Правила хорошего стиля программирования

Время, когда программирование было уделом отдельных талантливых одиночек, закончилось более 30 лет назад. Сегодня крупные программные проекты разрабатываются большими коллективами, причем развитие и сопровождение проекта часто длится годами.

За это время успевает обновиться коллектив разработчиков, нередки случаи, когда инициаторы проекта к моменту его завершения могут отсутствовать не только в фирме, но и в стране.

В этих условиях естественным является требование унификации стиля программирования. Код, написанный различными программистами, должен читаться как единое целое и быть понятен не только его авторам. Отсюда следует, что программа должна быть рационально написана и тексты программы должны содержать достаточно комментариев для понимания сути алгоритмов другими программистами.

Никлаус Вирт

Ничто так не раздражает программиста–профессионала, как небрежно написанный и плохо документированный код.

Так же как невозможно представить инженерный чертеж, оформленный без учета требований стандартов, так и программа должна иметь некий унифицированный вид.

Приведенные ниже правила и рекомендации не являются обязательным стандартом, но их применение желательно и весьма приветствуется в профессиональном сообществе.

Существует несколько стилей написания и оформления программ, одним из которых является стиль *Microsoft*. Поскольку большинство профессиональных программистов, пишущих программы для *MS Windows*, разрабатывает свой код средствами *Microsoft Visual Studio*, рекомендуется применять этот стиль в качестве образца и придерживаться его при выполнении программных проектов.

Часто приходится сталкиваться с ситуацией, когда код сначала пишется и отлаживается и только потом документируется. Хорошо известно, что такая практика неверна, и вот почему:

- прежде чем начинать кодировать, необходимо продумать структуру программы и зафиксировать ее в документации;
- на завершающем этапе разработки времени всегда не хватает, в результате комментированием программы либо вообще пренебрегают, либо делают это небрежно, лишь бы «отвязаться». Комментирование программы должно происходить одновременно с написанием кода.

Технически процесс вставки «шапок» перед модулями и классами может быть реализован с помощью макросов. В этом случае комментирование выполняется буквально «в два щелчка» мыши.

Приведенные ниже рекомендации являются общепринятыми. В некоторых софтверных компаниях принятый стиль программирования доводится до сведения каждого сотрудника и является своего рода внутренним стандартом компании, а также элементом фирменного стиля.

Однако, даже если отсутствуют формализованные требования к принятому стилю программирования, все равно такие требования неявно присутствуют, хотя и носят более размытый, интуитивный характер.

В любом случае, плохой стиль программирования сразу бросается в глаза и свидетельствует о недостаточном уровне профессионализма разработчиков кода.

Стандартизация кода программного обеспечения

Стандартизация становится неизбежной, когда процесс разработки программного обеспечения принимает промышленный характер. То есть:

- когда в работе над проектом участвует достаточно многочисленный коллектив разработчиков;
- когда проект состоит из нескольких частей, и отдельные исполнители не знают детально, чем занимаются другие члены проектной команды. Каждый делает свою часть проекта. Проект целиком представляет себе руководитель проекта и несколько ведущих специалистов;
- когда в процессе работы происходит ротация персонала, кто-то увольняется, появляются новые сотрудники, которых нужно быстро ввести в курс дела;
- должны быть некие общие правила поведения, без которых невозможна слаженная работа коллектива разработчиков и их взаимодействие с заказчиками.

Одним словом, если вы участвуете в разработке достаточно большого и длительного проекта, вам придется следовать правилам работы того коллектива, в котором вы работаете. Чем более «оригинально» вы оформите свой код, тем меньше окажется желающих в нем разобраться.

Наконец, если вы пишете программу «для себя», вам неизбежно придется неоднократно возвращаться к ней для устранения ошибок или развития программы. Нередки случаи, когда, заглянув в свой проект через пару недель, программист уже не очень хорошо помнит, с какой целью он написал тот или иной фрагмент кода.

Такого не случится, если исходный код программы будет оформлен и прокомментирован надлежащим образом.

Выбор имен

Имя – результат размышлений, в основе которых лежит понимание работы системы в целом. Имя должно соответствовать тому объекту, свойству или процессу, который оно обозначает.

- Имена должны быть осмысленными и понятными;
- Имена должны быть построены на основе английских слов;
- Избегайте использования имен переменных и функций, составленных целиком из больших букв;
- Имена констант следует задавать только большими буквами.

Имена классов

Имя класса должно соответствовать тому объекту, который описывает данный класс.

Пример:

```
class CInterpolator        // интерполятор
class CSummer              // сумматор
class CConicInterpolator   // конический интерполятор
class ClinearInterpolator  // линейный интерполятор
```

Имя класса должно начинаться с буквы "*C*", что означает "*class*". Это традиция фирмы *Microsoft*, другие фирмы могут использовать другую литеру. Например, фирма *Borland* использует букву "*T*". Если имя класса состоит из нескольких слов, каждое слово должно начинаться с заглавной буквы.

Пример:

```
class CConnectionPointForMyOcx;
```

Имена, объединяющие в себе более трех слов, использовать не рекомендуется. Длинные идентификаторы затрудняют чтение программы.

Пример:

```
class CSemaphorForMyNewPort;   //слишком длинно и сложно
class CNewPortSemaphor;        // уже лучше
```

Имена производных классов не должны содержать в себе имен родительских классов, каждый класс должен иметь собственное имя, кем бы он ни порождался.

Имя класса должно начинаться с заглавной буквы. Если имя класса состоит из нескольких слов, каждое слово должно начинаться с заглавной буквы, которая служит разделителем слов.

Имена библиотек классов

Чтобы избежать конфликта имен с другими библиотеками классов, используйте для своей библиотеки некоторый уникальный префикс. Длина префикса не должна превышать трех символов.

Классы, производные от стандартных *C++* библиотек, должны иметь тот же префикс, который используется этими библиотеками. Например, префикс "*C*" используется библиотекой *MFC (class CWnd)*, префикс "*T*" используется библиотекой *OWL (class TWindow)*.

Для классов и функций, разрабатываемых в конкретной фирме, в качестве префикса можно использовать сокращенное имя фирмы.

Пример:

```
class ArcNewPortSemaphor      // Arc – от “ARCADIA”
class NitOldPortSemaphor      // Nit – от “NITA”
```

Имена функций и методов класса

Используйте те же правила, что и для имен классов. Имя не должно напоминать ругательство. Длинное имя обычно лучше, чем короткая аббревиатура, однако лаконичностью не следует злоупотреблять.

Обычно каждый метод и функция выполняет некоторое действие, поэтому имя должно описывать это действие:

CheckForErrors() вместо *ErrorCheck(),*
DumpDataToFile() вместо *DataFile*().

Пример имен функций:

```
BOOL AddChildWindow(HWND hChildWnd);
socket GetListenSocket(unsigned short uListenPort);
BOOL IsTransactionValid(TRANS_ID trid);
```

Следующие суффиксы иногда могут быть полезны:

- *Max* - означает наибольшее значение чего либо.
- *Cnt* - означает текущее количество чего либо.
- *Key* – ключевое значение.

Пример:

RetryMax – означает максимальное число попыток.

RetryCnt – означает номер текущей попытки.

Следующие префиксы иногда могут быть полезны:

Is – используется, чтобы задать некоторый вопрос: *IsEnable(), IsWindow(), IsActive();*

Get – используется для считывания некоторого значения: *GetData(), GetWndRect();*

Set – используется для задания нового значения: *SetData(), SetWindowText();*

On – используется при задании обработчика событий: *OnCreate(), OnClose().*

Пример:

```
class CNameOneTwo
{
   public:
      int DoIt();
      void HandleError();
      BOOL IsItPrintable();
      long OnWmCreate(WPARAM wParam1, LPARAM lParam2);
      const char* GetFirstName() const;
      void SetSecondName(const char* pszSecName);
};
```

Имена аргументов функций и методов класса

Используйте те же правила, что и для имен классов.

Пример:

```
class CTest
{
    public:
       int RunTest(int nParam1, long lParam2);
};
```

Имена переменных

Используйте так называемую «упрощенную венгерскую нотацию», которая представлена в таблице ниже.

Prefix	Type	Description	Example
Basic Types			
ch, c	**Char**	8-bit character	*ChGrade*
sz, str	**char[]**	8-bit string	*szBuffer[]*
s	**Short**	16-bit signed integer	*sIndex*
n, l	**Int**	Integer (size dependent on operating system)	*nLength*
u	**unsigned**	Unsigned integer (size dependent on operating system)	*uSize*
l	**Long**	32-bit signed integer	*lSum*
ul	**usigned long**	32-bit unsigned integer	*ulFreeSpace*
flt	**Float**	Floating point	*fltResult*
dbl	**Double**		*DblResult*
ldbl	**long double**		*ldblResult*
Aliased Types			
by	**BYTE**	Byte (unsigned char)	*byLower*
b	**BOOL**	Boolean value	*bEnabled*
n, u	**UINT**	Unsigned value (size dependent on operating system)	*nLength*
w	**WORD**	16-bit unsigned value	*wPos*
l	**LONG**	32-bit signed integer	*lOffset*
dw	**DWORD**	32-bit unsigned integer	*dwRange*
h	**Handle**	Handle to Windows object	*hWnd*
Pointers			
p	**void***	Ambient memory model pointer	*pDoc*
pp	**void****	Pointer to pointer	*ppDoc*
lp	**FAR***	Far pointer	*lpDoc*
np	**NEAR***	Near pointer	*npDoc*
lpsz	**LPSTR**	32-bit pointer to character string	*lpszName*
lpfn	**callback**	Far pointer to CALLBACK function	*lpfnAbort*
p(type-prefix)	**basic_type***	Pointer to basic type variable	
Derived Types			
pt	**POINT**	Points	*ptCurPos*
rc	**RECT**	Rectangle	*rcViewPort*
Obj	**OBJECT**	Object	*objOld*
Arr	**ARRAY**	Array	*arrItems*

Имена свойств класса

Используйте упрощенную венгерскую нотацию. Имена свойств (скрытых данных класса) желательно предварять префиксом 'm_'. После 'm_' следует имя свойства, имеющее ту же структуру, что и имя переменной. Префикс 'm_', если он применяется, должен предшествовать любым другим модификаторам, например, 'p' для указателей.

Пример:

```
class CNameOneTwo
{
    public:
        int VarAbc();
        int ErrorNumber();

    private:
        int m_nVarAbc;
        int m_nErrorNumber;
        char* m_pszName;
};
```

Имена указателей

Имени указателя должен предшествовать модификатор ‘p’. Символ ‘*’ следует помещать сразу после типа указателя, а не перед именем указателя.

Пример:

```
class CTest
{
    public:
        void PrintString(char* pszString);

    private:
        char* m_pszClassName;
};
```

Имена ссылок

Ссылке может предшествовать символ “r”, это позволит различать модифицируемые и немодифицируемые объекты.

Пример:

```
class CTest
{
    public:
        void DoSomething(StatusInfo& rStatus);
        const StatusInfo& GetStatus() const;

    private:
        StatusInfo& m_rStatus;
};
```

Имена глобальных переменных

Используйте упрощенную венгерскую нотацию. Глобальной переменной должен предшествовать префикс “g_”.

Пример:

```
long g_lCounter;
char* g_pszAppName;
```

Имена статических переменных

Используйте упрощенную венгерскую нотацию. Статической переменной должен предшествовать префикс “s_”.

Пример:

```
class CTest
{
    private:
        static StatusInfo ms_Status;
};
static int s_nReadBytes;
static char* s_pClassName;
```

Имена констант

Константы в *C++* можно задать тремя операторами: *const, #define, enum*. Константы должны состоять из заглавных букв и сепараторов “_”. Имена констант не должны совпадать, независимо от того, каким способом вы задали константы.

Пример:

```
#define GLOBAL_CONSTANT 5
const int default_lval = 4;
```

Имена макросов

Константы, определяемые оператором #*define*, должны состоять из заглавных букв и сепараторов “_”. Рекомендуется использовать префиксы, чтобы сгруппировать константы в категории.

Пример:

```
#define MAX(a,b) …..
#define IS_ERR(err) …..
#define ERRMSG_NO_RAM 0
#define ERRMSG_WRITE 1
#define ERRMSG_READ 2
```

Имена перечислений enum

Если *enum* используется вне класса для задания констант, следует убедиться, что имя константы не совпадает с другой константой, уже заданной операторами *const* или #*define*.

Имена констант в перечислении задаются или только заглавными буквами, или только строчными буквами (с использованием сепаратора ‘_’). Смешение стилей не допускается. Предпочтительным является использование строчных букв для того, чтобы не смешивать константы с макросами. Если *enum* задается при определении класса, то для чтения константы необходимо указать имя класса, в котором задана *enum*, например так: *Aclass::pin_off.* Можно использовать *enum* для задания неинициализированных или ошибочных состояний.

Пример:

```
enum PinStateType
{
   pin_off = 0,
   pin_on
};
enum {state_err, state_open, state_running, state_dying};
```

Имена типов

Имена типов, созданных при помощи *typedef*, следует писать только строчными буквами (с использованием сепаратора "_") для отличия от макросов, созданных директивой #*define*. Это отличие весьма важно, поскольку, например, объявление

```
typedef void *(ptr_to_func) ( int );
```

позволяет писать как

```
(ptr_to_func) (p);          // convert p into ptr_to_func
```

так и

```
ptr_to_func f( long );     // f returns a pointer to a function
```

В то же время

```
#define PTR_TO_FUNCTION (void ( * ) ( int ))
```

позволяет осуществлять приведение типов:

```
(PTR_TO_FUNCTION)(p);
```

но не определяет функцию.

Пример:

```
typedef unsigned int module_id;
typedef unsigned long system_type;
```

Имена меток

Использования оператора *goto Label*; следует избегать всегда, когда это только возможно. Имена меток следует задавать строчными или прописными буквами с использованием сепаратора "_". Смешивание стилей недопустимо.

Пример:

```
FIRST_JUMP:
    goto restart_point;
```

Имена файлов

Файлы исходного кода для языков *C/C++* должны иметь расширения *".c"/".cpp"*, соответственно. Заголовочные файлы для языков *C/C++* должны иметь расширения *".h"*. Файлы, содержащие *inline*-определения, должны иметь расширения *".inl"*. Файлы ресурсов для *Windows*-проектов должны иметь расширения *".rc"*. *Include* – файлы, содержащие идентификаторы для ресурсов, могут иметь расширения *".h", ".rh", ".hm"*.

Используйте для имен и расширений файлов так называемое «соглашение 8.3» (8 символов – имя, и 3 символа - расширение), когда это возможно. Некоторые операционные системы и архиваторы все еще не поддерживают длинные имена; используя соглашения 8.3, вы избежите ненужных конфликтов.

Если Вы используете *Wizard* для создания своего проекта, не изменяйте без крайней необходимости имена файлов, которые он предложит. Всегда старайтесь выбрать уникальное имя для своих файлов, насколько это возможно. Старайтесь включать имя класса в имя файла, в котором он определяется.

Форматирование кода

Общие требования к форматированию кода

На одной строке должно находиться не более одного оператора *C/C++.*

Если вызов функции, операция инициализации, список и т.п. занимают более одной строки, перенос на следующую строку должен следовать сразу после запятой.

Если выражение занимает несколько строк, переход на следующую строку должен выполняться сразу после бинарной операции.

Максимальная длина строки

Максимальная длина строки не должна превышать 70 символов. Хотя большие мониторы могут отображать и более длинные строки, печатающие устройства более ограничены в своих возможностях.

Чем больше размеры окна, тем меньше окон можно одновременно наблюдать на экране дисплея. Исходя из требований удобства отладки, считается, что возможность одновременно иметь несколько открытых окон важнее, чем возможность иметь одно, но большое окно. Из этого соображения также следует ограничение на длину строки.

Использование табуляции и пробелов для организации отступов

Для организации отступов предпочтительно использование табуляции вместо пробелов.

На размер отступа не накладывается ограничений, но злоупотреблять отступами не стоит. Если размер отступа более 4 или 5 пробелов, исходный код может не уместиться по ширине страницы. Большинство редакторов позволяет задавать размер отступа табуляции.

Комментарии должны находиться на уровне того оператора, которому они соответствуют. Комментарии справа от операторов желательно выравнивать с помощью пробелов, а не табуляции.

Пример:

```
int Func(int nParam1, long lParam2)
{
    int nVariable;          // in-line comment
    char sBuffer[10];       // in-line comment

     // comment
    if (something_bad)
    {
        if (another_thing bad)
        {
            while (more_input)
            {
            }
        }
    }

    // comment
    for (int i = 0; i < 10; i++)
        nVariable += i;

    return nVariable;
}
```

Пунктуация

Точке с запятой ';' не должны предшествовать пробелы или табуляция. После запятой должен следовать пробел.

Правильное совместное использование операторов и пробелов проиллюстрировано в следующей таблице.

Operator	L	R	Comments	Example
!	.	-	Not	`if (!TRUE)`
!=	+	+	Arithmetic inequality	`if (a != b)`
#	.	-	Preprocessor directive	`#define YES 1`
##	+	+	Token-paste	`#define cat(x, y) x ## y`
%	+	+	Modulus	`a = b % c;`
%=	+	+	Modulus and assign	`a %= b;`
&	+	+	Bitwise AND	`if (mask & attrib)`
&	.	-	Address of an operand	`a = &b;`
&&	+	+	Logical AND	`while (a && b)`
&=	+	+	Bitwise AND assign	`a &= mask;`
()	.	-	Cast operators	`a = (INT)b;`
*	+	+	Multiply	`a = b * c;`
*	.	-	Indirection operator	`a = *b;`
*=	+	+	Multiply and assign result	`a *= b;`
+	+	+	Add	`a = b + c;`
++	.	-	Increment prefix	`++a;`
++	-	.	Increment postfix	`a++;`
+=	+	+	Add assign result	`a += b;`
,	-	+	Evaluate an rvalue	`a = b, b = c, c = a;`
-	+	+	Subtract	`a = b - c;`
--	.	-	Decrement prefix	`--a;`
--	-	.	Decrement prefix	`a--;`
-=	+	+	Subtract and assign	`a -= b;`
->	-	-	Struct/union pointer offset	`a = b->c;`
.	-	-	Select struct/union member	`a.b = 2;`
/	+	+	Divide	`a = b / c;`
/=	+	+	Divide and assign	`a /= b;`
<	+	+	Less then	`if (a < b)`
<<	+	+	Left shift	`a = b << 2;`
<<=	+	+	Left shift and assign	`a <<= 4;`
<=	+	+	Less than or equal	`if (a <= b)`
=	+	+	Assignment operator	`a = b;`
==	+	+	Equality	`if (a == b)`
>	+	+	Greater than	`if (a > b)`
>=	+	+	Greater than or equal	`if (a >= b)`
>>	+	+	Right shift	`a = b >> 2;`
>>=	+	+	Right shift and assign	`a >>= 4;`
?	+	+	if/else (cf. :)	`a = (b > c) ? b : c;`
:	+	+	if/else (cf. ?)	`a = (b > c) ? b : c;`
[	-	-	Array subscript (cf.])	`a = s[i];`
]	-	.	Array subscript (cf. [)	`a = s[i];`
^	+	+	Exclusive OR	`a = b ^ c;`
^=	+	+	Exclusive OR and assign	`a ^= b;`
sizeof	.	-	Size in bytes	`a = sizeof(INT);`
|	+	+	Bitwise OR	`a = b | c;`
|=	+	+	Bitwise OR and assign	`a |= b;`
||	+	+	Logical OR	`if (a || b)`
~	.	-	One's complement	`a = ~b;`

В таблице использованы следующие обозначения:

L: левый пробел;

R: правый пробел;

-: пробел недопустим;

+: пробел обязателен;

(.) означает, что пробел не обязателен, но допустим.

Замечание: Оператор ‘##’ использовать не рекомендуется, поскольку не все компиляторы поддерживают этот оператор.

Выравнивание блока объявлений

Блок объявлений должен быть выровнен по типу, имени, начальному значению и комментарию. Для выравнивания используйте как пробелы, так и табуляции. Целью выравнивания является улучшение ясности и читаемости кода.

Символы ‘&’ и ‘*’ должны следовать без пробелов за объявлением типа объекта, а не стоять перед именем объекта.

Пример:

```
void Func()
{
    DWORD   dwLength;       // in-line comment
    DWORD*  pdwLength;      // in-line comment
    char    szName[12];     // in-line comment
    char*   pszName;        // in-line comment

    dwLength  =  0;
    pdwLength =  NULL;
    szName[0] =  '\0';
    pszName   =  NULL;
  ...
}
```

Расстановка фигурных скобок

Существует несколько традиций расстановки скобок, наиболее употребительными являются следующие:

традиция *Microsoft*:

```
if(condition)
{
    ...
}
```

```
while(condition)
{
    ...
}
```

традиция *Unix*:

```
if(condition) {
    ...
}
```

```
while(condition) {
    ...
}
```

Традиция *Microsoft* более предпочтительна, поскольку иногда бывает полезно точно знать, где находится граница блока.

Пример:

```
if (very_long_condition && second_very_long_condition)
{
    ...
}
else if (...)
{
    ...
}
```

Расстановка круглых скобок

Ключевое слово, заключенное в круглые скобки, должно начинаться и заканчиваться пробелом. Исключением является оператор *sizeof()*.

Недопустимо ставить пробелы сразу после имени функции. Не применяйте скобки в операторе *return* без необходимости. Не используйте пробелы перед текстом, заключенным в кавычки, а также после него.

Каждое выражение, за исключением арифметических операций, должно использовать скобки, чтобы явно задать порядок выполнения операций. В арифметических выражениях также лучше всегда использовать скобки, поскольку разные компиляторы по-разному выполняют разбор арифметических выражений. Такого не должно быть, но это имеет место - это экспериментальный факт.

Пример:

```
int Func(char* pszParam1, char* pszParam2)
{
    int     nAddrSize   = sizeof(char*);
    if (pszParam1 == NULL)   return 0;
    if (pszParam2 == NULL)   return 0;
    strcpy(pszParam1, pszParam2);
    return 1;
}

int GetMaxValue(int nValue1, int nValue2)
{
    return ((nValue1 >= nValue2) ? nValue1 : nValue2);
}
```

Форматирование операторов *if-else*

Выравнивать *if-else* рекомендуется так, как это показано в приведенном ниже примере.

Пример:

```
if (condition)                  // in-line comment
{
    ...
}
else if (condition)             // in-line comment
{
    ...
}
else                            // in-line comment
{
    ...
}
```

При сравнении с константой ее лучше размещать справа от оператора “**==**” или “*!=*”:

```
if (nErrorNum == 6)
 ...
```

Форматирование операторов *switch-case*

Если оператор *break* не заканчивает блок команд, следующих после оператора *case*, отметьте в комментарии, что это не случайный факт, а ваше сознательное решение.

Если на вход оператора *switch* должны попадать только данные, перечисленные в case-ловушках, то оператор *default* должен обязательно присутствовать и генерировать сообщение об ошибке.

Пример:

```
switch (...)
{
case 1:                 // in-line comment
    ...
// fall-through comment must be here
case 2:                 // in-line comment
    ...
    break;
case 3:                 // in-line comment
{
    int v;
        ...
}
break;
default:
        break;
 }
```

Комментарии

Строка комментария должна иметь такой же отступ, как и операция, которую она комментирует. *In-line* комментарии должны размещаться справа от комментируемого кода. Символы комментария “/*” и “//” должны отделяться от текста комментария по крайней мере одним пробелом. *In-line* комментарии желательно выравнивать с использованием пробелов, а не табуляций.

Пример:

```
POINT ptCurrentMenuArea;                    // in-line comment
 // check if designated point is in the menu area
if (!IsMenuArea(ptCurrentMenuArea))
{
    // out of the menu area
    return NO_MENU_ID;
}
```

Методы и функции

Формат задания формальных аргументов функции зависит от количества этих аргументов. Если список аргументов помещается на одной строке, то функция записывается в одну строку, в противном случае каждый аргумент функции следует писать на новой строке, при этом полезно применять *in-line* комментирование.

Пример: Функция с коротким списком формальных аргументов:

```
int Func(int nParam1, long lParam2, char* pszParam3)
{
    ...
}
```

Пример: Функция с длинным списком формальных аргументов:

```
int Func
(
    int     nParam1,        // Integer argument
    long    lParam2,        // Long argument
   char*    pszParam3       // String argument
)
{
    ...
}
```

Документирование исходного кода

В этом разделе описываются меры, способствующие повышению читабельности исходного кода и облегчающие его сопровождение в течение всего процесса реализации проекта.

Общие требования

Документация к исходному коду, содержащаяся в комментариях, должна быть достаточной для полного понимания кода и его дальнейшего сопровождения как для разработчиков (проектировщиков) программного обеспечения, так и для программистов. Следует иметь в виду, что недокументированный код не имеет смысла, а следовательно, и права на существование.

Каждый файл исходного кода программы должен иметь вводную часть, содержащую в форме комментария информацию об авторе программы, имени файла и его содержании.

Исходный код должен включать в себя заявление о защите авторских прав. Если программа разрабатывается в течение нескольких лет, указывается каждый год.

Все комментарии рекомендуется писать на английском языке. Программистов, не владеющих английским, в природе не существует, а вот компиляторы, не поддерживающие кириллицу, к сожалению, встречаются часто.

Исходный код необходимо снабжать комментариями, основываясь на принципе «разумной достаточности». Для начала, следует корректно выбрать имена переменных, функций и классов (см. раздел «Выбор имен»). Далее, необходимо правильно форматировать и структурировать исходный код (см. раздел «Форматирование кода»), такой код требует меньше комментариев.

Помните, что комментарии в заголовочных *h*-файлах предназначены для пользователей классов, тогда как комментарии в *cpp*-файлах предназначены для разработчиков классов.

Комментарии подразделяются на стратегические и тактические.

В стратегических комментариях описывается общее назначение функций или фрагментов кода, и они вставляются в текст программы блоком из нескольких строк.

Тактический комментарий задается одной строкой и описывает операцию на следующей строке. Его также можно размещать справа от комментируемой операции. Избыток тактических комментариев ухудшает читабельность кода, ими не следует злоупотреблять. Вместе с тем, точки ветвления и циклы должны быть снабжены комментариями в обязательном порядке.

Исходные файлы *(cpp-files)*

Каждый исходный файл должен содержать реализацию только одного класса или группы функций, близких по своему назначению.

Каждый файл должен иметь заголовок, а информация должна быть единообразным образом структурирована.

Каждый *.cpp*-файл должен включать в себя соответствующие заголовочные файлы *(.h-files)*, которые обычно содержат:

- объявление типов и функций, которые используются функциями или методами класса, реализуемого в данном *.cpp-файле*;
- объявление типов, переменных и методов классов, которые реализуются в данном *.cpp-файле.*

Заголовок исходного файла

Заголовок исходного *CPP-файла* представляет собой закомментированный текст, помещенный в начало файла. Этот текст имеет следующую структуру.

```
/*************************************************************\
FILE….....:     filename.ext
AUTHOR….....:   Name(s) of the programmer(s)
DESCRIPTION.…:  The description of the file.
CLASSES…….....: List of classes implemented in this module.
FUNCTIONS……:   List of functions implemented in this module.
SWITCHES……...: Preprocessor switches and their meaning.
NOTES….....:    Other useful information (compilers, portability, updates, and so on).
COPYRIGHT…. ..: Copyright information.
HISTORY…... …..: DATE  COMMENT
                 -------- ---------------------------------------
                 mm-dd-yy Comments – Author.
\*************************************************************/
```

Поля FILE, AUTHOR, DESCRIPTION, COPYRIGHT и HISTORY являются обязательными для заполнения. Заполнение остальных полей остается на усмотрение программиста.

Структура исходного файла

Каждый исходный файл в проекте должен иметь следующую структуру.

```
/*************************************************************\
 FILE….....:    filename.ext
 AUTHOR….....:  Author's name.
 COPYRIGHT…...:  Copyright information.
 DESCRIPTION…:  Description of the source file.
 HISTORY….....:  DATE COMMENT
                 -------- -------------------------------------
                 mm-dd-yy Comments – Author.
\*************************************************************/

/*============================[ SPECIAL ]=======================*/
/*============================[ IMPORT DECLARATIONS ]==========*/
/*============================[ PUBLIC DECLARATIONS ]==========*/
/*============================[ PRIVATE DECLARATIONS ]=========*/
/*============================[ ..PRIVATE CONSTANTS ]==========*/
/*============================[ ..PRIVATE TYPES ]==============*/
/*============================[ ..PRIVATE VARIABLES ]==========*/
/*============================[ ..PRIVATE FUNCTIONS ]==========*/
/*============================[ ..PRIVATE PSEUDO FUNCTIONS ]=====*/
/*============================[ PUBLIC VARIABLES ]=============*/
/*============================[ PRIVATE VARIABLES ]============*/
/*============================[ CLASS LIFECYCLE ]==============*/
/*---------------------------[ CLASS_NAME1 ]---------------*/
 /*---------------------------[ CLASS_NAME2 ]---------------*/

/*============================[ CLASS OPERATORS ]==========*/
/*---------------------------[ CLASS_NAME1 ]---------------*/
/*---------------------------[ CLASS_NAME2 ]---------------*/
/*============================[ PUBLIC CLASS METHODS ]======*/
/*---------------------------[ CLASS_NAME1 ]---------------*/
/*---------------------------[ CLASS_NAME2 ]---------------*/
/*============================[ PROTECTED CLASS METHODS ]===*/
/*---------------------------[ CLASS_NAME1 ]---------------*/
/*---------------------------[ CLASS_NAME2 ]---------------*/
/*============================[ PRIVATE CLASS METHODS ]=====*/
/*---------------------------[ CLASS_NAME1 ]---------------*/
/*---------------------------[ CLASS_NAME2 ]---------------*/
/*============================[ PUBLIC FUNCTIONS ]=========*/
/*============================[ PRIVATE FUNCTIONS ]=========*/
/** (END OF FILE : filename.ext) **************************/
```

В таблице ниже приведено описание каждого раздела. Если некоторый раздел не содержит информации, его можно опустить, но порядок следования разделов менять не рекомендуется.

Section	Description
SPECIAL	В этом разделе должны находиться блоки условной компиляции (#ifndef … и т.п.).
IMPORT DECLARATIONS	Все системные, библиотечные и заголовочные файлы (h-файлы) должны находиться в данном разделе.
PUBLIC DECLARATIONS	Эта строка должна предшествовать объявлением *public* методов и данных класса.
PRIVATE DECLARATIONS	Эта строка должна предшествовать объявлением *private* методов и данных класса.
..PRIVATE CONSTANTS	Все частные #defines и constants должны быть объявлены в данном разделе.
..PRIVATE TYPES	Все частные типы, которые используются в данном исходном файле, должны быть объявлены в данном разделе.
..PRIVATE VARIABLES	В этом разделе объявляются все частные переменные.
..PRIVATE FUNCTIONS	Все частные функции, используемые в данном исходном файле, должны быть объявлены в этом разделе.
..PRIVATE PSEUDO FUNCTIONS	Все макросы должны быть объявлены в этом разделе.
PUBLIC VARIABLES	Все глобальные переменные должны быть заданы в этом разделе.
PRIVATE VARIABLES	Этот раздел должен содержать все частные переменные данного исходного файла.
CLASS LIFECYCLE	Раздел жизненного цикла предназначен для размещения объектов, которые управляют жизненным циклом объекта. Обычно это конструкторы, деструкторы и методы, изменяющие состояние объекта.
CLASS_NAME	Если исходный файл содержит реализацию нескольких классов, то их методы должны быть разделены данной секцией с именем класса в квадратных скобках.
CLASS OPERATORS	Все оператора класса должны находится в этом разделе.
PUBLIC CLASS METHODS	Все *public* методы класса должны находится в этом разделе.
PROTECTED CLASS METHODS	Все *protected* методы класса должны находится в этом разделе.
PRIVATE CLASS METHODS	Все *private* методы класса должны находится в этом разделе.
PUBLIC FUNCTIONS	Раздел для реализации *public* функций.
PRIVATE FUNCTIONS	Раздел для реализации *private* функций.

Наличие комментария, обозначающего начало нового раздела и содержащего имя этого раздела, не обязательно, но весьма желательно. Дело в том, что имеется целый ряд утилит для автоматической генерации документации по коду, комментированному в соответствии с формальными правилами. Такие утилиты помогают тем, кто, не являясь разработчиком, должен модифицировать код.

В любом случае обязательными являются строки комментария, обозначающими конец и начало файла. Их отсутствие должно сигнализировать о том, что «что-то не в порядке» (например, в результате ошибки записи часть файла оказалась потеряна).

Заголовочные файлы *(h-files)*

Каждый класс должен быть задан в своем *.h-файле*. Следующие объекты должны задаваться в своих отдельных *.h-файлах*:

- базовые классы;
- объекты (классы), которые являются данными других классов;
- классы, которые передаются в качестве формальных параметров, функций или методов класса или возвращаются оператором return;
- прототипы функций или методы классов, реализованные, как inline.

Описания классов, доступных только через указатели (*) или ссылки (&), не должны включаться из .h-файлов. Используйте опережающие ссылки.

Чтобы избежать многократного включения заголовочных файлов, содержимое каждого .h-файла должно находиться между следующими операторами условной компиляции:

```
#if !defined(FILENAME_H)
#define FILENAME_H

 /* contents of FILENAME.H */

 #endif /*FILENAME_H */
```

Никогда не используйте абсолютные имена путей, например:

```
#include c:\project\sources\include\project.h
```

Используйте относительные пути или задавайте путь в опциях компилятора (в последнем случае необходимость задания путей в опциях должна быть документирована, в том числе и в заголовке программы).

Никогда не включайте при помощи *#include* файлы исходного кода *(.cpp)*. Включать надо только заголовочные файлы *(.h)*.

Включите в начало *h-файла* следующий фрагмент текста.

```
/*************************************************************\
FILE….....:     filename.ext
AUTHOR…….....:  Name(s) of the programmer(s)
DESCRIPTION…:   The description of the file.
CLASSES…….....: List of classes declared in this module.
FUNCTIONS…....: List of functions declared in this module.
SWITCHES……...: Preprocessor switches and their meaning.
NOTES….....:    Other useful information (compilers, portability, updates and so on)
COPYRIGHT…....: Copyright information.
HISTORY….....:  DATE COMMENT
                -------- -----------------------------------
                mm-dd-yy Comments – Author.
\*************************************************************/
```

Поля FILE, AUTHOR, DESCRIPTION, COPYRIGHT, HISTORY подлежат обязательному заполнению, остальные – на ваше усмотрение.

Каждый h-файл должен иметь следующую структуру.

```
/*****************************************************************\
 FILE…..................:     filename.ext
 AUTHOR……....:     Author's name.
 COPYRIGHT…...:     Copyright information.
 DESCRIPTION…:     Description of the source file.
 HISTORY……....:     DATE COMMENT
                    ------- ----------------------------------------
                    mm-dd-yy Comments – Author.
\*****************************************************************/
/*=================================[ REDEFINITION DEFENCE ]=======*/

    #if !defined( FILENAME_EXT)
    #define FILENAME_EXT
/*=================================[ SPECIAL ]====================*/
/*=================================[ PUBLIC CONSTANTS ]===========*/
/*=================================[ PUBLIC TYPES ]===============*/
/*=================================[ FORWARD REFERENCES ]=========*/
/*=================================[ PUBLIC VARIABLES ]===========*/
/*=================================[ PUBLIC FUNCTIONS ]===========*/
/*=================================[ PSEUDO/INLINE FUNCTIONS ]====*/
/*=================================[ END REDEFINITION DEFENCE ]===*/

    #endif /* FILENAME_EXT */

/** (END OF FILE : filename.ext) ***************************/
```

Описание каждого раздела приведено ниже.

Section	Description
REDEFINITION DEFENCE	Гарантирует однократное включение h-файла
SPECIAL	Блок операторов условной компиляции
PUBLIC CONSTANTS	Публичные константы
PUBLIC TYPES	Публичные типы
FORWARD REFERENCES	Ссылки вперед
PUBLIC VARIABLES	Публичные переменные
PUBLIC FUNCTIONS	Публичные функции
PSEUDO/INLINE FUNCTIONS	Inline функции (методы класса)
END REDEFINITION DEFENCE	Окончание однократно включаемого h-файла

Если некоторый раздел не содержит информации, его можно опустить, но порядок следования разделов менять не рекомендуется.

Классы

Заголовок класса представляет собой закомментированный текст, предшествующий определению класса.

Он имеет следующий вид.

```
/*****************************************************************\
CLASS….............:      Name of the class.
DESCRIPTION…:      The description of the class and its purpose.
\*****************************************************************/
```

Пример:

```
/*****************************************************************\
CLASS……..…..:      CdiskMngr
DESCRIPTION…:      Provides low-level file I/O operations.
\*****************************************************************/
   class CdiskMngr : public CcheckSumObject
   {
       ...
   };
```

Методы

Заголовок метода представляет собой закомментированный текст, предшествующий определению метода. Он имеет следующий вид:

```
/*****************************************************************\
METHOD….....:      Name of the method.
DESCRIPTION…:      Purpose of the method.
ATTRIBUTES…..:     Method's attributes
ARGUMENTS…...:     Arguments of the method, their type and meaning.
RETURNS……....:     Possible return values and their meaning
NOTES….....:       For instance, explanation of the usage and/or an
                   example, limitations, pseudocode, etc.
\*****************************************************************/
```

Поля METHOD, DESCRIPTION, ARGUMENTS, RETURNS являются обязательными, остальные могут отсутствовать.

Каждому аргументу метода может предшествовать один из следующих префиксов:

- IN – означает, что данный аргумент – только для чтения и изменению не подлежит;
- OUT – означает, что данный аргумент будет изменен при работе метода, однако его начальное значение методом не используется;
- INOUT - означает, что данный аргумент будет изменен при работе метода, а его начальное значение методом используется.

Поскольку не все компиляторы поддерживают указанные префиксы (например, *MS Visual C++* не поддерживает), предпочтительно использовать данные слова в *in-line* комментариях.

Пример:

```
/*****************************************************************\
 METHOD.............:     Read
 DESCRIPTION….:     Reads nCount bytes to pBuffer beginning from
                    the given file position.
 ATTRIBUTES…...:     Public
 ARGUMENTS.....:     fpStart - the file position to be read from,
                    pBuffer - pointer to a buffer that is to receive
                        read data from the file,
                    nCount - maximum bytes to be read.
 RETURNS.............:     Returns the number of bytes read. If the method
                        tries to read at end of file, it returns 0.
                        If the reading failed, the method return -1.
                        To get extended error information, call
                        GetLastError() method.
 NOTES.....................:     The following error codes GetLastError() can
                    return after the method failed:
                       ERR_MEMORY - not enough memory,
                       ERR_FILELOCK - the file is locked to read
\*****************************************************************/

   int CDiskMngr::Read
   (
       /* IN */       FilePtr  fpStart,       // Начало
       /* OUT */      void*    pBuffer,       // Адрес буфера
       /* IN */       int      nCount         // Счетчик
   )
   {
      ...
   }
```

Функции

Заголовок функции представляет собой закомментированный текст, предшествующий определению функции. Он имеет следующий вид:

```
/*****************************************************************\
 FUNCTION..........:     Name of the function.
 DESCRIPTION...:     Purpose of the function.
 ARGUMENTS.....:     Arguments of the function, their type and meaning.
 RETURNS.............:     Possible return values and their meaning.
 EXTERNS.............:     Global data used/affected in the function.
 NOTES.....................:     For instance, explanation of the usage and/or an
                    example, limitations, pseudocode, etc.
\*****************************************************************/
```

Поля FUNCTION, DESCRIPTION, ARGUMENTS, RETURNS являются обязательными, остальные могут отсутствовать.

Каждому аргументу метода может предшествовать один из следующих префиксов:

- IN – означает, что данный аргумент – только для чтения и изменению не подлежит.

- OUT – означает, что данный аргумент будет изменен при выполнении метода, однако его начальное значение методом не используется.
- INOUT - означает, что данный аргумент будет изменен при выполнении метода, а его начальное значение методом используется.

Поскольку не все компиляторы поддерживают эти префиксы (например, *MS Visual C++* не поддерживает), предпочтительно использовать данные слова в in-line комментариях.

Пример:

```
/*****************************************************************\
 FUNCTION.....:  AbbreviateFileName
 DESCRIPTION….:  Makes an abbreviate file name from the given full path name.
 ARGUMENTS…...:  lpszCanon - pointer to a full path name to be abbreviated,
                 ichMax - max number of characters of the resultant file name,
                 bAtLeastName - specify whether the file nameshould be left at least after
                 abbreviation.
 RETURNS.....:   None.
 NOTES.....:     Below is the example how the function works.
                 lpszCanon = “C:\MYAPP\DEBUGS\C\TESWIN.C”;
                  ichMax     bAtLeastName    Result (lpszCanon)
                 ------- ------------ --------------------------
                    1-7          FALSE         <empty>
                    1-7          TRUE          TESWIN.C
                    8-14           x           TESWIN.C
                    15-16          x           C:\...\TESWIN.C
                    17-23          x           C:\...\C\TESWIN.C
                    24-25          x           C:\...\DEBUGS\C\TESWIN.C
                    26+            x           C:\MYAPP\DEBUGS\C\TESWIN.C
\*****************************************************************/
      void PASCAL AbbreviateFileName
  (
     /* INOUT */     LPTSTR    lpszCanon,
     /* IN */        int        ichMax,
     /* IN */        BOOL       bAtLeastName
  )
  {
     ...
  }
```

Документирование каталогов

Каждый каталог должен содержать README файл, в котором описано:

- Назначение каталога и его содержимое.
- Одна строка комментария на каждый файл. Комментарий обычно извлекается из поля NAME оглавления файла.
- Описание процедуры инсталляции.
- Перечень ресурсов и ссылки на них:
  - каталоги исходных файлов;
  - оперативная документация (справочная система и другие доступные в электронной форме руководства);
  - перечень бумажных документов (т.е. документов, представленных не в электронной форме).

Требования к комментариям

Комментарии должны формулироваться таким образом, чтобы, если их извлечь из кода программы, они составили осмысленное связное изложение – описание функционирования программы.

Комментарии должны документировать логику работы программы и принимаемые решения (что делается в данном месте кода и почему это делается).

Комментарии должны быть лаконичными и понятными. Следующие встроенные ключевые слова (*gotchas*) можно использовать, чтобы отметить места, чреватые проблемами.

- *Gotcha Keywords* – ключевые слова в комментарии.

  :TODO: topic. Означает: ЗДЕСЬ НАДО ДОРАБОТАТЬ. НЕ ЗАБУДЬ.

  :BUG: [bugid] topic. Означает: ЗДЕСЬ НАХОДИТСЯ ИЗВЕСТНЫЙ BUG. Объясните его причину и присвойте ему идентификатор (*bug ID*)

- *:KLUDGE:* Если вы сделали что-то неудачное, объясните, как это произошло, и что вы намерены делать впредь, чтобы избежать этого.
- *:TRICKY:* Предупреждение, что следующая часть кода очень сложная, ее не следует изменять без тщательного обдумывания.
- *:WARNING:* Предупреждение о чем-либо.
- *:COMPILER:* Иногда возникают проблемы с компилятором. Опишите эти проблемы. Проблемы могут быть решены позднее.
- *:ATTRIBUTE: value*. Общий вид атрибута, вставленного в комментарий. Вы можете задавать собственные атрибуты, которые затем могут быть извлечены из комментария.

В дополнение вы можете использовать возможности препроцессора, но имейте в виду, что не все компиляторы их поддерживают в равной мере.

При использовании ключевых слов *gotcha* само ключевое слово *Gotcha* должно быть первым словом комментария.

Комментарий может состоять из нескольких слов, но первое слово должно отражать смысл комментария, а последующие слова служат лишь для уточнения.

Имя автора и дата внесения замечания должны быть частью комментария.

Часто *gotcha* остаются в программе дольше, чем это необходимо.

Данные о дате и авторе изменений позволят другим программистам лучше ориентироваться и принимать решения. Зная имя автора изменений, вы сможете связаться с ним и получить консультацию.

Пример:

```
// :TODO: tmh 960810: possible performance problem
// We should really use a hash table here but for now we'll use a linear search.

// :KLUDGE: tmh 960810: possible unsafe type cast
// We need a cast here to recover the derived type. It should
// probably use a virtual method or template.
```

Разное

Оператор *if…else*

Если *if* завершается оператором *return*, то не используйте *else*. Код получается более понятным. Так, вместо

```
 if (condition) return 0;
else
{
   do_something();
}
```

лучше написать

```
if (condition) return 0;
do_something();
```

В этом случае последним *return* может быть выдача кода ошибки (это позволит отловить «случайное» попадание в ненужную ветвь алгоритма). Необходимо стараться проверять все альтернативные возможности (в том числе все «предельные случаи»).

Циклы

По возможности следует избегать циклов *do…while*. Они опасны тем, что тело цикла исполняется всегда хотя бы один раз. Кроме того, при чтении программы очень трудно отыскать проверяемое условие, т.к. *while* в такой конструкции располагается «внизу», т.е. в конце тела цикла. По этим причинам никогда не следует употреблять *do…while* для организации бесконечного цикла – для этой цели намного лучше использовать *while(1)*.

Всегда используйте оператор *for*, если присутствуют любые два из инициализирующего, условного или инкрементного выражений. Это сокращает код и делает его более понятным.

Препроцессор

Обязательно заключайте в скобки тела сложных макросов и их параметры. Например, следует писать

```
#define TWO_K (1024+1024)          //correct!
```

а не

```
#define TWO_K 1024+1024            //incorrect!
```

В последнем случае *TWO_K*10* будет вычислено на этапе компиляции как *1024+1024*10=11264* вместо ожидаемого *20480*.

Из приведенного примера видно, что макросы может быть достаточно трудно сопровождать. Поэтому:

- для задания констант, используемых в программе (кроме системо-зависимых констант) по возможности следует использовать модификатор *const* или перечисление *enum* и избегать использования *#define*;
- вместо параметризованных макросов *#define* лучше использовать *inline*-функции.

Константы и перечисления

Следует помнить, что *const int* на самом деле не задает константу, а резервирует память под *int* (хотя и запрещает менять содержащееся в ней значение). Под перечисление *enum* память никогда не выделяется, поэтому использование *enum* может быть предпочтительнее.

Мобильность

Мобильность программ – это свойство, позволяющее выполнять программы на разных компьютерах, под управлением разных операционных систем, с минимальными изменениями кода.

Естественно, мобильность предусматривает и возможность трансляции программы разными компиляторами. Думать о мобильности нужно даже в том случае, если предполагается, что программа пишется, скажем, только «под» *Windows NT* и только с использованием *MS Visual C++* (т.е. никогда не планируется использовать другой компилятор). Дело в том, что аппаратное и программное обеспечение не стоит на месте, и вполне возможно, что через несколько лет написанную вами программу придется полностью переделывать под новую версию операционной системы и, соответственно, под новый компилятор.

Зависимость от компилятора

Некоторые детали в *C/C++* не стандартизованы. Например:

- не определен порядок вычисления операндов большинства бинарных операций, таких как сложение и умножение. Следовательно, не определен порядок появления возможных побочных эффектов;
- некоторые директивы и ключевые слова (в основном это директивы препроцессора) поддерживаются только определенными компиляторами.
- старайтесь избегать явного или неявного использования нестандартных возможностей, которые предоставляет конкретный компилятор.

Зависимость от компьютера

Не следует полагаться на определенный размер машинного слова, он может быть разным у разных компьютеров.

Размеры данных базовых типов (*int, float, double, char* и т.п.) в байтах и в машинных словах могут быть разными у разных компьютеров и в разных операционных системах. Гарантировано только, что размер *short* не меньше размера *int*, размер *float* не меньше размера *double* и т.п. Размеры данных могут сказаться на обработке двоичных масок. Например, во фрагменте

```
#define MASK 0177770 // incorrect!
int x;
    ...
x &= MASK;
```

три правых бита целого *x* будут обнуляться только в том случае, если данные типа *int* занимают 16 бит.

Но если размер данных типа *int* больше 16 бит, то обнуляться будут левые биты *x*. Поэтому предпочтительнее следующий код:

```
#define MASK (~07) // correct!
int x;
 ...
x &= MASK;
```

По возможности следует вообще избегать использования двоичных масок. Это рудимент «ассемблерного» стиля программирования. Лучше использовать битовые поля.

Используйте *sizeof*() для определения размера объектов, в том числе и переменных базовых типов.

Следует тщательно проверять операции двоичного сдвига. Максимальное число бит, которые могут быть сдвинуты вправо или влево, различается на разных компьютерах и в разных операционных системах.

Максимальный размер битового поля зависит от размера машинного слова и, следовательно, не является стандартным.

При работе с объектами разных типов данных (классов) используйте преобразование типа. Особенно важно использовать преобразование типов для указателей.

Используйте макросы *#define* для задания системо-зависимых именованных констант.

Не полагайтесь на внутреннюю кодировку целых и вещественных типов данных. Например, не следует полагаться на использование дополнительного или обратного кода для представления целых типов.

Не являются стандартными порядок и число байт в машинном слове, число бит в байте (данная константа определена в файле *<values.h>)*.

Не следует полагаться на конкретные значения основных констант, таких как *NULL* и *EOF*. Например, *NULL*, как правило, является константой 0, но это вовсе не обязательно. Поэтому

```
if ( lval == NULL ) { ... }
```

лучше, чем

```
if ( !lval ) { ... }
```

Символы (*char*) могут иметь знак. Для повышения мобильности можно использовать явное описание *unsigned char* или преобразовывать символы перед обработкой к типу *unsigned char*.

Нельзя полагаться на конкретную кодировку символов, в том числе на определенную последовательность символов в кодировке.

Например,

```
if ( islower(ch) ) { ... }
```

лучше, чем

```
if ( ch >= 'a' && ch <= 'z' ) { ... }
```

так как использует стандартную библиотечную функцию *islower*(), а не полагается на предположение о последовательности строчных букв в кодировке.

Для получения и освобождения динамической памяти лучше использовать операции *new* и *delete*, а не функции *malloc*() и *free*().

О правильной организации исходного кода

Исходный код программы организован правильно, если его легко читать, модифицировать, эксплуатировать и, следовательно, переносить на другие компьютеры и в другое операционное окружение. Для правильной организации исходного кода имеет смысл придерживаться следующих рекомендаций.

1. Все определения, связанные с конкретным компьютером и/или операционной средой, следует оформлять при помощи директивы препроцессора *#define* и помещать в отдельный заголовочный файл. Туда же следует помещать системо-зависимые определения типов данных (для этой цели стоит использовать *typedef*). Стандартные системные определения типов содержатся в *<sys/types.h>*.
2. Для локализации системо-зависимых программных фрагментов следует использовать директивы условной компиляции и директиву *#define*. Для локализации системо-зависимых определений типов данных следует использовать *typedef*.
3. Следует особенно тщательно проверять число и тип аргументов, передаваемых функциям (методам классов). Для мобильного определения функций с переменным числом аргументом используйте средства, описанные в файле *<varargs.h>*. Тип передаваемых в функцию (метод) аргументов должен соответствовать типу формальных аргументов; добиться этого необходимо при помощи приведения типов.
4. Стоит активно пользоваться модификатором *const* применительно к передаваемым в функцию (метод) аргументам для указания того, что эти аргументы не могут быть изменены. Например, описание *void func (const SomeClass &);* запрещает модификацию переданного по ссылке объекта функцией *func*. В идеале все ссылочные аргументы функций (методов) должны быть описаны как *const*.
5. Следует тщательно следить за инициализацией указателей и переполнением их значений. Нельзя полагаться на инициализацию переменных (в т.ч. указателей) по умолчанию.
6. Имеет смысл пользоваться классом *String* для работы со строками. Использование *char** в качестве синонима *String* – это потенциально опасный рудимент «чистого» *C*.
7. Не следует описывать тело метода внутри определения класса. Исключения могут составлять очень короткие методы и короткие перегруженные операции. Методы, тело которых включено в определение класса, как правило, реализуются компилятором как *inline-функции*. При необходимости задать *inline*-метод (функцию) лучше сделать это путем явного объявления.
8. Следует стремиться к тому, чтобы в используемых классах все данные были *private*. Желательно избегать дружественных (*friend*) функций и классов.