Министерство образования Российской Федерации
Санкт-Петербургский государственный институт
точной механики и оптики (технический университет)

Кафедра компьютерных технологий

Е.О.Степанов, С.В.Чириков

Стиль программирования на C++

Санкт-Петербург

2001

УДК 681.3.06

Е.О.Степанов, С.В.Чириков. Стиль программирования на C++. Учебное пособие к курсам “Вычислительная геометрия и C++” и “Технология программирования”– СПб : СПб ГИТМО(ТУ), 2001. – 48 с.

В пособии рассмотрены основные требования к стилю программирования на C++ и оформлению программного кода. Выполнение предложенных рекомендаций позволит создать правильно задокументированный листинг программы. Пособие адресовано студентам программистских специальностей и может быть полезно всем разработчикам программного обеспечения.

Рекомендовано советом факультета Информационных Технологий и Программирования, протокол N 2 от 14.12.2000.

ВВЕДЕНИЕ

В настоящее время разработка программного обеспечения является интенсивно развивающейся отраслью промышленности. Время, когда программирование было уделом отдельных талантливых одиночек, закончилось уже лет 30 назад. Сегодня крупные программные проекты разрабатываются большими коллективами, причем развитие и сопровождение проекта часто длятся 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 хорошо откомментирована. Отметим, ничто так не раздражает программиста–профессионала, как небрежно написанный и плохо откомментированный код. Так же как невозможно представить инженерный чертеж, оформленный без учета требований стандартов, так и программа должна иметь некий унифицированный вид. Целью данного пособия и является разработка свода рекомендаций, следуя которым вы сможете представить свой код как документ, имеющий общую форму и доступный для изучения, а также для дальнейшего сопровождения и развития.

Правила и рекомендации, приведенные в данном пособии, не являются догмой, но мы советуем их придерживаться. Существует несколько стилей написания и оформления программ. В данном пособии описан стиль Microsoft (стиль MFC). Поскольку большинство профессиональных программистов, пишущих программы для MS Windows, разрабатывает свой код средствами Microsoft Visual Studio, мы рекомендуем взять этот стиль за образец и придерживаться его при выполнении программных проектов.

Часто приходится сталкиваться с ситуацией, когда программа сначала пишется и отлаживается и лишь затем документируется. Это неправильный подход. Во-первых, прежде чем начинать кодировать, необходимо продумать структуру программы и зафиксировать ее в документации. Во-вторых, 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 программами и классами может быть реализован с помощью макросов. В этом случае комментирование выполняется буквально «в два

щелчка» мышкой. Есть поговорка: «Посеешь привычку – пожнешь характер. Посеешь характер – пожнешь судьбу». Вырабатывайте в себе привычку все делать набело, ничего не откладывая на потом, и ваша профессиональная судьба сложится удачно.

Стоит привести еще одно соображение, интересное изучающим программирование и, в частности, студентам программистских специальностей. Многие из них задумываются о трудоустройстве. Ознакомившись с объявлениями о вакансиях, обнаруженными в Internet или в прессе, можно заметить, что все они имеют одно общее требование. Претенденту на вакансию программиста предлагается предъявить образец программного кода. Обратите внимание, речь идет не об исполняемом модуле, который может продемонстрировать ваше умение разрабатывать приложения, а именно о тексте программы. Цель этого требования очевидна. Взглянув на листинг программы, опытный программист сразу поймет, с кем он имеет дело - с хорошо обученным и имеющим опыт работы профессионалом или с любителем (пусть и очень «продвинутым»), кодирующим «для души». Излишне спрашивать - у кого больше шансов занять вакансию. Возиться с новичками никто не желает, а если их все же и берут на работу, то платят заметно меньше, чем более опытным коллегам. Если вы внимательно ознакомитесь с предлагаемыми ниже рекомендациями, ваши шансы найти место программиста и получить достойную оплату своего труда существенно возрастут. Во всяком случае, вас не отсеют сразу, при первом же взгляде на листинг вашей тестовой программы.

Приведенные в данном пособии рекомендации являются «почти» общепринятыми. В некоторых программистских фирмах аналогичного содержания документы доводятся до сведения каждого сотрудника и являются своего рода внутренним стандартом фирмы. Однако, даже если в фирме отсутствует такой документ, фиксирующий требования к стилю программирования, все равно такие требования имеют место, хотя и носят более размытый, интуитивный характер. В любом случае плохой стиль сразу заметен и корректируется административными мерами. Следуя нашим рекомендациям, вы избежите ненужных конфликтов с руководителем проекта.

1. СТАНДАРТИЗАЦИЯ – ЭТО ВАЖНО

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 «подпроектов», и отдельные исполнители не знают детально, чем, собственно, они занимаются. Каждый делает свой фрагмент, а целую картину представляет себе лишь руководитель проекта и несколько ведущих специалистов. Такая система работы особенно популярна в «почтовых ящиках» – организациях, работающих по государственным, как правило, оборонным заказам;
- когда в процессе работы происходит ротация персонала, кто-то увольняется, появляются новые сотрудники, которых нужно быстро ввести в курс дела;
- наконец, должны быть некие общие правила поведения, без которых невозможна слаженная работа коллектива разработчиков и их взаимодействие с заказчиками.

Одним словом, если вы занимаетесь разработкой достаточно большого и длительного проекта, вам придется следовать правилам работы того коллектива, в котором вы работаете.

Если проект имеет, например, научный (а не «промышленный») характер, все равно вам следует придерживаться общепринятого стиля, если вы, конечно, ожидаете, что ваша работа заинтересует других, и кто-то захочет разобраться в ваше программе. 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 помнит, с какой целью он написал тот или иной фрагмент кода. Такого не случится, если программа будет прокомментирована.

2. ИМЕНА

Имя – результат размышлений, в основе которых лежит понимание работы системы в целом. Имя должно соответствовать тому 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.

2.1. 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.

2.2. 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 данной фирме, в качестве префикса можно использовать сокращенное имя фирмы.

  Пример:

```
class ArcNewPortSemaphor     // Arc – от "ARCADIA"
class NitOldPortSemaphor     // Nit – от "NITA"
```

2.3. Имена методов класса

- Используйте те же правила, что и для имен классов.
- Имя не должно напоминать ругательство. Длинное имя обычно лучше, чем короткая аббревиатура, однако не следует злоупотреблять.
- Обычно каждый метод и функция выполняет некоторое действие, поэтому имя должно описывать это действие:

`CheckForErrors()` вместо `ErrorCheck()`,

`DumpDataToFile()` вместо `DataFile()`.

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);
};
```

2.4. Имена функций

- Используйте те же правила, что и для имен методов классов.

Пример:

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

2.5. Имена аргументов функций и методов класса

- Используйте те же правила, что и для имен классов.
- Используйте упрощенную Венгерскую нотацию (см. *Приложение 1*).

  Пример:

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

2.6. Имена переменных

- Используйте упрощенную Венгерскую нотацию (см. *Приложение 1*).

2.7. Имена свойств класса

- Используйте упрощенную Венгерскую нотацию (см. *Приложение 1*).

- Имена свойств должны иметь префикс '*m_*'.
- После '*m_*' следует имя свойства, имеющее ту же структуру, что и имя переменных.
- '*m_*' всегда должно 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;
};
```

2.8. 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;
};
```

2.9. 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;
};
```

2.10. Имена глобальных переменных

- Используйте упрощенную Венгерскую нотацию (см. *Приложение 1*).
- Глобальной переменной должен предшествовать префикс “g_”.

Пример:

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

2.11. Имена статических переменных

- Используйте упрощенную Венгерскую нотацию (см. *Приложение 1*).
- Статической переменной должен предшествовать префикс “s_”.

Пример:

```
class CTest
{
  private:
       static StatusInfo ms_Status;
};

static int    s_nReadBytes;
static char*  s_pClassName;
```

2.12. 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 nLocalConst     = 4;
```

2.13. Имена макроопределений #define

- Константы, определяемые оператором `#define,` должны состоять из заглавных букв и сепараторов “_”.

- Рекомендуется использовать префиксы, чтобы сгруппировать константы в категории.

  Пример:

```
#define MAX(a,b)          blah
#define IS_ERR(err)       blah

#define ERRMSG_NO_RAM     0
#define ERRMSG_WRITE      1
#define ERRMSG_READ       2
```

2.14. Имена констант const

- 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, чтобы не смешивать константы с макросами.

Пример:

```
const int default_lval=0;
```

2.15. Имена перечислений enum

- Некоторые программисты используют `enum` вне класса для задания константы, в этом случае следует убедиться, что имя данной константы не совпадает с другой константой, заданной операторами `const` или `#define`.

  Пример:

```
enum PinStateType
{
  pin_off=0,
  pin_on
};
```

- 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`:

  `Aclass::pin_off.`

- enum можно использовать для задания неинициализированных или ошибочных состояний.

Пример:
```
enum{state_err, state_open, state_running, state_dying};
```

2.16. Имена типов данных

Имена типов данных, 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 привести тип

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

но не определить функцию.

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

2.17. Имена меток

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

  Пример:
  ```
  FIRST_JUMP:
  goto restart_point;
  ```

2.18. Имена файлов

- Файлы программ 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 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.

3. ФОРМАТИРОВАНИЕ

3.1. Общие требования

- 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.

3.2. 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. Принимая во внимания требование удобства отладки, мы утверждаем, что возможность одновременно иметь несколько открытых окон важнее, чем возможность иметь одно, но большое окно. Из этого соображения также следует ограничение на длину строки.

3.3. Использование для организации отступов табуляции

и пробелов

- Используйте для организации отступов табуляцию вместо пробелов.
- На размер отступа не накладывается ограничений, но злоупотреблять отступами не стоит. Если размер отступа более 4 или 5 пробелов, ваш 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 должны быть выровнены 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;
}
```

3.4. 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;`

Здесь:

- 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.

3.5. Выравнивание блока объявлений

- Блок объявлений должен быть выровнен по типу, имени, начальному значению и комментарию.
- Для выравнивания используйте пробелы, а не табуляции.
- Символы ‘&’ и ‘*’ должны следовать без пробелов за объявлением типа, а не имени.
- Целью выравнивания является улучшение ясности и читаемости кода.

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;

         ...
}
```

3.6. Расстановка фигурных скобок

Существует несколько традиций расстановки скобок, наиболее употребительными являются следующие:

- традиция фирмы 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 (...)
{
     ...
}
```

3.7. 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);
}
```

3.8. Форматирование операторов if-else

- При сравнении с константой ее лучше размещать справа от оператора “==” или “!=”.

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

- Выравнивать `if/else` следует так, как это показано в приведенном ниже примере.

Пример:

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

3.9. 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;
}
```

3.10. Комментарии

- Строка комментария должна иметь такой же отступ, как и операция, которую она комментирует.
- `In-line` комментарии должны размещаться справа от комментируемого кода.
- Операторы комментария “`/*`” и “`//`” должны отделяться от текста комментария по крайней мере одним пробелом.
- `In-line` комментарии должны быть выровнены 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;
}
```

3.11. Методы и функции

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

  Пример: Функция с коротким списком формальных параметров:

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

  Функция с длинным списком формальных параметров:

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

4. ДОКУМЕНТИРОВАНИЕ ИСХОДНОГО КОДА

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

4.1. Общие требования

- Документация к исходному коду, содержащаяся в комментариях, должна быть достаточной для полного понимания кода и его дальнейшего сопровождения как для разработчиков (проектировщиков) системы, 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 документировать. Следует правильно выбирать имена переменных, функций и классов. Необходимо структурировать код, такой код требует меньше комментирования.
- Имейте в виду, что комментарии в заголовочных h-файлах предназначены для пользователей классов, тогда как комментарии в CPP-файлах реализации предназначены для разработчиков этих 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 комментарных строк. Тактический комментарий задается одной строкой и описывает операцию на следующей строке. Его также можно размещать справа от комментируемой операции. Слишком много тактических комментариев делает код программы нечитабельным, по этой причине рекомендуется применять стратегическое комментирование. Общее представление важнее деталей реализации кода.

4.2. 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 должны содержать:
    - 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.

4.3. 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 information supposed to be useful, e.g.,
                      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.

Смотри *Приложение 2*: пример заголовка исходного файла.

4.4. 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)
*****************************/
```

Приведем ниже описание каждого раздела.

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.

Section	Description
..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.

- Если некоторый раздел не содержит информации, его можно опустить, но порядок разделов менять не разрешается.
- 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 откомментированному в соответствии с формальными правилами тексту программы (например, Cweb, описанная в книге Donald. E. Knuth, S. Levy. The Cweb system of structured documentation. Reading: Addison-Wesley, 1994). Такие утилиты очень помогают, в особенности, тем, кто, не являясь разработчиком, должен модифицировать программу.
- 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).

  Пример: смотри *Приложение 2*.

4.5. 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:

```
#ifdef   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 задокументирована, 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 information supposed to be useful, e.g.,
                    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.

Пример: смотри *Приложение 2*.

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
]=======*/
     #ifndef 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 разделов менять не разрешается.

Пример: Смотри *Приложение 2*.

4.6. **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
{
     …
};
```

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

Однако не все компиляторы их поддерживают (например, MS Visual C++ их не поддерживает). Поэтому 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 re-
ceive
                                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
)
{
    ...
}
```

4.9. 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 параметр – только для чтения и изменению не подлежит.
  - ***OUT*** – означает, что данный параметр будет изменен при выполнении метода, однако его начальное значение методом не используется.
  - ***INOUT -*** означает, что данный параметр будет изменен при выполнении метода, а его начальное значение методом используется.

  Однако не все компиляторы их поддерживают (например, MS Visual C++ их не поддерживает). Поэтому 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 name
                                 should 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
)
{
     ...
}
```

4.10. 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 др. доступные в электронной форме руководства);
  - перечень бумажных документов (т.е. документов, представленных не в электронной форме).

4.11. 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 принимаемые вами решения (что вы делаете в данном месте программы и почему).
- Комментарии должны быть лаконичными и понятными. 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 в будущем.
- **:TRICKY:** Предупреждение, что следующая часть программы очень сложная, ее не следует изменять без тщательного обдумывания.
- **:WARNING:** Предупреждение о чем-либо.
- **:COMPILER:** Иногда требуется решить проблему с компилятором. Задокументируйте ее. Проблема может быть решена 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.

```
#define chSTR(x)        #x
#define chSTR2(x)       chSTR(x)
#define TODO(desc)      message(":TODO:("__FILE__",     \
                            "chSTR2(__LINE__) ") " #desc)
#define BUG(desc)       message(":BUG:("__FILE__",\
                        "chSTR2(__LINE__) ") " #desc)
```

Когда вы вставляете эти строки в ваш `.h/.cpp` – файл, вы можете использовать их следующим образом:

```
main()
{
  if (condition)                // in-line comment
  {
    #pragma TODO("Add error handling here")
    ...
  }
  else if (condition)           // in-line comment
  {
    ...
  }
  else                          // in-line comment
  {
    #pragma TODO("There is no error handling here")
    ...
  }
}
```

Это позволит видеть все потенциальные проблемы программы во время ее компиляции.

- Форматирование кода программы при использовании ключевых слов Gotcha.
- Ключевое слово Gotcha должно быть первым словом комментария.
- Комментарий может состоять из нескольких слов, но первое слово должно содержать 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 принять другим программистам решение. Зная имя автора, 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.
```

5. ОПЕРАТОРЫ

5.1. 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”).

5.2. 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.

5.3. Препроцессор

- Обязательно заключайте в скобки тела сложных макросов и их аргументы. 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 системозависимых параметров) по возможности следует использовать модификатор `const` или перечисление `enum` и избегать использования `#define`;
- вместо параметризованных макрооопределений `#define` лучше использовать `inline`-функции.

5.4. 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.

6. 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 «старой» операционной системы и, соответственно, под новый компилятор.

6.1. 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 опреде-

лен порядок появления возможных побочных эффектов;

- некоторые директивы и ключевые слова (в основном это директивы препроцессора) поддерживаются только определенными компиляторами.

Не следует явно или неявно пользоваться нестандартными возможностями, предоставляемыми конкретным компилятором.

6.2. 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, в фрагменте

```
#define MASK 0177770  // incorrect!

int x;
  …
x &= MASK;
```

  три правых бита целого `x` будут обнуляться только в 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

```
#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 макроопределения `#define` для задания системозависимых 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 выделения/освобождения памяти лучше использовать `new/delete`, а не `malloc()/free()`.

7. ПРАВИЛЬНАЯ ОРГАНИЗАЦИЯ ПРОГРАММ

Программа организована правильно, если ее легко читать, модифицировать, эксплуатировать и, следовательно, переносить на другие компьютеры и в другое операционное окружение. Для правильной организации программ имеет смысл придерживаться следующих рекомендаций.

- 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 системозависимые определения типов данных (для этой цели стоит использовать `typedef`). Стандартные системные определения

типов содержатся в <sys/types.h>.

- Для локализации системозависимых программных фрагментов следует использовать директивы условной компиляции и директиву #define. Для локализации системозависимых определений типов данных следует использовать typedef.
- 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 параметров; добиться этого необходимо при помощи приведения типов.
- Стоит активно пользоваться модификатором const применительно к передаваемым в функцию (метод) параметрам для указания того, что эти параметры 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.
- Следует тщательно следить за инициализацией указателей и переполнением их значений. Нельзя полагаться на инициализацию переменных (в т.ч. указателей) по умолчанию.
- Имеет смысл пользоваться классом String для работы со строками. Использование char* в качестве синонима String – это потенциально опасный рудимент «чистого» C.
- Не следует описывать тело метода внутри определения класса. Исключения могут составлять очень короткие методы и короткие перегружаемые операторы. Методы, тело которых включено в определение класса, как правило, реализуются компилятором как inline. При необходимости задать inline-метод (функцию) лучше сделать это путем явного объявления.
- Следует стремиться к тому, чтобы в используемых классах все данные были private. Желательно избегать дружественных (friend) функций и классов.

8. ПОЛЕЗНЫЕ КНИГИ И СТАТЬИ

1. Steve Maguire. Writing Solid Code. //Microsoft Press, 1993.
2. Jocelyn Garner, Senior Technical Writer for MFC. Microsoft Foundation Class Library Development Guidelines. Microsoft Corporation, August 1995.

3. Saks, Dan, and Tom Plum. C++ Programming Guidelines. //Cardiff, NJ: Plum Hall, 1991.
4. Meyers, Scott. Effective C++. // MA: Addison-Wesley, 1992.
5. Murray, Rob. C++ Strategies And Tactics. // MA: Addison-Wesley, 1993.

ПРИЛОЖЕНИЕ 1: ВЕНГЕРСКАЯ НОТАЦИЯ

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	*lpfnAbort*

Prefix	Type	Description	Example
		function	
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*

ПРИЛОЖЕНИЕ 2: ПРИМЕРЫ ИСХОДНЫХ И ЗАГОЛОВОЧНЫХ ФАЙЛОВ

```
/*************************************************************
\
  FILE.........: abrfname.cpp
  AUTHOR.......: Dmitriy Viktorov
  DESCRIPTION...: The module contains function for getting an
                  Abbreviate file name from given name.
  FUNCTIONS.....: AbbreviateFileName(), GetFileName()
  SWITCHES......: WIN32    - if defined, 32-bit version is
                  compiled, otherwise 16-bit edition is compiled.
                  _UNICODE - if defined, UNICODE support is
                  provided.
  COPYRIGHT.....: Copyright (c) 1996-97, Arcadia Inc.
  HISTORY.......: DATE     COMMENT
                  --------  ------------------------------------
-
                  12-23-96 Created - Dmitriy
                  02-12-97 Fixed bug with long file names -
                           Sergey
\*************************************************************
/


/*===============================[                          SPECIAL
]==================*/

#define STRICT

/*===============================[       IMPORT       DECLARATIONS
]===+====*/

#include "windows.h"

/*===============================[       PUBLIC       DECLARATIONS
]========*/

#include "abrfname.h"
```

```
/*================================[       PRIVATE       DECLARATIONS
]=======*/


/*================================[       ..PRIVATE       CONSTANTS
]========*/

const TCHAR cNull   = _T('\0');
const TCHAR cRSlash = _T('/');
const TCHAR cLSlash = _T('\\');
const TCHAR cColon  = _T(':');


/*================================[       ..PRIVATE       FUNCTIONS
]========*/

UINT PASCAL GetFileName(LPCTSTR lpszPathName, LPTSTR lpszTitle,
                        UINT nMax);




/*================================[   ..PRIVATE   PSEUDO   FUNCTIONS
]=*/

#ifndef _tcsinc
     #ifdef _WIN32
          #define _tcsinc(x) CharNext(x)
     #else
          #define _tcsinc(x) AnsiNext(x)
     #endif // _WIN32
#endif

#ifndef _MFC_VER  // MFC specific
     #define ASSERT ((void)0)
#endif

/*================================[          PUBLIC          FUNC-
TIONS]===========*/

/*****************************************************************
\
  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 name
                                 should 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
(
     LPTSTR lpszCanon,      // INOUT
     int    ichMax,         // IN
     BOOL   bAtLeastName    // IN
)
{
     int ichFullPath;
     int ichFileName;
     int ichVolName;

     const TCHAR* lpszCur;
     const TCHAR* lpszBase;
     const TCHAR* lpszFileName;

     ASSERT(lpszCanon != NULL);
     if(lpszCanon == NULL)
          return;

     lpszBase     = lpszCanon;
     ichFullPath  = lstrlen(lpszCanon);
     ichFileName  = GetFileName(lpszCanon, NULL, 0) - 1;
     lpszFileName = lpszBase + (ichFullPath - ichFileName);

     // If ichMax is more than enough to hold the full path name,
     // we'redone.
     // This is probably a pretty common case, so we'll put it
     // irst.

     if (ichMax >= ichFullPath)
          return;

     // If ichMax isn't enough to hold at least the basename,
     // we're done

     if (ichMax < ichFileName)
     {
```

```
        lstrcpy(lpszCanon, (bAtLeastName) ? lpszFileName :
&cNull);
        return;
      }
      // Calculate the length of the volume name. Norm. = two
      chars
      // (e.g., "C:", "D:",etc.), but for a UNC name, it could be
      // more (e.g., "\\server\share").
      //
      // If ichMax isn't enough to hold at least <volume_name>\
      ...
      // \<base_name>, the result is the base filename.
}

/*==============================[        PRIVATE        FUNCTIONS
]==========*/

/*****************************************************************
\
  FUNCTION......: GetFileName
  DESCRIPTION...: Returns the file name from the given full path
                  name.
  ARGUMENTS.....: lpszPathName - pointer to the given full path
                  name,
                  lpszTitle    - pointer to a buffer where the
                                 file name will be placed,
                  nMax         - max size of the buffer for the
                                 resultant file name.
  RETURNS.......: The return value is the number of bytes copied
                  to lpszTitle. If lpszTitle is NULL, the func-
          tion
                  returns the number of bytes enough to hold the
                  file name.
\*****************************************************************
/

UINT PASCAL GetFileName
(
      LPCTSTR lpszPathName,   // IN
      LPTSTR  lpszTitle,      // INOUT
      UINT    nMax            // IN
)
{

#ifdef _MFC_VER  // MFC specific
      ASSERT(lpszTitle == NULL ||
            AfxIsValidAddress(lpszTitle, _MAX_FNAME));
            ASSERT(AfxIsValidString(lpszPathName));
#endif

      // always capture the complete file name including extension
      // (if present)
      LPTSTR  lpszTemp = (LPTSTR)lpszPathName;
      for( LPCTSTR lpsz = lpszPathName; *lpsz != cNull;
```

```
          lpsz =_tcsinc(lpsz))
     {
          // remember last directory/drive separator
          if (cLSlash == *lpsz ||
              cRSlash == *lpsz || cColon == *lpsz)
               lpszTemp = (LPTSTR)_tcsinc(lpsz);
     }

     // lpszTitle can be NULL which just returns
     // the number of bytes
     if (lpszTitle == NULL)
          return (lstrlen(lpszTemp) + 1);

     // otherwise copy it into the buffer provided
     return lstrcpyn(lpszTitle, lpszTemp, nMax);
}

/** (END OF FILE  : abrfname.cpp)
*****************************/

/*****************************************************************
\
  FILE.........: abrfname.h
  AUTHOR.......: Dmitriy Viktorov
  DESCRIPTION...: The header file contains AbbreviateFileName
                  function declaration.
  FUNCTIONS.....: AbbreviateFileName()
  SWITCHES......: WIN32    - if defined, 32-bit version is
                  compiled, otherwise 16-bit edition is compiled.
                       _UNICODE - if defined, UNICODE support is
                       provided.
  COPYRIGHT.....: Copyright (c) 1996-97, Arcadia Inc.
  HISTORY.......: DATE     COMMENT
                  -------- ------------------------------------
-
                  12-23-96 Created - Dmitriy
\*****************************************************************
/



/*================================[      REDEFINITION      DEFENCE
]=======*/

#ifndef ABRFNAME_H
#define ABRFNAME_H

/*================================[                         SPECIAL
]===================*/

#if !defined(_INC_WINDOWS) || !defined(_WINDOWS_)
     #error include 'windows.h' before including this file
#endif
```

```
#ifdef WIN32
     #include <tchar.h>
#else
     #define LPCTSTR LPCSTR
     #define LPTSTR  LPSTR
     #define TCHAR   CHAR
     #define _T(x)
#endif

/*================================[         PUBLIC         FUNCTIONS
]===========*/

void PASCAL AbbreviateFileName(LPTSTR lpszCanon, int ichMax,
                                       BOOL bAtLeastName);

/*================================[    END    REDEFINITION    DEFENCE
]===*/

#endif /* ABRFNAME_H */
/**       (END       OF       FILE              :       abrfname.h)
*********************************/
```

Евгений Олегович Степанов,
Сергей Васильевич Чириков
Стиль программирования на C++
Учебное пособие

Компьютерная верстка С.В.Чириков
Дизайн С.В.Чириков
Редакционно-издательский отдел СПб ГИТМО(ТУ)
Зав. РИО Н.Ф.Гусарова
Лизензия ИД N00408 от 05.11.99
Подписато к печати 06.12.00
Отпечатано на ризографе Заказ N 292

Тираж 150 экз.