Учебно-методические пособия Учебно-научного центра ОИЯИ Дубна

УНЦ-2009-41

В. А. Калинников

РАЗРАБОТКА WINDOWS-ПРИЛОЖЕНИЙ С ПОМОЩЬЮ API-ФУНКЦИЙ НА ОСНОВЕ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ЯЗЫКА C++

2009

Учебно-научный центр ОИЯИ

В. А. Калинников

РАЗРАБОТКА WINDOWS-ПРИЛОЖЕНИЙ С ПОМОЩЬЮ API-ФУНКЦИЙ НА ОСНОВЕ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ЯЗЫКА C++

Учебное пособие

Калинников В. А.

К17 Разработка Windows-приложений с помощью API-функций на основе объектно-ориентированного языка C++: Учебное пособие. — Дубна: ОИЯИ, 2009. — 99 с.

В пособии рассмотрена технология разработки Windows-приложений с использованием интерфейса прикладного программирования (API-функции) в среде Microsoft Visual Studio C++ 6.0. Подробно описана структура и технология создания разрабатываемого приложения и его основных компонентов: дочерних и диалоговых окон, меню, панелей инструментов, управляющих элементов, подключаемых библиотек и редактора ресурсов. Приводятся примеры программ с подробными комментариями.

Kalinnikov V. A.

The Programming Windows Applications with the API Functions on the Basis of Object-Oriented Implementation of C++: Textbook. — Dubna: JINR, 2009. — 99 p.

This manual describes the technology for building applications for the Windows platform based on the application programming interface (API functions) in the Microsoft Visual Studio C++ 6.0 environment. It provides a detailed description of the structure and technology for building applications and its basic components, such as: child and dialogue windows, menu, toolbars, control elements, dynamic link libraries and AppWizard. It also contains examples of applications with detailed comments.

Оглавление

1. Введение

Целью курса является получение навыков разработки приложений для Windows (все программы для Windows называются приложениями) с использованием интерфейса прикладного программирования (Application Programming Interfaces) API и специальной библиотеки классов MFC (Microsoft Foundation Classes). Для этого необходимо иметь представление об архитектуре и основных возможностях функций API и MFC, а также надо уметь пользоваться средой разработки.

В настоящее время для разработки Windows-приложений в качестве пользовательской среды в основном используются MS Visual C++ и Borland C++ Builder. По оценкам специалистов, MS Visual C++ является самой мощной программой такого класса, кроме того, в ней разработана специальная библиотека классов MFC, которая значительно облегчает процесс программирования.

Для учебных целей мы будем пользоваться средой MS Visual C++ 6. Тем не менее инструменты разработки достаточно "типичные", чтобы впоследствии (при необходимости) перейти к использованию других инструментальных средств. Например, программы, написанные в среде Win32 API, достаточно легко переносятся в среду LINUX и обратно.

Возрастающая сложность API-программ привела к широкому распространению объектно-ориентированных языков программирования и появлению специальных библиотек классов для взаимодействия с операционной системой (ОС) Windows. Поэтому программирование только на уровне API, с точки зрения требуемых программистских усилий, для большинства задач представляется слишком сложным и неэффективным. Программирование же приложений для Windows в MFC-среде с использованием специальной программы AppWizard позволяет автоматизировать

процесс написания кодов программы, в связи с чем существенно облегчается процедура программирования.

В данном курсе мы научимся создавать Windows-приложения «в ручном режиме» на базе API-интерфейса и в среде MFC с использованием универсального генератора динамических шаблонов (мастер AppWizard), который связан с библиотекой MFC и автоматически генерирует объектно-ориентированный код программы.

2. Объектно-ориентированное программирование на языке C++

2.1. Концепция объектно-ориентированного программирования (ООП)

Концепция ООП была сформулирована еще в конце 40-х годов, но реализовать в полной мере эту идею удалось только в последние десятилетия. Первые программы на языке BASIC часто представляли собой многостраничные листинги, в которых не прослеживалась четкая структура. Огромные программные блоки связывались друг с другом посредством одно- или двухбуквенных меток, по которым осуществлялись переходы с помощью многочисленных команд goto, а анализ и отладка таких программ, не говоря уже об их модификации, была нелегким делом.

В 60-х годах был реализован метод структурного программирования, основанный на использовании переменных с осмысленными именами, существования глобальных и локальных переменных и процедурного проектирования приложений по принципу "сверху вниз". Благодаря этой концепции программы стали понятнее. Упростился процесс отладки текста программ, так как появилась возможность контролировать выполнение не отдельных команд, а целых процедур. Примерами языков, ориентированных на подобный процедурный подход, были Си и Pascal.

Объектно-ориентированное программирование (ООП) обеспечивает программистам три важных преимущества:

- *упрощение программного кода и улучшение его структуризации* (программы стали проще для чтения и понимания, а код описания классов, как правило, отделен от кода основной части программы, благодаря чему над ними можно работать по отдельности);
- *модернизация программ*, т. е. добавление и удаление программных блоков, становится более простой задачей (чаще всего она сводится к добавлению нового класса, который наследует все свойства одного из имеющихся классов и содержит требуемые дополнительные методы);
- *одни и те же классы можно много раз использовать в разных программах* (удачно созданный класс можно сохранить в библиотечном файле, и его добавление в новую программу, как правило, не требует внесения серьезных изменений в ее текст).

2.2. Объекты

В основу объектно-ориентированного программирования заложены совершенно иные принципы и другая стратегия написания программ. ООП представляет собой уникальный подход к написанию программ, когда задачи и решения формулируются путем описания схемы взаимодействия связанных объектов. С помощью объектов можно смоделировать реальный процесс, а затем проанализировать его программными средствами. Каждый объект является определенной структурой данных, поэтому переменную типа struct в C/C++ можно рассматривать как простейший объект.

Взаимодействие между объектами осуществляется путем отправки им сообщений. Сообщения по своей сути аналогичны функциям в процедурном программировании. Когда объект получает сообщение, то выполняется хранящийся в нем метод, который возвращает результат вычислений в программу. Методы также называют *функциями-членами*, и

они напоминают обычные функции за тем исключением, что являются неразрывной частью объекта и не могут быть вызваны отдельно от него.

2.3. Классы

В C++ *класс* выступает в качестве шаблона, на основе которого создаются объекты. *Объект* – это экземпляр класса, доступный для выполнения над ним требуемых действий. Он создается путем описания класса как нового типа данных. Класс является расширенным вариантом структуры и содержит ряд констант, переменных, а также операций (функций-членов, или методов), выполняемых над ними. Чтобы произвести какое-либо действие над данными объекта, необходимо вызвать один из методов класса.

От любого класса можно «породить» один или несколько *подклассов*, в результате чего сформируется *иерархия классов*. Родительские классы содержат методы общего характера, тогда как решение специфических задач выполняется производными классами. Доступ к отдельным частям класса регулируется с помощью специальных ключевых слов: public (*открытая часть*), private (*закрытая часть*) и protected (*защищенная часть*). Методы, расположенные в открытой части, формируют интерфейс класса и могут свободно вызываться всеми пользователями класса. Переменные-члены класса обычно находятся в части private, поэтому в классе должны существовать интерфейсные методы, позволяющие читать и модифицировать значение каждой переменной. Доступ к переменным-членам закрытой части класса возможен только из его собственных методов, а к защищенной – также из методов классов-потомков.

Инициализация переменных-членов класса и резервирование памяти выполняется при помощи функции-члена *конструктор*. Имя конструктора совпадает с именем класса, которому он принадлежит. Конструкторы

могут принимать аргументы и быть перегруженными. При создании объекта класса нужный конструктор вызывается автоматически.

Деструктором называется специальная функция-член класса, которая в основном служит для освобождения динамической памяти, занимаемой удаляемым объектом. Деструктор, как и конструктор, носит имя класса, которое в качестве префикса содержит знак тильды (~). Деструктор вызывается автоматически, когда в программе встречается оператор delete с указателем на объект класса или когда объект выходит за пределы своей области видимости. В отличие от конструкторов, деструкторы не принимают никаких аргументов и не могут быть перегружены.

2.4. Примеры создания простых объектно-ориентированных программ

Применение классов, объектов и функций-членов в объектно-ориентированном C++ рассмотрим на примере простых программ.

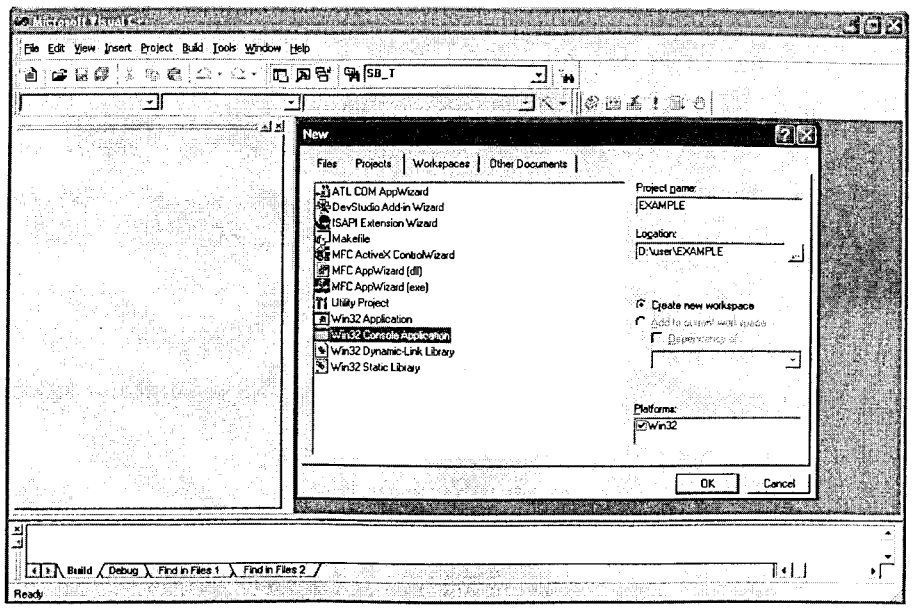

Рис.1. Создание проекта EXAMPLE в среде MS Visual C++ в Win32 Console Application

Создадим в MS Visual C++ новый проект EXAMPLE (программные задачи в MS Visual C++ оформляются в виде проектов). Для этого в MS Visual C++ командой *File* → *new*, далее в окне *new* в *Projects*→*Win32 Console Application* (рис.1) создаем новый проект с названием в текстовом поле *Project name* → EXAMPLE, а расположение задаем в текстовом поле *Location* →*d:\\VisualC6\2009*.

Итак, мы создали проект EXAMPLE, с файлом example.dew (который определяет параметры новой рабочей области) и файл exampl.dsp - с параметрами этого проекта. Затем в *Projects*→*Add To Projects* → *new* → *Files* → *C++Source File* добавляем в наш проект новый Си-файл с именем example.cpp. В созданном файле объявим новый класс с именем *institute*:

```
class institute
{
➧  // структура класса
};
```

Зададим структуру класса *institute*. Пусть класс содержит две переменные и один метод (функцию) обработки. Объявим переменные двух типов: закрытую, которую можно использовать только внутри данного класса, и открытую, которую можно использовать в любой части программы. Для этого используем модификаторы доступа *private* – для закрытых и *public* – для открытых объектов нашего класса. В нашем классе мы будем использовать открытый метод обработки. Тогда структура класса *institute* будет

```
class institute
{
private:
➧   int privatedata;
public:
➧   int publicdata;
➧   int publicmethod();
};
```

где int privatedata – целочисленная переменная закрытого типа, а int publicdata – открытого типа; int publicmethod(void) – метод обработки открытого типа. Так-так у метода в скобках *void* – это означает, что метод не получает никаких параметров, а *int* указывает, что он возвращает целочисленный параметр после выполнения.

Теперь надо инициализировать объекты нашего класса institute. Начинаем с инициализации метода publicmethod(). Вначале укажем, к какому классу принадлежит этот метод: → int institute::publicmethod(); т. е. метод принадлежит классу *institute*. Так как в классе institute имеется закрытая переменная int privatedata, то пусть наш метод возвращает ее значение в общую программу. Тогда код инициализации метода будет:

```
int institute::publicmethod()
{
➧   return privatedata;
}
```

Таким образом, код структуры нашего класса имеет следующий вид:

```
class institute
{
private:
➧ int privatedata;
public:
➧   int publicdata;
➧   int publicmethod();
};
int institute::publicmethod()    // метод
{
➧   return privatedata;
}
```

В классе *institute* объявлены две переменные, но так как класс не содержит параметров, то их инициализацию (передачу параметров) выполняем при помощи функции *конструктора*. В конструкторе удобно инициализировать данные класса. Имя конструктора совпадает с именем

класса (он не возвращает никакого значения). Пусть конструктор передает значение закрытой переменной класса, т. е. он получает значение, которое должно быть передано переменной *privatedata*:

```
class institute
{
private:
      int privatedata;
public:
      int publicdata;
      int publicmethod();
➧     institute(int value);   // объявляем конструктор
};
int institute::publicmethod()
{
      return privatedata;
}

institute::institute(int value)  // конструктор
{
➧     privatedata=value;       // определяем код конструктора
}
```

Мы создали новый класс и инициализировали его объекты. Теперь создадим новый объект на базе данного класса *institute*. Включим в программу функцию main(). Функция main() содержит код программы, выполняемой при запуске. Объявим новый объект *gruppa* класса *institute* и в этом объекте передадим значение конструктору для *privatedata*, т. е.

```
➧   void main()
    {
      ➧ institute gruppa(1);
    }
```

Чтобы передать значение открытой переменной int publicdata, воспользуемся оператором точка, в языке C++ → gruppa.publicdata = 6; → переменной *publicdata* предаем значение, равное 6. Теперь нам осталось

только вывести результаты на экран. Используем для этого стандартную библиотеку iostream.h языка C++ для вывода данных:

- cout << "publicdata = " << gruppa.publicdata << "\n";
- cout << "publicmethod = " << gruppa.publicmethod() << "\n";

Таким образом, код нашей программы проекта EXAMPLE будет

```
#include <iostream.h>
class institute
{
private:
        int privatedata;
public:
        int publicdata;
        int publicmethod();
        institute(int value);
};
int institute::publicmethod()
{
        return privatedata;
}
institute::institute(int value)
{
        privatedata=value;
}
void main()
{
institute gruppa(1);
gruppa.publicdata =6;
        cout << "publicdata = " << gruppa.publicdata << "\n";
        cout << "publicmethod = " << gruppa.publicmethod() << "\n";
}
```

На рис.2 показан результат выполнения этой программы в консольном окне.

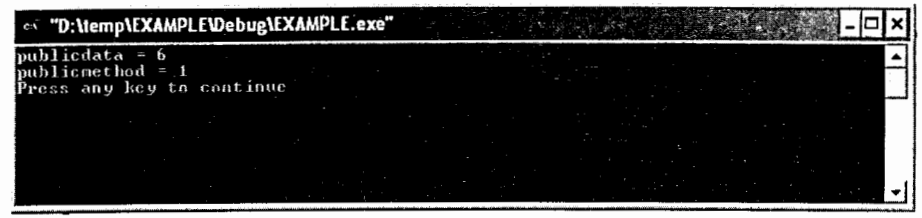

Рис.2. Результат выполнения программы проекта EXAMPLE в Visual C++

Рассмотрим реальное применение объектов и классов в объектно-ориентированном C++ на примере программы по вычислению среднего возраста и роста в группе студентов. Для этого в MS Visual C++, по аналогии с предыдущим примером, создадим новый проект с названием EXAMPLE2 и добавим в него новый Си-файл с именем example2.cpp. В созданном файле example2.cpp объявляем новый класс *Gruppa*. В этом классе для хранения информации о возрасте и росте объявляем о выделении требуемого массива (области) памяти Data и вводим указатель на начало этого массива. В языке C++ указатель обозначается оператором * перед массивом (т. е. *Data). Затем вводим индексную переменную int DataIndex для этого массива.

В классе Gruppa объявляем функции-члены: конструктор Gruppa(int NumberStudents) для выделения необходимого массива памяти Data (конструктор получает параметр int NumberStudents, по значению которого программе будет выделена требуемая память ОС Windows); метод AddData(int value) для добавления данных в массив Data; метод GetData(int Index) для извлечения данных из массива Data; метод AverageData(void) для вычисления средних значений над данными в массиве Data, т. е.

```
class Gruppa
{
        int *Data;
        int DataIndex;
    public:
```

```
    Gruppa(int NumberStudents);
    ~ Gruppa (void);
    void    AddData(int value);
    int     GetData(int Index);
    float   AverageData(void);
};
```

Необходимый массив памяти для хранения данных выделяем с помощью оператора *new* в конструкторе Gruppa(), и одновременно здесь же инициируем индексную переменную массива DataIndex = 0. Для правильного удаления объектов в класс Gruppa вводим функцию-член деструктор ~Gruppa(void), которая по завершении освободит выделенную для программы область памяти Data, т. е.

```
Gruppa::Gruppa (int NumberStudents)    // конструктор
{
    Data = new int[NumberStudents];
    DataIndex = 0;
}
Gruppa::~ Gruppa (void)                // деструктор
{
    delete Data;
}
```

Запишем код для метода AddData(). В массив Data по значению индексной переменной DataIndex заносится целочисленное значение value, а значение индексной переменной DataIndex увеличивается на 1, т. е.

```
void Gruppa::AddData(int value)
{
    Data[AgeDataIndex++] = value;
}
```

Метод GetData возвращает данные в ячейке памяти по значению индексной переменной DataIndex (если оно не превышает значение выделенной области памяти):

```
int Gruppa::GetData(int Index)
{
    if(Index <= DataIndex)
```



```
    {
        return Data[Index];
    }
else
    {
        return -1;
    }
}
```

Метод AverageData() вычисляет среднее значение данных в вещественной переменной float Sum, выбранных в цикле от index2 = 0 до index2 < AgeDataIndex. Так как данные в массиве Data целочисленные, а переменная Sum вещественная, то выполняем преобразование формата с помощью оператора float:: →(float)Data[index2]. Затем вычисляем среднее значение выбранных данных и возвращаем его в программу:

```
float Gruppa::AverageScore(void)
{
float Sum = 0;
        for(int index2 = 0; index2 < AgeDataIndex; index2++)
            {
            Sum += (float) Data[index2];
            }
        return Sum / (float) DataIndex;
}
```

Для вычисления средних значений возраста и роста студентов в функции main() объявим объекты *Age* (средний возраст) и *Height* (средний рост) класса *Gruppa*. С помощью этих объектов для выделения требуемого массива памяти *Data* через указатель (*точка*) передаем значение параметра *NumberStudents* функции-члену конструктор:

```
void main()
{
    Gruppa Age(9);
    Gruppa Height (9);
```

Затем с помощью метода AddScore() запишем в массив данные по возрастам и росту:

```
void main()
{
        Gruppa Age(9);     Gruppa Height (9);
    ➧ Age.AddData(20);   Age.AddData (21);   Age.AddData (22);
    ➧ Age.AddData (33);   Age.AddData (34);   Age.AddData (25);
    ➧ Age.AddData (26);   Age.AddData (27);   Age.AddData (28);
    ➧ Height.AddData(210);   Height.AddData (180);
    ➧ Height.AddData (160);   Height.AddData (175);
    ➧ Height.AddData (160);   Height.AddData (166);
    ➧ Height.AddData (170);   Height.AddData (180);
    ➧ Height.AddData (179);
}
```

Теперь осталось только вывести полученный результат на консоль. Как и в предыдущем примере, для этого используем стандартный оператор вывода данных библиотеки iostream.h:

```
void main()
{
        Gruppa Age(9);
        Gruppa Height (9);
        Age.AddData(20);   Age.AddData (21);   Age.AddData (22);
        Age.AddData (33);   Age.AddData (34);   Age.AddData (25);
        Age.AddData (26);   Age.AddData (27);   Age.AddData (28);
        Height.AddData(210);   Height.AddData (180);
        Height.AddData (160);   Height.AddData (175);
        Height.AddData (160);   Height.AddData (166);
        Height.AddData (170);   Height.AddData (180);
        Height.AddData (179);
    ➧ cout << "Average for Age = " << Age.AverageData() << "\n";
    ➧ cout << "Average for Height=  " << Height.AverageData() << "\n";
}
```

На рис.3 показан результат выполнения этой программы.

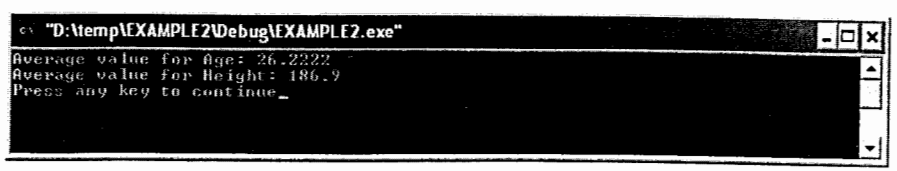

Рис.3. Результат выполнения программы проекта EXAMPLE2 в Visual C++

На этом закончим краткий обзор основных понятий классов, объектов и функций-членов в объектно-ориентированном языке C++.

3. Разработка Win API-приложений в операционной среде Windows

3.1. Особенность создания приложений в среде Windows

Рассмотрим особенности создания Windows-приложений с использованием интерфейса прикладного программирования Win32-API (Application Programming Interfaces). Приложения, написанные на базе Win32 API, поддерживаются 32-разрядными ОС Windows, легко переносятся на платформу Linux и обратно и совмещают в себе достоинства структурного и объектно-ориентированного языка C++.

Windows представляет собой графическую многозадачную операционную систему. Все программы, разработанные для этой среды, должны соответствовать определенным стандартам и требованиям. Это касается, прежде всего, внешнего вида окна программы и принципов взаимодействия с пользователями.

Особенность создания приложений в среде Windows заключается в том, что здесь применяются специальные методики программирования и используется специфическая терминология, которую можно разделить на две большие категории: терминология, связанная с пользовательским интерфейсом (меню, диалоговые окна, значки и т.д.), и терминология, относящаяся непосредственно к программированию (сообщения, вызовы функций).

Приложения Windows представляют целостную систему различных элементов, таких как окна, меню, диалоговые панели и другие объекты Windows, а каждый элемент задается множеством параметров и состояний. Поэтому основной задачей при разработке Windows-приложений является создание целостной программы управления этими параметрами и состояниями.

3.2. Преимущества Windows

Среди наиболее важных следует указать стандартизированный графический интерфейс пользователя, многозадачность, совершенные средства управления памятью, аппаратную независимость, возможность широкого применения библиотек динамической компоновки (DLL).

3.3. Операционная система Windows

3.3.1. Многозадачность Windows

В операционной среде MS-DOS запущенная программа получала полный контроль над предоставляемыми ей ресурсами компьютера, включая устройства ввода/вывода, память, монитор и центральный процессор. В среде Windows все эти ресурсы динамически распределяются между запущенными приложениями. Многозадачность Windows состоит в том, что одновременно можно запустить несколько приложений или открыть сразу несколько сеансов работы с одним приложением. В любой момент времени пользователь может переместить одно из окон в иное место экрана, перейти от одного окна к другому, изменить их размер или произвести обмен данными между приложениями.

Считается, что приложения выполняются одновременно, однако в действительности это не так. В текущий момент времени только одно приложение может использовать ресурсы процессора. Многозадачность реализуется за счет того, что процессор переключается между выполняющимися заданиями в течение очень коротких промежутков времени. Приоритеты выполнения одновременно запущенных программ также неодинаковы. В текущий момент времени активным, т. е. принимающим данные от пользователя, может быть только одно приложение, хотя инструкции для процессора поступают сразу от всех открытых приложений независимо от их состояния. Задачу определения приоритетов приложений и распределения времени работы процессора берет на себя Windows.

Многозадачность Windows обеспечивается посредством выделения квантов времени для запущенных в системе *потоков*. Поток – это набор значений внутренних регистров процессора, которые содержат информацию о том, какая машинная команда выполняется процессором в данный момент и где расположены локальные переменные.

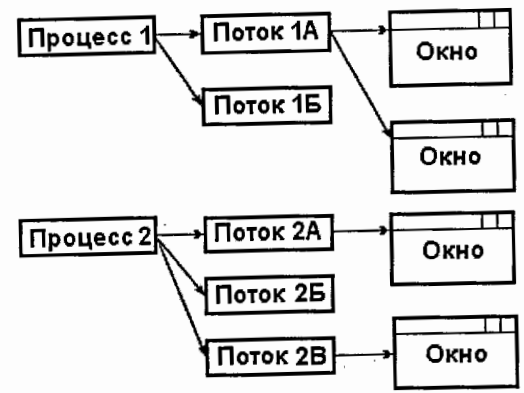

Рис.4. Процессы, потоки и окна в ОС Windows

При запуске приложения в ОС Windows (рис.4) происходит создание *процесса*. Процессу принадлежат открытые файлы, участки оперативной памяти и другие ресурсы. Кроме того, ему принадлежит *программный поток*. Одному процессу могут принадлежать несколько потоков, а создание нового потока требует немного системных ресурсов. Потоки одного процесса имеют доступ ко всем его объектам и отличаются друг от друга лишь точкой входа и локальными переменными, расположенными в общем адресном пространстве процесса. Потоки обладают правом доступа к памяти и другим ресурсам, принадлежащим этому процессу. Поэтому между потоками одного процесса легче организовать обмен данными и взаимодействие, чем между потоками разных процессов.

Потоки, принадлежащие разным процессам, не имеют между собой ничего общего, однако они могут получить доступ к общим ресурсам ОС и памяти, используя механизмы *межпроцессного взаимодействия*.

В начале работы каждый процесс обладает единственным *первичным потоком*. Информация о первичном потоке передается ОС в виде адреса функции. Поэтому все Windows-приложения содержат вызываемую при запуске функцию WinMain(), адрес которой и передается в качестве адреса первичного потока.

3.3.2. Сообщения, обработка и генерация сообщений в ОС Windows

a) Сообщения в Windows

В основе работы многозадачной операционной системы Windows лежит механизм *генерации и обработки сообщений*. С точки зрения приложений сообщения являются формой уведомления о произошедших событиях, на которые приложение должно (или не должно) каким-то образом реагировать. Пользователь может инициировать событие щелчком или перемещением указателя мыши, изменением размера окна или выбором команды из меню. Другие события могут быть инициализированы самим приложением или генерироваться автоматически самой Windows.

b) Генерирование сообщений

Именно реализация концепции обмена сообщениями позволяет Windows быть многозадачной средой. Работа Windows основана на передаче, приеме и обработке сообщений. Существует четыре основных источника, от которых приложение может получить сообщение. Это пользователь, Windows, само приложение, другие приложения Windows.

Сообщения пользователей включают информацию о вводе текста, перемещении и нажатии кнопок мыши, выборе команд меню, перемещении бегунка полосы прокрутки и т. д. Большую часть времени приложение занято обработкой именно таких сообщений. Получение сообщения от пользователя означает, что человек, запустивший программу, хочет изменить ход ее выполнения.

Системные сообщения от Windows посылаются программе при изменении ее состояния. Например, щелчок на значке приложения означает, что пользователь хочет сделать данное приложение активным. В этом случае Windows сообщает приложению, что на экране открывается его окно, размер и положение окна изменяются. В случае завершения работы Windows также посылает каждому открытому приложению уведомление о необходимости проверить сохранность данных и закрыть свои окна.

Собственные сообщения могут быть инициализированы самим приложением. Например, в электронной таблице завершение вычислений в ячейках может сопровождаться автоматическим обновлением диаграммы, основанной на данном блоке ячеек. В таком случае при завершении вычислений генерируется сообщение для этого же приложения о необходимости обновления диаграммы (перерисовки окна приложения).

Сообщения от других приложений. Сообщения этого типа в настоящее время применяются достаточно редко. Примером межзадачного обмена сообщениями может служить протокол динамического обмена данными DDE (Dynamic DataExchange).

c) Обработка сообщений в ОС Windows

Рассматривая роль сообщений в Windows, необходимо отметить, что именно благодаря системе сообщений становится возможной многозадачность Windows. Система сообщений позволяет распределять время работы процессора между всеми открытыми приложениями. Каждый раз, когда Windows посылает сообщение приложению, она на некоторое время открывает этому приложению доступ к процессору. Единственная возможность для приложения получить доступ к процессору – это получить сообщение от Windows. Кроме того, сообщения позволяют приложению прореагировать определенным образом на событие, произошедшее в системе. Это событие могло быть вызвано самим приложением, другим

приложением, выполняющимся в это же время в Windows, пользователем или операционной системой. Каждый раз, когда происходит то или иное событие, Windows оповещает о нем все заинтересованные приложения, открытые в настоящий момент.

В ОС Windows существует только один механизм обработки сообщений – *системная очередь сообщений*. На последовательность обработки сообщения в ней влияют два фактора: очередь, в которой находится сообщение и приоритет его самого. Сообщения могут поступать из двух очередей – системной и очереди приложения. Но даже если сообщение поступило от самого приложения, оно все равно будет помещено в системную очередь. Когда подойдет черед сообщения, оно будет направлено в очередь соответствующего приложения. Такая организация работы системной очереди позволяет Windows контролировать прохождение всех сообщений и ограничивает ответственность приложений обработкой только тех сообщений, которые относятся непосредственно к ним.

Сообщения обычно помещаются в очередь по принципу FIFO (first in, first out – первым поступил, первым обслужен). Такого типа сообщения называются *синхронными*. Но иногда Windows вставляет сообщение сразу в голову очереди. Сообщения такого типа называются *асинхронными*. Существует несколько видов асинхронных сообщений, среди них: сообщения о перерисовке, сообщения таймера и сообщения о завершении программы. Например, сообщение таймера может запускать некоторое действие в определенный момент времени независимо от того, какие сообщения сейчас находятся в очереди; оно имеет наивысший приоритет и передается раньше всех других сообщений.

Если сообщения поступают одновременно, то эта проблема в Windows решается одним из двух способов. Первый способ состоит в

задании приоритетов. Когда загружается некоторое приложение, для него автоматически устанавливается нулевой приоритет. В процессе работы приложения его приоритет может быть изменен. Любые конфликты разрешаются в пользу того приложения, чей приоритет выше (изменять приоритет приложения, заданный по умолчанию, обычно не рекомендуется).

Второй способ определяет очередность передачи сообщений группе приложений с одинаковым приоритетом. Если ОС Windows видит, что у приложения образовался длинный список необработанных сообщений, то она автоматически приостанавливает передачу этому приложению новых сообщений, хотя продолжает передавать сообщения другим, более свободным приложениям.

Функции модуля USER обеспечивают передачу сообщений из системной очереди в модули *цикла обработки сообщений* открытых приложений, причем каждое открытое приложение может организовать свою собственную очередь, в которой накапливаются сообщения, адресованные любому окну, открытому данным приложением.

d) Формат и тип сообщений

Сообщения используются для информирования приложения о том, что в системе произошло то или иное событие. На практике сообщение направляется не столько самому приложению, сколько определенному окну, открытому этим приложением.

Сообщения в Windows описываются с помощью структуры *msg*:

```
typedef struct tagMSG
{
 HWND      hwnd;      // Идентификатор окна-получателя
 UINT      message;   // Идентификатор сообщения
 WPARAM    wParam;    // Дополнительная информация, смысл
 LPARAM    lParam;    //которой зависит от типа сообщения
 DWORD     time;      // Время посылки сообщения
```

```
    POINT       pt;          // Местоположение указателя мыши
}
```

Независимо от типа все сообщения характеризуются четырьмя параметрами: дескриптором окна (которому адресуется данное сообщение), типом сообщения и еще двумя 32-разрядными параметрами.

Дескриптором окна называется уникальный номер, который присваивается всем системным объектам, таким как окна, элементы управления, меню, значки, перья и кисти, а также областям памяти, устройствам вывода. Переменная hwnd – это уникальный идентификатор окна, которому было послано сообщение.

Тип сообщения задается идентификатором, который определен в файле заголовков <windows.h>. Идентификаторы начинаются, как правило, с двухсимвольного префикса, за которым следует символ подчеркивания. Так, *оконные сообщения* начинаются с префикса WM_ (например: WM_CREATE, WM_PAINT, WM_CLOSE, WM_COPY, WM_PASTE и т. д.), сообщения от кнопок имеют префикс BM_, полей – EM_, списков – LB_.

Два 32-разрядных параметра wParam и lParam несут дополнительную информацию, wParam используется для описания старшего параметра сообщения, а lParam – для описания младшего аргумента сообщения. Содержание параметров wParam и lParam может изменяться в зависимости от типа сообщения, например, где находился указатель мыши или бегунок полосы прокрутки. Посредством этих параметров передается информация о том, какая клавиша нажата, причем: младшее слово wParam – идентификатор кнопки LOWORD (wParam) = ID_BUTTON; старшее wParam – код извещения (по нему судят о совершенном над кнопкой действии). Например, при нажатии на кнопку передается параметр HIWORD(wParam) = BN_CLICKED, а lParam содержит дескриптор окна кнопки lParam = (HWND)hButton.

3.4. Базовые концепции программирования, типы и структура данных в Windows

Особенность создания приложений в среде Windows заключается в том, что здесь применяются специальные методики программирования и используется своя терминология. Чтобы эффективно общаться с ОС Windows и легко понимать друг друга, необходимо внимательно изучить все описанные ниже в табл.1 и 2 термины и понятия.

Таблица 1. Часто используемые типы данных Windows

Тип	Описание
handle	32-разрядное беззнаковое число, используемое в качестве дескриптора
HDC	Дескриптор контекста устройства
hwnd	Дескриптор окна
LONG	32-разрядное целое число со знаком
lparam	Используется для описания младшего аргумента сообщения
lpstr	32-разрядный указатель на строку
LPCSR	То же, что и lpstr, но используется для указания константных строк
lpvoid	Обобщенный тип указателя, эквивалентен void
lresult	Используется для возвращения значений из оконных процедур
unit	32-разрядное беззнаковое целое число
wchar	16-разрядный символ UNICODE для представления печатных символов
wparam	Используется для описания старшего аргумента сообщения

Таблица 2. Стандартные структуры Windows

Структура	Что определяет
Msg	Параметры сообщения
PAINTSTRUCT	Область окна, требующая перерисовки
rect	Прямоугольная область окна
WNDCLASS	Класс окна

3.5. Окна в Windows

3.5.1. Типы окон в Windows

В Windows существует большое количество разных типов окон. Некоторые из них очевидны, например, главное окно приложения и диалоговые окна. Менее очевидно, что большинство элементов управления в окнах приложений и диалоговых окнах тоже являются окнами. Каждая кнопка, строка ввода, полоса прокрутки, список, пиктограмма и даже фон рабочего стола – рассматриваются ОС Windows как окна. Все элементы окна, его размер и внешний вид контролируются открывающей его программой. У каждого окна в Windows есть уникальный *дескриптор окна* (имя окна). Переменные для хранения оконных дескрипторов обычно имеют тип hwnd.

Окно – это не только область на экране, а конкретный объект, предназначенный для организации взаимодействия между пользователем и приложением. На практике сообщение направляется не столько самому приложению, сколько окну, открытому этим приложением. Windows отслеживает события пользовательского интерфейса и преобразует их в сообщения. В структуру сообщения помещается и дескриптор окна получателя. Затем сообщение передается непосредственно в оконную процедуру окна-получателя сообщения.

Для управления окнами ОС Windows хранит информацию о них в иерархической структуре, упорядоченной по принадлежности, которая бывает двух типов: *родительское/дочернее окно* и *владелец/собственное окно*. В корне иерархии находится *окно рабочего стола*, создаваемое Windows в процессе загрузки. Рабочий стол является родительским окном для *окон верхнего уровня*. У *дочерних окон* родительским окном может быть окно верхнего уровня или другое *дочернее окно, расположенное выше в иерархии*.

Отношения *родительское окно/дочернее* и *владелец/собственное окно* отличаются тем, что дочернее окно ограничено областью родительского окна. Собственное окно выводится на экран поверх окна-владельца и исчезает, когда окно-владелец сворачивается. Типичным примером отношения *владелец/собственное окно* является отображение диалогового окна. Диалоговое окно не является дочерним окном, так как оно не ограничено клиентской областью главного окна приложения, но принадлежит главному окну приложения.

3.5.2. Компоненты окна

Окно – это специальная прямоугольная область экрана. Все элементы окна, его размер и внешний вид контролируются открывающей его программой. Каждый щелчок пользователя на каком-нибудь элементе окна вызывает ответные действия приложения. Многозадачность в Windows заключается, в частности, в возможности одновременно открывать окна сразу нескольких приложений или нескольких окон одного приложения. Активизируя с помощью мыши или клавиатуры то или иное окно, пользователь дает системе понять, что последующие команды и данные следует направлять именно этому окну.

Стандартный внешний вид окон всех приложений Windows и предсказуемость работы различных их компонентов позволяют пользователям чувствовать себя уверенно с новыми приложениями и легко разбираться в принципах их работы. Основные компоненты любого окна:

Рамка. Обычно каждое окно заключается в небольшую рамку. Функции рамки сводятся не только к отделению окна от остальных частей экрана. Рамка является и средством масштабирования. Размер окна приложения можно изменить, если поместить указатель мыши на рамку и перетащить его, удерживая нажатой левую кнопку мыши.

Строка заголовка. Имя приложения, которому принадлежит открытое окно, отображается в строке заголовка в верхней части окна. Строка заголовка является обязательным элементом всех окон приложений и позволяет пользователю легко определить, какому приложению принадлежит данное окно, если в системе запущено сразу несколько приложений. Строка заголовка активного окна выделяется альтернативным цветом, чтобы его легко можно было отличить от неактивных окон.

Значок приложения. Другим обязательным элементом любого окна является расположенный в его верхнем левом углу значок приложения. Этот значок обычно представляет собой маленький логотип приложения. Щелчок на значке приводит к открытию системного меню.

Системное меню. Системное меню открывается при щелчке мышью на значке приложения. В нем представлены стандартные команды управления окном: **Restore** (Восстановить), **Move** (Переместить), **Size** (Размер), **Minimize** (Свернуть), **Maximize** (Развернуть) и **Close** (Закрыть).

Кнопка свертывания. В правом верхнем углу большинства окон приложений имеется три кнопки. Крайняя левая кнопка предназначена для свертывания окна в кнопку на панели задач.

Кнопка развертывания/восстановления. Средняя кнопка в правом верхнем углу либо разворачивает окно на весь экран, либо восстанавливает прежние его размеры, если окно уже развернуто.

Кнопка закрытия. Крайняя правая кнопка в углу предназначена для закрытия приложения. После закрытия окна активным автоматически становится окно следующего приложения.

Вертикальная полоса прокрутки. В некоторых случаях окно приложения может содержать вертикальную полосу прокрутки, которая располагается по правому краю окна. В верхней и нижней частях полосы находятся кнопки со стрелками, направленными соответственно вверх и

вниз. Вдоль самой полосы располагается бегунок. Положение бегунка показывает, какая часть окна или документа отображается в данный момент на экране. Перемещая бегунок, можно выбрать нужную часть многостраничного документа. Щелчок мышью на кнопке со стрелкой приведет к смещению содержимого окна на одну строку вверх или вниз, а щелчок на свободном пространстве выше или ниже бегунка – на одну экранную страницу вверх или вниз.

Строка меню. В окнах большинства приложений сразу под строкой заголовка находится строка меню, содержащая наборы команд и опций программы. Обычно для выбора команд меню используется мышь, хотя эти действия можно выполнить и с помощью клавиатуры. Каждому элементу меню, как правило, соответствует клавиша быстрого вызова, выделенная подчеркиванием в названии элемента. Чтобы выбрать данный элемент, нужно нажать соответствующую клавишу в сочетании с клавишей [Alt]. Так, комбинация клавиш [Alt+F] открывает меню **File.**

Рабочая область. Рабочая область обычно занимает большую часть окна. Именно в эту область программа выводит результаты своей работы.

3.5.3. Классы окон

Чтобы два окна одной и той же программы выглядели и работали совершенно одинаково, они оба должны базироваться на общем классе окна. В приложениях, написанных на С, где используются традиционные вызовы функций, класс окна регистрируется программой в процессе инициализации. При создании окна класс указывается в качестве параметра функции CreateWindow(). Зарегистрированный новый класс становится доступным для всех программ, запущенных в данный момент в системе. Благодаря тому, что окна приложения создаются на основе общего базового класса, значительно сокращается объем информации, которую при этом следует указывать. Поскольку класс окна содержит в

себе описания элементов, общих для всех окон данного класса, нет необходимости повторять эти описания при создании каждого нового окна. К тому же все окна одного класса используют одну оконную процедуру, что также позволяет избежать дублирования кода.

3.6. Принципы обработки сообщений в приложении

Основным компонентом любого приложения Windows является цикл обработки сообщений. Работа приложения организуется следующим образом. Приложение сначала создает и открывает свои окна, затем запускается цикл обработки сообщений и, в конце концов, при получении сообщения типа WM_CLOSE работа программы завершается. Особенность этого процесса состоит в том, что приложение должно быть *полностью ориентировано на прием и обработку сообщений.* Программа должна быть готова в любой момент принять сообщение, определить его тип, выполнить соответствующую обработку и вновь перейти в режим ожидания до поступления следующего сообщения.

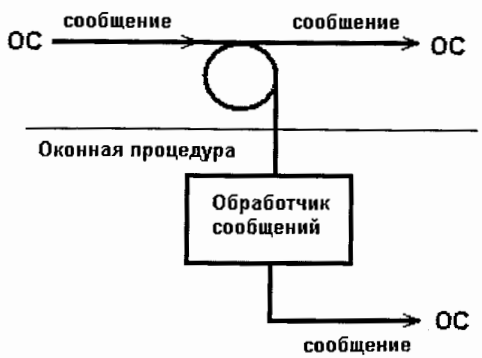

Рис.5. Цикл обработки сообщений в Windows-приложении

Типичное приложение Windows строится на базе каркаса (рис.5), содержащего *цикл обработки сообщений.* Цикл обработки сообщений отвечает за передачу поступающих от Windows сообщений соответствующим оконным процедурам. В этом цикле выполняются прием сообщений

и передача их в соответствующие *функции-обработчики сообщений.* Сообщения используются для информирования приложения о том, что в системе произошло то или иное событие. Активизируя с помощью мыши или клавиатуры то или иное окно, пользователь дает системе понять, что последующие команды и данные следует направлять именно этому окну. Приложение может обрабатывать не все сообщения, а только некоторые. Необработанные сообщения "по умолчанию" передаются обработчику сообщений в ОС Windows.

Приложения Windows, написанные на C/C++, содержат *специальные процедуры обработки* для всех типов сообщений, которые могут быть посланы программе. Разные окна программы могут по-разному реагировать на одни и те же сообщения. Это достигается за счет того, что обработчики сообщений пишутся отдельно для каждого окна, но Windows знает, какому именно окну адресовано сообщение. Таким образом, приложения содержат процедуры обработки не только сообщений разных типов, но и сообщений для разных окон. Например, ***сообщения для кнопки***:

WM_LBUTTONDOWN	*– была нажата левая кнопка мыши*
WM_PAINT	*– требуется перерисовать кнопку OK, т. к. теперь она нажата*
WM_LBUTTONUP	*– левая кнопка мыши была отпущена*
WM_PAINT	*– перерисовать кнопку OK, т. к. теперь она отпущена*

или ***сообщения для окна***:

WM_WINDOWPOSCHANGED	*– положение окна на экране только что было изменено*
WM_ACTIVATE	*– была активизирована клиентская область окна*
WM_KILLFOCUS	*– у окна будет отключен фокус ввода*
WM_DESTROY	*– окно уничтожается*

3.7. Вызовы функций Windows

Приложения Windows существенно отличаются от приложений, написанных для MS-DOS. Ни одно приложение в Windows не отображает свои данные непосредственно на экране. Точно так же ни одно приложение не обрабатывает напрямую прерывания устройств и не выводит данные непосредственно на печать. Вместо этого приложение вызывает встроенные функции Windows и ожидает от системы соответствующих сообщений. Приложение обращается к ОС Windows при помощи системных вызовов или "функций". Они составляют интерфейс прикладного программирования API. Функции API располагаются в системных динамических библиотеках (DLL) операционной системы Windows.

Существуют функции для выделения памяти, управления процессами, окнами, файлами, для рисования графических примитивов и др. Обращение к функциям API из большинства сред разработки на C++ осуществляется очень просто, так как API специально спроектирован для использования в среде C/C++. Для этого в текст программы надо включить заголовочный файл <windows.h>, который содержит описания функций API и в процессе компоновки подключает необходимые библиотеки, и затем в программу можно включать любые обращения к функциям API.

"Базовый" набор системных вызовов или функций API в Windows можно разделить на четыре группы:

- *Функции модуля KERNEL.DLL* – обеспечивают управление потоками, процессами, ресурсами, файлами и памятью, выделение памяти для ресурсов, загрузку ресурсов (с диска или из исполняемого файла приложения), удаление ресурсов из памяти;
- *Функции модуля USER.DLL* – обеспечивают работу с пользовательским интерфейсом:

a) *управления окнами* – позволяют изменять размеры, местоположение на экране и внешний вид окна, заменять оконную процедуру, разрешать и запрещать работу окна, получать информацию о свойствах окна;

b) *работа с меню* – создание, отображение, изменение строки меню;

c) *управление формой и положением указателя мыши и текстового курсора*;

d) *работа с буфером обмена*;

e) *управление сообщениями и очередью сообщений потока*. В модуле USER есть функции для управления дочерними окнами программ с многодокументным интерфейсом (MDI).

- *Функции модуля GDI.DLL* – обеспечивают процесс рисования графических примитивов аппаратно-независимым способом. Для абстрагирования от конкретного устройства введено понятие *контекста устройства*. Контекст устройства – это системный объект, обеспечивающий интерфейс с конкретным графическим устройством. С помощью контекста можно выполнять графический вывод на устройство или получать информацию о его свойствах (имя устройства, разрешение, цветовые возможности и т. д.).

При рисовании примитивов (отрезков, прямоугольников, эллипсов, текста) функциям рисования передается *дескриптор контекста устройства*. Контекст устройства преобразует общие, независимые от устройства, графические команды в набор команд конкретного устройства. Например, когда приложение вызывает из GDI функцию «Ellipse», то контекст устройства сначала определяет, какой драйвер будет выполнять это действие. Затем выбранный драйвер может передать вызов аппаратному ускорителю (если он есть в видеоадаптере) или нарисовать эллипс по точкам.

Контексты GDI представляют широкий спектр устройств. Типичными такими контекстами являются *дисплейный контекст устройства* (для выполнения вывода непосредственно на экран компьютера) и *принтерный контекст устройства* (при выводе в этот контекст вызовы приложения преобразуются в управляющие коды принтера и посылаются на принтер). Рисование в контексте устройства обычно производится в *логических координатах* посредством аппаратно-независимых физических единиц. Например, при задании прямоугольника можно указать его размеры в сантиметрах, а контекст GDI автоматически выполнит преобразование логических координат в физические координаты устройства.

В качестве параметров преобразования приложение может задать начало координат и размер области вывода в логических и физических координатах. От положения начала координат в обеих системах зависит горизонтальное и вертикальное смещение примитивов в области вывода. Размер области вывода определяет ориентацию и масштаб примитивов после преобразования.

При рисовании примитивов часто указываются их свойства: цвет, толщина и стиль линии. Для этого служат объекты GDI: перья, кисти, шрифты, растровые изображения и палитры. Их можно создавать и выбирать в контексте устройства для того, чтобы рисуемые впоследствии примитивы выглядели нужным образом.

- *Дополнительные API.* Windows содержит также большое количество вспомогательных API. Есть API для работы с электронной почтой (MAPI), модемами (TAPI), базами данных (ODBC), DirectX API – для некоторых приложений (в особенности, игр), который ускоряет доступ к аппаратуре. В ОС Windows есть специальные API для работы с сетями, например: WinSock (библиотека сокетов), RAS (Remote Access Service – сервис удаленного доступа) и RPC (библиотека вызова удаленных процедур) и др.

3.8. Файл WINDOWS.H

Файл заголовков WINDOWS.H открывает доступ к тысячам описаний, констант, структур, типов данных и прототипов функций. Он включается в большинство приложений Windows и содержит директивы включения множества других файлов заголовков. Это одна из причин, почему приложения Windows компилируются так долго.

3.9. Структура и этапы разработки Windows-приложения

3.9.1. Структура приложения

Приложения Windows содержат два общих элемента: функцию winMain() и оконную процедуру. Функция winMain() составляет основу любого приложения. Она служит как бы точкой входа в приложение и выполняет ту же роль, что и функция main() в обычных программах на C++. Оконная процедура в приложении выполняет роль диспетчера (обработчика) сообщений. Приложение никогда не получает к ней прямого доступа. Для этого им нужно сделать запрос к Windows на выполнение соответствующего действия. Поэтому все оконные процедуры *являются функциями обратного вызова.* Подобная функция регистрируется в Windows и вызывается всякий раз, когда выполняется некоторое действие над окном.

Структура Windows-приложения:

Функция WinMain();

- структура WNDCLASS;
- определение класса окна;
- создание окна;
- отображение и обновление окна;
- цикл обработки сообщений;

Оконная процедура;

- обработка кодов сообщения с префиксом WM_: типа WM _ PAINT, WM_SIZE, WM_COMMAND и т. д.;

-обработка сообщения WM _ DESTROY;

-функция DefWindowProc().

3.9.2. Этапы разработки Windows-приложения

Разработку приложения можно разбить на следующие этапы:

- создание функции WinMain() и других базовых функций;
- создание меню и других ресурсов, включение их в файл ресурсов;
- создание с помощью редактора ресурсов уникальных указателей мыши, значков и других растровых изображений (не обязательный);
- создание необходимых диалоговых окон;
- компиляция проекта.

3.10. Основные компоненты приложения

3.10.1. Функция WinMain()

Функция winMain() является обязательным компонентом любого приложения Windows. С нее начинается выполнение программы и ею же обычно заканчивается. Функция WinMain() отвечает за следующее:

- создание и инициализацию цикла обработки сообщений (который имеет доступ к очереди сообщений приложения);
- начальную инициализацию приложения;
- регистрацию класса окна приложения;
- завершение выполнения программы (обычно в результате получения сообщения WM_QUIT).

В функцию WinMain() передаются четыре параметра:

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPreInst, LPSTR IpszCmdLine, int nCmdShow)

Первый параметр – hInst – представляет собой дескриптор экземпляра приложения. Второй – hPreInst – всегда содержит значение NULL, что указывает на отсутствие других, запущенных ранее, экземпляров приложения. Третий параметр – IpszCmdLine – служит указателем на строку,

завершающуюся нулевым символом и содержащую командную строку программы. Последний параметр – nCmdShow – задает одну из возможных констант, определяющих варианты отображения окна, например, SW_SHOWNORMAL, SWSHOWMAXIMIZED или SWSHOWMINIMIZED.

а) Структура WNDCLASS

Одна из задач функции WinMain() заключается в регистрации класса окна приложения. Класс окна содержит комбинацию выбранных пользователем свойств окна, дескрипторы значка и указателя мыши, а также прочие атрибуты и выступает в роли шаблона при создании всех экземпляров окон приложения. Чтобы определить класс окна, приложение должно объявить переменную wc, типа WNDCLASS, которая впоследствии заполняется информацией о классе окна.

Структура WNDCLASS описывается комбинацией различных полей следующего вида:

```
typedef struct _WNDCLASS
        {
          INT           style
          WNDPROC       lpfnWndProc
          int           cbClsExtra
          int           cbWndExtra
          HANDLE        hInstance
          HICON         hIcon
          HCURSOR       hCursor
          HBRUSH        hbrBackground
          LPCTSTR       lpszMenuName
          LPCTSTR       lpszClassName
        } WNDCLASS
```

***Поля структуры* WNDCLASS**

Особенно важны первые две переменные структуры WNDCLASS – style и lpfnWndProc. Большая часть свойств, делающих окно уникальным и сложным объектом, управляется именно этими переменными. В style

хранится стиль оконного класса, а в lpfnWndProc адрес оконной процедуры, которая будет обрабатывать все сообщения, посылаемые окну.

- *Style.* Стиль и свойства окна задаются в поле wc.style при помощи набора констант с префиксом CS_, которые могут объединяться с помощью побитовой операции ИЛИ, например: wc.style= CS_HREDRAW|CS_VREDRAW, где константы указывают Windows, что рабочую область окна нужно перерисовывать всякий раз, когда размер окна изменяется по вертикали или горизонтали. Возможные свойства поля style перечислены в табл.3.

Таблица 3. Свойства полей *style*

Свойство	Описание
CS_ BYTEALIGNCLIENT	Задает выравнивание по границе байта размера рабочей области по горизонтали
CS_ YTEALIGNWINDOW	Задает выравнивание по границе байта размера окна по горизонтали
CS_CLASSDC	Указывает, что все окна данного класса должны использовать один и тот же контекст устройства
CS_ DBLCLKS	При установке этого флага окну будут посылаться сообщения о двойном щелчке
CS_ HREDRAW	Задает операцию перерисовки окна при изменении горизонтального размера
CS_ NOCLOSE	Делает неактивной команду close в системном меню
CS_PARENTDC	Указывает, что область отсечения дочернего окна будет равна области отсечения родительского окна
CS_SAVEBITS	Служит указанием системе сохранять часть окна, перекрываемую другим окном, чтобы впоследствии не посылать перекрытому окну сообщения WM_PAINT, а восстанавливать его содержимое самостоятельно
CS_ VREDRAW	Задает операцию перерисовки окна при изменении вертикального размера

- *IpfnWndProc* – содержит указатель на оконную процедуру, которая будет обрабатывать все сообщения, посылаемые окну.
- *cbClsExtra* – содержит информацию о количестве байтов, выделяемых системой после создания структуры класса окна. Значение этого параметра может быть равным NULL.
- *cbWndExtra* – содержит информацию о количестве байтов, выделяемых системой после создания экземпляра окна. Этот параметр может быть равным NULL.
- *Hlnstance* – содержит дескриптор экземпляра приложения, регистрирующего класс окна.
- *Hlcon* – содержит дескриптор значка приложения.
- *hCursor* – содержит дескриптор указателя мыши, используемого в окне приложения. Этот параметр может быть равным NULL.
- *hbrBackground* – определяет кисть, используемую для заливки фона окна и может содержать дескриптор реальной кисти или одну из следующих констант: color_activeborder; color_activecaption; color_background; color_btntext; color_btnshadow; color_captiontext; color_highlighttext; color_highlight; color_inactiveborder; color_inactivecaption; color_menu; color_menutext; color_scrollbar; color_window; color_windowframe; color_windowtext. Если значение параметра hbrBackground равно NULL, то приложение должно самостоятельно управлять заливкой фона окна при получении сообщения WM_ERASEBKGND.
- *lpszMemiName* – содержит указатель на строку, завершающуюся нулевым символом и представляющую собой имя ресурса меню. Этот параметр может быть равным NULL.
- *lpszClassName* – содержит указатель на строку, завершающуюся нулевым символом и представляющую собой имя класса окна.

В Windows-приложении класс окна определяется структурой и атрибутами полей этой структуры. Можно в приложении создать собственный класс окна, описав для него структуру соответствующего типа и заполнив поля структуры атрибутами разрабатываемого приложения. Следующий фрагмент файла swa.cpp демонстрирует, как можно создать и проинициализировать структуру WNDCLASS:

```
// эта строка используется при регистрации оконного класса;
char ProgName[] = "ProgName";
char Title[] =    "Вывод текста в оконное приложение Windows ";
HWND          hWnd;       // Описатель окна
MSG           msg;        // Структура, содержащая описание сообщений
// Структура атрибутов класса окна
WNDCLASS      wc;
// Поля структуры:
wc.lpszClassName  = ProgName;   // Имя класса окна
wc.hInstance      = hInst;      // Описатель экземпляра приложения
wc.lpfnWndProc    = WndProg;    // Указатель на оконную процедуру
wc.hCursor    = LoadCursor(NULL, IDC_ARROW);  // Описатель курсора
wc.hIcon          = NULL;       // Описатель значка
wc.lpszMenuName   = NULL;       // Имя ресурса меню
// Описатель кисти для фона окна
wc.hbrBackground  = (HBRUSH)GetStockObject(WHITE_BRUSH);
// Набор битов, описывающих стиль окна
wc.style      = CS_HREDRAW | CS_VREDRAW;
wc.cbClsExtra = 0;  //Количество дополнительных байтов после структуры
wc.cbWndExtra = 0;  // ..... дополнительных байтов после экземпляра  окна
//  Регистрация созданного оконного класса:
    if (!RegisterClass (&wc))
    return FALSE;           // Уход в случае неудачи
```

Имя программы хранится в переменной ProgName и записывается в поле wc.IpszClassName. Второму полю структуры WNDCLASS, называемому wc.hinstance, передается аргумент hlnst функции winMain(), хранящий дескриптор текущего экземпляра приложения. Полю wc.lpfnWndProc

присваивается указатель на оконную процедуру, которая будет обрабатывать все сообщения, посылаемые окну. В файле swa.cpp эта функция называется WndProg(). Имя WndProg() задается программистом и является стандартным. Прототип функции WndProg() *должен быть описан до выполнения операции присваивания.*

В поле wc.hCursor хранится дескриптор указателя мыши текущего экземпляра приложения. В рассматриваемом примере создается указатель IDC_ARROW (соответствует обычному указателю мыши в виде наклоненной стрелки), дескриптор которого возвращается функцией LoadCursor(). Поскольку в приложении нет устанавливаемого по умолчанию значка, полю wc.hIcon присваивается значение NULL.

Если поле wc.lpszMenuName содержит NULL, то Windows считает, что у приложения нет собственного меню. В противном случае должно быть указано название ресурса меню, взятое в кавычки.

Функция GetStockObject() возвращает дескриптор кисти, которая используется для заливки фона окна, созданного на базе данного класса. В нашем случае применяется стандартная белая кисть – WHITEBRUSH.

В поле wc.style свойства окна задаются как CS_HREDRAW в сочетании с CS_VREDRAW. Константы с префиксом CS_ определены в файле WINUSER.H и могут объединяться с помощью операции побитового ИЛИ (|). Константы CS_HREDRAW и CS_VREDRAW указывают Windows, что рабочую область окна нужно перерисовывать всякий раз, когда размер окна изменяется по вертикали или горизонтали.

Последние два поля, wcA.cbClsExtra и wc.cbWndExtra, обычно устанавливают равными 0. Их предназначение – указание дополнительного количества байтов памяти, которые должны быть зарезервированы после создания класса окна или экземпляра самого окна.

Затем в функцию RegisterClass передается указатель на только что созданную структуру класса окна. Если Windows не может зарегистрировать новый класс (скажем, в случае нехватки памяти), функция возвращает 0, вследствие чего выполнение программы завершается.

b) Создание окна

Регистрация класса окна еще не свидетельствует о создании самого окна. Окно создается в результате вызова функции CreateWindow(). В то время как класс окна определяет общие свойства всех окон данного семейства, параметры функции CreateWindow() задают описание конкретного экземпляра окна, например, его размеры, местоположение и внешний вид. Если функция CreateWindow() выполняется успешно, она возвращает дескриптор созданного окна, в противном случае – NULL.

Функция CreateWindow() вызывается следующим образом:

```
hWnd    = CreateWindow( ProgName,Title, WS_OVERLAPPEDWINDOW|
          WS_HSCROLL|WS_VSCROLL, 0, 0, CW_USEDEFAULT,
          CW_USEDEFAULT, NULL, NULL, hInst, NULL );
```

Первый параметр, ProgName, содержит название класса окна, чьи свойства наследует новое окно. Это может быть класс, зарегистрированный функцией RegisterClass, или один из стандартных оконных классов, например, классов элементов управления: BUTTON, SCROLLBAR, EDIT. За ним указывается содержимое строки заголовка окна – Title. Третий параметр задает стиль окна – WS_OVERLAPPEDWINDOW. Это стандартный стиль Windows, определяющий обычное масштабируемое окно со строкой заголовка, системным меню, кнопками свертывания, развертывания и закрытия, а также с рамкой.

Следующие шесть параметров (CW_USEDEFAULT или NULL) определяют координаты начальной точки окна, его размеры, а также дескрипторы родительского окна и меню. Всем этим параметрам в нашем

примере присвоены значения по умолчанию. Десятый параметр, hinst, содержит дескриптор экземпляра приложения. Так как окну не передаются никакие дополнительные значения, то последний параметр равен NULL.

c) Отображение и обновление окна

Для отображения окна на экране предназначена функция ShowWindow(hWnd, nCmdShow). В этой функции дескриптор окна hwnd уже создан функцией CreateWindow(), а параметр nCmdShow устанавливает режим начального отображения окна приложения. Так, например, задание константы SW_SHOWMINNOACTIVE в функции ShowWindow приведет к отображению окна в виде значка на панели задач, а константа SW_SHOWMAXIMIZED выполняет развертывание окна приложения на весь экран. Режим отображения SW_SHOWMINIMIZED аналогичен SW_SHOWMINNOACTIVE за исключением того, что окно остается активным (имеет фокус).

Последний этап выведения окна на экран реализует функция UpdateWindow(hWnd). Вызов функции ShowWindow() с установленным параметром SW_SHOWNORMAL приводит к заливке фона окна выбранной кистью, а функция UpdateWindow() генерирует сообщение WM_PAINT, указывающее на формирование содержимого рабочей области окна.

d) Цикл обработки сообщений

После отображения окна программа готова к выполнению своей непосредственной задачи обработке сообщений. В Windows информация приложению не передается напрямую, а в виде сообщений помещается в очередь. Цикл обработки сообщений в Windows-приложении организуется с помощью следующей конструкции:

```
while (GetMessage(&msg, NULL, 0, 0) ) {
    ➧ TranslateMessage(&msg);
    ➧ DispatchMessage(&msg); }
```

Функция *GetMessage* читает очередное сообщение из очереди и записывает его в структуру *msg*, адресуемую указателем &msg (структура *msg*, описана в разделе 4.3.1). Установка второго параметра в функции GetMessage равным NULL свидетельствует о том, что читаться должны сообщения, адресованные любому окну приложения. Два других параметра формируют фильтр сообщений, который ограничивает получаемые сообщения определенным диапазоном (например, только сообщения мыши). Если эти параметры равны нулю, как в нашем случае, то принимаются любые сообщения, адресованные данному приложению.

После запуска цикла только одно сообщение может остановить его выполнение. Как только в цикл передается сообщение WM_QUIT, функция GetMessage() возвращает значение FALSE, в результате чего цикл завершается, выполняются заключительные операции функции winMain() и работа приложения прекращается.

Функция *TranslateMessage*. Сообщения о нажатии виртуальных клавиш могут быть преобразованы в символьные сообщения с помощью функции TranslateMessage(), которая формирует сообщение WM_CHAR из пары WM_KEYDOWN/WM_KEYUP. Эта функция необходима только тем приложениям, которые обрабатывают ввод данных с клавиатуры. Важность функции заключается в том, что она позволяет пользователям выбирать команды меню путем нажатия клавиш, а не только щелчками мыши.

Функция *DispatchMessage*. Функция DispatchMessage() направляет полученное сообщение соответствующей оконной процедуре, автоматически определяя окно, которому адресовано сообщение.

3.10.2. Оконная процедура

Оконная процедура регистрируется в системе (wc.lpfnWndProc = WndProg() и вызывается всякий раз, когда Windows выполняет какую-либо

операцию над окном приложения. Структура простой оконной процедуры имеет следующий вид:

```
LONG WINAPI WndProg(HWND hWnd, UINT msg, WPARAM wParam,
LPARAM lParam) {
    PAINTSTRUCT  ps;
    HDC          hDC;
      switch(msg)                    // Переход по сообщению
        {
          case WM_PAINT:
            hDC = BeginPaint(hWnd,&ps);
            TextOut(hDC,100,90,"Разработка windows-приложений", 29);
            EndPaint(hWnd,&ps);
            break;
          case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
        return (DefWindowProc(hWnd,msg,wParam,lParam));
      }
 return(0);
}
```

Первым параметром этой процедуры WndProg() является дескриптор окна hWnd, которому посылается сообщение. Поскольку одна процедура может обрабатывать сообщения сразу нескольких окон в приложении, этот дескриптор позволяет определить, какому именно окну адресовано сообщение. Второй параметр содержит идентификатор сообщения. Два оставшихся параметра, wParam и lParam, несут дополнительную информацию, зависящую от конкретного типа сообщения.

В процедуре WndProg() определяются две локальные переменные: hDC, содержащая дескриптор контекста устройства (экрана) и ps, задающая структуру PAINTSTRUCT. Эта структура необходима для

хранения информации, отображаемой в рабочей области окна. Обработка самих сообщений осуществляется с помощью стандартной конструкции *switch*(), которая может быть довольно большой (размер ее зависит от числа обрабатываемых сообщений в оконной процедуре).

В Windows существует более двухсот различных сообщений, которые могут посылаться окнам. Все они имеют префикс WM_, например, WM_CREATE, WM_SIZE, WM_PAINT.

В рассматриваемом примере в оконной процедуре WndProg() обрабатываются только два сообщения WM_PAINT и WM_DESTROY. Сообщение WM_PAINT посылается окну приложения, когда его части нуждаются в перерисовке. Это сообщение также посылается, когда другое приложение или диалоговое окно может открыть свое окно поверх окна этого приложения, что делает закрытую область окна некорректной, нуждающейся в перерисовке. Тогда Windows посылает приложению сообщение WM_PAINT с координатами этой области и функцию UpdateWindow(), которая обновляет эту область. Сообщение WM_PAINT посылается также при вызове функции InvalidateRect() или InvalidateRgn(), когда происходит изменение размеров окна, и при вызове функции ScrollWindow().

При обработке сообщения WM_PAINT приложение вызывает функцию BeginPaint, которая подготавливает указанное окно к операции рисования, заполняет структуру PAINTSTRUCT данными об обновляемой области и возвращает дескриптор контекста устройства, связанного с этим окном. После всех необходимых операций рисования приложение вызывает функцию EndPaint для информирования Windows о том, что приложение закончило обработку и имеющийся контекст устройства можно удалить. Вызов функции invalidateRect() заставляет Windows пометить указанную область окна как недействительную и сгенерировать для нее сообщение WM_PAINT.

В нашем примере (файл swa.cpp) после вызова функции BeginPaint, с использованием функции TextOut(), выполняется вывод текстовой строки в рабочую область окна приложения. Структура этой функции имеет следующий вид:

TextOut (HDC hDC, int nXStart, int nYStart, LPCTSTR lpString, int cbString);

где hDC – идентификатор контекста устройства; int nXStart и int nYStart – координаты начала строки, lpString – указатель на строку, которая будет выведена; int cbString – количество символов в выводимой строке. Затем вызывается функция EndPaint, которая ставит Windows в известность, что приложение закончило обработку.

Когда пользователь выбирает в системном меню команду Close или нажимает кнопку закрытия окна, Windows посылает приложению сообщение WM_DESTROY. В ответ на это приложение завершает свою работу, вызывая функцию PostQuitMessage(), которая направляет в очередь сообщений приложения сообщение WM_QUIT, вследствие чего цикл обработки завершается.

Функция DefWindowProc() обычно вызывается в блоке default оператора switch. Она возвращает все нераспознанные или необработанные сообщения обратно системе, гарантируя таким образом, что все посылаемые приложению сообщения будут корректно обработаны.

В системе Windows определены понятия *дескриптор окна* – hWnd (handle window), *дескриптор контекста устройства* – hDC (handle device context) и *дескриптор объекта* (handle object). С помощью hWnd элементам окна передаются сообщения и данные, а с помощью hDC производятся графические операции с содержимым самого объекта.

Контекст устройства – это структура, которая определяет набор графических объектов, их свойств и способов воздействия на объект. Графические объекты включают перо (pen) для линии, кисть (brush) для

заполнения графического объекта, битовую плоскость (bitmap) для копирования или прокручивания частей экрана, палитру (palette) для определения набора доступных цветов, регион (region) для усечения областей и других действий.

К дескрипторам объекта (OBJ) относятся перья системы рисования (Pen – цвет пера, толщина и тип линии), кисть, заполняющая область (Brush – цвет кисти, стиль и вид шаблона заполнения) и т. д. Это независимый объект, и он может существовать отдельно от hWnd и hDC (эти же работают только в паре). Для осуществления каких-либо операций необходимо затребовать требуемый дескриптор. Например, для рисования графических объектов нам потребуется дескриптор hDC. По окончании работы с ним мы обязаны вернуть его системе, иначе по истечении времени Windows прекратит все операция рисования и просто зависнет. Общий принцип работы в контексте устройства можно свести к следующему:

- получить через hWnd дескриптор контекста устройства hDC;
- выбрать в контексте hDC нужный объект (pen, brush,..) и сохранить его предыдущее состояние;
- произвести необходимую операцию;
- вернуть старое содержимое контекста в систему Windows.

3.11. Пример разработки простого Windows-приложения

В данном пункте рассмотрим пример создания простой программы swa.cpp (Simple Windows Application), которая создает окно Windows и выводит в нем текстовую строку. На базе данного примера мы узнаем, как эту программу и связанные с ней файлы можно использовать в качестве шаблона оконного класса, то есть собственного оконного класса, который можно использовать при создании других приложений Windows. Четкое понимание структуры этого примера позволит сэкономить время при

создании новых программ в будущем, так как рассматриваемые компоненты составляют основу практически любого приложения.

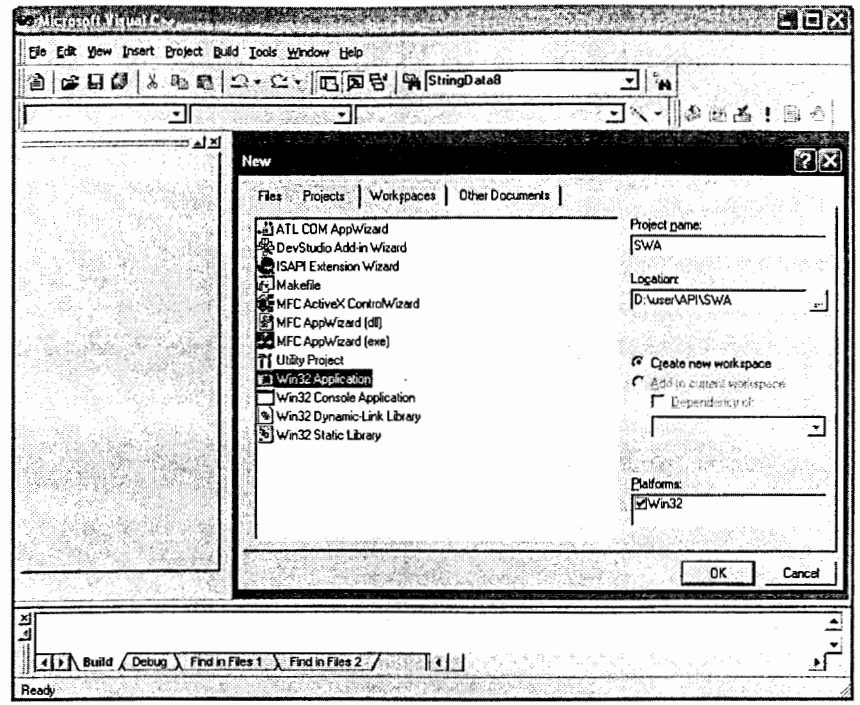

Рис.6. Создание в MS Visual C++ проекта SWA в среде Win32 Application

Создадим в MS Visual C++ новый проект SWA. Для этого в MS Visual C++ командой *File → new*, далее в окне *new → в Projects →Win32 Application* (рис.6) создаем новый проект с названием в текстовом поле *Project name→ SWA*, расположение которого задаем в текстовом поле *Location →d:\\User\API*. Мы создали проект SWA с файлом swa.dsw, который определяет параметры новой рабочей области и файл swa.dsp – с параметрами нового проекта. Затем в *Projects→Add To Projects → new →Files→ C++Source File* добавляем в наш проект новый файл с именем swa.cpp.

В созданный файл swa.cpp, с учетом информации, полученной в пунктах 4.3.1. и 4.3.2. помещаем код нашей первой программы:

```
// «Простое оконное Windows приложение»
#include <windows.h>
// Прототип оконной процедуры WndProg()
LONG WINAPI WndProg (HWND,UINT,WPARAM,LPARAM);
// Функция WinMain, с нее начинается выполнение Windows приложения
int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE, LPSTR, int
nCmdShow) {
char ProgName[] = "Используется при регистрации оконного класса";
char Title[]    = "Вывод текста в оконное приложение Windows ";
HWND            hWnd;  // Описатель окна
MSG             msg;   // Структура, содержащая описание сообщений
// Структура атрибутов класса окна
WNDCLASS        wc;
// Поля структуры:
wc.lpszClassName    = ProgName;  // Имя класса окна
wc.hInstance        = hInstance;  // Описатель экземпляра приложения
wc.lpfnWndProc      = WndProg;  // Указатель на оконную процедуру
wc.hCursor          = NULL;      // Описатель курсора
wc.hIcon            = LoadIcon(hInstance,ProgName); // Описатель значка
wc.lpszMenuName  = NULL;      // Имя ресурса меню
// Описатель кисти для фона окна
wc.hbrBackground  = (HBRUSH)GetStockObject(WHITE_BRUSH);
// Набор битов, описывающих стиль окна
wc.style            = CS_HREDRAW | CS_VREDRAW;
wc.cbClsExtra  = 0; // Количество дополнительных байт после  структуры
wc.cbWndExtra  = 0; // ..... дополнительных байт после экземпляра окна
// Попытка зарегистрировать оконный класс
            if (!RegisterClass (&wc))
                return FALSE;     // Уход в случае неудачи
// Создаем окно и запоминаем его описатель
hWnd  =  CreateWindow( ProgName,Title,   WS_OVERLAPPEDWINDOW|
```



```
                WS_HSCROLL|WS_VSCROLL,0,0,CW_USEDEFAULT,
                CW_USEDEFAULT,  NULL, NULL, hInstance, NULL );
// Показываем окно
      ShowWindow(hWnd,nCmdShow);
      UpdateWindow(hWnd);
// Цикл ожидания сообщений
      while (GetMessage(&msg,0,0,0)) {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
            }
      return (msg.wParam);
}
      // ======== Оконная процедура ========//
LONG WINAPI WndProg(HWND hWnd, UINT msg, WPARAM wParam,
LPARAM lParam) {
PAINTSTRUCT   ps;
HDC           hDC;
      switch(msg)    // Обработка сообщений (переход по сообщению)
         {
          case WM_PAINT:
              hDC = BeginPaint(hWnd,&ps);
            // Функция TextOut выводит строку с заданной позиции
              TextOut(hDC,100,90,"Разработка windows-приложений",29);
              EndPaint(hWnd,&ps);
              break;
         case WM_DESTROY:
        // Сообщаем системе, что поток требует закрытия
              PostQuitMessage(0);
              break;
        default:
              return (DefWindowProc(hWnd,msg,wParam,lParam));
         }
 return(0);
}
```

После получения исходного файла swa.cpp приступаем к созданию исполняемого файла swa.exe. Согласно терминологии, в VisualC++ этот процесс называется *построением программы*. Для этого в меню выбираем команды: *Build → Rebuild All*, которые компилируют и компонуют все файлы проекта. Различие между командами *Build* и *Rebuild All* состоит в том, что команда *Rebuild All* не проверяет дату создания используемых в проекте файлов, а компилирует и компонует все файлы проекта независимо от того, когда они были созданы. Затем запускаем созданную программу и на экран выведется окно нашего приложения (рис.7) с текстовой строкой.

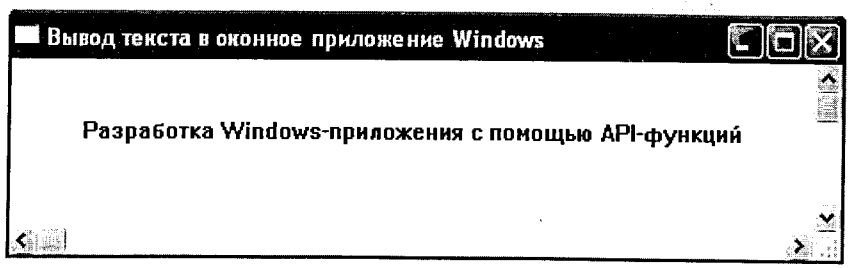

Рис.7. Окно Windows-приложения программы swa.cpp

Проекты для Windows, как правило, включают большое число файлов, но в нашем приложении мы создали проект с программой, состоящей из единственного файла swa.cpp. Этого вполне достаточно, чтобы разобраться с основными этапами создания полноценного Windows приложения на C/C++.

3.12. Создание «шаблона» класса окна для Windows API-приложений

3.12.1. Создание «шаблона» класса окна

На примере программы paint.cpp (которая создает окно и рисует в нем графические объекты) покажем, как созданный в программе swa.cpp оконный класс можно использовать в качестве шаблона для нашего приложения и что необходимо включить в программу paint.cpp для создания и отображения на экране окна графических объектов.

По аналогии с предыдущим примером создадим в MS Visual C++ новый проект PAINT. Для этого в MS Visual C++ командой *File → new*, далее в окне *new → в Projects →Win32 Application* (рис.6) создаем проект с названием *Project name*→PAINT и расположением →*d:\\User\API*. Затем в *Projects→Add to Projects → new →Files→ C++Source File* добавляем в проект paint.dsw новый файл с именем paint.cpp и копируем в него содержимое файла swa.cpp.

В отличие от проекта SWA этот проект, кроме вывода текстовой строки на экран, должен рисовать графические примитивы: отрезки линий, прямоугольник и эллипс. Рисование графических объектов в контексте устройства производится в *логических координатах* от начала координат окна, при этом указываются их свойства: цвет, толщина и стиль линии. Для этого служат объекты GDI: перья, кисти, шрифты, растровые изображения, палитры. Контекст устройства hDC преобразует общие, независимые от устройства графические команды в набор команд конкретного устройства. Например, при создании в нашем приложении объекта «эллипс» (при вызове из GDI функции *Ellipse*) контекст устройства сначала определяет, какой драйвер будет выполнять это действие, затем выбранный драйвер контекста устройства может передать вызов аппаратному ускорителю (если он есть в видеоадаптере) или нарисовать этот эллипс по точкам.

3.12.2. Создание графического объекта «линия»

При рисовании линий контекст устройства использует значение текущей позиции пера GetCurrentPosition(hDC,lpPoint), где параметр lpPoint указывает на структуру, куда будут записаны координаты текущей позиции. Чтобы нарисовать прямую линию, надо вызвать функцию LineTo(hDC, int X, int Y), которая рисует линию из текущей позиции до точки с координатами X, Y. Для изменения текущей позиции используется

функция MoveToEx(hDC, int X, int Y , NULL), где X, Y - координаты точки, в которую перемещается перо.

3.12.3 Создание графического объекта «прямоугольник»

Для рисования прямоугольника вызывают функцию Rectangle (hDC, int l, int t, int b, int h), где l, t, r, b – логические координаты соответственно левого верхнего и правого нижнего угла прямоугольника.

3.12.4 Создание графического объекта «эллипс»

Для рисования эллипса или окружности вызывают функцию Ellipse(hDC, int l, int t, int b, int h), где l, t, r, b – логические координаты соответственно левого верхнего и правого нижнего угла прямоугольника, в который вписывается эллипс или окружность.

Для вывода графических объектов в нашем приложении в функцию оконной процедуры WndProg() в структуру обработки сообщения WM_PAINT добавляем код создания графических примитивов в контексте устройства hDC:

```
.....
switch(msg)    // Обработка сообщений (переход по сообщению)
    case WM_PAINT:
       hDC = BeginPaint(hWnd,&ps);
     // вывод символьной строки с текущей или с заданной позиции
       TextOut(hDC,100,90,"Разработка windows-приложений",29);
       с помощью API-функций",51);
    //создание линий с заданной позиции
       MoveToEx(hDC, 10, 10, NULL);
       LineTo(hDC, 600,400);
       MoveToEx(hDC, 300, 10,NULL);
       LineTo(hDC, 50, 300);
    // создание прямоугольника
       Rectangle(hDC, 200,500,500,400);
     // создание эллипса
       Ellipse(hDC, 550, 500, 650, 400);
```



```
       EndPaint(hWnd,&ps);
       break;
.....
```

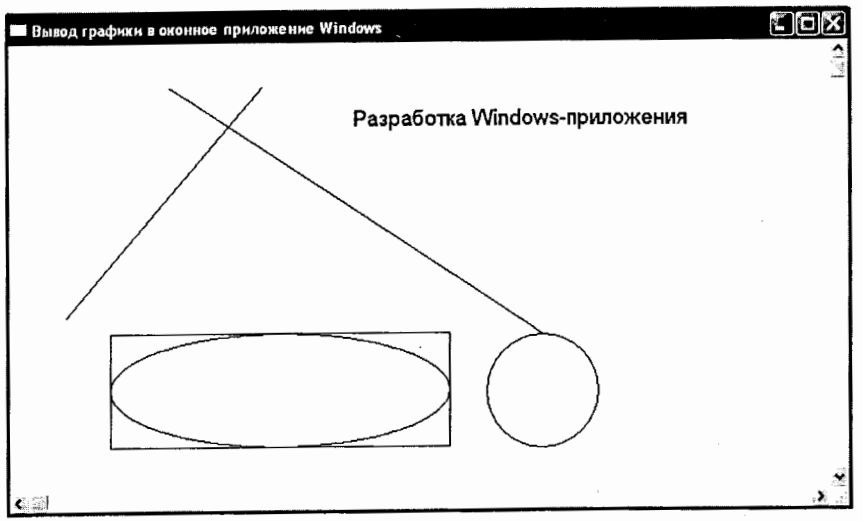

Рис.8. Окно Windows-приложения проекта программы PAINT

После получения кода исходного файла paint.cpp мы создаем исполняемый файл paint.exe. Для этого в меню выбираем команды *Build → Rebuild All*, которые компилируют и компонуют все файлы проекта. Затем запускаем нашу программу, и на экран выведется окно созданного приложения (рис.8).

При создании Windows-приложения PAINT мы использовали код программы swa.cpp, в котором изменили содержимое только оконной процедуры WndProg(), а структуру оконного класса мы оставили без изменения. Следовательно, мы использовали созданный в программе swa.cpp оконный класс в качестве «шаблона» для нашего приложения. Поэтому этот оконный класс можно сохранить отдельно в библиотечном файле, что позволит использовать его в других различных программах Windows-приложений. Это существенно сэкономит время при создании новых программ Windows-приложений.

3.12.5. Создание библиотечного файла для нашего оконного класса

Созданный в пункте 3.11 оконный класс можно сохранить в виде библиотечного файла и использовать при разработке различных Windows-приложений. Для этого создадим два файла: заголовочный файл *win.h* и Си-файл *win.cpp*, в которых будет содержаться структура нашего оконного класса.

Описание файла win.h:

```
#include <windows.h>      // Подключаем библиотеку описания функций API
  class WinClass          // объявляем класс окна WinClass
  {
   public:
     WinClass(LPCTSTR windowName, HINSTANCE hInst, int cmdShow,
     LRESULT (WINAPI *pWndProc) (HWND,UINT, WPARAM, LPARAM),
     LPCTSTR menuName = NULL,int x = CW_USEDEFAULT, int y = 0,
     int width = CW_USEDEFAULT, int height = 0,  UINT classStyle =
     CS_HREDRAW | CS_VREDRAW, DWORD windowStyle =
     WS_OVERLAPPEDWINDOW, HWND hParent = NULL);
     HWND GetHWnd() {
                        return hWnd;
                        }
  protected:
// объявляем переменную дескриптора контекста устройства  hwnd
       HWND hWnd;
// объявляем переменную типа: WNDCLASS  wc для регистрации класса
       WNDCLASSEX wc;
  };
```

Описание файла win.cpp:

```
#include "win.h"          // Подключаем заголовочный файл "win.h"
WinClass::WinClass (LPCTSTR windowName, HINSTANCE hInstance, int
cmdShow, LRESULT(WINAPI*WndProg)(HWND,UINT,WPARAM,LPARAM),
LPCTSTR menuName, int x, int y, int width, int height,UINT classStyle,
DWORD windowStyle, HWND hParent)
```



```
{
char ProgName[]   = "используется при регистрации оконного класса";
char Title[]      = "Вывод текста в оконное приложение Windows ";
wc.lpszClassName  =  ProgName;
wc.hInstance      =   hInstance;
wc.lpfnWndProc    =   WndProg;
wc.hCursor        =   LoadCursor(NULL, IDC_ARROW);
wc.hIcon          =   LoadIcon(NULL, IDI_APPLICATION);
wc.lpszMenuName   =   menuName;
wc.hbrBackground  =   (HBRUSH)GetStockObject(WHITE_BRUSH);
wc.cbSize         =   sizeof(wc);
wc.style          =   classStyle;
wc.style          =   CS_HREDRAW | CS_VREDRAW;
wc.cbClsExtra     =   0;
wc.cbWndExtra     =   0;
wc.hIconSm        =   LoadIcon(hInstance,ProgName);
     if (!RegisterClassEx(&wc))
        {
        char msg[100] = "Cannot register class: ";
        strcat(msg, ProgName);
        MessageBox(NULL, msg, "Error", MB_OK);
       return;
     }
hWnd= CreateWindow(ProgName,windowName,WS_OVERLAPPEDWINDOW|
WS_HSCROLL|WS_VSCROLL,x,y,CW_USEDEFAULT,CW_USEDEFAULT,
hParent, (HMENU)NULL, hInstance, NULL);
  if (!hWnd) {
             char text[100] = "Cannot create window: ";
             strcat(text, windowName);
             MessageBox(NULL, text, "Error", MB_OK);
             return;
             }
  ShowWindow(hWnd, cmdShow);
}
```

Теперь нам осталось в файле paint.cpp удалить код программы, описывающий структуру окна, подключить библиотечный оконный файл win.h и переписать структуру функции main():

```
#include <windows.h>
#include "win.h"
 LONG WINAPI WndProg (HWND,UINT,WPARAM,LPARAM);
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)
{
MSG msg;
➡ WinClass    mainWnd("Графика", hInstance, nCmdShow, WndProg, NULL,
10, 10, 400, 300);
       while (GetMessage(&msg, NULL, 0, 0))
       {
             TranslateMessage(&msg);
             DispatchMessage(&msg);
             }
             return msg.wParam;
}
       // ======== оконная процедура ========//
LONG WINAPI WndProg(HWND hWnd, UINT msg, WPARAM wParam,
LPARAM lParam) {
PAINTSTRUCT    ps;
HDC                hDC;
      switch(msg)
         {
           case WM_PAINT:
             hDC = BeginPaint(hWnd,&ps);
            // вывод текста  с заданной позиции
             TextOut(hDC,100,90," Разработка Windows-приложения",28);
            // создание линий с заданной позиции
              MoveToEx(hDC, 10, 10, NULL);   LineTo(hDC, 600,400);
              MoveToEx(hDC, 300, 10,NULL);  LineTo(hDC, 50, 300);
```



```
            // создание прямоугольника
              Rectangle(hDC, 200,500,500,400);
            // создание эллипса
              Ellipse(hDC, 550, 500, 650, 400);
              EndPaint(hWnd,&ps);
              break;
        case WM_DESTROY:
             PostQuitMessage(0);
             break;
       default:
             return (DefWindowProc(hWnd,msg,wParam,lParam));
      }
     return(0);
}
```

3.13. Компоненты графических элементов в Win API-приложении

3.13.1. Создание графических объектов по координатам, вводимым с помощью мыши

В среде Visual C++ мы научимся разрабатывать программы Windows-приложений, содержащие основные типы окон (родительские, дочерние, диалоговые), различные компоненты ресурсов (меню, пиктограммы, курсоры, клавиши и т. п.), реализующие операции по выводу текста и графических элементов с помощью функций GDI. В этом разделе разберем основные процедуры и методы создания Windows-приложений с обработкой сообщений от мыши.

Обработка сообщений от кнопок мыши

Чтобы создать графический объект по координатам, вводимым с помощью мыши, необходимо написать специальный программный код, обрабатывающий поступающие сообщения от мыши. Если кнопка мыши *нажимается* или *отпускается* внутри клиентской области окна, то

функция оконной процедуры получает сообщения: WM_LBUTTONDOWN или WM_LBUTTONUP. Если же мышь перемещается, то оконная процедура получает сообщение WM_MOUSEMOVE.

В нашем приложении MOUSE при нажатии на левую клавишу мыши оконная процедура получает сообщение

```
case WM_LBUTTONDOWN:
    strcpy(Text,"нажата: левая клавиша");
    InvalidateRect();
```

где в программном коде по его обработке при помощи функции strcpy() в переменную *Text* типа *string* копируется константная строка "нажата левая клавиша", которая затем выводится на экран при обработке сообщения WM_PAINT:

```
case WM_PAINT:
   {
     PAINTSTRUCT  ps;
     HDC          hDC;
     hDC = BeginPaint(hWnd,&ps);
     TextOut (hDC,300,500,Text,23);
     EndPaint(hWnd,&ps);
    }
```

Функция InvalidateRect(), вызывая функцию UpdateWindow(hWnd), перерисовывает содержимое клиентской области для вывода сообщения от кнопки. Ниже приведен код оконной процедуры программы *mouse.cpp* по обработке сообщений от кнопок мыши:

```
LONG WINAPI WndProg(HWND hWnd, UINT msg, WPARAM wParam,
LPARAM lParam)
PAINTSTRUCT  ps;
HDC          hDC;
    switch(msg) {
        case WM_PAINT:
            {
```



```
            hDC = BeginPaint(hWnd,&ps);
            TextOut (hDC,300,500,Text,23);
            EndPaint(hWnd,&ps); return 0;
            }
        case WM_LBUTTONDOWN:
            {
            strcpy(Text,"нажата: левая клавиша ");
            InvalidateRect(hWnd,0,TRUE);  return 0;
            }
        case WM_RBUTTONDOWN:
            {
            strcpy(Text,"нажата:  правая клавиша");
            InvalidateRect(hWnd,0,TRUE);  return 0;
            }
}
```

3.13.2. Создание эластичных графических объектов с помощью мыши

При создании эластичных графических объектов с помощью мыши необходимо после получения сообщения WM_MOUSEMOVE стереть предшествующее изображение объекта, а затем, получив новые координаты позиции мыши, перерисовать его изображение, используя для этого полученные координаты.

Для стирания текущего изображения объекта в контексте устройства устанавливается режим рисования R2_NOTXORPEN. В этом режиме повторный вызов функции вывода объекта приводит к восстановлению состояния клиентской области окна, которое было перед первым его выводом.

Чтобы рисуемые объекты были прозрачными (т. е. не закрашенными), в начале программы при обработке сообщения WM_CREATE в контексте устройства выбирается пустая кисть (HOLLOW_BRUSH).

В приложении ELASTICRECT для создания эластичных графических объектов выбирается дескриптор объекта дисплея hDC со спецификатором

static. Приложение получает требуемый контекст устройства при помощи функции *GetDC* в блоке обработки сообщения WM_CREATE, а освобождает его с помощью функции *ReleaseDC* в блоке обработки сообщения WM_DESTROY:

```
case WM_CREATE:
    ➡ hDC = GetDC(hWnd);
.....
case WM_DESTROY:
    ➡ ReleaseDC(hWnd, hDC);
```

Это позволяет обновлять клиентскую область непосредственно в ответ на пользовательский ввод информации с помощью мыши, *не прибегая к механизмам, генерирующим сообщение* WM_PAINT.

Статический класс памяти для переменной hDC здесь необходим потому, что требуется сохранять значение атрибута «текущая позиция пера», используемого и модифицируемого функцией LineTo, между двумя последовательными вызовами функции WndProg.

Процесс создания эластичных прямоугольников начинается с нажатия левой кнопки мыши. Программа обработки сообщения WM_LBUTTONDOWN устанавливает флаг слежения за мышью *bTracking* в значение *TRUE* и начальную позицию пера x1 = x2 = LOWORD(lParam), y1 = y2 = HIWORD(lParam). Для того чтобы прямоугольники рисовались различным цветом, задаем цвет пера HPEN hPen =CreatePen (PS_SOLID, 3,color), где long color = RGB(red,green,blue), а цветовую гамму выбираем случайным образом → BYTE red= rand()%256; BYTE green= rand()%256; BYTE blue= rand()%256. При обработке сообщения WM_MOUSEMOVE программа рисует прямоугольник Rectangle(hDC, x1, y1, x2, y2), используя получаемые текущие позиции курсора мыши: x2=LOWORD(lParam), y2= HIWORD(lParam). Когда пользователь отпускает кнопку мыши, флаг *bTracking* сбрасывается в *FALSE* и рисование прекращается.

При обработке сообщения WM_LBUTTONUP программа, получив конечные позиции курсора: x2=LOWORD(lParam), y2= HIWORD(lParam), возвращает растровую операцию по умолчанию R2_COPYPEN и рисует окончательный вид прямоугольника с пером (hOldPen).

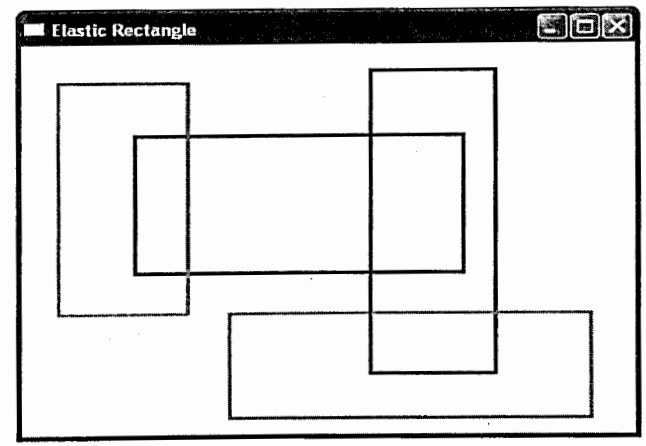

Рис.9. Окно Windows-приложения ELASTICRECT

Вид окна приложения ELASTICRECT показано на рис.9, а полный текст кода оконной процедуры приведен ниже:

// *ElasticRectangl.cpp*:

```
LRESULT CALLBACK WndProg(HWND hWnd, UINT msg, WPARAM
wParam, LPARAM lParam)
{
static HDC hDC; int i;
static  int x1, y1, x2, y2, N=12;
static BOOL bTracking = FALSE;
static HBRUSH hOld, hNew;
static HPEN hOldPen, hNewPen;
      switch (msg)
            {
        ➤ case WM_CREATE:
            ➡ hDC = GetDC(hWnd);
```

```
    hOld = (HBRUSH)SelectObject(hDC,
                GetStockObject(HOLLOW_BRUSH));
    break;
case WM_LBUTTONDOWN:
    bTracking = TRUE;
    SetROP2(hDC, R2_NOTXORPEN);
      for (i=0; i<N; i++)
          {
          BYTE red= rand ()%256; BYTE green= rand()%256;
          BYTE blue = rand()%256;
          long color = RGB(red,green,blue);
          HPEN hPen = CreatePen(PS_SOLID, 3,color);
          HPEN hOldPen = (HPEN) SelectObject(hDC, hPen);
          }
        x1 = x2 = LOWORD(lParam);
        y1 = y2 = HIWORD(lParam);
        break;
case WM_LBUTTONUP:
        if (bTracking)  {
          bTracking = FALSE;
        SetROP2(hDC, R2_COPYPEN);
        x2 = LOWORD(lParam); y2 = HIWORD(lParam);
          SelectObject(hDC, hOldPen);
        Rectangle(hDC, x1, y1, x2, y2);
        }
        break;
case WM_MOUSEMOVE:
        if (bTracking)  {
        Rectangle(hDC, x1, y1, x2, y2);
      // нарисовать новый прямоугольник
        x2 = LOWORD(lParam);  y2 = HIWORD(lParam);
        Rectangle(hDC, x1, y1, x2, y2);
        }
        break;
```



```
case WM_DESTROY:
            SelectObject(hDC, hOldPen);
          ReleaseDC(hWnd, hDC);
            PostQuitMessage(0);
            break;
    default:
        return DefWindowProc(hWnd, msg, wParam, lParam);
    }
    return 0;
}
```

Аналогично выполняется программа проекта LINE, которая рисует линии по координатам, вводимым с помощью мыши. Процесс рисования очередной линии начинается с нажатия левой кнопки мыши. В функции WM_LBUTTONDOWN устанавливаются начальные координаты линии MoveToEx(hDC, x, y, NULL), а текущие и конечные мы получаем в процессе обработки сообщения WM_MOUSEMOVE(). После отпускания кнопки мыши флаг *bTracking* сбрасывается в *FALSE* и процесс рисования прекращается. Ниже приведен фрагмент описания кода обрабатываемых функций для этой программы:

```
switch (msg)
    {
    case WM_PAINT:
      {
       hDC = BeginPaint(hWnd,&ps);
        MoveToEx(hDC, x1, y1, NULL); LineTo(hDC, x, y);
        EndPaint(hWnd,&ps);  return 0;
      }
    case WM_LBUTTONDOWN:
          bTracking = TRUE;
            x1 = LOWORD(lParam); y1= HIWORD(lParam);
            break;
```

```
➤ case WM_LBUTTONUP:
        if (bTracking)
            bTracking = FALSE;
            break;
➤ case WM_MOUSEMOVE:
        if (bTracking)
            {
            ➧ x = LOWORD(lParam); y = HIWORD(lParam);
            ➧ InvalidateRect(hWnd,0,TRUE);
            }
        break;
}
```

3.14. Диалоговые окна

3.14.1. Модальные диалоговые окна

В Windows-приложениях, кроме главного окна со строкой меню и специфическим для приложения содержимым, для взаимодействия между пользователем и программой применяются диалоговые окна. Диалоговое окно, как правило, содержит набор элементов управления, которые по типу являются дочерними окнами. С их помощью пользователь производит определенные управляющие воздействия на режим работы программы.

В Win32 API есть набор функций для создания отображения и управления содержимым диалогового окна. Внешний вид диалогового окна разрабатывается с помощью редактора ресурсов Visual C++.

Диалоговые окна Windows бывают трех типов: *модальные* (приложение приостанавливает работу на время активации диалогового окна); *системные модальные* (после запуска этого окна приостанавливают работу все приложения); *немодальные* (не блокируют работу приложений). Диалоговые окна 1-го и 2-го типов для завершения собственного цикла обработки сообщений в теле функций своего окна должны содержать

вызов функции EndDialog(). Немодальное диалоговое окно закрывается с помощью вызова функций DestroyWindow().

При отображении модального диалогового окна его *окно-владелец* приостанавливает работу приложения. Пользователь сможет продолжить работу с приложением только после завершения работы с модальным окном. Для создания и активизации модального окна предназначена функция *DialogBox*. Эта функция создает диалоговое окно по данным из ресурсного файла и выводит это окно на экран в модальном режиме. Приложение при обращении к DialogBox() передает ей адрес функции обратного вызова. Эта функция (процедура диалогового окна) является оконной процедурой. DialogBox() возвращает управление приложению только после завершения процедуры диалогового окна. Обычно это делается с помощью функции *EndDialog* при обработке какого-то сообщения от пользователя, например, по нажатии кнопки *OK*.

3.14.2. Немодальные диалоговые окна

При отображении немодального диалогового окна (в отличие от модальных окон) работа его *окна-владельца* не прерывается, т. е. приложение продолжает работать в обычном режиме. Но немодальное окно выводится поверх своего владельца, даже когда окно-владелец получает фокус ввода. Немодальные окна удобны для непрерывного отображения важной для пользователя информации.

Немодальное окно создается функцией *CreateDialog*. В Win32-API нет аналога функции DialogBox() для немодальных окон, следовательно, приложение самостоятельно должно обрабатывать диспетчеризацию и получение сообщений для немодальных окон. Большинство приложений делают это в своем главном цикле обработки сообщений с помощью функции *IsDialogMessage*. Эта функция проверяет, предназначено ли

сообщение заданному диалоговому окну, и при необходимости передает его в процедуру диалогового окна.

Немодальное окно не возвращает никакого значения своему владельцу, но при необходимости немодальное окно и его владелец могут взаимодействовать с помощью функции *SendMessage*. В процедуре немодального диалогового окна не надо вызывать функцию *EndDialog*. Такие окна уничтожаются вызовом функции *Destroy Windows*.

3.14.3. Описание процедуры диалогового окна

Диалоговые окна в среде Visual C++6 легко создаются с помощью редактора ресурсов. Изучение процедуры создания диалоговых окон, их включение в код программы приложения мы сделаем в разделе «создание приложений с вводом пользовательских данных в диалоговом окне». В этом пункте мы только рассмотрим создание кода процедуры *диалогового окна* на примере проекта DIALOG.

Создадим с помощью Visual C++ 6 новый проект DIALOG, используя опцию меню *Projects→Add To Projects → new →Files*. Добавляем в него два «пустых» файла dialog.cpp и Resource.rc и подключаем созданный оконный класс (добавляем в проект DIALOG файлы win.cpp и win.h). Затем копируем в наш файл ресурсов содержимое файла Resource.rc:

```
//  Resource.rc.
#include "resource.h"
#define APSTUDIO_READONLY_SYMBOLS
#include "afxres.h"
#undef APSTUDIO_READONLY_SYMBOLS
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_RUS)
#ifdef _WIN32
LANGUAGE LANG_RUSSIAN, SUBLANG_DEFAULT
#pragma code_page(1251)
#endif //_WIN32
```



```
// Dialog
IDD_DIALOG1 DIALOG DISCARDABLE  0, 0, 186, 90
STYLE DS_MODALFRAME|WS_POPUP|WS_CAPTION | WS_SYSMENU
CAPTION "Dialog"
FONT 8, "MS Sans Serif"
    BEGIN
        DEFPUSHBUTTON   "OK",IDOK,129,7,50,14
        PUSHBUTTON      "Cancel",IDCANCEL,129,24,50,14
        EDITTEXT        IDC_EDIT1,53,42,60,14,ES_AUTOHSCROLL
        LTEXT           "Data",IDC_STATIC,23,46,16,8
    END
#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
    BEGIN
        IDD_DIALOG1, DIALOG
            BEGIN
                LEFTMARGIN, 7
                RIGHTMARGIN, 179
                TOPMARGIN, 7
                BOTTOMMARGIN, 83
            END
    END
#endif    // APSTUDIO_INVOKED
// Menu
IDR_MENU1 MENU DISCARDABLE
    BEGIN
        MENUITEM "menu", ID_MENU
    END
#ifdef APSTUDIO_INVOKED
1 TEXTINCLUDE DISCARDABLE
    BEGIN
        "resource.h\0"
    END
```

```
2 TEXTINCLUDE DISCARDABLE
    BEGIN
        "#include ""afxres.h""\r\n"
        "\0"
    END
3 TEXTINCLUDE DISCARDABLE
    BEGIN
        "\r\n"
        "\0"
    END
#endif    // APSTUDIO_INVOKED
#endif    // Russian resources
#ifndef APSTUDIO_INVOKED
#endif    // not APSTUDIO_INVOKED.
```

Копируем содержимое файла swa.cpp в пустой файл dialog.cpp, добавляем в него оконную процедуру обработки сообщений COMMAND и функцию обработки диалогового окна ViewDlgProc(). Ниже приведен код полученного файла программы dialog.cpp.

```
    // dialog.cpp
#include <windows.h>
#include "win.h"
#include "resource.h"
static HDC hDC;
char Buff[21];
// Прототипы оконной процедуры и функции диалогового окна
   LONG WINAPI WndProg (HWND,UINT,WPARAM,LPARAM);
   BOOL CALLBACK ViewDlgProg(HWND, UINT, WPARAM, LPARAM);
    int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)
{
MSG msg;
```



```
    WinClass mainWnd("Диалоговое окно в приложении Windows ",
     hInstance, nCmdShow, WndProg,  MAKEINTRESOURCE (IDR_MENU1),
     100, 100, 400, 300);
          while (GetMessage(&msg, NULL, 0, 0))  {
               TranslateMessage(&msg);
               DispatchMessage(&msg); }
          return msg.wParam;
}
     // ======== Оконная процедура ========//
LONG WINAPI WndProg(HWND hWnd, UINT msg, WPARAM wParam,
LPARAM lParam)
{
HINSTANCE hInst;
   switch(msg)
    //Процедура обработки сообщения COMMAND  от диалогового окна
    {
       case WM_CREATE:
               hDC = GetDC(hWnd);
               break;
      case WM_COMMAND:
         switch (LOWORD(wParam))
           {
             case ID_MENU1:                    // меню
                  hInst = GetModuleHandle(NULL);
                  DialogBox(hInst, MAKEINTRESOURCE
                  (IDD_DIALOG1), hWnd, ViewDlgProg);
                  break;
           }
           break;
      case WM_DESTROY:
         PostQuitMessage(0);
         break;
      default:
               return (DefWindowProc(hWnd,msg,wParam,lParam));
    }
```

```
        return(0);
}
➤    // ========= Функция обработки Диалогового окна =========//
BOOL CALLBACK ViewDlgProg(HWND hDlg, UINT msg, WPARAM
wParam, LPARAM lParam)
{
char  Title[]="Введенные данные",  text[200];
  switch (msg)
       {
        ➧ case WM_COMMAND:
             switch (LOWORD(wParam))
             ➧ case IDOK:
                    {
                    ➧ GetDlgItemText(hDlg, IDC_EDIT1, Buff, 20);
                    ➧ sprintf(text, "% s", Buff);
                    ➧ TextOut(hDC, 200,200, text, strlen(text));
                    ➧ InvalidateRect(hDlg,0,TRUE);
                      return 0;
             ➧ case IDCANCEL:
                      EndDialog(hDlg, TRUE);
                      return TRUE;
                    }
       }
       return FALSE;
}
```

Выполнение работы диалогового окна начинается в оконной процедуре WndProg(), когда при выборе команды «меню» генерируется сообщение WM_COMMAND. Если это сообщение имеет подтип case: ID_MENU, то вызывается функция диалогового окна ViewDlgProc(), которая обрабатывает сообщения WM_INITDIALOG и WM_COMMAND. В ответ на сообщение WM_INITDIALOG выполняется инициализация элементов управления диалогового окна и вывод его на экран.

Большинство элементов управления диалогового окна посылают своим окнам-владельцам сообщения WM_COMMAND. Программа диалогового окна анализирует его с тем, чтобы определить, какой именно элемент послал это сообщение и какое действие для этого необходимо выполнить. Для этого выполняется анализ значения идентификатора «wParam» в параметре (LOWORD(wParam)). Если сообщение имеет подтип case: IDOK, выполняется процедура функции диалогового окна. В нашем приложении строка набранного в диалоговом окне текста при помощи функции TextOut(hDC, 200,200, text, strlen(text)); выводится на экран. Эта строка определяется по идентификатору IDC_EDIT1 вызовом функции GetDlgItemText(hDlg, IDC_EDIT1, Buff,20).

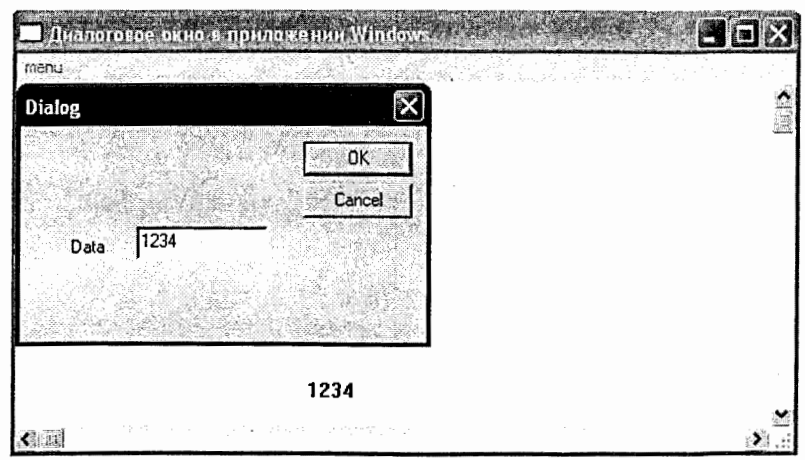

Рис.10. Окно приложения программы DIALOG

По завершении работы с диалоговым окном функция *DialogBox* возвратит управление процедуре WndProg(). Это выполняется с помощью функции *EndDialog* или нажатием на кнопку *Cancel* (case: ID_CANCEL). На этом процедура работы с диалоговым окном заканчивается. После создания кода исполняемого файла dialog.exe, запускаем нашу программу, и на экран выведется окно созданного приложения (рис.10).

3.15. Информационные окна

Это специальные модальные диалоговые окна, в которых выводится короткое сообщение для пользователя, например заголовок или некоторая комбинация стандартных кнопок и пиктограмм. Эти окна предназначены для вывода коротких текстовых сообщений и запроса у пользователя ответа из нескольких стандартных вариантов (*Да, Нет, Отмена, OK*). Например, информационные окна часто применяются для уведомления пользователя об ошибках программы и для запроса варианта реакции на ошибку: повторение или отмена операции.

Информационное окно создается и выводится на экран функцией *MessageBox*. Процедура начинается с сообщения WM_INITDIALOG, а при вызове функции *MessageBox* приложение передает ей строку сообщения и набор флагов, определяющих тип и внешний вид информационного окна. Далее идет обработка сообщения MessageBox(). Если нажата клавиша *YES* (wParam = IDYES), то выполняется процедура «Уничтожение окна»:

```
//==========Информационное окно ==========
 switch(msg)
      {
➤    case WM_INITDIALOG:
          return FALSE;
➤    case WM_CLOSE:
     //     Если нажата клавиша YES (wParam =IDYES)
          ➧ if (MessageBox (hWnd,"Закрыть окно ?", "Уходим",
            MB_YESNOCANCEL) == IDYES)
          ➧ DestroyWindow(hWnd);  // Уничтожение окна
            else
                 return 0;
            break;
      }
```


Для демонстрации работы информационного окна добавим эту процедуру в созданный проект PAINT в оконную процедуру WndProg():

```
     // ======== Оконная процедура ========//
LONG WINAPI WndProg(HWND hWnd, UINT msg, WPARAM wParam,
LPARAM lParam)
{
PAINTSTRUCT  ps;
HDC          hDC;
   switch(msg)
     {
      case WM_PAINT:
          hDC = BeginPaint(hWnd,&ps);
            TextOut(hDC,100,90,"Разработка windows-приложений",28);
            MoveToEx(hDC , 10, 10, NULL);LineTo(hDC, 600,400);
            MoveToEx(hDC, 300, 10,NULL); LineTo(hDC, 50, 300);
            Rectangle(hDC, 200,500,500,400);
            Ellipse(hDC, 550, 500, 650, 400);
          EndPaint(hWnd,&ps);
      break;
➤    //========Информационное окно ========
      ➧ case WM_INITDIALOG:
            return FALSE;
      ➧ case WM_CLOSE:
          if (MessageBox (hWnd,"Закрыть окно ?", "Уходим",
                MB_YESNOCANCEL) == IDYES)
                DestroyWindow(hWnd);      // Уничтожение окна
            else
                 return 0;
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
```

```
        return (DefWindowProc(hWnd,msg,wParam,lParam));
        }
    return(0);
}
```

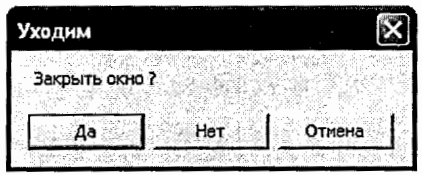

Рис.11. Информационное окно приложения PAINT

Вид информационного окна этого приложения показан на рис.11.

3.16. Разработка приложения для вывода графических объектов с вводом пользовательских данных в диалоговом окне

Создадим на базе оконного класса win.h новое Windows-приложение PAINT_DIALOG для вывода графика значений функций: $F_1 = \sin(x) \times A_1$, $F_2 = \cos(x) \times A_2$ и $F_3 = F_1 + F_2$ в диапазоне углов $\varphi = 0^0 \div 700^0$. Ввод параметров в этом приложении выполняется пользователем в диалоговом окне, а процедура создания графических объектов выполняется в контексте устройства с использованием функций графического модуля GDI. Обработка сообщения WM_PAINT в оконной процедуре WinProg() приложения PAINT_DIALOG выполняется с помощью конструкции *switch*:

```
switch(msg)
    {
  ➤ case WM_PAINT:
    ➤ hDC = BeginPaint(hWnd, &ps);
      ➧ DrawGraph (hWnd, hDC); //функция рисования графиков
      ➧ DrawAxis (hWnd, hDC);   // функция рисования осей
    ➤ EndPaint(hWnd, &ps);
    }
```


В этой конструкции вычисление функций и построение графиков по их значениям выполнено в функции DrawGraph(), а осей координат – в функции DrawAxis(). Контекст устройства, кроме характеристик вывода графических примитивов, содержит указатели на выбранные в контексте инструменты для рисования, такие как *Pen*, *Color*, *Width*, *Style*. С их помощью задается настройка цвета графических и текстовых объектов.

Для выбора цвета вызывается функция COLORREF() или DWORD():

```
static DWORD dwColor[7]=
        {
            RGB(0, 0, 0),          // черный
            RGB (245,0, 0),        // красный
            RGB(0, 245, 0),        // зеленый
            RGB(0, 0, 245),        // синий
            RGB(245,245, 0),       // желтый
            RGB(127,127, 127),     // серый
            RGB(245, 245, 245)     // белый
        };
```

которые возвращают значение цвета в виде интенсивности красной (R), зеленой (G) и голубой (B) составляющих цвета. Интенсивность составляющих цвета может быть в диапазоне от 0 до 255. Перо рисования задается функцией CreatePen(fnPenStyle, nWidth, crColor), где fnPenStyle – тип линии (может быть PS_SOLID – сплошная, PS_DOT – пунктирная и.т. д.), nWidth – ширина линии и crColor – тип цвета. Выбор пера для рисования выполняется вызовом функции SelectObject(hDC, hPen0). Например, для сплошной линии зеленого цвета и шириной nWidth =3:

```
HPEN hPen    = CreatePen(PS_SOLID, 3, RGB(0, 245,0));
HPEN hOldPen = (HPEN)SelectObject(hDC, hPen1);
```

а задание цвета для текстовой строки – вызовом функции SetTextColor():

```
SetTextColor(hDC, RGB(245,0,245)).
```

В функции void DrawGraph() значение функций $F_1 = \sin(x) \times A_1$ и $F_2 = \cos(x) \times A_2$ в диапазоне углов $\varphi = 0 \le i < 700$ заносим в массивы buf1[1024], buf2[1024] следующим образом:

```
void DrawGraph (HWND hWnd, HDC hDC)
{
double buf1[1024], buf2[1024];
int i;
```

➤ *// запись данных в массив*

```
a1 = atoi(Buff3); a3 = atoi(Buff4);
    for (i=0; i < 700; i++)
        {
        ➧ y1=(sin(a1*i*pi/180));      buf1[i] = y1;
        ➧ y2=(cos(a3*i*pi/180));      buf2[i] = y2;
        };
```

Частотные параметры функций a1 и a2 задаем в диалоговом окне. Затем, выбирая требуемое перо для рисования, строим графики функций, используя данные из массивов buf1 и buf2:

➤ *// построение графика синусоиды*

```
HPEN hPen0 = CreatePen(PS_SOLID, 3, RGB(0, 160, 0));
HPEN hOldPen = (HPEN)SelectObject(hDC, hPen0);
MoveToEx(hDC, 100, 200, NULL);
for (i=0; i < 700; i++)
    {
    ➧ y1 = buf1[i]*a2;
    ➧ LineTo(hDC, i+100,  (int) (200.0 - y1));
    }
```

Параметры масштабирования для функций $a1 \rightarrow y1 = buf1[i] \times a2$, $a2 \rightarrow y1 = buf2[i] \times a4$ также задаем в диалоговом окне. В функции *DrawAxis* выполняем построение осей координат и надписей для выводимых графических объектов:


```
void DrawAxis(HWND hWnd, HDC hDC)
{
```

➤ *// вывод текста*

```
➧ TextOut (hDC,70,200,"0",1); TextOut (hDC,70,150,"100",3);
➧ SetTextColor(hDC, RGB(245,0, 245));
➧ TextOut (hDC,250,20,"График функций",14);
```

➤ *// вывод осей координат*

```
  HPEN hPen3 = CreatePen(PS_SOLID, 2, RGB(127,127, 127));
  SelectObject(hDC, hPen3);
➧ MoveToEx(hDC, 100, 200, NULL); LineTo(hDC, 800,200);
➧ MoveToEx(hDC, 100, 20, NULL);   LineTo(hDC, 100,400);
}
```

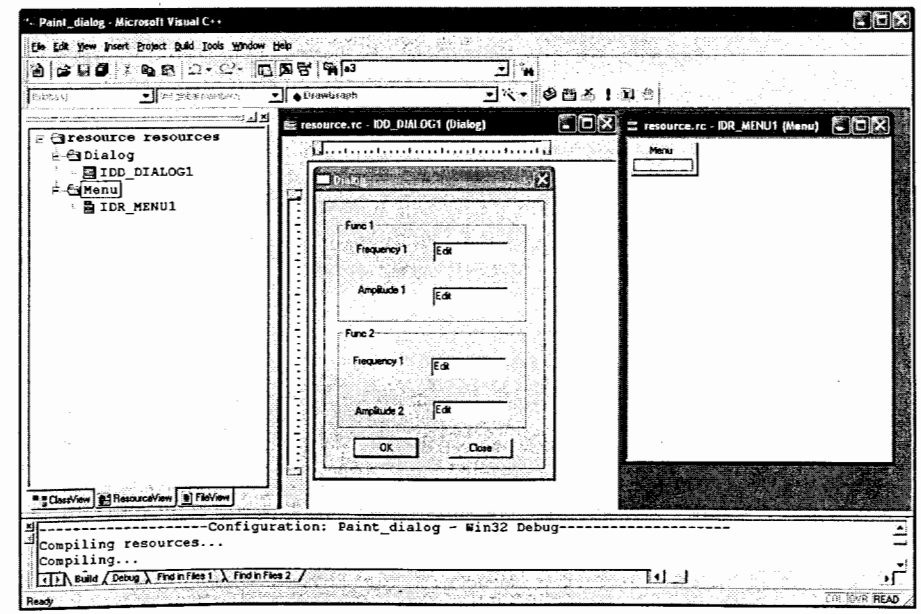

Рис.12. Создание диалогового окна с помощью редактора ресурсов

С помощью редактора ресурсов создаем в приложении Paint_dialog диалоговое окно и окно меню. Для этого в меню Visual C++6 (рис.12) выбираем опцию *Insert* и команду *Resource*. Затем из списка выбираем

необходимый тип ресурса «Menu» (рис.12), а в редакторе свойств (рис.13) задаем имя системного идентификатора меню ID_MENU.

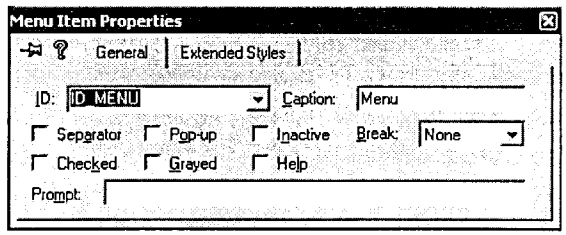

Рис.13. Системный идентификатор «Menu»

Аналогично, в редакторе ресурсов создаем диалоговое окно и присваиваем ему системный идентификатор IDD_DIALOG1. Затем в диалоговом окне, используя панель инструментов, создаем четыре текстовых поля редактирования (рис.12) для ввода значений параметров функций, которые необходимы программе. Редактор ресурсов автоматически присваивает им идентификаторы: IDC_EDIT1, IDC_EDIT2, IDC_EDIT3, IDC_EDIT4 соответственно. Поскольку эти параметры будут доступны в приложении только после окончания работы с диалоговым окном, то используем для этого кнопки *OK* и *Cancel*.

При нажатии кнопки *OK* запускается процедура обработки данных, введенных пользователем, тогда как при нажатии кнопки *Cancel* программа возвращается к исходному состоянию, а все введенные данные отменяются. В редакторе Visual C++6 с кнопками *OK* и *Cancel* связаны стандартные идентификаторы IDOK и IDCANCEL. После окончания работы в редакторе ресурсов, выбирая в меню → *File* команду → *Save As*, сохраняем этот файл ресурсов в нашем проекте с именем «resource.rc».

После создания файла ресурсов в программу оконной процедуры WinProg() включаем код обработки диалогового окна, т. е. программу обработки сообщения WM_COMMAND подтипа case: ID_MENU, которая

вызывает процедуру ViewDlgProc(), выполняющую инициализацию элементов диалогового окна.

После ввода параметров процедура диалогового окна анализирует сообщение WM_COMMAND с тем, чтобы определить, какое действие необходимо выполнить. Если это сообщение имеет подтип case: IDOK, то выполняется процедура присвоения значений текстовой строки в окне редактирования переменной типа *char* и тем самым в программу приложения вводятся параметры управления:

```
GetDlgItemText(hDlg,IDC_EDIT1, Buff1,5);
GetDlgItemText(hDlg, IDC_EDIT2, Buff2, 5);
GetDlgItemText(hDlg, IDC_EDIT3, Buff3, 5);
GetDlgItemText(hDlg, IDC_EDIT4, Buff4, 5);
```

Для использования этих параметров в качестве масштабирующих коэффициентов необходимо выполнить преобразование типа переменных: a1 = atoi(Buff1); a2 = atoi(Buff2); a3 = atoi(Buff3); a4 = atoi(Buff4).

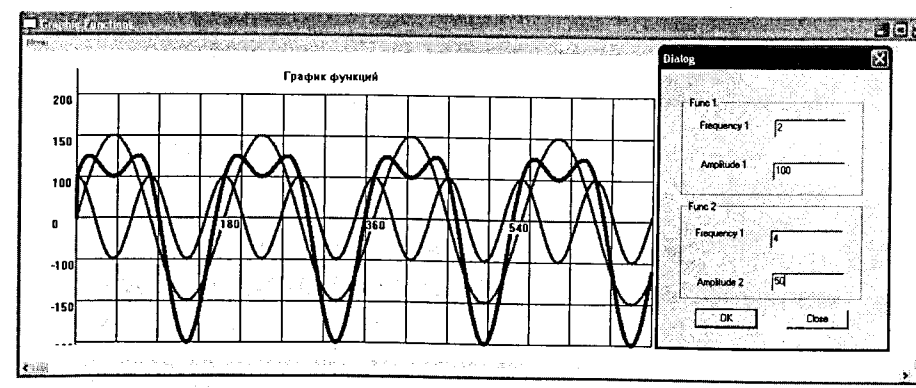

Рис.14. Окно приложения для вывода графики функций $F_3 = F_1 + F_2$ и диалоговое окно для ввода параметров управления

На рис.14 показано окно приложения для вывода графики функций $F_3 = F_1 + F_2$ и открытое диалоговое окно для ввода данных, а листинг этой программы приведен ниже.

//Листинг программы для вывода графики функций $F_3 = F_1 + F_2$:

```
#include <windows.h>
#include "win.h"
#include "resource.h"
#include <math.h>
char Buff1[5],Buff2[5],Buff3[5],Buff4[5];
int     a1, a2, a3, a4;
LONG WINAPI WinProg (HWND,UINT,WPARAM,LPARAM);
BOOL CALLBACK ViewDlgProg(HWND, UINT, WPARAM, LPARAM);
void DrawGraph(HWND, HDC);
void DrawAxis(HWND, HDC);
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)
{
MSG msg;
WinClass mainWnd("Graphic Functions", hInstance, nCmdShow, WinProg,
                        MAKEINTRESOURCE (IDR_MENU1));
      while (GetMessage(&msg, NULL, 0, 0)) {
               TranslateMessage(&msg);
               DispatchMessage(&msg);
               }
            return msg.wParam;
}
      // ======== Оконная процедура ========//
  ➤ LONG WINAPI WinProg (HWND hWnd, UINT msg, WPARAM
wParam, LPARAM lParam)
{
PAINTSTRUCT   ps;
HDC           hDC;
HINSTANCE     hInst;
      switch(msg)
          {
          ➤ case WM_PAINT:
```



```
               hDC = BeginPaint(hWnd, &ps);
             ➧ DrawGraph(hWnd, hDC);
             ➧ DrawAxis(hWnd, hDC);
               EndPaint(hWnd, &ps);
               break;
        ➤ case WM_COMMAND:
               switch (LOWORD(wParam))
                    {
                  ➧ case ID_MENU: // меню
                    hInst = GetModuleHandle(NULL);
                  ➧ DialogBox(hInst,MAKEINTRESOURCE
                              (IDD_DIALOG1), hWnd, ViewDlgProg);
                  ➧ InvalidateRect(hWnd,0,TRUE);
                    break;
                    }
               break;
        ➤ case WM_DESTROY:
               PostQuitMessage(0);
               break;
          default:
               return (DefWindowProc(hWnd,msg,wParam,lParam));
          }
          return 0;
}
➤     // ======== Функция DrawGraph ========//
 void DrawGraph(HWND hWnd, HDC hDC)
 {
double y1, y2, y3;
const double pi = 3.14159265359;
double buf1[1024], buf2[1024];
int i;
a1 = atoi(Buff1); a2 = atoi(Buff2); a3 = atoi(Buff3); a4 = atoi(Buff4);
          ➤ // запись данных в массив
```

```
        for (i=0; i < 700; i++)
              {
              y1=(sin(a1*i*pi/180));   buf1[i]=y1;
              y2=(cos(a3*i*pi/180));   buf2[i]=y2;
               }
      // построение синусоиды
        HPEN hPen0 = CreatePen(PS_SOLID, 3, RGB(0, 160, 0));
        HPEN hOldPen = (HPEN)SelectObject(hDC, hPen0);
        MoveToEx(hDC, 100, 200, NULL);
        for (i=0; i < 700; i++)
              {
              y1 = buf1[i]*a2; LineTo(hDC,  i+100, (int) (200.0-y1));
              }
        HPEN hPen1 = CreatePen(PS_SOLID, 3, RGB(0,0,245));
        SelectObject(hDC, hPen1);
        MoveToEx(hDC, 100, 200, NULL);
        for (i=0; i < 700; i++)
              {
              y2 = buf2[i]*a4; LineTo(hDC, i+100, (int) (200.0-y2));
              }
        HPEN hPen2 = CreatePen(PS_SOLID, 5, RGB(245,0, 0));
        SelectObject(hDC, hPen2);
        MoveToEx(hDC, 100, 200, NULL);
        for (i=0; i < 700; i++)
              {
              y3 = buf2[i]*a4+buf1[i]*a2;
              LineTo(hDC, i+100,(int) (200.0-y3));
              }
}
    // ======== Функция DrawAxis ========//
  void DrawAxis(HWND hWnd, HDC hDC)
      {
      // вывод текста
    TextOut (hDC,70,200,"0",1);        TextOut(hDC,70,150,"100",3);
```



```
TextOut (hDC,70,100,"150",3);      TextOut (hDC,70,50,"200",3);
TextOut (hDC,70,250,"-100",4);     TextOut (hDC,70,300,"-150",4);
TextOut (hDC,70,350,"-200",4);     TextOut (hDC,275,200,"180",3);
TextOut(hDC,450,200,"360",3);      TextOut(hDC,625,200,"540",3);
SetTextColor(hDC, RGB(245,0, 245));
TextOut (hDC,250,20,"График функций",14);
// вывод осей координат
HPEN hPen3 = CreatePen(PS_SOLID, 2, RGB(127,127, 127));
SelectObject(hDC, hPen3);
MoveToEx(hDC, 100, 200, NULL); LineTo(hDC, 800,200);
MoveToEx(hDC, 100, 20, NULL);  LineTo(hDC, 100,400);
HPEN hPen4 = CreatePen(PS_SOLID, 1, RGB(0,0, 0));
SelectObject(hDC, hPen4);
MoveToEx(hDC, 100, 50, NULL);   LineTo(hDC, 800,50);
MoveToEx(hDC,100,100,NULL);     LineTo(hDC,800,100);
MoveToEx(hDC, 100, 150, NULL); LineTo(hDC, 800,150);
MoveToEx(hDC,100,250,NULL);     LineTo(hDC,800,250);
MoveToEx(hDC,100,300, NULL);    LineTo(hDC, 800,300);
MoveToEx(hDC, 100, 350, NULL); LineTo(hDC, 800,350);
MoveToEx(hDC, 150, 50, NULL);   LineTo(hDC, 150,350);
MoveToEx(hDC, 200, 50, NULL);   LineTo(hDC, 200,350);
MoveToEx(hDC, 250, 50, NULL);   LineTo(hDC, 250,350);
MoveToEx(hDC, 300, 50, NULL);   LineTo(hDC, 300,350);
MoveToEx(hDC, 350, 50, NULL);   LineTo(hDC, 350,350);
MoveToEx(hDC, 400, 50, NULL);   LineTo(hDC, 400,350);
MoveToEx(hDC, 450, 50, NULL);   LineTo(hDC, 450,350);
MoveToEx(hDC, 500, 50, NULL);   LineTo(hDC, 500,350);
MoveToEx(hDC, 550, 50, NULL);   LineTo(hDC, 550,350);
MoveToEx(hDC, 600, 50, NULL);   LineTo(hDC, 600,350);
MoveToEx(hDC, 650, 50, NULL);   LineTo(hDC, 650,350);
MoveToEx(hDC, 700, 50, NULL);   LineTo(hDC, 700,350);
MoveToEx(hDC, 750, 50, NULL);   LineTo(hDC, 750,350);
MoveToEx(hDC, 800, 50, NULL);   LineTo(hDC, 800,350);
}
```

```
➤   // ======== Функция обработки Диалогового окна ========//
  BOOL CALLBACK ViewDlgProg(HWND hDlg, UINT msg, WPARAM
wParam, LPARAM lParam)
{
        switch (msg)
          {
          case WM_COMMAND:
                switch (LOWORD(wParam))
          case IDOK:
                {
                GetDlgItemText(hDlg, IDC_EDIT1, Buff1, 21);
                 GetDlgItemText(hDlg, IDC_EDIT2, Buff2, 21);
                 GetDlgItemText(hDlg, IDC_EDIT3, Buff3, 21);
                 GetDlgItemText(hDlg, IDC_EDIT4, Buff4, 21);
                 return 0;
          case IDCANCEL:
                 EndDialog(hDlg, TRUE);
                 return TRUE;
                 }
                break;
         }
          return FALSE;
}
```

3.17. Элементы управления

3.17.1. Создание образа кнопки

Элемент управления – это дочернее окно специального типа, применяемое для того, чтобы пользователь мог с его помощью выполнить какое-то простое действие. В результате этого действия элемент управления посылает окну-владельцу сообщение. Например, у *нажимаемой кнопки* – когда пользователь нажимает кнопку, то она посылает своему окну-владельцу или диалоговому окну сообщение WM_COMMAND. В приложении можно

создавать собственные элементы управления (разрабатывать "с нуля"), или их можно наследовать от стандартных оконных классов.

Рассмотрим процедуру создания элемента *кнопка*. Кнопка с независимой фиксацией может пребывать в одном из двух состояний: *включенном* или *выключенном* (кнопки с *зависимой фиксацией* часто используются в виде группы кнопок, позволяющей выбрать одно из нескольких взаимно исключающих состояний). Для создания образа кнопки вызываем функцию CreatWindow():

Button = CreateWindow(lpszClassName,ProgName,WS_CHILD|WS_VISIBLE| BS_DEFPUSHBUTTON, 0, 0, 200, 30, hWnd, (HMENU)ID_BUTTON, hInstance, NULL),

где аргумент *lpszClassName* задает имя класса BUTTON, а *ProgName* определяет надпись на кнопке. Аргумент *style* задает стиль кнопки: WS_CHILD|WS_VISIBLE|BS_DEFPUSHBUTTON (т. е. дочернее/видимое /имеет черную толстую обводку). Следующие четыре аргумента – это координаты прямоугольника, описывающие кнопку, затем дескриптор дочернего окна – *hWnd* и идентификатор кнопки – *ID_BUTTON*, с помощью которого распознается кнопка. Идентификатор описывают следующим образом:

#define ID_BUTTON 3330.

Дескриптор дочернего окна описывают в теле функции родительского окна: static HWND hButton. При помощи функции SetFocus(hButton) создается сообщение для главного окна о том, что оно получило фокус ввода от кнопки. Ниже приведена оконная процедура приложения BUTTON (файл button.cpp):

```
LONG WINAPI WndProg (HWND hWnd, UINT msg, WPARAM wParam,
LPARAM lParam)
{
```

```
static HWND hButton;
  switch(msg)
      {
       case WM_CREATE:
         {
        ➧ hButton = CreateWindow("BUTTON","Кнопка",WS_CHILD|
          WS_VISIBLE|BS_DEFPUSHBUTTON,0,0,200,30,hWnd,(HMENU)
          ID_BUTTON, hInstance, NULL);
               if(!hButton) return 1;
              SetFocus(hButton);return 0;
         }
      case WM_DESTROY:
         PostQuitMessage(0);
          break;
      default:
         return (DefWindowProc(hWnd,msg,wParam,lParam));
      }
           return 0;
}
```

3.17.2. Обработка сообщений от кнопки

Рассмотрим пример обработки сообщений двух кнопок (файл two_button.cpp). На рис.15 показан пример приложения TWO_BUTTON с двумя кнопками управления.

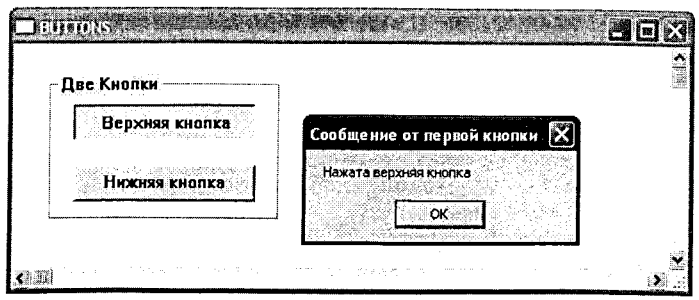

Рис.15. Приложение TWO_BUTTON с двумя кнопками управления

По аналогии с приложением BUTTON, в оконной процедуре WinProg() с помощью функции CreatWindow() создаем две кнопки: «Верхняя кнопка» и «Нижняя кнопка». Функция оконной процедуры WinProg() получает сообщение от этих кнопок WM_COMMAND, которое информирует программу о том, что с ними сделали, где параметр wParam описывает идентификатор кнопки, например wParam = ID_BUTTON1, а lParam описывает код извещения, по которому судят о совершенном над кнопкой действии. Нажатое состояние кнопки передает сообщение BM_SETSTATE с параметрами: wParam = TRUE и lParam = 0. Для получения значения параметра о состоянии кнопки посылают сообщение BM_GETSTATE, например, SendMessage(hButton1, BM_GETSTATE, 0, 0L), а с помощью функции BM_SETSTATE устанавливают состояния этих кнопок, например, SendMessage(hButton1, BM_SETSTATE, FALSE, 0L).

Если нажать на *нижнюю кнопку*, то функция окна получит сообщение WM_COMMAND с параметром LOWORD(wParam) = ID_BUTTON2. При его обработке приложение выдаст сообщение MessageBox(hWnd, "Нажата нижняя кнопка", MB_OK), которое по параметру lParam посылается кнопке2 → SendMessage((HWND)lParam, BM_SETSTATE, TRUE, 0L). По этому сообщению *нижняя кнопка* устанавливается в нажатое состояние. Затем производится анализ состояния *верхней кнопки*. Если она находится в нажатом состоянии, т. е. if(nSt==BST_PUSHED), то переводится в отжатое состояние → SendMessage(hButton1, BM_SETSTATE, FALSE, 0L).

Ниже приведена программа оконной процедуры (two_button.cpp) по обработке сообщений от кнопок:

➤ LONG WINAPI WndProg(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)

```
{
static HWND hButton1, hButton2;
   switch(msg)
```

```
{
  case WM_CREATE:
  {
  CreateWindow("BUTTON","Две кнопки",WS_CHILD|WS_VISIBLE|
  BS_GROUPBOX,30,25,190,120,hWnd,(HMENU)0,hInstance,NULL);
  hButton1= CreateWindow("BUTTON","Верхняя кнопка", WS_CHILD|
        WS_VISIBLE|BS_PUSHBUTTON,50,50,150,30,hWnd,(HMENU)
        ID_BUTTON1, hInstance,NULL);
  hButton2 =CreateWindow("BUTTON","Нижняя кнопка", WS_CHILD|
        WS_VISIBLE|BS_PUSHBUTTON,50,100,150,30,hWnd,(HMENU)
        ID_BUTTON2, hInstance,NULL); return 0;
  }
 case WM_COMMAND:
    {
    WORD nSt;
    switch(LOWORD(wParam))
       {
      case ID_BUTTON1:
         {
        MessageBox(hWnd," Нажата верхняя кнопка ",
                  "Сообщение от первой кнопки",MB_OK);
        SendMessage((HWND)lParam,BM_SETSTATE,TRUE,0L);
        nSt=(WORD)SendMessage(hButton2,BM_GETSTATE,0,0L);
          if(nSt==BST_PUSHED)
          SendMessage(hButton2,BM_SETSTATE,FALSE,0L);
          return 0;
         }
      case ID_BUTTON2:
         {
        MessageBox(hWnd,"Нажата нижняя кнопка",
              "Сообщение от первой кнопки",MB_OK);
        SendMessage((HWND)lParam,BM_SETSTATE,TRUE,0L);
        nSt=(WORD) SendMessage(hButton1,BM_GETSTATE,0,0L);
          if(nSt==BST_PUSHED)
              SendMessage(hButton1, BM_SETSTATE,FALSE,0L);
```



```
              return 0;
            }
          }
          return 0;
       }
  case WM_DESTROY:
   PostQuitMessage(0);
        break;
    default:
        return (DefWindowProc(hWnd,msg,wParam,lParam));
    }
return 0;
}
```

3.18. Создание приложения со специализированным элементом управления и с имитацией движения графического объекта

В практическом применении по программированию часто возникает необходимость визуального анализа поведения исследуемого объекта во времени или в пространстве. Это требует в программе приложения изменять положение графического объекта на экранной области с последующей ее перерисовкой. Для того чтобы понять, как выполняется эта процедура, рассмотрим пример разработки приложения MOVE_OBJECT с имитацией движения прямоугольника в заданном направлении и запуском движения объекта от элемента управления кнопка «Пуск».

На рис.16 показано окно приложения MOVE_OBJECT с кнопкой управления «Пуск». В этой программе имитация движения объекта выполнена за счет:

- стирания предшествующего изображения прямоугольника Rectangle (hDC, x1, y1, x1+20, (y1+20)) на экране с координатами положения: x1, y1;
- перерисовки изображения Rectangle(hDC, x + x1, y(x, a) + y1, x + 20 + x1, (y(x, a) + 20 + y1)) после получения новых текущих координат x, y.

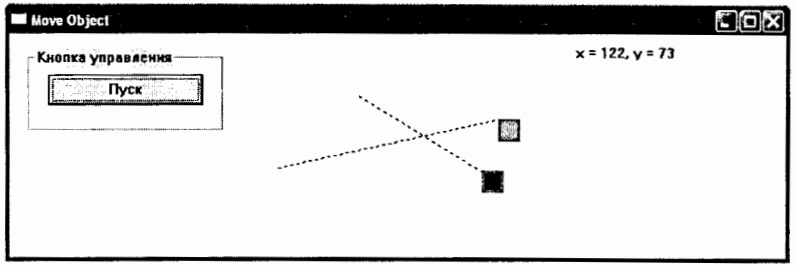

Рис.16. Приложение MOVE_OBJECT с имитацией движения объекта и кнопкой управления

Стирание изображения объекта в приложении может быть организовано, если в программе для контекста устройства hDC установлен режим рисования R2_XORPEN или R2_NOTXORPEN. В этом случае повторный вывод объекта приводит к восстановлению того состояния экрана, которое было перед первым выводом этого объекта. Результаты растровых операций после применения функций R2_XORPEN и R2_NOTXORPEN приведены в табл. 4.

Таблица 4. Операция R2_XORPEN и R2_NOTXORPEN

Объект		Результат операции:	
		R2_XORPEN	R2_NOTXORPEN
Цвет экрана Цвет пера	0 0	0	1
Цвет экрана Цвет пера	0 1	1	0
Цвет экрана Цвет пера	1 0	1	0
Цвет экрана Цвет пера	1 1	0	1
примечание	монохромная система с кодировкой 1 бит/пиксель «0» – черный цвет, «1» – белый цвет		

Так как мы рисуем прямоугольник черным пером на белом экране, то в соответствии с табл.4 необходимо использовать бинарную растровую операцию R2_NOTXORPEN. Если мы рисуем цветным пером в формате RGB, то в случае R2_NOTXORPEN значения одного бита, приведенные в табл.4, просто тиражируются по всем разрядам. В нашем приложении MOVE_OBJECT для удаления и перерисовки прямоугольника мы использовали функцию SetROP2(hDC, R2_NOTXORPEN):

```
SetROP2(hDC, R2_NOTXORPEN);
 for(x = 0; abs(x) <= (int)x3; x++)
  {
   Rectangle(hDC, -x + x1, y(x, a) + y1, -x +20 + x1, (y(x, a) +20 + y1));
   Sleep(8);
   Rectangle(hDC, -x + x1, y(x, a) + y1, -x +20 + x1, (y(x, a) +20 + y1));
  }
 Rectangle(hDC, -x + x1, y(x, a) + y1, -x +20 + x1, (y(x, a) +20 + y1));
 return 0;
```

По аналогии с предыдущим примером с помощью функции CreatWindow() создаем кнопку «Пуск» с идентификатором ID_BUTTON1. При нажатии на кнопку окно получит сообщение WM_COMMAND с параметром wParam, по которому запускается программа имитации движения прямоугольника. Ниже приведен листинг данной программы.

```
// Листинг программы приложения Move_Object.cpp
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include "win.h"
#define ID_BUTTON1 3001
char text[50];

LRESULT CALLBACK WndProg(HWND, UINT, WPARAM, LPARAM);
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)
{
```

```
MSG msg;
WinClass mainWnd("Move Object ", hInstance, nCmdShow, WndProg);
        while (GetMessage(&msg, NULL, 0, 0))
        {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
        }
        return msg.wParam;
}
int y (int x, float a)
{
        return (int)(a * x);
}
//====Оконная процедура=============================
LRESULT CALLBACK WndProg(HWND hWnd, UINT msg, WPARAM
wParam, LPARAM lParam)
{
static HDC hDC;  static  int x1, y1, x2, y2;
static BOOL bTracking = FALSE;
static HBRUSH hOld, hNew; static HPEN hOldPen, hNewPen;
static HWND hButton1;
int i = 0, x = 0;
float a = 0, x3 = 0, y3 = 0;
static int N=12;
    switch (msg)
     {
       case WM_CREATE:
       hDC = GetDC(hWnd);
       hOld=(HBRUSH)SelectObject(hDC, GetStockObject(WHITE_BRUSH));
       CreateWindow("BUTTON","Кнопка управления",
       WS_CHILD|WS_VISIBLE|BS_GROUPBOX|BS_DEFPUSHBUTTON,
       30, 25,190,80, hWnd, (HMENU)0,NULL,NULL);
       hButton1 =  CreateWindow("BUTTON","Пуск",
                WS_CHILD|WS_VISIBLE|BS_DEFPUSHBUTTON,
                50,50,150,30,hWnd,(HMENU)ID_BUTTON1,NULL,NULL);
```



```
break;
case WM_COMMAND:
  {
  switch(LOWORD(wParam))
    {
    case ID_BUTTON1:
        {
         for (i=0; i<N; i++)
         {
         BYTE red=  rand()%256;  BYTE green= rand()%256;
         BYTE blue= rand()%256;
         long color = RGB(red,green,blue);
         hNew = CreateSolidBrush (color);
         hOld  = (HBRUSH)SelectObject(hDC,hNew);
         HPEN hPen = CreatePen(PS_SOLID, 3,color);
         HPEN hOldPen=(HPEN)SelectObject(hDC, hPen);
         }
        x3 = (float)abs(x2 - x1);  y3 = (float)(y2 - y1);  a = y3 / x3;
        SelectObject(hDC, hOld);
         for(x = 0; abs(x) <= (int)x3; x++)
           {
         if (x2 < x1)
           {
          Rectangle(hDC,-x+x1, y(x, a)+y1,-x+20+x1,(y(x, a)+20+y1));
          Sleep(6);
           if (abs(x) != x3) Rectangle(hDC,-x+x1,y(x,a)+y1,-x+20+x1,
                        (y(x, a)+20+y1));
              sprintf(text, "                              ");
              TextOut(hDC, 300, 60, text, strlen(text));
              sprintf(text, "x = %d, y = %d",-x, y(x, a));
             TextOut(hDC, 300, 60, text, strlen(text));
           }
         else
          {
          Rectangle(hDC,x+x1,y(x,a)+y1,x+20+x1,(y(x,a)+20+y1));
```

```
                Sleep(6);
                 if(abs(x)!=x3) Rectangle(hDC,x+x1,y(x,a)+y1,x+20+x1,
                                (y(x,a)+20+ y1));
                 sprintf(text, "                                ");
                 TextOut(hDC, 300, 60, text, strlen(text));
                 sprintf(text, "x = %d, y = %d", x, y(x, a));
                 TextOut(hDC, 300, 60, text, strlen(text));
                 }
            } DeleteObject(hNew);
        }
    }  break;
  }
  case WM_LBUTTONDOWN:
        bTracking = TRUE;
        SetROP2(hDC, R2_NOTXORPEN);
        x1 = x2 = LOWORD(lParam); y1 = y2 = HIWORD(lParam);
  break;
  case WM_LBUTTONUP:
        if (bTracking)
                {
                bTracking = FALSE;
                x2 = LOWORD(lParam); y2 = HIWORD(lParam);
                HPEN hPen = CreatePen(PS_DOT, 1,RGB(0,0,245));
                HPEN hOldPen = (HPEN)SelectObject(hDC, hPen);
                MoveToEx(hDC, x1, y1, NULL);
                LineTo(hDC, x2, y2); DeleteObject(hNew);
                }
        break;
case WM_DESTROY:
        SelectObject(hDC, hOld); SelectObject(hDC, hOldPen);
        ReleaseDC(hWnd, hDC);
        PostQuitMessage(0);
        break;
  default:
        return DefWindowProc(hWnd, msg, wParam, lParam);
```



```
  }
  return 0;
  }
```

На этом мы заканчиваем первый этап изучения разрабоки Windows-приложений с использованием API-интерфейса.

Рекомендуемая литература

1. Ганеев Р.М. Проектирование интерфейса пользователя средствами Win32 API. М.: Горячая линия – Телеком, 2007.
2. Рихтер Дж. Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows. СПб.: БХВ – Петербург, 2001.
3. Давыдов В. Visual C++. Разработка Windows-приложений с помощью MFC- и API-функций. СПб.: БХВ – Петербург, 2008.
4. Липпман Стэнли Б. Основы проектирования на C++. М.: Издательский дом Вильямс, 2002.
5. Круглински Д., Уингоу С., Шеферд Дж. Программирование на Microsoft Visual C++ 6.0 для профессионалов. СПб.: БХВ–Питер, 2000.
6. Павловская Т.А. C/C++ Программирование на языке высокого уровня. Учебник для вузов. СПб.: БХВ – Петербург, 2002.
7. Лафоре Р. Объектно-ориентированное программирование в C++ . СПб.: БХВ – Петербург, 2007.

Учебное издание

Калинников Владимир Александрович

Разработка Windows-приложений с помощью API-функций на основе объектно-ориентированного языка C++

УНЦ-2009-41

Редактор *М. И. Зарубина*

Получено 19.10.2009. Подписано в печать 03.12.2009.
Формат 60 × 90/16. Усл. печ. л. 6,37. Уч.-изд. л. 4,95. Тираж 85. Заказ № 56803.

Издательский отдел Объединенного института ядерных исследований
141980, г. Дубна, Московская обл., ул. Жолио-Кюри, 6
E-mail: publish@jinr.ru
www.jinr.ru/publish/