует. Например, если функция получает данные через указатель и этот указатель в принципе не может быть равен 0, то проверять указатель на ноль не имеет смысла. Первое обращение к этому указателю приведёт к генерации исключения, что позволит быстро найти ошибку.

3.5.6. Специальные значения

Одним из признаков плохо спроектированного интерфейса является наличие у методов параметров, специальные значения которых сильно меняют семантику этих методов. Также вредны специальные возвращаемые значения. Специальные значения плохи тем, что делают семантику интерфейса крайне запутанной, и это приводит к ошибкам.

Использование специального значения параметра оправдано в редких исключениях и должно в каждом случае специально обосновываться. Избежать этого можно, к примеру, введением дополнительных входных параметров или дополнительных методов.

Использованию специальных возвращаемых значений следует предпочесть систему обработки ошибок либо введение дополнительных выходных параметров.

3.5.7. Управление динамической памятью

Запрещается использовать какие-либо средства распределения памяти кроме **`new`** и **`delete`** или умных указателей (приоритетнее).

3.5.8. Функции с переменным числом параметров

Функции с переменным числом параметров (например, функция `printf`) обычно определяют количество своих параметров и их тип в результате анализа значения других параметров этой функции. Контроль типа параметра компилятором отсутствует. Это может приводить к ошибкам, которые проявляются только во время выполнения программы. Использовать функции с переменным числом параметров запрещается без разрешения руководителя проекта.

3.5.9. Функции scanf, strcpy, strcat

Применение этих функций часто приводит к порче памяти, так как невозможно проконтролировать отсутствие переполнения буфера, в который копируется строка. Эти функции использовать запрещено.

Для работы со строками нужно использовать класс `std::string`. В случае крайней необходимости работы с низкоуровневыми строками нужно использовать функцию `strncpy`.

3.5.10. Оператор goto

Использование оператора **`goto`** является классической темой для holy war.

В некоторых языках, например FORTRAN, использование данного оператора полностью оправданно.

В языке C++ использование оператора **`goto`** может быть оправданно только в нескольких случаях: досрочном прерывании нескольких вложенных циклов и использования legacy-кода. В обоих случаях показан code-review. Запрещается использовать оператор **`goto`** в иных случаях.

3.5.11. Целая арифметика

Опасна беззнаковая арифметика, где переполнение происходит в области малых по модулю чисел. Вычитания беззнаковых чисел следует избегать, а если нельзя — производить крайне осторожно, с предварительным сравнением. Можно также преобразовывать беззнаковый тип в знаковый большего размера, делать вычисления, а при обратном преобразовании производить проверку на диапазон.

3.5.12. Использование ассемблера

Использование ассемблера относится к опасным приемам программирования. Оно допускается только при обоснованной необходимости оптимального кода или невозможности достичь требуемого результата высокоуровневыми средствами. Кроме того, написать на ассемблере процедуру, более эффективную, чем соответствующая процедура на C, достаточно трудно.

3.5.13. Формулировка условий цикла

Следует очень тщательно подходить к формулированию условий циклов. В качестве условия цикла следует использовать логическое выражение, которое

выполняется только для допустимых значений переменной цикла. Например, не следует вместо `i < 100` писать `i != 100`, т.к. такое условие может привести к ошибке, если после каждой итерации цикла значение переменной цикла увеличивается на отличное от единицы число.

3.5.14. Использование статических объектов

Использования сложных статических объектов следует избегать по следующим причинам:

1) Зависимости между различными статическими объектами приводят к тому, что работа программы зависит от порядка вызова конструкторов и деструкторов этих объектов, а этот порядок для объектов из разных модулей не определен. Наличие зависящих друг от друга статических объектов практически всегда является следствием ошибок в архитектуре системы.

2) о время вызова конструкторов и деструкторов статических объектов затруднена обработка исключений. Исключение, сгенерированное в конструкторе или деструкторе статического объекта и не перехваченное в этом конструкторе или деструкторе, приведет к завершению приложения без адекватной диагностики.

Поэтому разрешается использовать статические объекты только простых типов с тривиальными конструкторами и деструкторами. Также разрешается использовать глобальные статические объекты библиотечных классов, таких как `std::string`. Использовать статические объекты пользовательских типов с нетривиальными конструкторами или деструкторами и статические массивы из таких объектов запрещается. Данный запрет относится к статическим и глобальным объектам, статическим полям классов и статическим переменным внутри функций.

Категорически запрещается определять статические переменные нетривиальных, в том числе библиотечных, типов внутри функций.

3.5.15. Использование деструктора массива

После использования оператора **new** для создания массива, необходимо пользоваться оператором **delete** для массива.

Пример:

```
char *p = new char[20];
...
delete[] p;  // Нельзя использовать delete p;
```

3.5.16. Использование шаблонов

С помощью шаблонных классов и функций довольно легко создать код, трудный для понимания и отладки. Поэтому создание собственных шаблонных классов и функций, не входящих в стандартные библиотеки, допускается только с разрешения руководителя проекта.

4. СТИЛЬ ПРОГРАММИРОВАНИЯ

4.1. Использование стандартных программных средств

Использование стандартных библиотечных средств облегчает процесс разработки, уменьшает число ошибок и повышает понятность кода. Кроме того, оно сильно облегчает перенос кода между различными операционными системами и вычислительными средствами.

Рекомендуется применять следующие стандартные средства и средства из состава репозитория операционной системы:

1) стандартная библиотека `std::`;
2) библиотека Boost [6];
3) библиотека Armadillo (работа с матрицами) [7].

4.2. Правила разработки классов на C++

4.2.1. О минимальной достаточности классов

Не следует пытаться разрабатывать чрезвычайно общие классы. Практика показывает, что разработать удачный класс с достаточно общей семантикой сложно даже для опытного разработчика. Общие классы обычно имеют сложную, плохо определенную семантику и крайне смутные правила использования.

Разработка прикладного класса должна начинаться с четкого определения целей и условий его использования. После этого разрабатывается семантика (описание методов) класса, реализующего поставленную задачу. Набор методов класса должен быть минимально достаточен. Необходимо, чтобы описание класса содержало не только семантику, но и правила использования в виде примеров с подробными и ясными комментариями.

4.2.2. Использования наследования

Наследование — сложное в использовании средство, поэтому на практике им легко злоупотребить. Излишнее использование наследования приводит к сложным иерархиям классов, в которых нелегко понять семантику отдельного класса и решить, в каком классе нужно реализовать тот или иной метод.

Можно выделить четыре случая правильного использования наследования.

1) **Конкретизация при классификации**. Это основное использование наследования. Примером может служить наследование класса `CTeacher` от `CPerson`. Наследование при конкретизации практически никогда не бывает множественным, поскольку сложные классификации, действительно требующие множественного наследования, встречаются очень редко. Очень желательно, чтобы предок-надкласс был первым предком.

2) **Использование библиотечных средств**. Распространенный способ использования библиотеки классов — наследование потомка из абстрактного библиотечного предка.

3) **Реализация интерфейсов**. Если интерфейс задан абстрактным классом, то объект, реализующий этот интерфейс, наследуется из абстрактного класса — интерфейса и реализует его виртуальные методы. Если объект реализует несколько интерфейсов, что бывает достаточно часто, используется множественное наследование. Если у класса есть обычный предок и предки — интерфейсы, то обычный предок должен быть первым предком, а интерфейсы — вторыми.

4) **Использование механизма**. Использование механизма путем наследования отличается от предыдущих случаев тем, что там наследование было открытое (**`public`**), а здесь — приватное (**`private`**) или защищенное (**`protected`**). Это вызвано тем, что механизм есть деталь реализации, несущественная для клиентов класса. Механизмы можно делать полями класса, но у наследования есть два преимущества:

– синтаксическая краткость;

– возможность переопределить виртуальные методы механизма и дать этим методам доступ к внутренним данным класса.

4.2.2.1. Разрешение конфликта имен при множественном наследовании

При реализации интерфейсов с помощью множественного наследования иногда возникает конфликт имен, когда методы разных предков имеют одинаковые имена и параметры. При этом методы могут иметь разную семантику, а в потомке их нельзя переопределить по-разному. Конфликт имен решается путем введения промежуточных предков:

```
class A
{
public:
    virtual int f();
};

class B
{
public:
    virtual int f();
};

class A_in_C : public A
{
public:
    virtual int f() { return A_f(); }
    virtual int A_f() = 0;
};

class B_in_C : public B
{
public:
    virtual int f() { return B_f(); }
    virtual int B_f() = 0;
};

class C : public A_in_C, public B_in_C
{
    virtual int A_f() { ... }
    virtual int B_f() { ... }
};
```

4.2.2.2. Виртуальное наследование

Виртуальное наследование нужно, чтобы предок при повторном наследовании был в потомке в единственном экземпляре. Данная ситуация известная также под названием ромбовидного наследования или «алмаза смерти» (diamond of death):

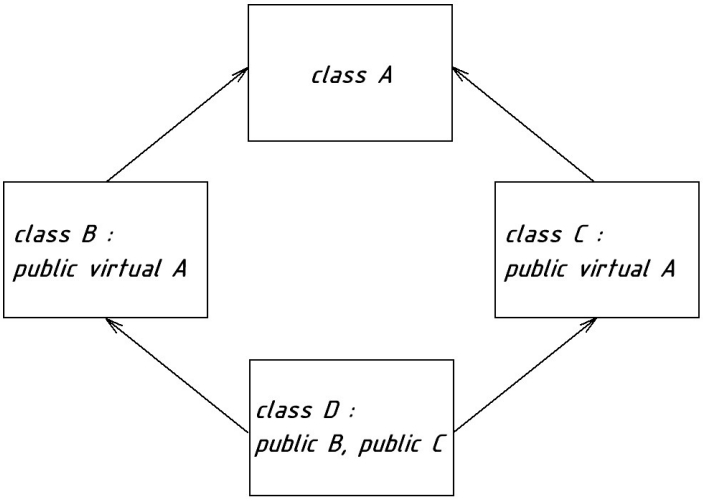

Рисунок 1 – Пример виртуального наследования

В приведенном примере класс D будет содержать одну копию класса A. Причины, по которым такое наследование названо «виртуальным», а не, например, «разделяемым» (shared), остаются за кадром. В стандарте принято ключевое слово **`virtual`** для такого наследования.

В большинстве случаев использование виртуального наследования не оправдано. В качестве примеров разделяемого наследования можно привести следующие иерархии:

1) Типы окон:

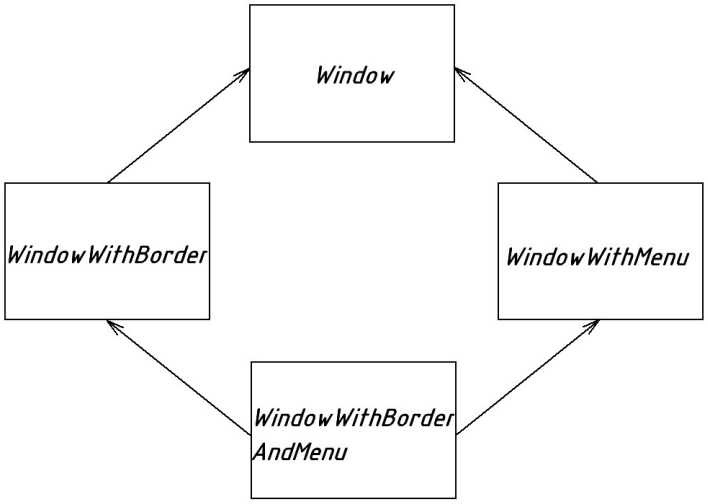

Рисунок 2 – Иерархия классов типов окон

2) Персонал университета:

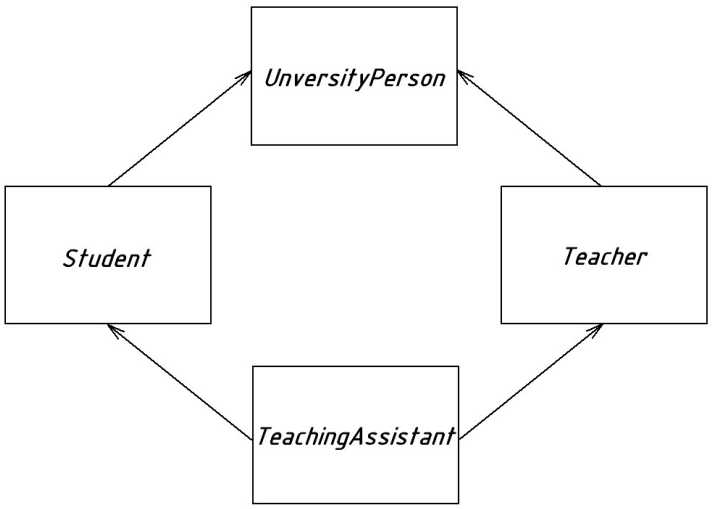

Рисунок 3 – Иерархия классов персонала университета

Первый случай — просто ошибка проектирования, которая приводит к серьезным проблемам в реализации например метода Draw потомка. Естественная реализация, а именно последовательный вызов соответствующих методов предков, приводит к тому, что само окно рисуется дважды. Гораздо проще сделать рамку и меню полями окна, и отказаться от наследования вовсе.

Во втором случае применение виртуального наследования кажется более оправданным. Тем не менее существует достаточно простой альтернативный подход:

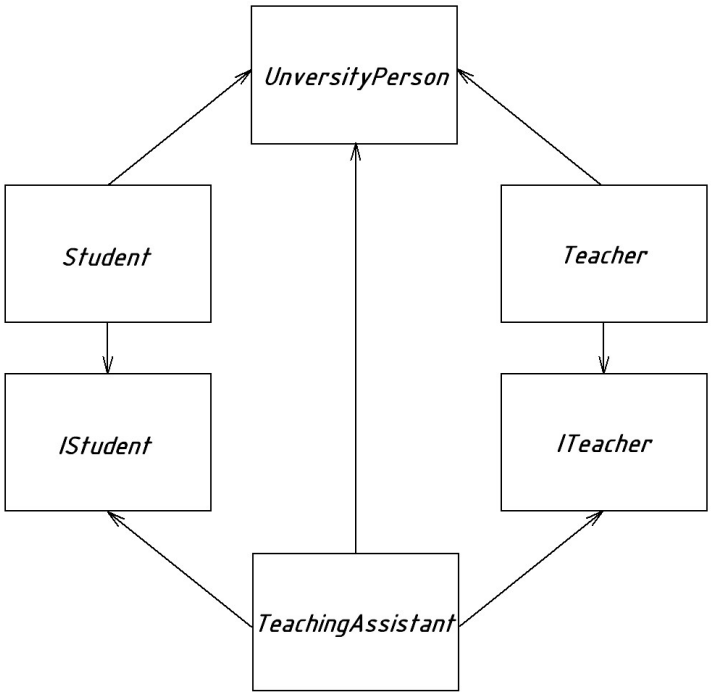

Рисунок 4 – Иерархия классов персонала университета с применением интерфейсов

Здесь `IStudent` и `ITeacher` — интерфейсы, частично содержащие реализацию. Хотя сущностей получается больше, не возникает технических проблем, неизбежных при разделяемом наследовании.

4.2.3. О конструкторе копирования и операторе присваивания

Если класс использует **`new`** для аллокации данных, у него должен быть описан конструктор копирования и оператор присваивания. Если, исходя из семантики класса, у него не должно быть конструктора копирования либо оператора присваивания, такой метод нужно описать как **`private`**, не определяя его. При этом нужно написать соответствующий комментарий.

4.2.4. О виртуальном деструкторе

У каждого класса, имеющего виртуальные функции, должен быть описан виртуальный деструктор. Класс, который может быть вторым предком другого класса, обязательно должен иметь виртуальный деструктор.

Исключением из этого правила являются интерфейсы в стиле COM с подсчётом ссылок, у которых традиционно нет виртуальных деструкторов.

4.2.5. О переопределении операций

Если у класса переопределяется оператор, нужно также переопределить близкие операторы.

Если у класса определён оператор `==`, нужно также определить и оператор `!=`. Если у класса переопределён **`operator`** `*`, нужно также переопределить и **`operator`** `->`.

Семантика переопределенных операторов должна быть подобна семантике встроенных операторов для базовых типов.

4.2.6. Использование explicit конструкторов

Если в классе определен конструктор с одним параметром, то этот конструктор может использоваться для неявного преобразования типа. В некоторых случаях это может привести к нежелательным последствиям. Например, у класса `CFile` может быть конструктор от строки, задающей имя файла. Этот конструктор открывает файл. Если неявное преобразование разрешено, то в функцию, получающую в качестве аргумента константную ссылку на файл можно будет передавать строковые константы. Такое преобразование затрудняет понимание кода программы и может быть источником ошибок. Для запрета неявного преобразования необходимо объявлять конструктор как **`explicit`**.

Конструкторы, выполняющие какие-либо нетривиальные действия (например, открытие файла), обязательно нужно объявлять **`explicit`**.

4.3. Использование ссылок и указателей

Отношение владения между объектами всегда описывается при помощи указателя. Не существует владеющей ссылки.

Если объекту нужно иметь доступ к другому объекту, который за время жизни первого объекта не будет уничтожен или заменен на другой, то нужно запоминать в первом объекте константную ссылку или константный указатель на второй объект. Это будет указывать на неизменность второго объекта за время жизни первого.

Если объект, на который указывают, нужно заменять, в качестве поля нужно использовать указатель, а методы доступа могут возвращать ссылку.

4.3.1. Использование указателей и ссылок в параметрах функции

Если в качестве параметра не нужно передавать нулевой указатель для обозначения пустого объекта, лучше использовать ссылку. Это облегчает чтение кода функции и позволяет избежать излишней проверки на равенство указателя нулю.

Ссылка, передаваемая в качестве параметра функции, может быть константной и не константной. Не константная ссылка на объект используется в качестве параметра функции только в том случае, если функция в соответствии со своей семантикой модифицирует этот объект.

Если такой объект не может модифицироваться функцией, то для передачи его в качестве аргумента функции следует использовать константную ссылку либо передавать его по значению. При этом атомарные типы следует всегда передавать по значению. Короткие структуры размером до четырёх байт с тривиальным конструктором копирования также лучше передавать по значению, если структура длиннее четырёх байт или имеет нетривиальный конструктор копирования, её следует передавать по константной ссылке.

В общем случае следует руководствоваться следующим правилом: атомарные типы всегда передаются по значению, структуры (классы) передаются

по константной ссылке. Неконстантная ссылка используется только в том случае, если функция использует параметр в качестве одного из возвращаемых значений.

4.4. О защитном стиле программирования

Защитным называется стиль программирования, направленный на облегчение тестирования и отладки и повышение надежности программы. Чтобы программировать защищено, нужно избегать опасных приемов программирования и использовать в нужных местах проверочный код.

4.4.1. Что такое assert

Для различных проверок во время выполнения программы используется макрос `assert`, параметром которого является логическое выражение. В момент выполнения программы это выражение вычисляется, и если получен результат **`true`**, то ничего не происходит и выполнение продолжается дальше. Если же получено значение **`false`**, то происходит аварийный останов и на экране печатается имя файла и номер строки, где произошла ошибка.

Столь простая команда приводит к сокращению времени отладки программы на порядок и многократному облегчению тестирования, поскольку правильность работы программы контролируется не только человеком по результатам работы, но и самой программой изнутри в течении всего времени ее работы.

4.4.2. Принцип взаимного недоверия

Каждый модуль для своего функционирования ожидает определенного поведения со стороны его окружения. Основной принцип состоит в том, что модуль проверяет результаты всех взаимодействий с другими единицами программы. Часто можно слышать оправдания вроде «Я сам написал тот модуль и я знаю, что в нем нет ошибок» или «Я знаю, что тот модуль проверяет результаты своей работы» или «Код становится слишком раздутым». Первые два возражения говорят о лености и о непонимании принципов построения надежной программы, а последнее почти полностью снимается системой обработки ошибок с помощью исключений (взамен обычных кодов возврата).

4.4.3. Где ставить assert?

В ходе написания метода, модуля, класса делается много явных и еще больше неявных предположений о состоянии внешней среды. Нужно наиболее полно уяснить себе, в каких предположениях работает данная часть кода, и явным образом отразить это в программе с помощью `assert`. В спецификациях методов нужно в явном виде указывать условия, которые должны быть выполнены перед вызовом, а в начале тела метода с помощью `assert` проверять эти условия. Не следует забывать проверять не только входные параметры методов, но и внешние переменные, поля объекта и другие компоненты окружения, используемые методом.

Таким образом рекомендуется ставить `assert` в следующих случаях.

1) Для проверки входных параметров функции/метода и других компонент окружения, используемых данной функцией/методом.

2) Если вызывается внешний для данного класса метод и от него ожидается какой-то определенный результат, проверять соответствие полученного ожидаемому. Особенно это актуально для внешних по отношению к разрабатываемой единице методов (другая подсистемы программы, операционная система). Следует явно формулировать постусловия методов и проверять их перед возвратом из метода.

Количество проверок, которые можно вставить в код программы, огромно. Проверки могут быть как тривиальными, так и очень сложными. Всегда необходимо находить разумный компромисс между числом проверок и степенью защищенности программы от ошибок.

К сложным проверкам, требующим существенных вычислений и объема кода, следует прибегать только в редких случаях, поскольку такие проверки сами служат источником ошибок.

Наиболее полезны тривиальные проверки, поскольку они почти не влияют на эффективность, не вносят дополнительных ошибок и достаточно хорошо обнаруживают сбои в программе.

Для проверок иногда полезны избыточные данные, например поля, указывающие на состояние объекта.

Код, исправляющий последствия ошибки, очень трудно отлаживается, и почти всегда сам содержит ошибки. Поэтому этот код не должен быть слишком сложен и должен делать только самые необходимые действия.

4.4.4. Assert и presume

Макрос `presume` в отличие от `assert` отключается в окончательной версии программы для повышения быстродействия. Можно указать несколько правил, когда ставить `assert`, а когда `presume`.

1) В некритических по быстродействию местах нужно ставить `assert`.
2) Если нарушение условия приводит к разрушению программы, например, выход индекса за границы массива при записи, нужно ставить `assert`.
3) Если условие служит отладочным целям, и нельзя стопроцентно гарантировать его выполнения, нужно ставить `presume`.
4) В деструкторах нужно использовать `presume`.

4.5. Классы как типы данных и механизмы

Существуют две разновидности классов объектов: типы данных и механизмы.

Тип данных представляет собой реализацию сущности предметной области с некоторым набором операций. Семантика операций может быть описана в виде набора аксиом. Тип данных не имеет внутреннего состояния в том смысле, что любая операция может быть применена к объекту данного типа в любое время.

Механизм представляет собой реализацию некоторого алгоритма. Обычно механизм имеет три основных типа операций: инициализацию, выполнение работы и извлечение результатов. При этом операции выполняются в строгой последовательности: механизм создается, инициализируется, работает, из него считывают результаты, он уничтожается. Например:

```
1  class CSpaceFinder
2  {
3  public:
4      CSpaceFinder( const Cline &line ); // Объект рассчитан на
5          // однократное использование, инициализация в
6          // конструкторе
7      void Process(); // Выполнение алгоритма
8      int GetSpaceWidth() const; // Получение результата
9
10 private:
11     ...
12 };
```

Поля механизма хранят настроечные параметры и промежуточные данные. В необъектном программировании механизму соответствует группа функций с промежуточными статическими данными.

Необходимость в специальном механизме возникает, когда появляются промежуточные данные, хранящие состояние алгоритма. Основное отличие механизма от нормального типа данных заключается именно в наличии промежуточных состояний.

В принципе возможны классы, имеющие свойства и типов данных, и механизмов. Можно представить механизм, который после работы ведет себя как нормальный тип данных. Однако лучше избегать таких ситуаций и выделять два класса: механизм и тип данных, порождаемый механизмом.

4.6. Внешние форматы и обеспечение обратной совместимости

Новая версия системы должна (но не обязательно!) поддерживать внешние форматы более старых версий. Совместимость новой версии с предыдущими будем называть обратной совместимостью. Для этого существует несколько приемов изложенных далее.

4.6.1. Флаги

Флаги (битовое множество) позволяют, пока есть место, добавлять новые бинарные атрибуты без изменения физической структуры данных. Нужно лишь инициализировать неиспользуемые флаги и игнорировать их значение.

4.6.2. Сохранение номера версии

Флаги годятся лишь для передачи бинарных атрибутов. Более общим решением является сохранение во внешнем формате номера версии для каждой структуры данных. Код считывания выглядит следующим образом. Сначала считывается номер версии. Затем этот номер сравнивается оператором **`switch`** с номером текущей версии и всех предыдущих. Для каждого случая вызывается соответствующий код считывания, а отсутствующие в старых версиях поля инициализируются значениями по умолчанию.

Если считанный номер версии превосходит текущий номер, то это означает что файл данных создан более новой версией программы и от его считывания нужно отказаться.

4.7. Интерфейсы между подсистемами

Различные подсистемы взаимодействуют друг с другом через интерфейсы. Интерфейсом называется абстрактный класс, содержащий только чисто виртуальные методы и не содержащий данных.

Одна подсистема может реализовывать несколько интерфейсов. Для реализации интерфейса необходимо создать класс — наследник абстрактного интерфейса, определить все его виртуальные методы и предоставить средство для создания объектов этого класса. Один объект может реализовывать несколько интерфейсов. В этом случае применяется множественное наследование. Для получения указателя на какой-либо интерфейс объекта по указателю на другой интерфейс необходимо использовать преобразование типа с помощью `dynamic_cast`.

4.8. Когда нужно заботиться об эффективности программы?

Есть два противоположных подхода к оптимизации программы:

1) Сначала программа должна заработать, а затем ее нужно оптимизировать

2) Об эффективности нужно думать с самого начала. Если проектные решения неэффективны, оптимизировать потом будет поздно.

В действительности оба подхода с определенной точки зрения справедливы и не противоречат друг другу. При проектировании и разработке программы нужно постоянно учитывать соображения эффективности, чтобы основные решения не были неисправимо неэффективны. Но на этом этапе не нужно тратить усилия на достижение максимальной эффективности и микрооптимизацию. После того, как программа заработала, можно снять профиль и провести требуемую оптимизацию.

Существует много приемов, дающих выигрыш в микроэффективности. Однако в подавляющем большинстве случаев микроэффективность никак не отражается на скорости работы программы. Дешевизна разработки и надежность гораздо важнее микроэффективности. Заботьтесь о надежности и читаемости Ваших исходных текстов, а об эффективности подумает руководитель проекта.

Настоятельно рекомендуется применять вместо обычных указателей на пользовательские объекты умные указатели (`unique_ptr, shared_ptr, weak_ptr, auto_ptr`). Умные указатели нужны для того, чтобы автоматизировать контроль за временем жизни ресурса (динамического объекта, как частный случай). Чтобы код был написан так, что забота о корректном освобождении ресурса ложилась на компилятор (посредством вызова деструктора). Это повышает надежность работы программы и облегчает процесс проектирования и отладки.

4.9. Работа с include файлами

4.9.1. Имена файлов

Рекомендуется использовать длинные имена файлов. Имена файлов пишутся с маленькой буквы (во избежание проблем при переносе кода на ОС типа UNIX). Файлу, содержащему описание либо реализацию некоторого класса, имя даётся по названию описываемого в файле класса без начальной буквы «C». В конце через точку приписывается необходимое расширение. Например, класс `CProgramCodeBuilder` должен быть объявлен в файле `programmcodebuilder.h`, а его реализация — в `programmcodebuilder.cpp`.

4.9.2. Оформление заголовочных файлов

Каждый заголовочный файл `<fileName_h>` должен начинаться со строк:

```
#ifndef fileName_h
#define fileName_h
...
#endif //fileName_h
```

Такая конструкция гарантирует, что текст не будет обработан компилятором дважды.

4.9.3. Включение include файла

Запрещено использовать в качестве имен пользовательских файлов полные либо относительные пути. Все нестандартные каталоги, из которых берутся включаемые файлы, описываются в опциях проекта.

```
#include <src/myheader.h> // Запрещено!
```

```
#include <myheader.h> // Правильно
```

Исключение может составлять использование библиотек типа Boost, где традиционно принято подключение вида:

```
#include <boost/asio.hpp>
#include <boost/core/noncopyable.hpp>
```

Имя включаемого заголовочного файла всегда указывается в угловых скобках. Использовать кавычки для указания имён файлов запрещено:

```
#include "myheader1.h" // Запрещено!
#include "../inc/myheader2.h" // Запрещено!
```

Первым включаемым в cpp-файл заголовочным файлом должен быть соответствующий этому cpp-файлу заголовок. Работа cpp-файла не должна зависеть от последовательности включения заголовочных файлов, следующих за первым заголовочным файлом.

В опциях проекта указываются пути для поиска включаемых файлов. При этом перечисляются все подкаталоги каталога проекта, в которых содержатся требуемые файлы. Кроме того, указываются стандартные каталоги, содержащие описания библиотек. Если включаемые файлы находятся в одном каталоге с проектом, то в списке путей надо указать символ «.».

Категорически запрещается указывать в опциях компиляции полные пути, содержащие имя диска или название компьютера. Для задания путей на каталоги, которые не являются подкаталогами проекта, необходимо использовать переменные окружения или относительные пути.

4.9.4. Использование предкомпиляции (precompiled headers)

Компилятор C++ поддерживает предкомпиляцию `include` файлов. Для эффективного использования этого средства каждый `*.cpp` файл проекта начинается со строк:

```
#include <common.h>
#pragma hdrstop
```

При этом для файла `common.cpp`, содержащего только указанные две строчки, устанавливается опция «Create precompiled header file (.pch)», а для остальных файлов проектов — «Use precompiled header file (.pch)». Имя загловочного, по которому осуществляется предкомпиляция указывать не нужно. Использовать опцию «Automatic use of precompiled headers» не рекомендуется, так как её использование существенно замедляет процесс компиляции.

Файл `common.h` — это специальный файл, который содержит в себе несколько строк, директив `include` для препроцессора. Обычно в него включают наиболее часто используемые в проекте `include` файлы стандартных библиотек. Кроме этого допустимо в `common.h` помещать директивы препроцессора, управляющие оптимизацией, например `#pragma inline_depth( 30 )`. Все содержимое `common.h` может быть удалено без изменения работоспособности программы, поэтому нельзя полагаться на то, что в `common.h` включаются какие-то файлы. Единственное его назначение — оптимизация времени компиляции.

Запрещается вносить в `common.h` какие либо другие строки кроме описанных выше.

4.9.5. Избыточные зависимости

При программировании на языке C++ возникает проблема избыточных зависимостей `*.cpp` файла от `*.h` файлов. Она возникает из-за того, что интерфейс класса и детали его реализации (**`protected`** и **`private`** поля, **`inline`** методы) должны быть размещены в одном `*.h` файле. При этом в `*.h` файл включаются другие `include`-файлы, необходимые для описания **`protected`** и **`private`** полей или для реализации **`inline`** методов. Это приводит к тому, что модуль перекомпилируется при изменении любого из включаемых в него `include`-файлов, хотя реальной зависимости от большинства из них нет. Основной принцип решения этой проблемы — отделять реализацию от интерфейса.

Кроме того, существуют следующие рекомендации.

1) Если в `include`-файле используется только указатель или ссылка на объект класса `X`, не следует включать заголовочный файл с описанием этого класса. Достаточно его объявить следующим способом: **`class`** `X;`

2) Не нужно выносить в `include`-файл реализацию нетривиальных **`inline`**-методов, поскольку возникает зависимость от реализации даже тех модулей, которые не используют эти методы. Если метод не критичен по быстродействию, то лучше сделать его обычным; если метод критичен по быстродействию, его нужно описать в отдельном файле с расширением `.inl` и включать этот файл только куда надо. Таким же образом можно бороться и с циклическими зависимостями файлов.

3) Не следует выносить в `include`-файл объявления тех констант, классов и типов, которые не относятся к интерфейсу. Их можно описывать прямо в модуле с реализацией либо в отдельном `include`-файле, если модулей несколько.

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ

ОС – операционная система

IDE – интегрированная среда разработки, англ. integrated development environment

ПЕРЕЧЕНЬ ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. ГОСТ 19.506-79. ЕСПД. Описание языка. Требования к содержанию и оформлению. — М. : Стандартинформ, 1979.

2. Б. Страуструп. Программирование. Принципы и практика с использованием C++. 2-е издание.: Пер. с англ. — М. : ООО «И.Д. Вильямс», 2016.

3. Н. Джосьютис. C++ Стандартная библиотека для профессионалов.: Пер. с англ. — СПб. : Питер, 2004.

4. Система документирования исходного кода Doxygen. — Режим доступа: `https://www.doxygen.nl/`.

5. Комментарии doxygen. — Режим доступа: `https://www.doxygen.nl/manual/docblocks.html`.

6. Boost. Собрание библиотек классов C++. — Режим доступа: `https://www.boost.org/`.

7. Armadillo. Библиотека линейной алгебры C++. — Режим доступа: `https://arma.sourceforge.net/`.