Кисти, карандаши и многое другое...

**Глава 8.**

**Кисти, карандаши и многое другое...**

* [Изменение начального размера окна](http://www.realcoding.net/articles/glava-8-kisti-karandashi-i-mnogoe-drugoe.html#1)
* [Создание графических объектов Windows](http://www.realcoding.net/articles/glava-8-kisti-karandashi-i-mnogoe-drugoe.html#2)
* [Создание шрифтов](http://www.realcoding.net/articles/glava-8-kisti-karandashi-i-mnogoe-drugoe.html#3)
* [Создание кистей](http://www.realcoding.net/articles/glava-8-kisti-karandashi-i-mnogoe-drugoe.html#4)
* [Создание битовых массивов](http://www.realcoding.net/articles/glava-8-kisti-karandashi-i-mnogoe-drugoe.html#5)
* [Создание карандашей](http://www.realcoding.net/articles/glava-8-kisti-karandashi-i-mnogoe-drugoe.html#6)
* [Работа с регионами](http://www.realcoding.net/articles/glava-8-kisti-karandashi-i-mnogoe-drugoe.html#7)
* [Создание регионов](http://www.realcoding.net/articles/glava-8-kisti-karandashi-i-mnogoe-drugoe.html#8)
* [Создание прямоугольных регионов](http://www.realcoding.net/articles/glava-8-kisti-karandashi-i-mnogoe-drugoe.html#9)
* [Создание эллиптических регионов](http://www.realcoding.net/articles/glava-8-kisti-karandashi-i-mnogoe-drugoe.html#10)
* [Создание сложных регионов на базе многоугольников](http://www.realcoding.net/articles/glava-8-kisti-karandashi-i-mnogoe-drugoe.html#11)
* [Создание регионов с закругленными углами](http://www.realcoding.net/articles/glava-8-kisti-karandashi-i-mnogoe-drugoe.html#12)
* [Комбинирование регионов](http://www.realcoding.net/articles/glava-8-kisti-karandashi-i-mnogoe-drugoe.html#13)
* [Изменение размеров окна](http://www.realcoding.net/articles/glava-8-kisti-karandashi-i-mnogoe-drugoe.html#14)
* [Обработка сообщения WM\_PAINT](http://www.realcoding.net/articles/glava-8-kisti-karandashi-i-mnogoe-drugoe.html#15)
* [Организация процесса рисования](http://www.realcoding.net/articles/glava-8-kisti-karandashi-i-mnogoe-drugoe.html#16)

                                                В проблеске гениальности видишь свою бездарность

                                                                                                   Станислав Ежи Леи

Начиная с этой главы, мы будем заниматься разработкой приложения, задача которого — отображать текущее время в виде аналоговых часов (т. е. часов с циферблатом и стрелками). Естественно, что основная цель преследуется несколько иная, а именно — показать использование большей части того, что предоставляет библиотека классов MFC в плане работы с графикой. По этой причине многие фрагменты кода написаны "не оптимально" — но это не должно вас смущать: благодаря такому подходу мы рассмотрим различные варианты решения одной и той же задачи (где-нибудь это вам обязательно пригодится).

Основа для нашего приложения у вас уже есть — это пример из главы 5, который, как вы помните, создавался именно для этой цели. Однако все-таки начнем по порядку. Класс приложения CClockApp пока трогать не будем, поскольку то, что создал мастер, нас вполне устраивает — связь с системой установлена.

Таким образом, у нас есть два класса — CMamFrame и CChildWnd. каждый из которых отвечает за свою часть работы приложения: первый — за общий вид, а второй за то, что пользователь будет видеть в окне.

Займемся сначала общим видом.

Изменение начального размера окна

Взгляните на следующий код, где полужирным шрифтом выделен фрагмент. который необходимо добавить к каркасу:

BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)

 {

if(ICFrameWnd::PreCreateWindow(cs))

return FALSE;

// Определяем размеры окна в зависимости от размеров экрана

**int nSize = min (GetSystetriMetrics (SM\_CXMAXIMIZED) ,**

**GetSystemMetrics(SM\_CYMAXIMIZED)) \* 3 / 5;**

// Задаем горизонтальный и вертикальный размеры окна

**cs.cx = nSize;**

**cs.су = nSize;**

cs.dwExStyle &= ~WS\_EX\_CLIENTEDGE;

 cs.lpszName = "Приложение 'Кварцевые часы'";

// Устанавливаем свою пиктограмму

cs.lpszClass = AfxRegisterWndClass(

CS\_HREDRAW | CS^HREDRAW | CS\_DBLCLKS,

::LoadCursor(NULL, IDC\_ARROW),

HBRUSH (COLOR\_WINDOW + 1) ,

AfxGetApp()->LoadIcon(IDR\_MAINFRAME));

return TRUE;

 }

Как видите, сначала мы определяем ширину и высоту экрана, выбираем из них минимальное значение и устанавливаем размеры окна равными трем пятым этой величины.

Вот, собственно, пока и все, что необходимо добавить в код класса CMainFrame, поскольку никакого взаимодействия с пользователем в данном приложении не предполагается.

Для вывода надписей нам понадобятся шрифты, а для создания внешнего вида часов — кисти и карандаши. Чтобы не создавать их каждый раз при выводе рисунка (и, соответственно, не удалять после окончания вывода), сделаем это непосредственно в классе. В этом случае они будут освобождены при удалении объекта класса CChildWnd.

class CChildWnd : public CWnd

{

private:

...

CFont m\_Font\_l;

CFont m\_Font\_2;

 CFont m\_Font\_3;

 CBrush m\_wndBrush;

СPen m wndPen;

CBrush m\_\_arrBrush;

CPen m\_arrPen;

CBrush m\_clkBrush;

CPen m\_clkPen;

CBrush m\_rndBrush;

CPen m\_rndPen;

CBrush m\_crnBrush;

CPen m\_crnPen;

...

};

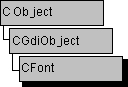
Зачем нам так много графических объектов — три шрифта и по пять кистей и карандашей — мы разберемся по ходу создания приложения, а пока запомним, что объекты Windows создаются в два этапа: сначала объект некоторого класса библиотеки MFC, а затем, вызовом соответствующей функции с префиксом Create, непосредственно объект Windows. Пока мы выполнили только первый этап — создали объекты соответствующих классов. По--•тому переходим ко второму этапу.

Создание графических объектов Windows

Как вы думаете, где лучше всего реализовать действия второго этапа по созданию графических объектов Windows? Надеюсь, что вы догадались. Действительно, в обработчике сообщения WM\_CREATE — функции On Create. Поэтому первое, что надо сделать — это определить обработчик On Create. Как это сделать, мы уже рассматривав. После этого можно переходить непосредственно к созданию графических объектов Windows. И начнем мы со шрифтов.

Создание шрифтов

Для работы со шрифтами в библиотеке MFC реализован класс СFont рис. 8.1), и для более ясного понимания дальнейших действий рассмотрим его несколько подробнее.



**Рис. 8.1**. Место класса CFont в иерархии библиотеки MFC

Класс CFont инкапсулирует графический объект Windows "шрифт". Для создания объекта этого класса имеется конструктор без параметров, однако чтобы создать собственно шрифт, необходимо вызвать одну из следующих рункций: CreateFont, CreateFontlndirect, CreatePointFont или CreatePointFontlndirect. Для использования созданного шрифта необходимо установить его в конкретный контекст устройства, а по завершении использования заменить ранее установленным шрифтом — все как и для других графических объектов.

Многие классы, инкапсулирующие графические объекты Windows, имеют в своем составе функции создания объектов при помощи структур. Именно с такого подхода мы и начнем. В данном случае это связано еще и с тем, что структура LOGFONT, используемая для создания шрифта, достаточно часто применяется не только вместе с классом CFont, но и, например, при выборе шрифта в блоке диалога, созданного при помощи класса CFontDialog.

Итак, для создания шрифта можно использовать функцию

BOOL CFont::**CreateFontlndirect** (const LOGFONT\* IpLogFont)

Параметр IpLogFont указывает на структуру LOGFONT, которая, в свою очередь, определяет шрифт, необходимый приложению.

Остановимся более подробно на составе структуры LOGFONT и описании ее полей.

typedef struct tagLOGFONT {

LONG ifHeight;

LONG IfWidth;

LONG IfEscapement;

LONG IfOrientation;

LONG IfWeight;

BYTE Ifltalic;

BYTE IfUnderline;

BYTE IfStrikeOut;

BYTE IfCharSet;

BYTE IfOutPrecision;

BYTE IfClipPrecision;

BYTE IfQuality;

BYTE IfPitchAndFamily;

TCHAR lfFaceName[LF\_FACESIZE];

 } LOGFONT;

Поле IfHeight определяет высоту (в логических единицах) знакомест или символов шрифта. Высота последних рассчитывается как разность между полной высотой знакоместа и размером области для отображения специальных знаков, например, ударения. Это поле структуры LOGFONT может интерпретироваться, в зависимости от знака, следующим образом:

> 0

Значение преобразуется в единицы устройства вывода и используется для выбора из имеющегося набора шрифта, соответствующего по высоте знакоместа

0

Для нахождения шрифта используется значение высоты по умолчанию

< 0

Значение преобразуется в единицы устройства вывода и используется для выбора из имеющегося набора шрифта, соответствующего по высоте символа

При поиске шрифта выбирается шрифт с наиболее подходящей высотой знакоместа или символа (высота должна быть меньше или равна запрашиваемой).

Для режима отображения ММ\_ТЕХТ высота может быть определена через размер задаваемого в точках (points) шрифта следующим образом:

IfHeighc = -MulDiv { PointSize,dc.GetDeviceCaps( LOGPIXELSY) ,72) ;

**Примечание**

Режимы отображения мы рассмотрим несколько позже.

Поле IfWidth определяет среднюю ширину символов в логических единицах. Нулевое значение поля IfWidth обеспечивает выбор из имеющегося набора шрифта, наиболее подходящего к конкретному устройству по возможностям масштабирования.

Поле IfEscapement определяет угол (в десятых лолях градуса) между базовой линией выводимого текста и горизонталью

Поле IfOrientation определяет угол (в десятых долях градуса) между базовой линией каждого выводимого символа и горизонталью

К сожалению, в рамках Windows 95-98 значения обоих полей IfEscapement и IfOrientation должны быть равны, т. е. существует возможность вывода под наклоном только текста всей строки, когда базовые линии символов лежат на базовой линии строки. 3 Windows NT при установке графического режима GM\_ADVANCED устанавливаемого функцией Л'т32 АР! ::SetGraphicsMode, возможности по выводу текста расширяются.

Поле IfWeight определяет толщину линий начертания символов и может принимать значения от 0 до 1000. При задании нуля используется значение по умолчанию. Некоторым значениям даны символьные наименования:

FW\_DONTCARE        О

FW\_THIN            100

FW\_EXTRALIGHT      200

FWJJLTRAUGHT       200

FWJJGHT            300

FW\_NORMAL          400

FW\_REGULAR         400

FW\_MEDiUM          500

FW\_SEMIBOLD        600

FW\_DEMiBOLD        600

FW\_BOLD            700

FW\_EXTRABOLD       800

FW\_ULTRABOLD       800

FWJHEAVY           900

FW\_BLACK           900

Поле Ifltalic определяет курсивное начертание шрифта.

Поле //Underline определяет подчеркнутое начертание шрифта.

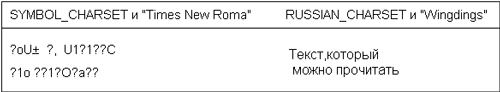
Поле IfStrikeOut определяет перечеркнутое начертание шрифта.

Поле IfCharSet определяет таблицу кодировки шрифта. Определены следующие таблицы: ANSI\_CHARSET. DEFAULT\_CHARSET, SYMBOL\_CHARSET, SHIFTJIS\_ CHARSET, GB2312\_CHARSET, HANGEUL\_CHARSET, CHINESEBIG5\_CHARSET. OEM\_CHARSET.

В рамках Windows 95/98 дополнительно определены следующие таблицы: JOHAB\_CHARSET, HEBREW\_CHARSET, ARABIC\_CHARSET, GREEK CHARSET, TURKISH\_CHARSET, THAI\_CHARSET, EASTEUROPE\_CHARSET, RUSSIAN\_CHAR-SET, MAC\_CHARSET, BALTIC\_CHARSET.

Значение OEM\_CHARSET определяет таблицу кодировки конкретной версии операционной системы. Значение DEFAULT\_CHARSET для идентификации шрифта требует задания имени (поле IfFaceName) и размера, однако отсутствие запрашиваемого шрифта может привести к непредсказуемым результатам, т. к. он будет подменен шрифтом, подходящим по другим параметрам, но неизвестной заранее кодировки.

При задании в явном виде имени шрифта особенно важно правильно задать таблицу кодировки, т. к. именно кодировка **в первую очередь определяет набор установленных шрифтов**, среди которых будет произведен поиск по имени (рис. 8.2).



**Рис. 8.2**. Пример доминирования заданной таблицы кодировки над именем шрифта

Поле IfOutPrecision определяет механизм выбора шрифта, т. е. насколько шрифт должен соответствовать заданным параметрам, прежде всего типу шрифта (растровый, системный или шрифт TrueType), а также ширине, высоте, ориентации и т. д. Это поле может принимать одно из следующих значений:

OUT\_CHARACTER\_PRECIS

 He используется

OUT\_DEFAULT\_PRECIS

Определяет механизм выбора, принятый по умолчанию

OUT\_DEVICE\_PRECIS

Определяет, что при установке в системе нескольких шрифтов с одним и тем же названием выбирается системный шрифт

OUT\_OUTLINE\_PRECIS

Только для Windows NT: выбор осуществляется из набора векторных или шрифтов TrueType

OUT\_RASTER\_PRECIS

Определяет, что при установке в системе нескольких шрифтов с одним и тем же названием выбирается растровый шрифт

OUT\_STRING\_PRECIS

Используется для получения списка растровых шрифтов

OUT\_STROKE\_PRECIS

Используется для получения списка векторных и шрифтов TrueType

OUT\_TT\_ONLY\_PRECIS

Задает выбор только шрифтов TrueType, если ни один такой шрифт не установлен, используется механизм выбора, принятый по умолчанию

OUT\_TT\_PRECIS

Определяет, что при установке в системе нескольких шрифтов с одним и тем же названием выбирается шрифт TrueType

Поле IfClipPrecision определяет механизм отсечения символов шрифта в случае, если символы частично оказываются вне области вывода. Это поле может принимать одно или несколько из следующих значений:

CLIP\_DEFAULT\_PRECIS

Определяет использование механизма, принятого по умолчанию

CLIP\_CHARACTER\_PRECIS

He используется

CLIP\_STROKE\_PRECIS

Используется только для получения списка растровых, векторных или шрифтов TrueType

CLIP\_MASK

He используется

CLIP\_EMBEDDED

Указывается при использовании внедренных шрифтов

CLIP\_LH\_ANGLES

Указывает, что отсчет направления угла поворота (по часовой стрелке или против) зависит от выбранной системы координат; если флаг не используется то поворот текста, выводимого с использованием системных шрифтов, производится против часовой стрелки, однако для всех остальных шрифтов направление поворота определяется системой координат

CLIP\_TT\_ALWAYS

 He используется

Поле IfQuality определяет желаемое качество вывода, которое зависит от выбранного шрифта. Определены следующие три значения качества вывода:

DEFAULT\_QUAI\_ITY

Качество вывода текста несущественно

DRAFT\_QUALITY

 Качество вывода текста менее важно, чем для PROOF\_QUALITY: для растровых шрифтов разрешено масштабирование; при необходимости возможно создание синтетических начертаний шрифтов — полужирного, курсивного, подчеркнутого и перечеркнутого

PROOF\_QUALITY

Качество вывода текста более важно, чем соответствие заданным атрибутам; для растровых шрифтов масштабирование запрещено, выбирается наиболее подходящий по размеру шрифт: при необходимости возможно создание синтетических начертаний шрифтов — полужирного, курсивного, подчеркнутого и перечеркнутого

Поле IfP/tchAndFamily определяет тип и семейство шрифтов. Это поле позволяет выбрать наиболее похожий на заданный по имени шрифт в случае, если последний не установлен в системе. Для задания типа и семейства шрифта используется операция OR (!) между определениями типа и семейства, соответственно

Тип шрифта определяется двумя младшими битами поля и может принимать одно из следующих значений:

DEFAULT\_PITCH

Тип, принятый по умолчанию

FIXED\_PITCH

 Непропорциональный шрифт, т. е. все символы шрифта имеют одинаковый размер

VARIABLE\_PITCH

Пропорциональный шрифт

Семейство определяется старшими четырьмя битами поля и может принимать одно из следующих значений:

FF\_DECORATIVE

Декоративный шрифт, например, Old English

FF\_DONTCARE

Семейство шрифта не имеет значения

FF\_MODERN

Непропорциональный шрифт с засечками или без них, например, CourierNew\*

FF\_ROMAN

Пропорциональный шрифт с засечками, например. MS\* Serif

FF\_SCRIPT

Шрифт, схожий по начертанию с рукописным, например, Script или Cursive

FF\_SWISS

Пропорциональный шрифт без засечек, например, MS\* Sans Serif

Поле IfFaceName содержит имя шрифта. Длина имени не может быть более 32 символов, включая завершающий нуль-символ. Если имя не указано или шрифт с таким именем не найден, поиск подходящего шрифта будет проводиться с использованием других заданных параметров. Однако в ряде случаев имя не является определяющим (см. комментарии к полю IfCharSet).

Таким образом, для создания шрифта, с помощью которого мы будем выводить название, нам необходимо заполнить поля структуры LOGFONT.

LOGFONT if = {

130,                     // Высота шрифта в  логических единицах

0,                       //В качестве ширины символов

                         // выбирается значение, наиболее

                        // подходящее конкретному устройству

0,                         // Текст будем выводить

0,                         // без наклона

FW\_BOLD,                   // полужирный,

0,                         // не курсивный,

0,                         // не подчеркнутый

0,                         // и не перечеркнутый шрифт

DEFAULT\_CHARSET,           // Кодировка, установленная

                           // по умолчанию

OUT\_DEFAULT\_PRECIS,        // Механизм соответствия заданным

                                // параметрам ло умолчанию

CLIP\_DEFAULT\_PRECIS,            // Механизм отсечения,

                                // используемый по умолчанию

PROOF\_QUALITY,                  // Важно качество шрифта

VARIABLE\_PITCH | FF\_DONTCARE,  // Пропорциональный шрифт, а какое

                            // семейство — не имеет значения

"Arial" // Имя шрифта

};

После того как структура заполнена, для создания соответствующего объекта Windows осталось только вызвать функцию CreateFontIndirect, передав ей в качестве, параметра указатель на эту структуру.

m\_Font\_l.CreateFontInctirect (&lf) ;

Кроме этой функции, для создания шрифта можно воспользоваться функцией CreateFonr, которая принимает в качестве параметров те же значения, что и поля структуры LOGFONT.

BOOL CFont: :CreareFont. (

int nHeight,

int nWidth,

int nEscapement,

int nOrientation,

int nWeight,

BYTE bltalic,

BYTE bUnderline,

BYTE cStrikeOut,

BYTE nCharSet,

BYTE nOutPrecision,

BYTE nCiipPrecision,

BYTE nQuality,

BYTE nPitchAndFamily,

LPCTSTR IpszFacename)

Воспользуемся этой функцией для создания второго шрифта:

m\_Forit\_2 . CreateFont (

100,

О,                       //В качестве ширины символов

                        // выбирается значение, наиболее

                        // подходящее конкретному устройству

О,                           // Текст будем выводить

О,                           // без наклона

FW\_BOLD,                     // полужирный,

О,                           //не курсивный,

О,                           //не подчеркнутый

О,                           // и не перечеркнутый шрифт

DEFAULT\_CHARSET,             // Кодировка, установленная

                             //по умолчанию

OUT\_DEFAULT\_PRECIS,        // Механизм соответствия заданным

                            // параметрам по умолчанию

CLIP\_DEFAULT\_PRECIS,        // Механизм отсечения,

                            // используемый по умолчанию

PROOFJ2UALITY,              // Важно качество шрифта

VARIABLE\_PITCH I FF\_SWISS,  // Пропорциональный шрифт

                            // без засечек

"Arial"                     // Имя шрифта

);

Как видите, особой разницы в рассмотренных способах создания шрифта нет. Однако в ряде случаев все же предпочтительнее пользоваться функцией CreateFontlndirect и вот почему.

Обратите внимание, что при создании двух шрифтов практически все их параметры совпадают. Конечно, это не обязательное условие, но в большинстве случаев именно так и бывает. Поэтому, чтобы не повторять каждый раз описание всех параметров, можно просто заменить значения в соответствующих полях уже созданной структуры LOGFONT и вызвать функцию CreatFontlndirect. Как вы помните, нам надо создать еще один, последний, шрифт. Воспользуемся для этого предложенным методом:

// Изменяем высоту

If.IfHeight =80;

// Делаем шрифт самым "тонким"

1f.lfWei ght = FW\_THIN;

// Устанавливаем пропорциональный шрифт без засечек

If.IfPinchAndFamily = VARIABLE\_PITCH [ FF\_SWISS;

Осталось только вызвать соответствующую функцию, что мы и делаем:

m\_Font \_3 .CreateFontlndirect (&lf) ;

Большой разницы в рассмотренных двух способах нет — какой вам больше понравился, таким и пользуйтесь. Мне же осталось рассказать еще о двух функциях.

В ряде случаев при выборе шрифта нет необходимости столь подробно указывать его ожидаемые параметры. В то же время использование большинства текстовых редакторов приучает к тому, что размер шрифта характеризуется вполне точным, если хотите, абсолютным значением в пунктах (points). А пункт — это вовсе не экранная точка, т. е. пиксел, а единица длины, равная 1/72 дюйма. Класс СFont имеет в своем составе функцию CreatePointFont, которая, с одной стороны, создает шрифт на основе только его имени, а с другой — устанавливает желаемый размер как раз в привычных всем единицах:

BOOL, CFor.t: : CreatePointFont (

int PointSize,

 LPCTSTR IpszFaceName,

 CDC\* pDC = NULL

Первый параметр nPointSize определяет величину шрифта в десятых долях пункта, например, при выборе шрифта с размером символов в 8 пунктов необходимо установить значение этого параметра в 80,

Имя шрифта задается параметром IpszFaceName, при задании нуля выбирается системно-независимый шрифт.

Последний параметр pDC нужен для правильного преобразования размера шрифта в логические единицы, которые определяются именно для контекста устройства. Нулевое значение параметра приводит к выбору в качестве контекста устройства контекст экрана и, соответственно, его логических единиц.

Аналог CreateFontIndirect — функция CreatePointFontIndirect — в качестве параметра также принимает указатель на структуру LOGFONT. однако единицами измерения в поле lfHeight этой структуры являются пункты.

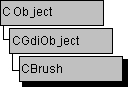
Мы не воспользовались этими функциями в частности еще и потому, что у нас пока не определен контекст устройства.

Итак, мы подготовили шрифты и переходим к следующему необходимому нам объекту — кисти.

Создание кистей

Процесс создания кисти значительно проще, чем создание шрифта. В библиотеке MFC реализован класс CBrush (рис. 8.3). который инкапсулирует графический объект Windows "кисть" для использования в качестве основы при закрашивании замкнутых фигур. Кисти могут быть однотонные, со штриховкой или созданные на основе шаблона.

Как и все другие классы графических объектов, класс CBrush имеет в своем составе конструктор без параметров — CBrush::CBrush(), который создает объект класса, но для его использования требует вызова одной из "создающих" функций — CreateSolidBrush, Create Hatch Brush, CreateBrush Indirect, CreatePatternBrush, CreateDIBPatternBrush, CreateSysColorBrush. Первые четыре функции полностью дублируют возможности конструкторов. Пятая функция создает кисть на основе битового массива, но, в отличие от четвертой, этот массив хранит изображение в независимом от устройств виде. Последняя функция создает кисть со штриховкой на основе системного цвета. Рассмотрим некоторые из этих функций более подробно.



**Рис. 8.3.** Место класса CBrush в иерархии библиотеки MFC

BOOL CBrush::**CreateSolidBrush**(COLORREF crColor)

Используется для создания однотонной кисти, цвет которой задается параметром crColor. Тип COLORREF определяет 32-разрядное значение, задающее цвет в системе RGB (красный, зеленый, синий) и имеющее формат OxOObbggrr.

Для задания параметра crColor можно воспользоваться определенным в Win32 API макросом RGB, который комбинирует интенсивности (в диапазоне от 0 до 255) трех цветов — красного, зеленого и синего и определяется следующим образом:

COLORREF **RGB**(int red, int green, int blue)

Итак, для создания однотонной кисти, которой мы будем закрашивать фон наших часов, нам необходимо задать ее цвет:

fdefine COLOR\_BCKGR     RGB(224, 224, 224)

и вызвать функцию CreateSolidBrush:

m\_wndBrush.CreateSolidBrush(COLOR\_BCKGR);

Аналогичную операцию проделаем и для других кистей, за исключением одной:

#define COLOR\_CLOCK RGB(64, 128, 128)

#define COLOR\_ROUND RGB (64., 128, 128)

#define COLOR\_CRNER RGB(64, 128, 128)

....

m\_clkBrush.CreateSolidBrush(COLOR\_CLOCK); m\_rndBrush.CreateSolidBrush(COLOR\_ROUND); m\_crnBrush.CreateSolidBrush(COLOR\_CRMER);

Для фона стрелок наших часов используем кисть со штриховкой. Создать ее так же просто, как и однотонную кисть:

BOOL CBrush:: **CreateHatchBrush** (

 int nIndex,

COLORREF crColor)

Здесь параметр nlndex определяет один из возможных вариантов штриховки:

HS\_BDIAGONAL

Штриховка слева направо под углом 45° и сверху вниз

HS\_CROSS

Горизонтальная и вертикальная штриховка крест накрест

HS\_DIAGCROSS

Штриховка крест накрест под углом 45°

 HS\_FDIAGONAL

 Штриховка слева направо под углом 45° и снизу вверх

HS\_HORIZONTAL

Горизонтальная штриховка

 HS\_VERTICAL

Вертикальная штриховка

a crColor— значение цвета линий штриховки.

Вот как это будет в нашем случае:

m\_ arrBrush.CreateHatchBrush (

HS\_DIAGCROSS,    // Штриховка крест накрест под углом 45°

CCLOR\_ARROW)     //цвет линий штриховки

Кисть для фона реализуем с помощью специатьного шаблона. В этом случае необходимо сначала создать или загрузить битовый массив, а затем вызвать функцию

CBrush:: **CreatePatternBrush**   (CBitmар\* pBimap)

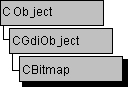
В качестве параметра pBitmap функции передается указатель на объект класса CBitmap. представляющего в библиотеке объект Windows "битовый массив".

При использовании этой функции надо иметь в виду, что размеры битового массива должны быть не меньше чем 8x8 пикселов, а сама кисть создается на основе левой верхней квадратной области битового массива размером также 8x8 пикселов.

Я думаю, вам теперь не составит труда самостоятельно разобраться с остальными имеющимися в классе CBrush функциями, а нам надо прерваться, чтобы познакомиться с классом CBitmap.

Создание битовых массивов

Для работы с битовыми массивами в библиотеке MFC реализован класс CBitmap (рис. 8.4), который инкапсулирует графический объект Windows "битовый массив".



**Рис. 8.4**. Место класса CBitmap в иерархии библиотеки MFC

Как обычно, для работы с объектом класса необходимо создать сам объект, а затем создать собственно битовый массив Windows путем вызова одной из функций CreateBitmap, Create Bitmap Indirect, CreateCompatibleBitmap или Create Discardable Bitmap, загрузить его из соответствующего ресурса (функции LoadBitmap, LoadOEMBitmap, Load Mapped Bitmap) или установить связь с ранее созданным (загруженным) битовым массивом (функция Attach). Кроме этого, класс предоставляет набор функций, позволяющих менять и узнавать содержимое (SetBitmapBits и GetBitmapBits) и размеры (SetBitmapDimension и GetBitmapDimension) массива.

Для нашего примера мы воспользуемся функцией загрузки битового массива, созданного в качестве ресурса:

BOOL CBitmap::**LoadBitanap** (LPCTSTR' дpszResourceName)

или

BOOL CBitmap:: **LoadBitmap** (UINT nIDResource)

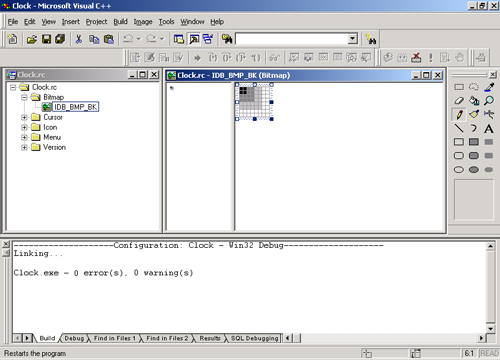
Функции позволяют загрузить битовые массивы, заданные текстовой строкой (параметр IpszResourceName), т. е. по имени, или числовым идентификатором (параметр nIDResource). Загрузка битовых массивов может осуществляться прямо из ресурсов исполняемого модуля текущего приложения.

Очевидно, что сначала требуется создать соответствующий ресурс. Вот как он будет выглядеть у нас (рис. 8.5).

Затем ресурс надо загрузить в объект класса CBitmap:

CBitmap bmp;

bmp.LoadBitmap(IDB\_BMP\_BK);



**Рис. 8.5.** На основе этого шаблона мы будем создавать "цвет фона" кисти часов

Конечно же. в классе СBitmap есть и другие функции, но рассмотренный подход — один из наиболее простых, и поэтому мы пока им и ограничимся — пора возвращаться к нашему примеру.

После того как объект класса СBitmap создан, нам осталось передать указатель на него в функцию CreatePatternBrush:

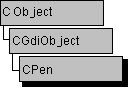
m\_clkBrush.CreateraLrernBrush. ( &bmp) ;

Результатом наших действии будет кисть на основе этого шаблона.

Теперь, когда мы научились создавать не только шрифты и кисти, но битовые массивы, перейдем к последнему (используемому нами, но не последнему из имеющихся) объекту, а именно, к карандашам.

Создание карандашей

Класс СРеn инкапсулирует графический объект Windows "карандаш" (рис. 8.6). Карандаши используются для рисования различных линий и контуров объектов. Эти объекты имеют наиболее богатые возможности для их создания — в Windows имеется целый набор различных предопределенных стилей, свойств и типов, которые позволяют использовать карандаши просто и эффективно и, соответственно, проводить с их помощью самые разнообразные линии.



**Рис. 8.6.**Место класса СРеn в иерархии библиотеки MFC

Как и все другие классы графических объектов, класс СРеn допускает как создание объектов класса, не связывая их при этом с объектами Windows, так и одновременное создание объекта класса и соответствующего графического объекта.

Что касается последней возможности, то она дублируется наличием функций создания карандаша Windows уже после создания объекта класса. Поскольку мы уже создали объекты этого класса, то не будем рассматривать имеющиеся в классе конструкторы, а подробно остановимся только на специальных функциях создания соответствующих объектов Windows.

СРеп::**CreatePen**  (

int nPenStyle,

int nWidth,

COLORREF crCo.lor)

и

СPen::**CreatePen** (

int  nPenStyle,

int nWidth,

const LOGBRUSH\* pLogBrush,

int nStyleCount = 0,

const DWORD\* IpStyle-= NULL)

Давайте рассмотрим отдельно для каждой версии функции CreatePen возможные значения параметра nPenStyle.

В результате использования первой версии функции создаются карандаши с предопределенными в Windows стилями для рисования следующих типов линий:

PS\_SOLID

Сплошная линия

PS\_DASH

Длинные штрихи, толщина которых равна 1

PS\_DOT

 Короткие штрихи, толщина которых равна 1

PS\_DASHDOT

 Штрихпунктир, толщина которого равна 1

PS\_DASHDOTDOT

Двойной штрихпунктир, толщина которого равна 1

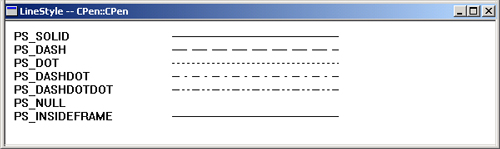
PS\_NULL

Прозрачная линия

PS\_INSIDEFRAME

Сплошная линия для рисования по внутренней границе замкнутой области, например, при использовании функций класса CDC — Rectangle, RoundRect, Ellipse, Chord и т. д.; при использовании карандаша данного типа для рисования линий, например, при помощи функции LineTo, результат будет тем же. что и для стиля PS\_SOLID. Еще одна область применения этого стиля — создание карандашей толщиной более одного пиксела и имеющих цвет, недоступный для текущего устройства. В этом случае линии будут состоять из пикселов доступных цветов, совокупность которых наиболее близко соответствует заданному цвету

Для наглядности продемонстрируем линии различных стилей. Обратите внимание, что при выводе линий одной длины некоторые штриховые линии фактически оканчиваются раньше сплошных (рис. 8.7), что вполне естественно — эти линии завершились в месте пустого пространства между штрихами. Такой эффект может негативно сказаться при решении конкретных задач.



**Рис. 8.7.** Различные стили линий

Использование второй версии функции CreatePen позволяет комбинировать в параметре nPenStyle стиль, тип, режим рисования концов линии и режим соединения линий. Каждый атрибут рисования может быть представлен только одним значением, заданные значения комбинируются при помощи оператора OR ([).

Карандаши Windows бывают двух типов — PS\_GEOMETRIC и PS\_COSMETIC, отличающихся использованием кистей для рисования линий и требованиями к заданию ширины. Кроме того, от выбранного типа зависит интерпретация параметров, задающих размеры штрихов для пользовательских стилей.

Вторая версия функции CreatePen поддерживает два дополнительных стиля — PS\_ALTERNATE (может использоваться только совместно с типом PS\_COSMETIC) и PS\_USERSTYLE, который для рисования линий требует задания массива длин штрихов.

Для карандашей также возможно задание режима рисования концов (естественно, это имеет смысл при рисовании широких линий). Таких режимов три, для первых двух фактическая длина линии может быть больше заданной:

PS\_ENDCAP\_ROUND

Концы линий закруглены

PS\_ENDCAP\_SQUARE

 Концы линий имеют прямоугольную форму

PS\_ENDCAP\_FLAT

 To же, что и для PS\_ENCAP\_SQUARE, но концы линии не выходят за заданные размеры

Для соединения линий можно выбирать один из следующих режимов:

PS\_JOIN\_BEVEL

 Соединение обрезано

PS\_JOIN\_MITER

Соединение определяется продолжением контура линий, если только расстояние между концами линий не превышает заданную функцией CDC::SetMHerLimit величину, в противном'случае соединение обрезается, как и в случае задания режима PS\_JOIN\_BEVEL

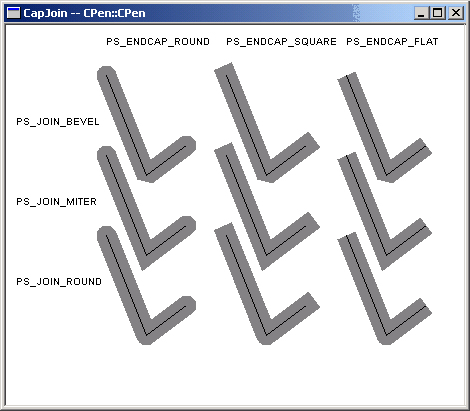
PS\_JOIN\_ROUND

Соединение закруглено

Примеры различных типов, стилей рисования и соединения линий представлены на рис. 8.8. Одно из требований при применении режима рисования концов линий и соединений заключается в том, что рисование линий при помощи карандашей, созданных с флагами PS\_ENDCAP\_SQUARE, PS\_ENDCAP\_FLAT. PS\_JOIN\_.BEVEL и PS JOIN\_MITER, можно осуществлять только при помощи механизма использования контуров. Обратите внимание, что для наглядности по оси каждой линии, имеющей тип PS\_GEOMETRIC, проведены сплошные тонкие линии с теми же координатами.

При создании карандаша необходимо также задать его ширину с помощью параметра nWidth. В первой версии функции CreatePen заданием ширины, равной 0, гарантируется вывод линии шириной в один пиксел независимо от режима вывода (mapping mode). При использовании второй версии функции ширина задается в логических единицах — для карандашей типа PS\_GEOMETRIC, а для карандашей типа PS\_COSMETIC ширина должна быть равна 1.

Для задания цвета карандаша используется параметр crColor.



**Рис. 8.8**. Различные стили рисования и соединения линий

Параметр pLogBrush является указателем на структуру LOGBRUSH для создания кисти, с помощью которой будет производиться рисование линий. Для карандашей типа PS\_COSMETIC элемент структуры LOGBRUSH::lbStyle должен быть равен BS\_SOLID.

Параметр IpStyle указывает на массив, содержащий nStyleCount чисел, определяющих попарно длину штриха и расстояние между штрихами для карандашей, имеющих стиль PS\_USERSTYLE.

Кроме рассмотренных, для создания карандашей могут быть использованы функции CreatePenlndirectили CreateStockObject. При помощи первой из них возможно создание только обычных карандашей, т. е. таких же, какие можно создавать при помощи первой версии функции CreatePen. Как и в случае создания кистей, никаких сложностей при создании карандашей нет. Вот как мы осуществляем это в нашем примере:

#define COLOR\_BCKGR RGB(224, 224, 224)

#define COLOR\_BLACK RGB( 0, 0, 0) •

#define COLOR\_ARROW RGB( 0, 128, 128)

#define COLOR\_CRNER RGB( 64, 128, 128)

...

m\_wndPen.CreatePen(PS^SOLID, 1, COLOR\_BCKGR); m\_arrPen.CreatePen(PSJSOLID, 1, COLOR\_ARROW) ;

:n.\_clkPen.CreatePen{?S\_SOLID, I, COLOR\_BLACK) ; m\_crnPen.CreatePen;PS\_SOLID, I, COLOR\_CRNER);

Единственное исключение мы сделаем для карандаша, которым будем рисовать "метки" для обозначения цифр. В этом случае используем вторую версию функции CreatePen. Для нее, как вы помните, нам необходим указатель на структуру LOGBRUSH. которая содержит параметры кисти, применяемой при рисовании "толстых" линии. Посмотрите, как можно получить параметры кисти, используемой в качестве основы карандаша:

LOGBRUSH lb;

m\_rndBrush.GetLogBrush;&1b);

После того как поля структуры заполнены, нам осталось только вызвать функцию создания карандаша, передав ей в качестве параметра указатель на эту структуру:

m\_rndPen.CreatePen(

PS\_SOLID I PS\_3EOKETRIC [  // "Сплошные",

PS\_ENDCAP\_SQUARE,          // с прямоугольными концами линии

20,                        // Ширина 20 пикселов

&1b);                      // Параметры кисти

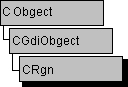
**Примечание**

Задание прямоугольных концов для линий, для которых создаем карандаш, в нашем примере не влияет на рисование, т. к. это свойство проявляется, когда рисование осуществляется при помощи создания контуров, а данный карандаш будет использоваться для рисования дуг, которые функцией CDC::StrokePath не отображаются.

Здесь мне хотелось бы рассмотреть еще один графический объект, который мы пока не создавали, но который понадобится нам при рассмотрении вопросов рисования. Речь идет о регионах.

Работа с регионами

Класс CAgn инкапсулирует объект Windows "регион", представляющий собой совокупность прямоугольных и эллиптических областей (рис. 8.9).



**Рис. 8.9.** Место класса СRgn в иерархии классов библиотеки MFC

Основное назначение регионов — определение области отсечения в пределах окна. Класс CRgn имеет в своем составе набор функций для создания регионов, схожих по своим параметрам с функциями рисования прямоугольников, эллипсов и многоугольников, а также функции, позволяющие комбинировать регионы.

Создание регионов

Для создания региона при помощи класса CRgn необходимо использовать конструктор без параметров, который создает объект класса CRgn. но не создает при этом собственно объект Windows. Для создания последнего можно воспользоваться функциями, описанными ниже.

Создание прямоугольных регионов

BOOL CRgn::**CreateRectRgn** (

int xl, i

nt yl,

int x2,

int y2)

и

BOOL CRgn::**CreateRectRgnlndirect** (LPCRECT IpRect)

Первая версия функции получает в качестве параметров координаты левого верхнего и правого нижнего углов прямоугольной области. Вторая версия функции получает указатель на структуру RECT, в которой указываются также координаты прямоугольной области.

Создание эллиптических регионов

BOOL CRgn::**CreateEllipticRgn** (

.int xl,

 int yl,

int x2,

int y2)

и

BOOL CRgn::**CreateEllipticRgnlndirect** (LPCRECT IpRect)

Обе версии функции получают информацию, аналогично версиям предыдущей функции, о прямоугольной области, в которую вписан эллипс.

Создание сложных регионов на базе многоугольников

BOOL CRgn::**CreatePolygonRgn** (

LPPOINT IpPoints,

int nCount,

int nPolyFillMode)

Функция принимает в качестве информации о многоугольнике массив координат (IpPoints) и количество его вершин (nCount).

BOOL СRgn::**CreatePolyPolygonRgn** (

LPPOINT ipPoints,

LPTNT lpPolyCounts,

int nCount,

int nPolyFiilMode)

Функция принимает в качестве информации массив координат вершин многоугольников (IpPoints] и массив, содержащий количество вершин для каждого из них (IpPolyCounts). Параметр nCount задает количество многоугольников.

Параметр nPolyFilIMode в этих функциях определяет режим закраски ALTERNATE или WINDING (см. следующую главу).

Создание регионов с закругленными углами

B00L CRgn::**CreateRoundRectRgn** (

int xl,

int yl,

int x2,

int у 2,

int x3,

int y3)

Первые четыре параметра определяют прямоугольную область, а параметры хЗ (л уЗ определяют ширину и высоту эллипса, на основе которого строятся закругленные углы.

Комбинирование регионов

Более подробно остановимся на функции, которая на основе комбинации двух регионов изменяет третий. Именно изменяет, т. к. объект класса, для которого вызывается эта функция и который, собственно, и будет хранить результат комбинации двух других регионов, обязательно должен существовать — так работает Windows, так работает и библиотека классов MFC.

**Примечание**

Объект класса CRgn, для которого вызывается функция, может совпадать с одним из тех, указатели на которые передаются в качестве параметров.

int CRgn : : CombineRgn (

CRgn\* pRgnl,

CRgn\* pRgn2,

int  nCombineMode,

Функция возвращает одно из следующих значений, показывающих, какого типа регион получился в результате комбинирования:

COMPLEXREGION

Регион состоит из набора более простых областей

 ERROR

Произошла ошибка

NULLREGION

Регион пуст

SIMPLEREGION

 Регион представляет собой одну простую область — прямоугольник или эллипс

Параметры pRgnl и рRgn2 являются указателями на объекты класса СПдп, которые должны быть связаны с регионами Windows, именно они и будут комбинироваться. Параметр nCombineMode определяет режим комбинирования. Таких режимов несколько:

RGN\_AND

 Регион будет являться пересечением обоих заданных, т. е. определяться областью, которая относится и к первому, и ко второму региону одновременно

RGN\_COPY

Регион будет точной копией первого региона (pRgnl)

RGN\_DIFF

Регион будет включать в себя только те области первого региона (pRgnl), которые не пересекаются со вторым (pRgn2)

RGN\_OR

 Регион будет являться объединением обоих заданных, т. е. будет включать все области, относящиеся и к первому, и ко второму региону

RGN\_XOR

Регион будет включать области первого региона (pRgnl) и второго региона (pRgn2), кроме тех, которые принадлежат и первому и второму одновременно

Несмотря на то, что Windows для осуществления комбинирования требует наличия объектов для всех трех регионов, возможно использование в качестве одного или даже обоих параметров указателя на объект класса CRgn, для которого и вызывается функция CombineRgn:

rgn.CombineRgn(&rgn, &rgnPrev);

Кроме вышеперечисленных функций, упомянем еще несколько. Класс CRgn имеет в своем составе функцию копирования CopyRgn, функцию преобразования текущего контура (path) в регион (CreateFromPath), функцию сравнения двух регионов (EqualRgn).

Для иллюстрации использования регионов приведем процедуру, в которой в результате комбинирования трех регионов производится закрашивание части замкнутых областей одним вызовом функции рисования (рис. 8.10).

void CRegionFrame::OnPaint()

{

CPaintDC dc(this);

CRect rect;

GetClientRect(rect);

// Определяем треугольную область с одной вершиной

//в центре клиентской части окна и двумя другими,

// совпадающими с левым верхним и левым нижним углами

POINT pt\_l[4] = { 0, 0,

rect.right / 2, rect.bottom / 2,

0, rect.bottom,

0, 0 } ;

CRgn rgn\_l;

rgn\_l.CreatePolygonRgn;pt\_\_l, 4, WINDING);

// Определяем треугольную область с одной вершиной

// в центре клиентской части окна и двумя другими,

 // совпадающими с правым верхним и правым нижним углами

POINT pt\_2[4] = < rect.right, 0,

rect.right / 2, rect.bottom / 2,

rect.right, rect.bottom,

rect.right, 0 };

CRgn rgn\_2;

rgn\_2.CreatePoiygonRgr, (pt\_2, 4, WINDING);

// Определяем прямоугольную область, лежащую точно

//в центре клиентской области окна и немного меньше ее

CRgn rgn;

rgn.CreateRectRgn/40, 40, rect.right-40, rect.cottom-40);

// Определяем регион как комбинацию трех заданных

rgn. CombineRgn (& rgn, &rgn\_l, RGN\_XOR) ;

 rgn.CombineRgn-irgn,( &rgn\_2, RGN\_XOR);

// Определяем параметры рисования

// кисть

CBrush brush(HS\_DIAGCROSS, RGB(0, 0, 0));

CBrush "pBrush = dc.SelectObnect(Sbrush);

/ цвет фона

dc.SetBkColor(RGB(123, 128, 128));

// область отсечения dc.SelectObject(&rgn);

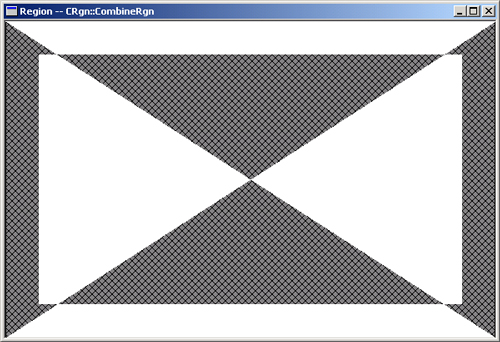
// Рисуем прямоугольник на всю клиентскую часть окна

dc.Rectangle(rect);

 // Восстанавливаем кисть

dc.SelectObject(pBrush);

}



**Рис. 8.10.** Результат работы функции CRegionFrame::OnPaint

К сожалению, даже на таком простом примере при выполнении программы видно, как медленно происходит рисование: такое впечатление, что для каждой точки проверяется возможность ее закрашивания.

На этом подготовительные работы по созданию необходимых нам графических объектов закончены, и мы переходим к следующему вопросу.

Изменение размеров окна

Вообще-то за изменение размеров окна в приложениях отвечает класс фрейма (у нас это CMainFrame). И более того, для поддержки этой возможности нам не пришлось писать ни строчки кода — все сделала библиотека MFC. Нас же вопрос изменения размеров интересует несколько с другой точки зрения: мы хотим, чтобы размер наших часов всегда совпадал с размерами окна.

Реализовать это чрезвычайно просто. Прежде всего необходимо определить две переменные, которые будут хранить текущие значения ширины и высоты окна:

class CChildWnd : public CWnd

{

...

private:

int m\_cxClient; // ширина клиентской области окна

 int m\_cyClient; // высота клиентской области окна

};

А для отслеживания текущих значений ширины и высоты необходимо определить обработчик сообщения WM\_SIZE. Вот как он будет выглядеть:

void CChildWnd: :OnSize (UINT nType, int ex, int. cy)

{

// He вмешиваемся в работу библиотеки

CWnd::OnSize(nType, ex, су);

// Запоминаем размер клиентской области

// окна, чтобы каждый раз при перерисовке

// не запрашивать размер заново

m\_cxClient = сх;

m\_cyClient = су;

}

Вот собственно и все. Теперь, как только нам понадобятся параметры размеров часов, мы всегда можем получить их текущие значения.

Обработка сообщения WM\_PAINT

Многие из вас, конечно же, ожидали, что в конце концов мы придем к созданию обработчика сообщения WM\_PAINT. И вы не ошиблись, хотя в нашем конкретном случае вполне можно было бы обойтись и без него. Дело в том, что рисование часов обладает особенностью — их надо перерисовывать не в зависимости от "поврежденности" окна, а тогда, когда истечет выбранный нами отрезок времени (через секунду, если будем рисовать секундную стрелку, или через минуту, если секундную стрелку рисовать не будем).

Однако для демонстрации "работы" этого наиболее часто используемого обработчика я решил определить в нем перерисовку фона. Вот как выглядит обработчик OnPaint в нашем примере:

void CChildWnd: : OnPaint ()

{

CPaintDC dc(this); // device context for painting

// контекст устройства для перерисовки

// Закрашиваем клиентскую область,

// для чего создаем однородную кисть и ...

CBrush brush(COLOR\_3CKGR);

// Рисуем прямоугольник

dc.FillRect (CRect (0, 0, m\_c;<Client, m\_cyClie-nf , &brush) ;

// Циферблат, текст и стрелки рисовать не будем,

 // этим займется функция CClockApp::OnIdle

 // Do not call CWnd:: OnPaint () for painting messages

 // Это едва ли не единственное место,

 // где не надо вызывать переопределенную функцию базового класса

}

**Примечание**

Английский текст в обработчике OnPaint вставляется мастером ClassWizard.

Вы наверняка обратили внимание на вызов функции FillRect, которая занимается закрашиванием прямоугольника размером с клиентскую область окна. И уж точно догадались, что она принадлежит одному из классов, отвечающих за контекст устройства. И это действительно так. Поэтому здесь мне бы хотелось прервать самого себя и рассмотреть более подробно класс CDC, поскольку ни одно действие по выводу чего-либо в окно (рисованию в окне) не обходится без участия этого класса (прямо или опосредованно).

Однако задержимся еще ненадолго, чтобы закончить со всеми остальными, отличными от рисования, вопросами.

Организация процесса рисования

Ясно, что нам нужна некоторая функция, которая будет заниматься собственно рисованием в окне. Назовем ее, например, Redraw (нам же периодически придется заниматься именно перерисовкой). Очевидно, что она должна принадлежать классу CChildWnd, отвечающему за окно, в котором мы будем рисовать.

Теперь необходимо определить, где и когда следует вызывать эту функцию. Первое, что приходит на ум: создать таймер и с заданной периодичностью вызывать Redraw. Безусловно, это один из вариантов. Однако я предлагаю рассмотреть другой способ, а работу с таймером мы разберем при обсуждении строки состояния.

Вернемся немного назад — к классу CWinApp, который мы рассматривали в главе 3. Как вы помните, на базе этого класса создается объект приложения. Некоторые основные его функции мы рассмотрели, нас же сейчас интересует еще одна, а именно:

virtual BOOL CWinApp::OnIdle (LONG ICount)

Параметр ICount увеличивается перед каждым вызовом функции, т. е. всякий раз, когда очередь сообщений приложения пуста, и сбрасывается в 0 при обработке нового сообщения. Этот параметр можно использовать для определения различных режимов обработки или относительного времени "простоя" приложения.

Функция CWinApp::OnIdle вызывается в цикле обработки сообщений, когда очередь сообщений приложения пуста. Функция, выполняемая по умолчанию, обновляет команды объектов интерфейса пользователя, такие как элементы меню и кнопки панелей инструментов, и очищает внутренние структуры данных. Если вы переопределяете эту функцию, то в новой версии необходимо сначала вызвать базовую функцию CWinApp::Onldle с параметром ICount.

Именно этой функцией мы и воспользуемся (естественно, сначала ее переопределив). Рассмотрим необходимые действия:

1. Откройте Class Wizard и в раскрывающемся списке Class name выберите класс нашего приложения — CClockApp. В окне Messages найдите и выделите виртуальную функцию Onldle (рис. 8.11).

2. Нажмите последовательно кнопки Add Function и Edit Code и в созданную заготовку внесите следующий код:

BOOL CClockApp::Onldle(LONG ICount)

 {

CWinApp::Onldle(ICount);

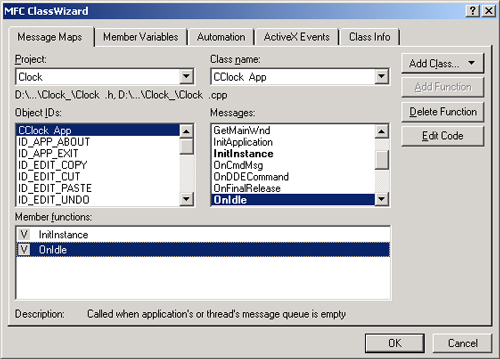
// Перерисовываем окно

((CMainFrame \*)m\_pMainWnd)->Redraw();

// чтобы не закончить цикл вызова, функции Cr.Idle возвращаем TRUE

return TRUE;

 }



**Рис. 8.11.** Переопределяем виртуальную функцию Onldle

3. В классе CMainFrame определяем функцию Redraw:

class CMainFrame : public CFraraeWrid

{

...

// Operations

 public:

void Redraw();

...

};

// Функция перерисовки часов может вызываться

// практически в любой функции программы

//У нас это будет происходить во время "простоя" приложения

void CMainFrame: :Redraw ()

m\_wndChild.Redraw();

}

4. В классе CChildWnd определяем свою функцию Redraw:

class CChildWnd : public CWnd

 {

...

// Operations

 public:

void Redraw();

...

};

void CChildWnd::Redraw()

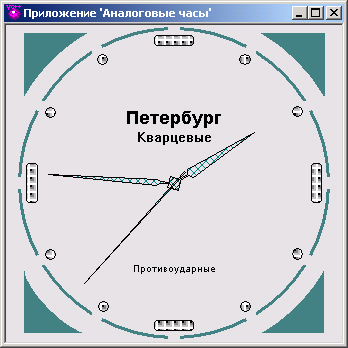
{

// Именно здесь будем реализовывать рисование часов

}

Вот как будет выглядеть окно приложения после того, как мы реализуем функции отображения часов (рис. 8.12).

И теперь мы спокойно можем переходить к знакомству с одним из основополагающих классов библиотеки MFC, а именно, классом CDC.



**Рис. 8.12**. Окно приложения "Аналоговые часы" при начальных размерах окна