МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ
(ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ)
Кафедра информатики и вычислительной математики

Е.П.Дербакова

Простая графическая задача на Microsoft Visual C++ с использованием библиотеки MFC

Учебно-методическое пособие

Москва
МФТИ
2016

УДК 517.8

Рецензент
Член-корр. РАН, доктор физико-математических наук *И.Б.Петров*

Простая графическая задача на Microsoft Visual C++ с использованием библиотеки MFC: Учебно-методическое пособие по курсу Информатика и применение компьютеров в научных исследованиях/ Сост.: Е.П. Дербакова, –М.:МФТИ, М., 2016. 37 с.

В пособии представлены основные понятия объектно-ориентированного программирования, основы языка С++, а также основная информация о классах и компонентах библиотеки *MFC* в среде программирования *Visual C++*. Приведен пример простой графической задачи в среде Visual C++ 2013.

Предназначено для студентов 2-го курса.

ОГЛАВЛЕНИЕ

1. Основные понятия 4

2. Библиотека *MFC* 7

3. Создание каркаса приложения 11
3.1. Создание проекта с использованием *мастера классов.* 11
3.2. Использование окна проекта 13
3.3. Класс *CwinApp* 15
3.4. Запуск приложения 16

4. Простая графическая задача 16
4.1. Класс *CApp1View* 16
4.2. Контекст устройства... 17
4.3. Средства работы с графикой 18
4.4. Первый рисунок 20
4.5. Добавление обработчика сообщений 20
4.6. Таймер 22
4.7. Обработчик сообщения *OnTimer* 24
4.8. Меню 25

ПРИЛОЖЕНИЕ 1 26
ПРИЛОЖЕНИЕ 2 31

СПИСОК ЛИТЕРАТУРЫ 35

1. Основные понятия

В названии среды программирования *Visual C++* указаны две основных черты современного программирования:

1. *Visual* — это визуальная среда программирования, т.е. с помощью мыши на экране монитора можно визуально менять содержимое окон программы и это будет автоматически зафиксировано в тексте программы.
2. C++ — означает, что в среде программирования используется не стандартный язык *C*, а созданный на его основе язык объектно-ориентированного программирования *C*++.

В программе, написанной на языке Си, данные и функции, предназначенные для их обработки, определяются отдельно. Такое разделение затрудняет структурированное программирование и создает дополнительные возможности для ошибок, которые трудно обнаружить.

В язык *C*++ введен новый тип данных — класс, который позволяет объединить в одной структуре данные и оперирующие ими функции. Такое объединение обычно называют инкапсуляцией данных и связанных с ними функций. Класс является расширением типа *struct* языка Си. Функции, оперирующие над данными класса, называются методами класса. И данные класса и методы класса называются членами класса.

Объявление класса может иметь следующее формальное описание:

Class type_classname: baseclassname{

*type*1 *var*1;

*type*2 *var*2;

…………….

typen varn;

Public:

*metod*1();

………..

protected:

```
        metod1();
        …………
        private:
        ………..
}
```

Здесь *type*1 *var*1…..*typen varn* — данные класса, *metod*1().— функции, оперирующие этими данными.

baseclassname — базовый класс для объявляемого класса. Базовым классом называется класс, от которого может порождаться другой производный (наследуемый) класс. При этом производный класс наследует структуру данных и поведение своего базового класса. Производный класс может переопределять и расширять поведение базового класса, а также добавлять свои данные — члены класса.

Наследование позволяет определять новые классы в терминах существующих классов.

Так же, как и для структур, после того как новый тип данных объявлен, можно определять объекты данного типа (класса), а также объекты, являющиеся производными от него. Например:

```
class Csple{
        int date;
        int func();
…………
};
…………
Csple cn1,              //переменная типа Csple
*pcl2,                  //указатель на переменную этого типа
arcln[5];               //массив элементов типа Csple
```

Обращение к элементам или методам класса как к элементам структур:

```
        cn1.date=10;
        pcl2—>func();
```

Термины *public*, *protected* и *private* называются модификаторами доступа к переменным и методам класса. К элементам и методам, следующим за модификатором *public,*

можно обращаться как из методов данного класса, так и из программы. Доступ к остальным членам класса открыт только для методов самого класса, а также для дружественных методов других классов. В случае модификатора *protected* — также для производных классов.

Для доступа к объекту класса можно использовать ключевое слово *this*, являющееся указателем на объект данного класса. Это ключевое слово нельзя использовать вне метода члена класса.

Для непосредственного указания метода или переменной класса в программе перед именем метода или переменной проставляется имя класса и знак ::

```
CWnd::GetParent();
```

Дружественные методы позволяют получить доступ к защищенным модификатором *private* членам класса из методов других классов. Если метод класса *A* внутри тела класса *B* объявлен с модификатором *friend*, то из него разрешен доступ ко всем членам класса *B*.

```
class B{
    friend A :: metod1();
}
```

Если объявить дружественным целый класс, то все его методы получают доступ ко всем переменным и методам другого класса.

```
class B{
    friend class A; }
```

Конструктором называется метод, одноименный с именем класса и вызываемый при создании объекта данного класса.

Деструктором называется метод, вызываемый при разрушении объекта данного класса. Имя этого метода начинается с символа ~ и, далее, совпадает с именем класса объекта. Один класс может иметь несколько конструкторов, отличающихся списком параметров.

Методы класса могут быть статическими и виртуальными. Статические методы объявляются с модификатором доступа *Static*, являются общими для всех объектов класса.

Статическими могут быть и переменные. Статические методы могут вызывать и использовать только другие статические методы и переменные.
Виртуальные методы объявляются в базовом классе с ключевым словом *virtual*, а в производном классе могут быть переопределены.

2. Библиотека *MFC*

Microsoft Foundation Classes (*MFC*) — это библиотека классов, которую можно использовать для программирования на языке *C*++ под *Windows*.
Библиотека позволяет программировать в терминах классов различные элементы *Windows*, такие, как окна, диалоги, элементы управления, а также графический интерфейс. Функции библиотеки используют функции *API* (программного интерфейса *Windows*), но есть возможность в *Visual C*++ использовать функции *API* и напрямую.
Библиотека *MFC* создана как иерархический набор классов. Базовым классом для большинства классов библиотеки является класс *Cobject,* т.е. класс объекта. В этом классе заключены основные общие представления об объекте: поддержка сохранения и восстановления данных (сериализация) объекта, вывод диагностики об объекте, информация о классе времени выполнения. Все эти свойства наследуются большинством классов библиотеки.
Непосредственно от класса *CObject* наследуются ряд классов, которые сами являются базовыми для остальных классов *MFC*. В первую очередь это класс *CCmdTarget*, представляющий основу структуры любого приложения. Объекты класса *CCmdTarget* обладают свойством получать от операционной системы сообщения и обрабатывать их. Структура классов, связанных с классом *CCmdTarget*, изображена на рис. 2.1.
Класс *CWinThread* представляет подзадачи приложения, а класс *CWinApp* является базовым классом для объекта

простого приложения. Этот класс предоставляет средства инициализации и управления приложением.

Класс *CDocument* служит для представления документов приложения, хранимых на диске в отдельных файлах.

Практически все приложения имеют пользовательский интерфейс, построенный на основе окон. Это может быть диалоговая панель, одно окно или несколько окон, связанных вместе. Основные свойства окон представлены классом *CWnd*, наследованным от класса *CCmdTarget*.

Класс *CDocTemplate* и наследуемые от него классы предназначены для синхронизации и управления основными объектами, представляющими приложение, — окнами, документами и используемыми ими ресурсами.

На рис. 2.2 представлена только небольшая часть дерева наследования класса *CWnd*. Класс *CFrameWnd* представляет окна, выступающие в роли обрамляющих окон, в том числе также главные окна. От этого класса наследуются классы *CMDIFrameWnd* и *CMDIChildWnd*, используемые для отображения окон многооконного интерфейса *MDI*. Класс *CMiniFrameWnd* применяется для отображения окон уменьшенного размера. Такие окна обычно используются для отображения в них панели управления.

Отдельно представлены классы органов управления: *CButton* — кнопка, *CStatic* — метка, *CEdit* — поле редактирования, *CListBox* — список и т.п. Эти элементы управления используются в диалоговых окнах и панелях для организации ввода–вывода данных и разветвления программы.

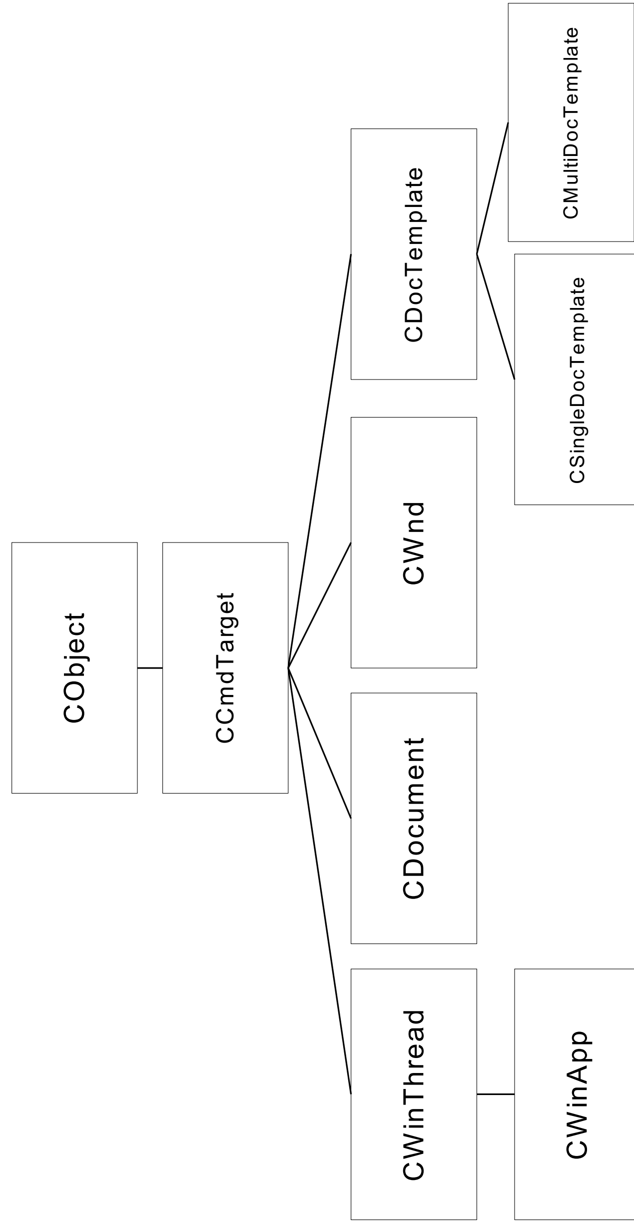

Рис. 2.1. Классы *CObject* и *CCmdTarget*

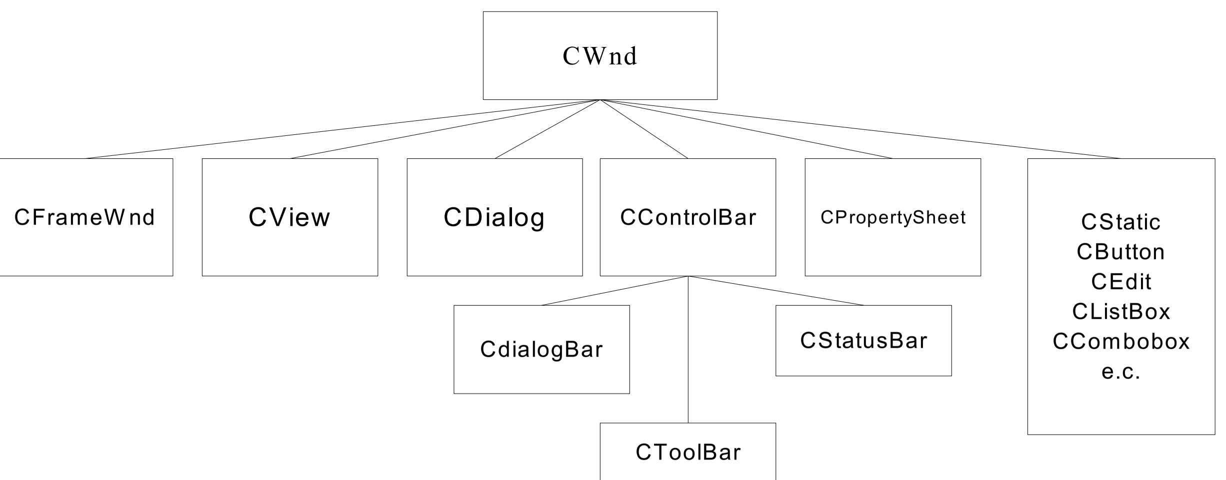

Рис. 2.2. Класс *CWnd*

Класс *CControlBar* и наследуемые от него классы предназначены для создания управляющих панелей. Такие панели могут содержать различные элементы управления и отображаются в верхней или нижней части главного окна приложения. Например, класс *CStatusBar* управляет панелью состояния. Панель состояния отображается в виде полосы в нижней части экрана, где приложение может выводить всевозможную информацию, например, краткую подсказку о выбранной строке меню.

Класс *CPropertySheet* представляет собой блокнот — диалоговую панель, содержащую несколько страниц.

Наибольший интерес имеет для нас класс *CView* и классы, наследуемые от него. Эти классы представляют окно просмотра документов приложения. Через это окно пользователь может изменять документ. Непосредственно наследуют классу *CView* два класса *CCtrlView* и *CScrollView*, которые используют для отображения стандартные классы элементов управления. Например, класс *CEditView*, наследующий классу *CCtrlView*, является готовым текстовым редактором. Класс *CScrollView* представляет окно просмотра, которое имеет полосы прокрутки. Класс *CFormView* позволяет создать окно просмотра документа, основанное на диалоговой панели.

3. Создание каркаса приложения

3.1. Создание проекта с использованием *мастера классов*

В среде *Visual C++* предусмотрено автоматическое создание каркаса приложения. В главном меню при наборе пункта *файл/ создать* открывается окно *создать*, имеющее четыре вкладки. На вкладке *проект* перечислены возможные типы проектов. Под проектом понимается набор файлов и конфигурация, необходимые для генерации программы. По умолчанию при создании проекта создаются две конфигурации: *Debug* и *Release*, т.е. работа проекта в режиме

отладки и без отладки. В проект может быть добавлено любое число конфигураций. Конфигурацией называется набор установок, определяющих режимы выполнения файла и опции компиляции файла.

На вкладке *проекты* приведены названия более 15 типов проектов. Из них три используют библиотеку *MFC*:

Приложение MFC — это приложение, поддерживающее полный графический интерфейс пользователя, реализованный на основе классов *MFC. Visual C*++ автоматически создаст файлы шаблона приложения с соответствующими классами и добавит эти файлы в проект. Выполнимый файл приложения будет иметь расширение *.exe*.

Библиотека DLL MFC — динамическая библиотека функций, реализованная на основе классов *MFC. Visual C*++ автоматически создаст файлы шаблона приложения с соответствующими классами и добавит эти файлы в проект. Файл приложения будет иметь расширение *.DLL*.

Элемент управления MFC — *ActivX*-элементы управления, реализованные на основе классов *MFC.* Файл приложения будет иметь расширение *.OCX.*

Мы будем создавать приложение *MFC*. т.к. нас интересует работа с графикой. Необходимо выделить имя этого типа проекта, затем определить имя проекта и каталог, в котором он будет размещен. На рис. 3.1. представлена первая страница, на которой записывается имя проекта. В левом столбце надо выбрать *Visual C++/MFC*. В центральном столбце - приложение *MFC*. После этого надо нажать на клавишу ОК. На следующей странице выбираются свойства проекта. Сначала надо выбрать какого типа проект вы хотите создать. Возможны три варианта: один документ (*single document*(*SDI*)), несколько документов (*multiple document*(*MDI*)) и на основе диалоговых окон (*dialog based)*. В первом случае проект создаст только одно окно представления документа. Во втором случае будет создано несколько

дочерних окон, каждое из которых соответствует отдельному документу. В третьем случае будет создано приложение на основе диалогового окна. Мы начнем с самого простого варианта — *single document.*

В следующие четыре свойства оставим все без изменения. Изменим только дополнительные параметры. Уберем галочки со следующих пунктов:

Печать и предварительный просмотр;

Элементы ActiveX;

Манифест стандартных элементов управления.

В первой программе нам не потребуются эти возможности. Справа предлагается выбрать стиль проекта. Мы выбираем стандарт *MFC*. Переходим к последнему свойству, появляется окно, где представлена информация о созданном проекте.

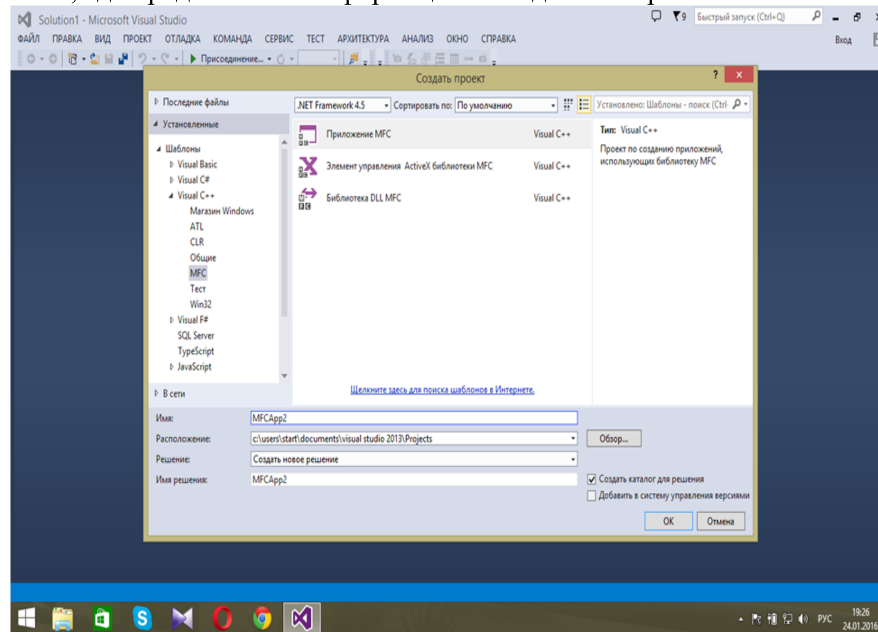

Рис. 3.1. Первая страница создания нового проекта

После того как вы нажали кнопку *OK*, мастер классов создаст каталог приложения, а в нем ряд файлов. Имя каталога совпадает с именем проекта.

3.2. Использование окна проекта.

Как только закроется окно создания проекта, справа откроется окно просмотра проекта (*Project Workspace*), которое имеет три кнопки — окно классов (*ClassView), ресурсы (ResourceView)* и обозреватель решений (*FileView)*. В первом окне представляется иерархическая схема классов проекта, во втором — схема ресурсов проекта и в третьем — схема файлов проекта. На рис. 3.2 представлена схема классов проекта, а на рис. 3.3 — схема файлов проекта.

Рассмотрим подробнее файлы проекта. Все файлы проекта разделены на три группы: файлы исходного кода (*Source Files), заголовочные файлы (Header Files),* файлы ресурсов *(Resource files)*.

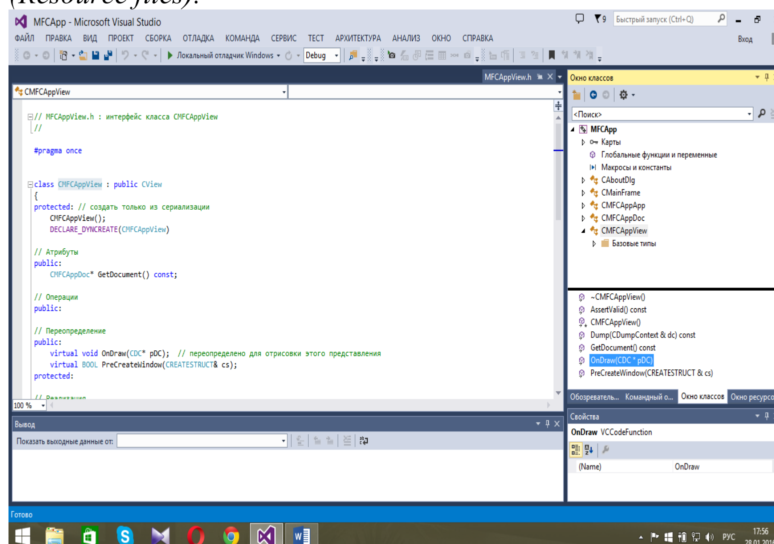

Рис. 3.2. Отображение схемы классов в окне проекта

Source Files — это файлы с расширением *.cpp*, *Header Files* — файлы с расширением *.h*. На самом деле одноименные файлы с расширением и *.cpp,* и *.h* относятся к описанию одного класса. В заголовочном файле описывается тип класса, а в файле с расширением *.cpp* — выполняемые функции класса.
Если мы теперь посмотрим подробнее заголовочные файлы, увидим, что всего было создано четыре основных класса: класс запуска приложения — *App1.h* , класс создания рамки окна — *MainFrm.h,* класс документа — *App1Doc.h,* класс отображения документа в окне приложения — *App1View.h.* Для того чтобы просмотреть содержимое файла, достаточно дважды щелкнуть мышкой по имени файла в окне проекта. Пример листинга файлов *App1.h* и *App1.cpp* приведен в ПРИЛОЖЕНИИ 1.

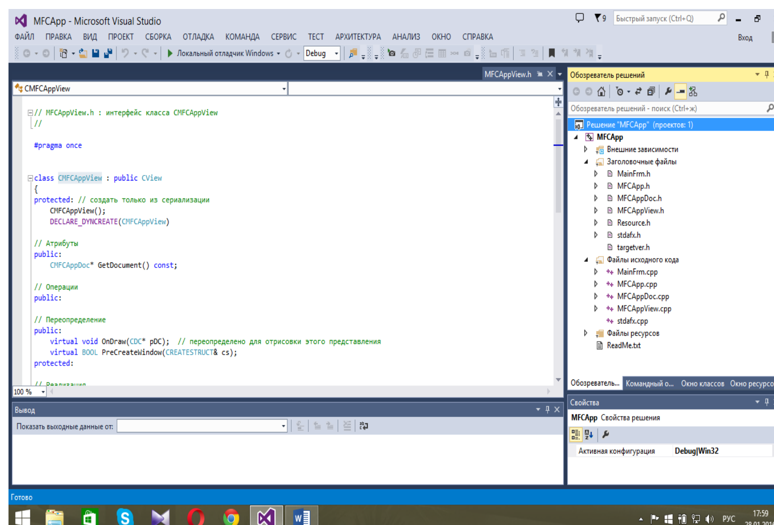

Рис. 3.3. Отображение имен файлов в окне проекта

3.3. Класс *CWinApp*

В заголовочном файле определяется класс *CApp1App,* который запускает на выполнение приложение, и является наследником класса *CWinApp.* Этот класс обязательно присутствует в любом *Windows*-приложении, создаваемом с помощью мастера классов. В нем инкапсулирован метод *WinMain.* При определении нового класса переопределяется виртуальная функция *virtual BOOL InitInstance*() и мастер классов сформирует код переопределения этого метода.
В выполняемой части файла *App1.cpp* интересным является определение шаблона документа *CSingleDocTemplate.* С помощью макроса *RUNTIME_CLASS* в шаблон документа объединены классы *Doc, View, MainFrm.*

3.4. Запуск приложения

Для компиляции и выполнения приложения воспользуемся пунктом меню «Отладка». В выпадающем меню выберем пункт «Начать отладку»*,* который позволяет скомпилировать сразу все файлы и ресурсы, создать выполняемую программу и запустить ее на счет. Результаты компиляции будут распечатываться в окне *"Вывод»,* которое располагается под окнами проекта и листинга программы. Если при компиляции возникли ошибки, сообщения о них распечатываются в этом же окне. Но в нашей программе не может быть ошибок, так как ее создал мастер классов. Результатом работы программы будет пустое окно с именем проекта в заголовке и стандартным меню.

4. Простая графическая задача

Имея каркас приложения, мы можем теперь создать на его основе любую однодокументную задачу. Однако поскольку наши знания о библиотеке *MFC* ограничены, попытаемся

создать приложение на базе функций этой библиотеки, не вдаваясь во все сложности иерархии классов и взаимосвязей объектов. Мы поработаем только с одним компонентом программы — классом «вид» (*class view*), тесно связанным с объектом «окно». Такие компоненты, как классы «приложение» (*application*), «рамка-окно» (*frame*) и «документ», можно пока игнорировать.
Рассмотрим следующую графическую задачу.

На клиентской области окна лежит цветной плоский шарик. Пользователь ударяет мышкой в любой точке окна, и шарик начинает двигаться в эту точку.

4.1. Класс *CApp1View*

Класс *CApp1View* — это класс, производный от класса *CView.* Как и для любого объекта C++, поведение объекта «вид» определяется функциями-членами (и переменными членами) его класса, включая и специфичные для приложения функции из производного класса, и стандартные функции, унаследованные от базовых классов. Используя *Visual C++,* можно создавать достаточно интересные приложения, просто добавляя код в производный класс «вид», сгенерированный мастером классов. В ПРИЛОЖЕНИИ 2 представлен код класса «вид», разделенный между двумя исходными модулями — заголовочным файлом (*.h*), и файлом реализации (*.cpp*).
Объект класса *CView* представляет собой прямоугольную клиентскую область окна, в которую производится вывод данных в графическом режиме. Если происходят изменения в клиентской области окна или изменения размеров окна, необходимо перерисовать окно. Для этого *Windows* посылает приложению сообщение *WM_PAINT.* Метод *OnPaint* класса «вид» создает контекст устройства класса *CPaintDC* и вызывает метод *OnDraw* производного класса *CApp1View,* передавая ему в качестве параметра контекст устройства. Следовательно, при внесении изменений в окне, т.е. при

рисовании в окне, необходимо переопределить метод *OnDraw,* внеся в него функции рисования. Дополнительно можно переопределить метод *OnInitialUpdate,* в который вносятся данные необходимые для рисования.

4.2. Контекст устройства

В *Visual C++* методы, обеспечивающие вывод на экран дисплея, используют контекст устройства. Контекст устройства является объектом класса *CDC.* При создании объекта данного класса в него заносится вся информация о графическом устройстве вывода. Указатель на контекст устройства передается как параметр при обработке метода *OnDraw.* Ниже представлен пример использования указателя на контекст устройства в функции *OnDraw:*

```
Void CApp1View::OnDraw(CDC* pDC)
    CString a;
    a=”text”;
    {pDC—>TextOutW(300,150, a);
   pDC—>Rectangle(20,20,300,200);}
```

Здесь *pDC* — указатель на контекст устройства,
TextOutW — метод вывода строки текста,
CString — класс строковых переменных,
Rectangle — метод отображает прямоугольник.

Класс *CDC* представляет собой набор методов, обеспечивающих работу с инструментами для рисования. Для того чтобы использовать контекст устройства, следует создать объект *CDC.* Библиотека *MFC* содержит несколько классов производных от *CDC.* В нашем приложении мы будем использовать только класс контекста устройства клиентской области *CClientDC.* Класс *CDC* содержит два контекста устройства: *m_hDC* и *m_hAttribDC.* Рассмотрим только те методы класса *CDC,* которые потребуются для нашей задачи. Остальные рекомендуем посмотреть в разделе *Help VisualC++.*

4.3. Средства работы с графикой

Библиотека *MFC* содержит следующие классы, инкапсулирующие работу с объектами графического интерфейса пользователя:
CPen (перо),
CBrush (кисть),
CFont (шрифт),
CBitmap (растровое изображение),
CPalett (палитра),
CRgn (область).
Каждый класс графического интерфейса имеет конструктор, выполняющий создание объекта. Затем объект инициализируется соответствующей функцией, для пера — *CreatePen,* и далее связывается с контекстом устройства с помощью метода *SelectObject,* возвращающего значение предыдущего объекта графического интерфейса. После завершения операций рисования можно вернуть предыдущее значение объекта графического интерфейса. Например:

```
Void CApp1View::OnDraw(CDC* pDC)
CPen myPen;
CPen* myOldPen;
myPen.CreatePen(PS_SOLID, 5, RGB(200,100,140));
myOldPen=pDC-> SelectObject(&myPen);
```

Рассмотрим метод *CreatePen.* Он содержит 3 параметра*:*
1. Структура линии, которой будем рисовать. *PS_SOLID* — сплошная, *PS_DOT* — пунктирная и т.д.
2. Толщина линии.
3. Цвет линии. Каждый цвет в *Windows* представляется сочетанием значений «красный», «зеленый» и «синий» (*RGB).* Три целочисленных параметра функции *RGB* меняют свое значение от 0 до 255. Ниже представлена таблица 16-ти

стандартных чистых цветов, записанных с помощью функции *RGB*.

Красный	Зеленый	Синий	Цвет
0	0	0	черный
0	0	255	синий
0	255	0	зеленый
0	255	255	бирюзовый
255	0	0	красный
255	0	255	малиновый
255	255	0	желтый
255	255	255	белый
0	0	128	темно-синий
0	128	0	темно-зеленый
0	128	128	темно-бирюзовый
128	0	0	темно-красный
128	0	128	темно-малиновый
128	128	0	темно-желтый
128	128	128	темно-серый
192	192	192	светло-серый

4.4. Первый рисунок

Теперь уже можно приступить к нашей задаче. Сначала нарисуем цветной шарик в клиентской области окна. Для этого исправим метод *OnDraw* в файле *App1View.cpp.* Шарик будем рисовать с помощью функции *Ellipse.* Воспользуемся также карандашом и кистью. Метод *OnDraw* будет иметь следующий вид:

```
        Void CApp1View::OnDraw(CDC* pDC)
{
        CPen myPen;
        CPen *myOldPen;
        myPen.CreatePen(PS_SOLID,2,RGB(250,128,200));
        myOldPen=pDC—>SelectObject(&myPen);
```

```
        CBrush myBrush;
        CBrush *myOldBrush;
        myBrush.CreateSolidBrush(RGB(255,255,0));
        myOldBrush=pDC—>SelectObject(&myBrush);

        pDC—>Ellipse(10, 10, 50, 50);
}
```

Запустим отладку. Если ошибок не было, на экране появится окно с меню, в клиентской области которого лежит небольшой плоский шарик желтого цвета с красной окантовкой.

4.5. Добавление обработчика сообщений

На следующем этапе мы должны заставить шарик двигаться. Для этого прежде всего заменим формулу для эллипса, вставив переменные координаты. Например, так:

```
pDC—>Ellipse(current_point.x, current_point.y,
current_point.x-10, current_point.y-10);
```

Переменная *current_point* будет указывать координаты шарика на клиентской плоскости в некоторый момент времени при его движении. Конечная точка движения будет находиться в месте удара мышью по клиентской области. Назовем ее *end_point.* Начальная точка положения шарика нам также понадобится, назовем ее *new_point*. Эти точки потребуются в разных методах внутри класса *View*. Поэтому определим их как глобальные в классе т.е. как члены класса. Но эти точки должны быть видны только в классе *View,* и, следовательно, их можно определить с модификатором *protected.*

В библиотеке *MFC* для определения точки на плоскости имеется класс *CPoint.* В этом классе каждая точка имеет координаты *x* и *y*.

Для определения переменной–члена класса проделаем следующую процедуру.

Обратимся к окну классов. Выделим класс *CApp1View*. Затем щелкнем по нему правой кнопкой мыши. Откроется контекстное меню. Нажмем строчку “Добавить”, затем – «Добавить переменную». Откроется окно, в котором надо указать класс определяемой переменной, имя переменной и модификатор доступа к переменной. Укажем *CPoint, current_point* и *protected.* После нажатия на кнопку ОК в файле *CApp1View.h* в разделе *protected* появится переменная *current_point.* Затем надо проделать то же самое с переменными *end_point, new_point.*

Следующий этап — добавление метода-обработчика сообщения *Windows* об ударе мыши по клиентской области окна. Для этого откроем мастер классов на вкладке *Сообщения* (Рис. 4.1). Прежде всего надо убедиться, что в окне *Имя класса* набрано имя класса *CApp1View.* В окне *Сообщения* ищем сообщение *WM_LBUTTONDOWN* , нажимаем кнопку *Добавить обработчик.* В окне *Существующие обработчики* появится имя функции *OnLbuttondown.* С помощью кнопки *Изменить код* можно сразу перейти в файл *App1View.cpp,* где уже имеется заголовок функции и фигурные скобки. Теперь запишем необходимые действия после удара мышкой по клиентской области окна. У метода *OnLbuttondown* имеются два параметра. Один из них — *point,* переменная типа *CPoint.*

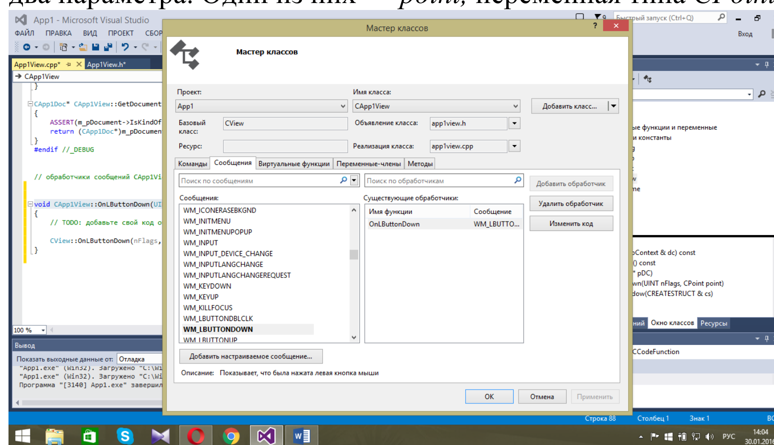

Рис. 4.1. Мастер классов

В этой переменной сохраняются координаты точки на клиентской области, где ударила мышь. Эти координаты необходимо переписать в глобальную переменную *end_point.* Теперь у нас есть начальные и конечные точки движения шарика. Для визуального представления движения воспользуемся таймером.

4.6. Таймер

Таймер *Windows* — это полезный элемент, который позволяет выполнять некоторые действия через равные промежутки времени. Для запуска таймера надо в тексте программы вставить вызов функции *CWnd::SetTimer* с параметром — интервалом времени. После этого с помощью *мастера классов* определить обработчик сообщения *WM_TIMER.* После запуска таймера с заданным интервалом в миллисекундах сообщения *WM_TIMER* постоянно посылаются окну до тех пор, пока не будет вызван метод *CWnd*::*KillTimer* или уничтожено окно. Функция *SetTimer* выглядит следующим образом:

UINT SetTimer(*UINT nIDEvent*, *UINT nElapse, void*
(*CALLBACK EXPORT* lpfnTimer*) (*HWND, UINT, UINT,*
DWORD));

Параметры:

nIDEvent — идентификатор таймера (целое, не равное нулю).

nElapse — интервал в миллисекундах.

lpfnTimer — определяет адрес приложения для функции обратного вызова, обрабатывающей сообщение *WM_TIMER.* Если значение параметра равно *NULL,* то сообщение будет поступать в очередь окна приложения и передаваться обработчику сообщения окна *CWnd.*

Вернемся к нашей задаче. Таймер мы установим в функции *OnLbuttondown,* которая будет выглядеть следующим образом:

```
void CApp1View::OnLButtonDown(UINT nFlags, CPoint point)
{	end_point = point;
	new_point=current_point;
	SetTimer(1, 10, NULL);
}
```

4.7. Обработчик сообщения *OnTimer*

Создадим обработчик сообщения *OnTimer* так же, как мы создавали обработчик сообщения *OnLbuttondown.* Эта функция будет выполняться через равные промежутки времени и каждый раз положение шарика на клиентской области окна должно продвигаться в направлении конечной точки, т.е. координаты шарика *current_point* будут двигаться равномерно по прямой, соединяющей эту точку с точкой *end_point.* Движение должно закончиться по достижении конечной точки. Уравнение движения шарика можно описать параметрическим уравнением прямой, проходящей через две точки. Для более точного вычисления координат шарика в выражении используются переменные типа float. Ниже приведена функция *OnTimer.*

```
void CApp1View::OnTimer(UINT nIDEvent)
{
	if((50*abs(end_point.x - current_point.x) >
abs(end_point.x- new_point.x)) && (50*abs(end_point.y -
current_point.y) > abs(end_point.y- new_point.y))){
	cpointx +=(float)( end_point.x- new_point.x)/50;
	cpointy +=(float)( end_point.y- new_point.y)/50;
	current_point.x=(int) cpointx;
	current_point.y =(int) cpointy;
	GetDocument()—>UpdateAllViews(NULL);
```

}

Последний метод *UpdateAllViews*(*NULL*) пересылает полученный документ на обработку в функцию *OnDraw.* Без этого метода мы не получим изображение шарика на экране.

4.7. Меню

Программа написана почти полностью, но если сейчас запустим последний вариант, ничего на клиентской области окна не увидим. Мы забыли определить начальное значение переменной *current_point,* которая собственно выводится в функции *OnDraw.* Инициализацию начальных данных обычно делают в методе *OnInitialUpdata*(). Но мы вставим новый пункт меню в окне нашего приложения. Нажав на этот пункт меню, увидим на экране шарик.

Для создания нового пункта меню следует выполнить два действия:

1. Создать элемент меню в ресурсе меню.
2. Создать обработчик этого элемента меню с помощью мастера классов.

Создание элемента меню начинается с открытия ресурса меню. Ресурс меню найдем в окне *ResourseView.* Щелкнув мышкой по слову «меню», увидим редактор меню. Пустая рамка следующего элемента меню всегда присутствует в окне редактора меню. Щелкнув мышью по новому элементу меню, увидим диалоговое окно свойств определяемого элемента меню. Здесь можно ввести заголовок элемента меню. Из свойств отметим только одно — *Pop-up.* По умолчанию этот переключатель включен и указывает, что элемент меню формируется как открывающий подменю.

Назовем элемент меню — *Function.* Появится подменю. Назовем элемент подменю — *Circle.*

Следующий этап — программирование обработчика команды меню. Для этого вызываем *мастер классов.* На вкладке *Имя*

класса укажем класс *CApp1View,* в котором будем создавать обработчик сообщений меню. Открываем вкладку *Команды*. В окне *Идентификаторы объектов* найдем идентификатор *ID_FUNCTION_CIRCLE.* В окне *Сообщения* появятся имена двух методов–обработчиков сообщений меню. Выберем метод *COMMAND* и нажмем кнопки *Добавить обработчик* и *Изменить код.* В окне обработчика сообщений наберем следующий код:

```
void CApp1View::OnFunctionCircle()
{
                current_point.x = 100;
                current_point.y = 100;
                cpointx = 100;
                cpointy = 100;
                GetDocument()->UpdateAllViews(NULL);

}
```

ПРИЛОЖЕНИЕ 1

Листинг файла *App1.h*.

```
// App1.h : главный файл заголовка для приложения App1
//
#pragma once

#ifndef __AFXWIN_H__
       #error "включить stdafx.h до включения этого
файла в PCH"
#endif

#include "resource.h"       // основные символы


// CApp1App:
// О реализации данного класса см. App1.cpp
//
```

```
class CApp1App : public CWinApp
{
public:
	CApp1App();


// Переопределение
public:
	virtual BOOL InitInstance();

// Реализация
	afx_msg void OnAppAbout();
	DECLARE_MESSAGE_MAP()
};

extern CApp1App theApp;
```

Листинг файла *App1.cpp*.

```
// App1.cpp : Определяет поведение классов для
приложения.
//

#include "stdafx.h"
#include "afxwinappex.h"
#include "afxdialogex.h"
#include "App1.h"
#include "MainFrm.h"

#include "App1Doc.h"
#include "App1View.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// CApp1App

BEGIN_MESSAGE_MAP(CApp1App, CWinApp)
```

```
	ON_COMMAND(ID_APP_ABOUT, &CApp1App::OnAppAbout)
	// Стандартные команды по работе с файлами
документов
	ON_COMMAND(ID_FILE_NEW, &CWinApp::OnFileNew)
	ON_COMMAND(ID_FILE_OPEN, &CWinApp::OnFileOpen)
END_MESSAGE_MAP()


// создание CApp1App

CApp1App::CApp1App()
{
	// поддержка диспетчера перезагрузки
	m_dwRestartManagerSupportFlags =
AFX_RESTART_MANAGER_SUPPORT_ALL_ASPECTS;
#ifdef _MANAGED
	// Если приложение построено с поддержкой среды
Common Language Runtime (/clr):
	//     1) Этот дополнительный параметр требуется
для правильной поддержки работы диспетчера
перезагрузки.
	//   2) В своем проекте для сборки необходимо
добавить ссылку на System.Windows.Forms.
	System::Windows::Forms::Application::SetUnhandled
ExceptionMode(System::Windows::Forms::UnhandledExceptio
nMode::ThrowException);
#endif

	// TODO: замените ниже строку идентификатора
приложения строкой уникального идентификатора;
рекомендуемый
	// формат для строки:
ИмяКомпании.ИмяПродукта.СубПродукт.СведенияОВерсии
	SetAppID(_T("App1.AppID.NoVersion"));

	// TODO: добавьте код создания,
	// Размещает весь важный код инициализации в
InitInstance
}

// Единственный объект CApp1App
```

```
CApp1App theApp;


// инициализация CApp1App

BOOL CApp1App::InitInstance()
{
	CWinApp::InitInstance();


	EnableTaskbarInteraction(FALSE);

	// Для использования элемента управления RichEdit
требуется метод AfxInitRichEdit2()
	// AfxInitRichEdit2();

	// Стандартная инициализация
	// Если эти возможности не используются и
необходимо уменьшить размер
	// конечного исполняемого файла, необходимо
удалить из следующего
	// конкретные процедуры инициализации, которые не
требуются
	// Измените раздел реестра, в котором хранятся
параметры
	// TODO: следует изменить эту строку на что-
нибудь подходящее,
	// например на название организации
	SetRegistryKey(_T("Локальные приложения,
созданные с помощью мастера приложений"));
	LoadStdProfileSettings(4);  // Загрузите
стандартные параметры INI-файла (включая MRU)


	// Зарегистрируйте шаблоны документов приложения.
Шаблоны документов
	//  выступают в роли посредника между
документами, окнами рамок и представлениями
	CSingleDocTemplate* pDocTemplate;
	pDocTemplate = new CSingleDocTemplate(
```

```
			IDR_MAINFRAME,
			RUNTIME_CLASS(CApp1Doc),
			RUNTIME_CLASS(CMainFrame),       // 
основное окно рамки SDI
			RUNTIME_CLASS(CApp1View));
		if (!pDocTemplate)
			return FALSE;
		AddDocTemplate(pDocTemplate);


		// Разрешить использование расширенных символов в 
горячих клавишах меню
		CMFCToolBar::m_bExtCharTranslation = TRUE;

		// Синтаксический разбор командной строки на 
стандартные команды оболочки, DDE, открытие файлов
		CCommandLineInfo cmdInfo;
		ParseCommandLine(cmdInfo);



		// Команды диспетчеризации, указанные в командной 
строке.  Значение FALSE будет возвращено, если
		// приложение было запущено с параметром 
/RegServer, /Register, /Unregserver или /Unregister.
		if (!ProcessShellCommand(cmdInfo))
			return FALSE;

		// Одно и только одно окно было инициализировано, 
поэтому отобразите и обновите его
		m_pMainWnd->ShowWindow(SW_SHOW);
		m_pMainWnd->UpdateWindow();
		return TRUE;
}

// обработчики сообщений CApp1App


// Диалоговое окно CAboutDlg используется для описания 
сведений о приложении
```

```
class CAboutDlg : public CDialogEx
{
public:
	CAboutDlg();

// Данные диалогового окна
	enum { IDD = IDD_ABOUTBOX };

protected:
	virtual void DoDataExchange(CDataExchange* pDX);
// поддержка DDX/DDV

// Реализация
protected:
	DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialogEx(CAboutDlg::IDD)
{
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialogEx::DoDataExchange(pDX);
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialogEx)
END_MESSAGE_MAP()

// Команда приложения для запуска диалога
void CApp1App::OnAppAbout()
{
	CAboutDlg aboutDlg;
	aboutDlg.DoModal();
}
```

ПРИЛОЖЕНИЕ 2

Листинг файла *App1View.h*

```
// App1View.h : интерфейс класса CApp1View
```

```
//

#pragma once
#include "atltypes.h"


class CApp1View : public CView
{
protected: // создать только из сериализации
	CApp1View();
	DECLARE_DYNCREATE(CApp1View)

// Атрибуты
public:
	CApp1Doc* GetDocument() const;

// Операции
public:

// Переопределение
public:
	virtual void OnDraw(CDC* pDC);  // переопределено
для отрисовки этого представления
	virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:

// Реализация
public:
	virtual ~CApp1View();
#ifdef _DEBUG
	virtual void AssertValid() const;
	virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Созданные функции схемы сообщений
protected:
	DECLARE_MESSAGE_MAP()
	CPoint end_point;
	CPoint current_point;
```

```
	CPoint new_point;
public:
	afx_msg void OnLButtonDown(UINT nFlags, CPoint
point);
	afx_msg void OnTimer(UINT_PTR nIDEvent);
protected:
	float cpointx;
	float cpointy;
public:
	afx_msg void OnFunctionCircle();
};

#ifndef _DEBUG  // отладочная версия в App1View.cpp
inline CApp1Doc* CApp1View::GetDocument() const
   { return reinterpret_cast<CApp1Doc*>(m_pDocument); }
#endif
```

Листинг файла ***App1View.cpp***.

```
// App1View.cpp : реализация класса CApp1View
//
#include "stdafx.h"
#include "App1.h"
#endif

#include "App1Doc.h"
#include "App1View.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// CApp1View

IMPLEMENT_DYNCREATE(CApp1View, CView)

BEGIN_MESSAGE_MAP(CApp1View, CView)
	ON_WM_LBUTTONDOWN()
	ON_WM_TIMER()
```

```
	ON_COMMAND(ID_FUNCTION_CIRCLE,
&CApp1View::OnFunctionCircle)
END_MESSAGE_MAP()

// создание/уничтожение CApp1View

CApp1View::CApp1View()
: end_point(0)
, current_point(0)
, new_point(0)
, cpointx(0)
, cpointy(0)
{
	// TODO: добавьте код создания

}

CApp1View::~CApp1View()
{
}

BOOL CApp1View::PreCreateWindow(CREATESTRUCT& cs)
{
	// TODO: изменить класс Window или стили
посредством изменения
	//  CREATESTRUCT cs

	return CView::PreCreateWindow(cs);
}

// рисование CApp1View

void CApp1View::OnDraw(CDC* pDC)
{
	CApp1Doc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);
	if (!pDoc)
		return;
	CPen myPen;
	CPen *myOldPen;
	myPen.CreatePen(PS_SOLID, 2, RGB(255, 0, 10));
```

```
	myOldPen = pDC->SelectObject(&myPen);
	CBrush myBrush;
	CBrush *myOldBrush;
	myBrush.CreateSolidBrush(RGB(225, 225, 0));
	myOldBrush = pDC->SelectObject(&myBrush);
	pDC->Ellipse(current_point.x, current_point.y,
current_point.x - 15, current_point.y - 15);
	// TODO: добавьте здесь код отрисовки для
собственных данных
}


// диагностика CApp1View

#ifdef _DEBUG
void CApp1View::AssertValid() const
{
	CView::AssertValid();
}

void CApp1View::Dump(CDumpContext& dc) const
{
	CView::Dump(dc);
}

CApp1Doc* CApp1View::GetDocument() const // встроена
неотлаженная версия
{
	ASSERT(m_pDocument-
>IsKindOf(RUNTIME_CLASS(CApp1Doc)));
	return (CApp1Doc*)m_pDocument;
}
#endif //_DEBUG


// обработчики сообщений CApp1View


void CApp1View::OnLButtonDown(UINT nFlags, CPoint
point)
{
```

```
	// TODO: дcpointyобавьте свой код обработчика 
сообщений или вызов стандартного
	end_point = point;
	new_point = current_point;
	SetTimer(1, 10, NULL);
	CView::OnLButtonDown(nFlags, point);
}


void CApp1View::OnTimer(UINT_PTR nIDEvent)
{
	// TODO: добавьте свой код обработчика сообщений 
или вызов стандартного
	if ((50 * abs(end_point.x - current_point.x) > 
abs(end_point.x - new_point.x)) && (50 * 
abs(end_point.y - current_point.y) > 
		abs(end_point.y - new_point.y))){
		cpointx += (float)(end_point.x - 
new_point.x) / 50;
		cpointy += (float)(end_point.y - 
new_point.y) / 50;
		current_point.x = (int)cpointx;
		current_point.y = (int)cpointy;
		GetDocument()->UpdateAllViews(NULL);

	}
	CView::OnTimer(nIDEvent);
}


void CApp1View::OnFunctionCircle()
{
	current_point.x = 100;
	current_point.y = 100;
	cpointx = 100;
	cpointy = 100;
	GetDocument()->UpdateAllViews(NULL);

	// TODO: добавьте свой код обработчика команд
}
```

СПИСОК ЛИТЕРАТУРЫ

1. Круглински Д., Уингоу С., Шеферд Дж. Программирование на *Microsoft Visual C++* 6.0 для профессионалов / Пер. с англ. — СПб: Питер; М.: Издательство торговый дом «Русская Редакция», 2001. — 864 с.
2. Баженова И.Ю. *Visual C++*. Уроки программирования. — М.: Диалог-МИФИ, 2001. — 416 с.
3. Тихомиров Ю.В. Самоучитель *MFC*. —СПб. : БХВ-Санкт–Петербург, 2000. — 640 с.
4.Хортон А. Visual C++ 2010: полный курс. –М.: ИД «Вильямс», 2011.
5.Пирогов В. Ю. Программирование на Visual C++.NET. – СПб.: БХВ – Петербург. 2003.–800с.
6.Майо Дж. Самоучитель Microsoft Visual Studio 2010. — СПб.: БХВ-Петербург, 2011. — 464с.
7.Дербакова Е.П. Клосс Ю.Ю. Начала программирования на Visual C++ с использованием библиотеки MFC: учеб.-метод. пособие. – М. :МФТИ, 2002, 36с.