Федеральное агентство образования
Димитровградский институт технологий, управления и дизайна

Курсовая работа
по курсу: “Программирование Windows”
на тему: “Сравнение технологий программирования «C Windows Api32» и «C#.NET»”

Выполнил студент гр. ВТ-31:
Потеренко А.Г.
Проверил преподаватель:
Наскальнюк А.Н.

Димитровград 2006г

Содержание

1. Основы технологий.

1.1. Синтаксис языков программирования (C# .NET)

Операторы

Категория операторов	Операторы
Арифметика	+ - * / %
Логический (boolean и bitwise)	& | ^ ! ~ && || true false
Соединение строк	+
Приращение, декремент	++ --
Изменение	<< >>
Относительный	== != < > <= >=
Назначение	= += -= *= /= %= &= |= ^= <<= >>=
Доступ к члену класса	.
Индексация	[]
Cast	()
Условный	?:
Связь делегата и удаление	+ -
Создание объекта	new
Информация о типе	is sizeof typeof
Проверка выхода за предел	checked unchecked
Адрес	* -> [] &

Алфавит C#

Алфавит (или множество литер) языка программирования C# составляют символы таблицы кодов ASCII. Алфавит C# включает:

- строчные и прописные буквы латинского алфавита (мы их будем называть буквами);
- цифры от 0 до 9 (назовем их буквами-цифрами);
- символ «_» (подчеркивание — также считается буквой);
- набор специальных символов: " { }, 1 [ ] + – %/ \; ' : ? < > = ! & #
- прочие символы.

Алфавит C# служит для построения слов, которые в C++ называются лексемами. Различают пять типов лексем:

- идентификаторы;
- ключевые слова;
- знаки (символы) операций;
- литералы;
- разделители.

Почти все типы лексем (кроме ключевых слов и идентификаторов) имеют собственные правила словообразования, включая собственные подмножества алфавита.

Лексемы обособляются разделителями. Этой же цели служит множество пробельных символов, к числу которых относятся пробел, табуляция, символ новой строки и комментарии.

Ключевые слова и имена

Часть идентификаторов C# входит в фиксированный словарь ключевых слов. Эти идентификаторы образуют подмножество ключевых слов (они так и называются ключевыми словами). Прочие идентификаторы после

специального объявления становятся именами. Имена служат для обозначения переменных, типов данных, функций. Обо всем этом позже.

Ниже приводится таблица со списком ключевых слов. Вы не можете использовать эти имена для образования классов, функций, переменных и других языковых структур.

abstract	enum	long	stackalloc
as	event	namespace	static
base	explicit	new	string
bool	extern	null	struct
break	false	object	switch
byte	finally	operator	this
case	fixed	out	throw
catch	float	override	true
char	for	params	try
checked	foreach	private	typeof
class	goto	protected	uint
const	if	public	ulong
continue	implicit	readonly	unchecked
decimal	in	ref	unsafe
default	int	return	ushort
delegate	interface	sbyte	using
do	internal	sealed	virtual
double	is	short	void
else	lock	sizeof	while

Комментарии

Часто бывает полезно вставлять в программу текст, который является комментарием только для читающего программу человека и игнорируется компилятором. В C# это можно сделать одним из двух способов. Символы **/*** начинают комментарий, заканчивающийся символами ***/**. Такая последовательность символов эквивалентна символу пропуска (например, символу пробела). Это особенно полезно для многострочных комментариев и изъятия частей программы при редактировании, однако следует помнить, что комментарии **/* */** не могут быть вложенными.

Символы **//** начинают комментарий, заканчивающийся в конце строки, на которой они появились. И здесь вся последовательность символов эквивалентна пропуску. Этот способ наиболее полезен для коротких комментариев.

Литералы

В C# существует четыре типа литералов:

- целочисленный литерал;
- вещественный литерал;
- символьный литерал;
- строковый литерал.

Литералы — это особая категория слов языка. Для каждого подмножества литералов используются собственные правила словообразования.

Целочисленный литерал служит для записи целочисленных значений и является соответствующей последовательностью цифр (возможно, со знаком '-'). Целочисленный литерал, начинающийся со знака 0, воспринимается как восьмеричное целое. В этом случае цифры 8 и 9 не должны встречаться среди составляющих литерал символов. Целочисленный литерал, начинающийся с 0x или 0X, воспринимается как шестнадцатеричное целое. В этом случае целочисленный литерал может включать символы от A или a, до F или f, которые в шестнадцатеричной системе эквивалентны десятичным значениям от 10 до 15. Непосредственно за литералом могут располагаться в произвольном сочетании один или два специальных суффикса: U (или u) и L (или l).

Вещественный литерал служит для отображения вещественных значений. Он фиксирует запись соответствующего значения в обычной десятичной или научной нотациях. В научной нотации мантисса отделяется от порядка литерой E (ли e). Непосредственно за литералом может располагаться один из двух специальных суффиксов: F (или f) и L (или l).

Значением ***символьного литерала*** является соответствующее значение ASCII кода (это, разумеется, не только буквы, буквы-цифры или специальные символы алфавита C#). Символьный литерал представляет собой последовательность одной или нескольких литер, заключенных в одинарные кавычки. Символьный литерал служит для представления литер в одном из форматов представления. Например, литера Z может быть представлена литералом «Z», а также литералами «\132» и «\x5A». Любая литера может быть представлена в нескольких форматах представления: обычном, восьмеричном и шестнадцатеричном.

Строковые литералы являются последовательностью (возможно, пустой) литер в одном из возможных форматов представления, заключенных в двойные кавычки. Строковые литералы, расположенные последовательно, соединяются в один литерал, причем литеры соединенных строк остаются различными. Так, последовательность строковых литералов «\xF» «F» после объединения будет содержать две литеры, первая из которых является символьным литералом в шестнадцатеричном формате «\xF», вторая — символьным литералом «F». Строковый литерал и объединенная последовательность строковых литералов заканчиваются пустой литерой, которая используется как индикатор конца литерала.

1.2. Типы данных (C Windows NT API)

Типы данных языка C описывать не имеет смысла. Опишем типы данных, которые приходится использовать при программировании под Windows на языке C. Типы данных, поддерживаемые Microsoft Windows используются для определения возвращаемых значений функциями, параметров функций и сообщений и членов структур. Они определяют размер и значение этих элементов.

Нижеследующая таблица содержит такие перечисляемые типы, как: символ, целое число, булево число, указатель и дескриптор. Типы символ, целое и булево число являются стандартными для большинства трансляторов с языка C. Большинство названий типа указателя начинаются с префикса P или LP. Дескрипторы ссылаются на ресурс, который был загружен в память.

Название	**Предназначение**
ATOM	Атом. За большей информацией обратитесь к статье Atoms (Атомы).
BOOL	Булева переменная (должна быть ИСТИНА (TRUE) или ЛОЖЬ (FALSE)).
BOOLEAN	Булева переменная (должна быть ИСТИНА (TRUE) или ЛОЖЬ (FALSE)).
BYTE	Байт (8 бит).
CALLBACK	Соглашение о вызовах для функций повторного вызова.
CHAR	8-битовый символ Windows (ANSI). За большей информацией обратитесь к статье Наборы символов, используемые шрифтами.
COLORREF	Красный (red), зеленый (green), голубой (blue) (RGB) значение цвета (32 бита). За большей информацией об этом типе обратитесь к статье COLORREF.
CONST	Переменная, значение которой должно остаться постоянным в ходе выполнения программного модуля.
CRITICAL_SECTION	Объект критической секции. За большей информацией обратитесь к статье Объекты критической секции.
DWORD	32-разрядное беззнаковое целое число.
DWORD_PTR	Тип беззнаковый дальний для точности указателя. Используется тогда, когда производится приведение указателя к дальнему типу, чтобы выполнить арифметические операции над указателями.
DWORD32	32-разрядное беззнаковое целое число.

DWORD64	64-разрядное беззнаковое целое число.
FLOAT	Переменная с плавающей точкой.
HACCEL	Дескриптор таблицы ускорителей (accelerator table).
HANDLE	Дескриптор объекта.
HBITMAP	Дескриптор точечного рисунка (bitmap).
HBRUSH	Дескриптор кисти.
HCONV	Дескриптор динамического обмена данными (DDE) в режиме диалога.
HCONVLIST	Дескриптор списка DDE в режиме диалога.
HCURSOR	Дескриптор курсора.
HDC	Дескриптор контекста устройства (DC).
HDDEDATA	Дескриптор данных DDE.
HDESK	Дескриптор рабочего стола.
HDROP	Дескриптор структуры вставки внутрь.
HDWP	Дескриптор структуры отложенной позиции окна.
HENHMETAFILE	Дескриптор усовершенствованного метафайла (enhanced metafile).
HFILE	Дескриптор открытого файла при помощи OpenFile, а не CreateFile.
HFONT	Дескриптор шрифта.
HGDIOBJ	Дескриптор объекта GDI.
HGLOBAL	Дескриптор блока глобальной памяти.
HHOOK	Дескриптор hook-точки.
HICON	Дескриптор пиктограммы.
HIMAGELIST	Дескриптор списка изображений.
HIMC	Дескриптор контекста ввода данных.
HINSTANCE	Дескриптор экземпляра исполняемого модуля.
HKEY	Дескриптор ключа реестра.
HKL	Идентификатор ввода данных национального языка.

HLOCAL	Дескриптор блока локальной памяти.
HMENU	Дескриптор меню.
HMETAFILE	Дескриптор метафайла.
HMODULE	Дескриптор модуля.
HMONITOR	Дескриптор монитора.
HPALETTE	Дескриптор палитры.
HPEN	Дескриптор пера.
HRGN	Дескриптор региона.
HRSRC	Дескриптор ресурса.
HSZ	Дескриптор строки DDE.
HWINSTA	Дескриптор оконного терминала.
HWND	Дескриптор окна.
INT	32-разрядное знаковое целое число.
INT_PTR	Целый знаковый тип для точности указателя. Используется тогда, когда производится приведение указателя к целому типу, чтобы выполнить арифметические операции над указателем.
INT32	32-разрядное знаковое целое число.
INT64	64-разрядное знаковое целое число.
LANGID	Идентификатор языка. За большей информацией обратитесь к статье Совокупность родственных национальных языков (Locales).
LCID	Идентификатор национального языка За большей информацией обратитесь к статье Совокупность родственных национальных языков (Locales).
LCTYPE	Тип информации о национальном языке. Список смотрите в статье Информация о стране и языке.
LONG	32-разрядное знаковое целое число.
LONG_PTR	Дальний знаковый тип для точности указателя. Используется тогда, когда производится приведение указателя к дальнему типу, чтобы выполнить арифметические операции над указателями.
LONG32	32-разрядное знаковое целое число.

LONG64	64-разрядное знаковое целое число.
LONGLONG	64-разрядное знаковое целое число.
LPARAM	Параметр сообщения.
LPBOOL	Указатель на BOOL.
LPBYTE	Указатель на BYTE.
LPCOLORREF	Указатель на значение COLORREF.
LPCRITICAL_SECTION	Указатель на CRITICAL_SECTION.
LPCSTR	Указатель на строковую константу с нулем в конце 8-разрядных символов (ANSI) Windows. За большей информацией обратитесь к статье Наборы символов, используемые шрифтами.
LPCTSTR	Если определен UNICODE, то тип LPCWSTR, иначе LPCTSTR.
LPCVOID	Указатель на константу любого типа.
LPCWSTR	Указатель на строковую константу с нулем в конце из 16-битовых символов Unicode. За большей информацией обратитесь к статье Наборы символов, используемые шрифтами.
LPDWORD	Указатель на DWORD.
LPHANDLE	Указатель на HANDLE (ДЕСКРИПТОР).
LPINT	Указатель на INT.
LPLONG	Указатель на LONG.
LPSTR	Указатель на строку с нулем в конце из 8-битовых символов Windows (ANSI). За большей информацией обратитесь к статье Наборы символов, используемые шрифтами.
LPTSTR	Если определен UNICODE, то тип LPWSTR, иначе LPSTR.
LPVOID	Указатель на любой тип.
LPWORD	Указатель на WORD.
LPWSTR	Указатель на строку с нулем в конце из 16-битовых символов Unicode. За большей информацией обратитесь к статье Наборы символов, используемые шрифтами.
LRESULT	Обозначает результат обработки сообщения.
LUID	Локальный уникальный идентификатор.

PBOOL	Указатель на BOOL.
PBOOLEAN	Указатель на BOOL.
PBYTE	Указатель на BYTE.
PCHAR	Указатель на CHAR.
PCRITICAL_SECTION	Указатель на CRITICAL_SECTION.
PCSTR	Указатель на строковую константу с нулем в конце из 8-разрядных символов (ANSI) Windows. За большей информацией обратитесь к статье Наборы символов, используемые шрифтами.
PCTSTR	Если определен UNICODE, то тип PCWSTR, иначе PCSTR.
PCWCH	Указатель на константу WCHAR.
PCWSTR	Указатель на строковую константу с нулем в конце из 16-битовых символов Unicode. За большей информацией обратитесь к статье Наборы символов, используемые шрифтами.
PDWORD	Указатель на DWORD.
PFLOAT	Указатель на FLOAT.
PHANDLE	Указатель на HANDLE (ДЕСКРИПТОР).
PHKEY	Указатель на HKEY.
PINT	Указатель на INT.
PLCID	Указатель на LCID.
PLONG	Указатель на LONG.
PLUID	Указатель на LUID.
POINTER_32	32-разрядный указатель. На 32-разрядной системе, это - родной указатель. На 64-разрядной системе, это -усеченный 64-разрядный указатель.
POINTER_64	64-разрядный указатель. На 64-разрядной системе, это - родной указатель. На 32-разрядной системе, это – знаковый расширенный 32-разрядный указатель.
PSHORT	Указатель на SHORT.
PSTR	Указатель на строковую константу с нулем в конце 8-разрядных символов (ANSI) Windows. За большей информацией обратитесь к статье Наборы символов, используемые шрифтами.

PTBYTE	Указатель на TBYTE.
PTCHAR	Указатель на TCHAR.
PTSTR	Если определен UNICODE, то тип PWSTR, иначе PSTR.
PTBYTE	Указатель на TBYTE.
PTCHAR	Указатель на TCHAR.
PTSTR	Если определен UNICODE, то тип PWSTR, иначе PSTR.
PUCHAR	Указатель на UCHAR.
PUINT	Указатель на UINT.
PULONG	Указатель на ULONG.
PUSHORT	Указатель на USHORT.
PVOID	Указатель на тип.
PWCHAR	Указатель на WCHAR.
PWORD	Указатель на WORD.
PWSTR	Указатель на строку с нулем в конце из 16-битовых символов Unicode. За большей информацией обратитесь к статье Наборы символов, используемые шрифтами.
REGSAM	Маска доступа системы безопасности для ключа системного реестра.
SC_HANDLE	Дескриптор менеджера сервисного управления базой данных. Для получения дополнительной информации, см.статью Дескрипторы SCM.
SC_LOCK	Дескриптор менеджера сервисного управления блокировкой базы данных. Для получения дополнительной информации, см. статью Дескрипторы SCM.
SERVICE_STATUS_ HANDLE	Значение дескриптора состояния модуля обслуживания. Для получения дополнительной информации, см. статью Дескрипторы SCM.
SHORT	Короткое целое число (16 бит).
SIZE_T	Максимальное число байтов, на которые указатель может указывать. Используется для счета, который должен охватить полностью диапазон указателя.
SSIZE_ T	Знаковый SIZE_T.

TBYTE	Если определен UNICODE, то тип WCHAR, иначе CHAR.
TCHAR	Если определен UNICODE, то тип WCHAR, иначе CHAR.
UCHAR	Беззнаковый CHAR.
UINT	Беззнаковый INT.
UINT_PTR	Беззнаковый INT_PTR.
UINT32	Беззнаковый INT32.
UINT64	Беззнаковый INT64.
ULONG	Беззнаковый LONG.
ULONG_PTR	Беззнаковый LONG_PTR.
ULONG32	Беззнаковый LONG32.
ULONG64	Беззнаковый LONG64.
ULONGLONG	64-битовое беззнаковое целое число.
UNSIGNED	Беззнаковый атрибут.
USHORT	Беззнаковый SHORT.
VOID	Любой тип.
WCHAR	16-битовый символ Unicode. За большей информацией обратитесь к статье Наборы символов, используемые шрифтами.
WINAPI	Соглашение о вызовах для системных функций
WORD	16-битовое беззнаковое целое число.
WPARAM	Параметр сообщения.

1.2. Типы данных (C# .NET)

Основные типы:

value-type:
- *struct-type*
- *enum-type*

struct-type:
- *type-name*
- *simple-type*

simple-type:
- *numeric-type*
- bool

numeric-type:
- *integral-type*
- *floating-point-type*
- decimal

integral-type:
- sbyte
- byte
- short
- ushort
- int
- uint
- long
- ulong
- char

floating-point-type:
- float
- double

enum-type:
- *type-name*

Пример использования и преобразования типов:

```
int i = int.MaxValue;
string s = i.ToString();
string t = 123.ToString();
string l="96";
long j = Convert.ToInt64(l,10);
```

C# является жестко типизированным языком. При его использовании вы должны объявлять тип каждого объекта, который создаете (например, целые числа, числа с плавающей точкой, строки, окна, кнопки, и т. д.), и компилятор поможет вам избежать ошибок, связанных с присвоением переменным значений только того типа, который им соответствует. Тип объекта указывает компилятору размер объекта (например, объект типа `int` занимает в памяти 4 байта) и его свойства (например, форма может быть видима и невидима, и т.д.).

Подобно языкам C++ и Java, C# подразделяет типы на два вида: встроенные типы, которые определены в языке, и определяемые пользователем типы, которые выбирает программист.

C# также подразделяет типы на две другие категории: размерные и ссылочные. Основное различие между ними — это способ, которым их значения сохраняются в памяти. Размерные типы сохраняют свое фактическое значение в стеке. Ссылочные типы хранят в стеке лишь адрес объекта, а сам объект сохраняется в куче. **Куча** — основная память программ, доступ к которой осуществляется на много медленнее чем к стеку. Если вы работаете с очень большими объектами, то сохранение их в куче имеет много преимуществ.

C# также поддерживает и указатели на типы, но они редко употребляются. Применение указателей связано с использованием неуправляемого кода.

Особенности использования стека и кучи

Стек — это структура данных, которая сохраняет элементы по принципу: первым пришел, последним ушел (полная противоположность очереди). Стек относится к области памяти, поддерживаемой процессором, в которой сохраняются локальные переменные. Доступ к стеку во много раз быстрее, чем к общей области памяти, поэтому использование стека для хранения данных ускоряет работу вашей программы. В C# размерные типы (например, целые числа) располагаются в стеке: для их значений зарезервирована область в стеке, и доступ к ней осуществляется по названию переменной.

Ссылочные типы (например, объекты) располагаются в куче. Куча — это оперативная память вашего компьютера. Доступ к ней осуществляется медленнее, чем к стеку. Когда объект располагается в куче, то переменная хранит лишь адрес объекта. Этот адрес хранится в стеке. По адресу программа имеет доступ к самому объекту, все данные которого сохраняются в общем куске памяти (куче).

«Сборщик мусора» уничтожает объекты, располагающиеся в стеке, каждый раз, когда соответствующая переменная выходит за область видимости. Таким образом, если вы объявляете локальную переменную в пределах функции, то объект будет помечен как объект для «сборки мусора». И он будет удален из памяти после завершения работы функции.

Объекты в куче тоже очищаются сборщиком мусора, после того как конечная ссылка на них будет разрушена.

Встроенные типы

Язык C# предоставляет программисту широкий спектр встроенных типов, которые соответствуют **CLS** (Common Language Specification) и отображаются на основные типы платформы .NET. Это гарантирует, что объекты, созданные на C#, могут успешно использоваться наряду с объектами, созданными на любом другом языке программирования, поддерживающем .NET CLS (например, VB.NET).

Каждый тип имеет строго заданный для него размер, который не может изменяться. В отличие от языка C++, в C# тип int всегда занимает 4 байта, потому что отображается к Int32 в .NET CLS.

В дополнение к этим примитивным типам C# может иметь объекты типа **enum** и **struct**.

Преобразование встроенных типов

Объекты одного типа могут быть преобразованы в объекты другого типа неявно или явно. Неявные преобразования происходят автоматически, компилятор делает это вместо вас. Явные преобразования осуществляются, когда вы «приводите» значение к другому типу. Неявные преобразования гарантируют также, что данные не будут потеряны. Например, вы можете неявно приводить от short (2 байта) к int (4 байта). Независимо от

того, какой значение находится в short, оно не потеряется при преобразовании к int.

Если вы делаете обратное преобразование, то, конечно же, можете потерять информацию. Если значение в int больше, чем 32.767, оно будет усечено при преобразовании. Компилятор не станет выполнять неявное преобразование от int к short.

Нельзя использовать неинициализированную переменную в C#. У вас может сложиться впечатление, что вы должны инициализировать каждую переменную в программе. Это не так. Вы должны лишь назначить переменной значение прежде, чем попытаетесь ее использовать.

Константы

Константа — это переменная, значение которой не может быть изменено. Переменные — это более гибкий способ хранения данных. Однако иногда вы хотите гарантировать сохранение значения определенной переменной.

Например, число pi. Как известно, значение этого числа никогда не изменяется. Следовательно, вы должны гарантировать, что переменная, хранящая это число, не изменит своего значения на протяжении всей работы программы.

Существует три разновидности константы: литералы, символические константы и перечисления. Символические константы устанавливают имя для некоторого постоянного значения. Вы объявляете символическую константу, используя ключевое слово const, и применяете следующий синтаксис для создания константы:

const тип идентификатор = *значение;*

Константа обязательно должна быть проинициализирована, и ее значение не может быть изменено во время выполнения программы.

Перечисления

Перечисления являются мощной альтернативой константам. Это особый тип значений, который состоит из набора именованных констант. Допустим, у вас есть список констант, содержащих годы рождения ваших знакомых. Для того чтобы запрограммировать это при помощи констант, вам придется написать:

```
const intfloat maryBirthday = 1955;
const intfloat ivanBirthday =- 1980;
const intfloat pavelBirthday - 1976;
```

У вас получились три совершенно несвязанные константы. Для того чтобы установить логическую связь между ними, в C# предусмотрен механизм перечислений.

Каждое перечисление имеет свой базовый тип, которым может быть любой встроенный целочисленный тип C# (int, long, short и т. д.), за исключением char.

Перечисление задается следующим образом:

```
[атрибуты] [модификаторы]
  enum идентификатор [: базовый тип] ( список перечислений);
```

Строковые константы

Для объявления в программе константной строки вам необходимо заключить содержимое строки в двойные кавычки ("My string"). Вы можете делать это практически в любом месте программы: в передаче параметров функции, в инициализации переменных.

1.3. Циклические операторы (C# .NET)

C# включает достаточно большой набор циклических операторов, таких как **for**, **while**, **do...while**, а также цикл с перебором каждого элемента **foreach**. Кроме того, C# поддерживает операторы перехода и возврата, например **goto**, **break**, **continue** и **return**.

Оператор **goto** был основой для реализации других операторов цикла. Но он был и базой многократных переходов, вследствие чего возникла запутанность кода программы. Поэтому опытные программисты стараются его не использовать, но для того чтобы узнать все возможности языка, рассмотрим и этот оператор. Он используется следующим образом:

1. Создается метка в коде программы Label1.

2. Организуется переход на эту метку goto Labell.

Цикл **while**. Эта циклическая инструкция работает по принципу: «Пока выполняется условие — происходит работа». Ее синтаксис выглядит следующим образом:

while (expression) statement

Как и в других инструкциях, выражение — это условие, которое оценивается как булево значение. Если результатом проверки условия является истина, то выполняется блок инструкций, в противном случае в результате выполнения программы while игнорируется.

Цикл **do... while**. Бывают случаи, когда цикл while не совсем удовлетворяет вашим требованиям. Например, вы хотите проверять условие не в начале, а в конце цикла. В таком случае лучше использовать цикл do...while.

do statement while (expression);

Подобно while, выражение — это условие, которое оценивается как булево значение. Это выражение можно прочитать как: «выполнить действие; если выполняется условие — повторить выполнение еще раз». Заметьте разницу между этой формулировкой и формулировкой работы цикла while. Разница состоит в том, что цикл do...while выполняется всегда минимум один раз, до того как произойдет проверка условия выражения.

Цикл **for**. Если еще раз внимательно посмотреть на примеры (while, do...while, goto), можно заметить постоянно повторяющиеся операции: первоначальная инициализация переменной i, ее наращивание на 1 внутри цикла, проверка переменной i на выполнение условия (i < 10). Цикл for позволяет вам объединить все операции в одной инструкции.

for ([initializers]; [expression]; [iterators]) statement

Операторы **break и continue**. Бывают ситуации, когда необходимо прекратить выполнение цикла досрочно (до того как перестанет выполняться условие) или при каком-то условии не выполнять описанные в теле цикла инструкции, не прерывая при этом цикла. Для таких случаев очень удобно использовать инструкции break и continue. Если вы хотите на каком-то шаге цикла его прекратить, не обязательно выполняя до конца описанные в нем действия, то лучше всего использовать break.

1.4. Массивы (C# .NET)

В C# массивы являются объектами, производными от базового класса **System.Array**. Поэтому, несмотря на синтаксис определения массивов, аналогичный языку C++, на самом деле происходит создание экземпляра класса, унаследованного от System.Array.

Одномерные массивы

Если вы объявляете массив как:

int[] arr;

то объявляете объект класса, производный от System.Array.

Как уже отмечалось, для создания объекта массива необходимо использовать оператор **new**.

```
int[] arr = new int[100];
arr[0]=1;
arr[1]=1;
arr[99]=1;
```

Многомерный массив — это массив, элементами которого являются массивы. Если массив имеет порядок N, то элементами такого массива являются массивы порядка N-1. Порядок еще называется рангом массива. Массив может иметь любой ранг, хотя вряд ли вам пригодится массив ранга 10, обычно используют одно, двухранговые (двумерные) массивы. В некоторых случаях применяются массивы с рангом три.

```
int[,] arr = new int[100,100];
arr[0,0]=1;
arr[1,0]=1;
arr[99,100]=1;
```

Как уже говорилось, любой массив является объектом класса, производным от System.Array. Как и любой другой класс, System.Array имеет свои методы и свойства. Одним из важнейших методов класса System.Array является метод **GetLength**(). Он позволяет получить размер определенного измерения многомерного массива. Другое важное достоинство класса System.Array — свойство **Rank**, которое позволяет программным путем определить ранг массива, то есть количество измерений массива. Кроме того, класс System.Array имеет свойство **Length**, которое возвращает общее количество элементов в массиве.

```
int g=arr.GetLength(0);
g=arr.Rank;
g=arr.Length;
```

Многомерные массивы могут иметь как одинаковый размер всех вложенных массивов, так и различный. Если используемый в программе массив будет иметь различную длину вложенных массивов, то об этом следует позаботиться заранее. Объявлять такой массив следует как:

int [] [] arr;

Такое объявление будет означать, что объявляется массив массивов значений типа int. Это определение невыровненных массивов. Пример таких массивов:

```
int[][] myJaggedArray = new int[3][];
myJaggedArray[0] = new int[5];
myJaggedArray[1] = new int[4];
myJaggedArray[2] = new int[2];
```

В Visual Basic уже давно встроен оператор для итерирования массивов и коллекций. Поэтому разработчики C# сочли полезным наделить такой возможностью и язык C#. Синтаксис оператора **foreach** следующий:

foreach (тип переменная in массив)

```
using System;
class MainClass
{
   public static void Main()
   {
      int odd = 0, even = 0;
      int[] arr = new int [] {0,1,2,5,7,8,11};
      foreach (int i in arr)
      {
         if (i%2 == 0)
            even++;
         else
            odd++;
      }
      Console.WriteLine("Found {0} Odd Numbers, and {1} Even Numbers.",odd,even);
   }
}
```

Для перебора всех элементов массива используется цикл for. Такой способ наиболее прижился среди программистов на C++. Однако такой способ имеет ряд проблем: необходимость инициализации переменной цикла, проверки булева значения, наращивания переменной цикла, использования переменной определенного типа для выделения элемента массива. Использование оператора foreach позволяет избежать всех этих неудобств.

Насколько проще и понятнее использование оператора foreach! Вам гарантирован обход всех элементов массива. Вам не нужно вручную устанавливать значение переменной для инициализации и проверять границы массива. Кроме того, foreach сам поместит необходимое значение в переменную указанного типа. Если вам нужно прервать цикл foreach, вы можете воспользоваться операторами break и return.

1.5. Классы. Наследование в C#

Классы — сердце каждого объектно-ориентированного языка. Как вы помните, класс представляет собой инкапсуляцию данных и методов для их обработки. Это справедливо для любого объектно-ориентированного языка, которые отличаются в этом плане лишь типами данных, хранимых в виде членов, а также возможностями классов. В том, что касается классов и многих функций языка, C# кое-что заимствует из C++ и Java и привносит немного изобретательности, помогающей найти элегантные решения старых проблем.

Определение классов

Синтаксис определения классов на C# прост, особенно если вы программируете на C++ или Java. Поместив перед именем вашего класса ключевое слово **class**, вставьте члены класса, заключенные в фигурные скобки, например:

```
using System;
public class Kid
{
   private int age;
   private string name;
   public Kid()
   {
      name = "N/A";
   }
   public Kid(string name, int age)
   {
      this.name = name;
      this.age = age;
   }
   public void PrintKid()
   {
      Console.WriteLine("{0}, {1} years old.", name, age);
   }
}
public class MainClass
{
   public static void Main()
   {
      Kid kid1 = new Kid("Craig", 11);
      Kid kid2 = new Kid("Sally", 10);
      Kid kid3 = new Kid();
      Console.Write("Kid #1: ");
      kid1.PrintKid();
      Console.Write("Kid #2: ");
      kid2.PrintKid();
      Console.Write("Kid #3: ");
      kid3.PrintKid();
   }
}
```

Состав классов

Типы, определенные в CTS поддерживаются как члены классов C# и бывают следующих видов:

- **Поле.** Так называется член-переменная, содержащий некоторое значение. В ООП поля иногда называют данными объекта. К полю можно применять несколько модификаторов в зависимости от того, как вы собираетесь его использовать. В число модификаторов входят static, readonly и const. Ниже мы познакомимся с их назначением и способами применения.
- **Метод.** Это реальный код, воздействующий на данные объекта (или поля). Здесь мы сосредоточимся на определении данных класса.
- **Свойства.** Их иногда называют «разумными» полями (smart fields), поскольку они на самом деле являются методами, которые клиенты класса воспринимают как поля. Это обеспечивает клиентам большую степень абстрагирования за счет того, что им не нужно знать, обращаются ли они к полю напрямую или через вызов метода-аксессора.
- **Константы.** Как можно предположить, исходя из имени, константа — это поле, значение которого изменить нельзя.
- **Индексаторы.** Если свойства — это «разумные» поля, то индексаторы — это «разумные» массивы, так как они позволяют индексировать объекты методами-аксессорами get и set. С помощью индексатора легко проиндексировать объект для установки или получения значений.
- **События.** Событие вызывает исполнение некоторого фрагмента кода. События — неотъемлемая часть программирования для Microsoft Windows.

Модификаторы доступа

Теперь, зная, что типы могут быть определены как члены класса C#, познакомимся с модификаторами, используемыми для задания степени доступа, или доступности данного члена для кода, лежащего за пределами его собственного класса. Они называются модификаторами доступа (access modifiers). Модификаторы доступа в C#:

Модификатор доступа	Описание
public	Член доступен вне определения класса и иерархии производных классов.
protected	Член невидим за пределами класса, к нему могут обращаться только производные классы.
private	Член недоступен за пределами области видимости определяющего его класса.
internal	Член видим только в пределах текущей единицы компиляции. Модификатор доступа internal в плане ограничения доступа является гибридом public и protected, зависимым от местоположения кода.

Если вы не хотите оставить модификатор доступа для данного члена по умолчанию (private), задайте для него явно модификатор доступа.

Метод Main

У каждого приложения на C# должен быть метод Main, определенный в одном из его классов. Кроме того, этот метод должен быть определен как public и static. Для компилятора C# не важно, в каком из классов определен метод Main, а класс, выбранный для этого, не влияет на порядок компиляции. Здесь есть отличие от C++, где зависимости должны

тщательно отслеживаться при сборке приложения. Компилятор C# достаточно «умен», чтобы самостоятельно просмотреть ваши файлы исходного кода и отыскать метод Main. Между тем, этот очень важный метод является точкой входа во все приложения на C#.

Инициализация классов и конструкторы

Одно из величайших преимуществ языков ООП, таких как C#, состоит в том, что вы можете определять специальные методы, вызываемые всякий раз при создании экземпляра класса. Эти методы называются конструкторами (constructors). C# вводит в употребление новый тип конструкторов — статические (static constructors), с которыми вы познакомитесь ниже в подразделе «Константы и неизменные поля».

Гарантия инициализации объекта должным образом, прежде чем он будет использован,— ключевая выгода от конструктора. Когда пользователь создает экземпляр объекта, вызывается его конструктор, который должен вернуть управление до того, как пользователь сможет выполнить над объектом другое действие. Именно это помогает обеспечивать целостность объекта и делать написание приложений на объектно-ориентированных языках гораздо надежнее.

Но как назвать конструктор, чтобы компилятор знал, что его надо вызывать при создании экземпляра объекта? Разработчики C# последовали в этом вопросе за разработчиками C++ и провозгласили, что у конструкторов в C# должно быть то же имя, что и у самого класса.

Значений конструкторы не возвращают. При попытке использовать с конструктором в качестве префикса имя типа компилятор сообщит об ошибке, пояснив, что вы не можете определять члены с теми же именами, что у включающего их типа. Следует обратить внимание и на способ создания экземпляров объектов в C#. Это делается при помощи ключевого слова new:

<*класс*> <*объект*> = *new* <*класс*>(*аргументы конструктора*)

Ссылочные типы создаются в куче, а размерные — в стеке. Поэтому ключевое слово new позволяет делать новые экземпляры класса, но не определяет место создания объекта.

Статические члены класса

Вы можете определить член класса как статический (static member) или член экземпляра (instance member). По умолчанию каждый член определен как член экземпляра. Это значит, что для каждого экземпляра класса делается своя копия этого члена. Когда член объявлен как статический, имеется лишь одна его копия. Статический член создается при загрузке содержащего класс приложения и существует в течение жизни приложения. Поэтому вы можете обращаться к члену, даже если экземпляр класса еще не создан.

Наследование

Чтобы построить (в терминах данных или поведения) класс на основе другого класса, применяется наследование. Оно принадлежит к правилам заменяемости, а именно к тому, которое гласит, что производный класс

может быть заменен базовым. Примером этого может служить написание иерархии классов базы данных. Допустим, вам был нужен класс для обработки баз данных Microsoft SQL Server и Oracle. Поскольку эти базы различны, вам понадобится по классу для каждой из БД. Однако в обеих БД достаточная доля общей функциональности, которую можно поместить в базовый класс, а два других класса сделать его производными, при необходимости подменяя или изменяя поведение, унаследованное от базового класса.

Чтобы унаследовать один класс от другого, используется синтаксис:

class <*производный_класс*>*:* <*базовый_класс*>

Наследование	**Пример**
Нет	`class ClassA { }`
Единственное	`class DerivedClass: BaseClass { }`
Наследование 2х интерфейсов	`class ImplClass: IFace1, IFace2 { }`
Единственное с одним интерфейсом	`class ImplDerivedClass: BaseClass, IFace1 { }`

1.6. Механизм индексаторов в C#

В C# существует особая возможность, позволяющая работать с объектами так, как если бы они были массивами,— это ***механизм индексаторов***.

Преимущество использования индексаторов

Вы можете спросить: «Зачем нужны индексаторы, в чем преимущество их использования?» Ответ такой: «Процесс написания приложений становится интуитивно более понятным». Индексаторы можно сравнить со свойствами. Последние позволяют обращаться к полям класса без использования методов аксессоров. При использовании свойств программисту нет необходимости задумываться о формате метода-установщика или метода-получателя. Программист выбирает простую инструкцию типа ***объект.поле***.

То же можно сказать и об индексаторах. Они позволяют обращаться к полям объекта так, как если бы это был массив.

В стремлении разработать максимально интуитивно понятный язык программирования они задумались над вопросом возможности обработки объекта, который по своей сути является массивом, как массива. В связи с этим и родилась концепция индексаторов.

Определение индексаторов

Как уже было сказано выше, индексаторы очень похожи на свойства. Формально синтаксис определения индексатора таков:

```
[attributes] [modifiers] indexer-declarator {accessor-declarations}
indexer-declarator:
type this [formal-index-parameter-list]
type interface-type.this [formal-index-parameter-list]
formal-index-parameter:
[attributes] type identifier
```

атрибуты — дополнительная информация об индексаторе. Наиболее значимым для индексаторов является атрибут ***Name***. Задание Name позволяет дать имя индексатору для того, чтобы его могли использовать другие языки, не поддерживающие индексаторы. По умолчанию все индексаторы вашего класса имеют Name, равный ***Item***; ***модификаторы*** — модификаторы доступа и директивы. К индексатору применимы почти все стандартные директивы C#. Он может быть скрыт, перегружен, сделан виртуальным, но есть одно исключение, индексатор не может быть ***static***; ***список формальных параметров*** — указывает параметры, посредством которых осуществляется индексация. Передается в get и set, которые используются в индексаторе так же, как в свойствах, get применяется для вычисления индексатора по заданному списку формальных параметров, а set — для изменения индексатора, set получает в качестве дополнительного параметра value того же типа, что и индексатор.

Следует отметить, что доступ к индексатору осуществляется посредством сигнатуры, в отличие от свойства, доступ к которому осуществляется посредством имени. Сигнатурой индексатора считаются

число и тип формальных параметров. Тип самого индексатора и имена параметров в сигнатуру не входят. Естественно, в классе не может быть двух индексаторов с одинаковой сигнатурой. К тому же, индексатор не считается переменной и не может быть передан в качестве ***ref*** или ***out*** параметра.

```
using System;
class IndexerClass
{
   private int [] myArray = new int[100];
   public int this [int index]
   {
      get
      {
         if (index < 0 || index >= 100)
            return 0;
         else
            return myArray[index];
      }
      set
      {
         if (!(index < 0 || index >= 100))
            myArray[index] = value;
      }
   }
}
public class MainClass
{
   public static void Main()
   {
      IndexerClass b = new IndexerClass();
      b[3] = 256;
      b[5] = 1024;
      for (int i=0; i<=10; i++)
      {
         Console.WriteLine("Element #{0} = {1}", i, b[i]);
      }
   }
}
```

1.7. Работа со строками (C# .NET)

C# обеспечивает встроенную поддержку работы со строками. Более того, C# обрабатывает строки как объекты, что инкапсулирует все методы манипуляции, сортировки и поиска, обычно применяемые к строкам символов.

Особенности типа Sytem.String

C# обрабатывает строки как встроенные типы, которые являются гибкими, мощными и удобными. Каждый объект строки — это неизменная последовательность символов Unicode. Другими словами, те методы, которые изменяют строки, на самом деле возвращают измененную копию, а первоначальная строк остается неповрежденной.

Объявляя строку на C# с помощью ключевого слова **string**, вы фактически объявляете объект типа **System.String**, являющийся одним из встроенных типов, обеспечиваемых .NET Framework библиотекой классов.

C# строка — это объект типа System.String.

Объявление класса System.String следующее:

```
public sealed class String : IComparable, ICloneable,
                             IConvertible, IEnumerable
```

Такое объявление говорит о том, что класс запечатан, что невозможно унаследовать свой класс от класса String. Класс также реализует четыре системных интерфейса — **IComparable**, **ICloneable**, **IConvertible** и lEnumerable — которые определяют функциональные возможности System. String за счет его дополнительного использования с другими классами в **.NET Framework**.

Интерфейс IComparable определяет тип, реализующий его как тип, значения которого могут упорядочиваться. ICloneable объекты могут создавать новые экземпляры объектов с теми же самыми значениями, как и первоначальный вариант. В данном случае возможно клонировать строку таким образом, чтобы создать новую с тем же самым набором символов, как и в оригинале. IConvertible классы реализуют методы для облегчения преобразования объектов класса к другим встроенным типам.

Создание строк

Наиболее общий способ создания строк состоит в том, чтобы установить строку символов, известную как строковый литерал, определяемый пользователем, в переменную типа string:

```
string newString = " Новая строка".
```

Строки могут также быть созданы с помощью дословной записи строки. Такие строки должны начинаться с символа (@), который сообщает конструктору String, что строка должна использоваться дословно, даже если она включает служебные символы. В дословном определении строки наклонные черты влево и символы, которые следуют за ними, просто рассматриваются как дополнительные символы строки.

System.Object.ToString()

Другой способ создать строку состоит в том, чтобы вызвать у объекта

метод ToString() и установить результат переменной типа string. Все встроенные типы имеют этот метод, что позволяет упростить задачу преобразования значения (часто числового значения) к строковому виду.

Класс System.String в .NET поддерживает множество перегруженных конструкторов, которые обеспечивают разнообразные методы для инициализации строковых значений различными типами. Некоторые из этих конструкторов дают возможность создавать строку в виде массива символов или в виде указателя на символы. При создании строки в виде массива символов CLR создает экземпляр новой строки с использованием безопасного кода.

При создании строки на основе указателя применяется «небезопасный» код, что крайне нежелательно при разработке приложений .NET.

Класс StringBuilder

Класс StringBuilder используется для создания и редактирования строк, обычно строк из динамического набора данных, например из массива байтовых значений. Очень важной особенностью класса StringBuilder является то, что при изменении значений в объекте StringBuilder происходит изменение значений в исходной строке, а не в ее копии.

2. Взаимодействие с ОС Windows NT.

2.1. Элементы управления Windows (C Windows NT API)

Создание элементов управления на чистом Api затруднительно. Сложный интерфейс программы приходится создавать “руками” в самой программе. Интерфейс главного окна уж точно придется делать программисту. Диалоговые окна можно создавать самому, а можно загружать из файла ресурсов, предварительно “собрав” его на этапе конструирования. Меню также можно загружать из файла ресурсов. Иначе можно сделать вывод – что нельзя загрузить из файла – придется делать самостоятельно в своей программе. Рассмотри пример создания кнопки в окне:

```
hwndButton = CreateWindow(
    "BUTTON",
    "OK",
    WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON,
    10,
    10,
    100,
    100,
    hwnd,
    NULL,
    (HINSTANCE) GetWindowLong(hwnd, GWL_HINSTANCE),
    NULL);
```

2.1. Элементы управления Windows (C# .NET)

Технология .NET позволяет визуально создавать компоненты для приложений пользователя. Это является, несомненно, большим плюсом при разработке приложений со сложным интерфейсом.

Создание элементов управления, обработка событий для них – все процедуры генерирует среда .NET. Программисту остается только сосредоточится на прикладной части программы.

2.2. Работа с файлами (C Windows NT API)

Функции файлового ввода - вывода, которые создают, открывают и удаляют файлы и каталоги, идентифицируют их своими именами. Эти функции сохраняют или ищут файл или папку в текущем каталоге на текущем дисководе, если имя явно не задает путь к другому каталогу, дисководу, или к обоим. Основные операции с файлами:

- *Открытие и создание файлов*
- *Чтение из файла и запись в файл*
- *Получение информации о временном файле*
- *Закрытие файлов*
- *Удаление файлов*
- *Защита файла и права доступа*

Функция **CreateFile** может создать новый или открыть существующий файл. Когда приложение создает новый файл, операционная система добавляет его в заданный каталог.

Когда приложение использует CreateFile, то должно определить, будет ли оно читать из файла, записывать в файл, или то и другое вместе.

Приложение должно также установить и какое выбрать действие, в зависимости от того, файл существует или нет. Например, прикладная программа может определить, чтобы функция CreateFile всегда использовалась для создания файла. В результате этого функция создает файл, если он не существует и переписывает его, если он действительно существует.

Функция CreateFile дает также возможность приложению и определять, допускает ли оно совместно использовать файл для чтения, записи, то и другое вместе, или не допускает ни того, ни другого действия. Файл, который не используется совместно, не может открываться более чем один раз первым приложением, а другим приложением до тех пор, пока первое приложение не закроет его.

Операционная система присваивает уникальный идентификатор, называемый ***дескриптором файла*** (file handle), каждому файлу, который открывается или создается. Приложение может использовать дескриптор файла в функциях, которые читают из файла, записывают в файл и характеризуют файл. Это правильно до тех пор, пока файл не закрыт. Когда приложение запускается, оно наследует все дескрипторы открытых файлов в процессе, который запустил программу, если дескрипторы наследуемы.

Приложение должно проверить возвращаемое значение функцией CreateFile перед попыткой использовать дескриптор, чтобы получить доступ к файлу. Если происходит ошибка, приложение может использовать функцию **GetLastError**, чтобы получить дополнительные сведения об ошибке.

Функция **DeleteFile** может быть использована для удаления файла. Но перед этим файл должен быть закрыт, чтобы любая попытка удалить его завершилась успешно.

Если вы удаляйте открытый файл или каталог на удаленной машине, а он ранее был открыт на ней без установки разрешения общего чтения, не вызывайте функцию CreateFile или OpenFile, чтобы открыть файл или каталог раньше удаления. Такое действие приведет к ошибке совместного доступа.

Приложение читает из файла и пишет в файл, используя функции **ReadFile** и **WriteFile**. Эти функции требуют дескриптора файла, открытого для чтения и записи, соответственно. Функции ReadFile и WriteFile читают и записывают указанное число байтов в месте, обозначенном указателем позиции в файле. Данные читаются и записываются именно так, как определены; функции не форматируют данные.

Схемы сборки - распределения данных могут также использоваться в операциях чтения и записи.

Когда указатель позиции в файле достигает конца, а приложение пытается читать из файла, ошибки не происходит, но никакие байты не читаются. Поэтому, прочитывание нулевых байтов без ошибки означает, что приложение достигло конца файла. Запись нулевых байтов не делает ничего.

Когда приложение пишет в файл, система обычно собирает записываемые данные во внутреннем буфере и пишет данные на диск на обычных основаниях. Приложение может заставить содержание этого буфера сбросить на диск.

Приложения могут получать уникальные имена файлов для временных файлов при помощи функции **GetTempFileName**. Функция же **GetTempPath** извлекает путь к каталогу, где будут создаваться временные файлы.

Чтобы эффективно использовать ресурсы операционной системы, приложение должно закрывать файлы, когда они больше не нужны, при помощи использования функции **CloseHandle**. Если файл является открытым, когда приложение завершает работу, система закрывает его автоматически.

Поскольку файлы - это защищенные объекты (**securable objects**), обращение к ним, регулируется моделью управления доступом, которая руководит доступом ко всем другим защищенным объектам в Windows.

Вы можете установить дескриптор безопасности (**security descriptor**) для файла или каталога, когда вызываете функцию CreateFile, CreateDirectory или CreateDirectoryEx. Если Вы для параметра lpSecurityAttributes определяете значение ПУСТО (NULL) , файл или каталог получает заданный по умолчанию дескриптор безопасности. Списки контроля доступа (ACL) в заданном по умолчанию дескрипторе безопасности для файла или каталога наследуются от его директории предыдущего уровня.

Чтобы извлечь дескриптор безопасности объекта файла или каталога, вызовите функцию **GetNamedSecurityInfo** или **GetSecurityInfo**. Чтобы изменить дескриптор безопасности объекта файла или каталога, вызовите функцию **SetNamedSecurityInfo** или **SetSecurityInfo**.

Правильные права доступа для файлов и каталогов включают в себя стандартные права (standard) **DELETE** (удаление), **READ_CONTROL** (управление чтением), **WRITE_DAC** (избирательный контроль за доступом к записи) и **WRITE_OWNER** (монопольная запись).

Пример для работы с файлами:

```
HANDLE hFile;
HANDLE hAppend;
DWORD  dwBytesRead, dwBytesWritten, dwPos;
char   buff[4096];
// Open the existing file.
hFile = CreateFile("ONE.TXT",     // open ONE.TXT
    GENERIC_READ,                 // open for reading
    0,                            // do not share
    NULL,                         // no security
    OPEN_EXISTING,                // existing file only
    FILE_ATTRIBUTE_NORMAL,        // normal file
    NULL);                        // no attr. template
if (hFile == INVALID_HANDLE_VALUE) {
    ErrorHandler("Could not open ONE.");  // process error
}
// Open the existing file, or if the file does not exist,
// create a new file.
hAppend = CreateFile("TWO.TXT",   // open TWO.TXT
    GENERIC_WRITE,                // open for writing
    0,                            // do not share
    NULL,                         // no security
    OPEN_ALWAYS,                  // open or create
    FILE_ATTRIBUTE_NORMAL,        // normal file

    NULL);                        // no attr. template
if (hAppend == INVALID_HANDLE_VALUE) {
    ErrorHandler("Could not open TWO.");    // process error
}
// Append the first file to the end of the second file.
// Lock the second file to prevent another process from
// accessing it while writing to it. Unlock the
// file when writing is finished.
do {
  if (ReadFile(hFile, buff, 4096, &dwBytesRead, NULL)) {
      dwPos = SetFilePointer(hAppend, 0, NULL, FILE_END);

      LockFile(hAppend, dwPos, 0, dwPos + dwBytesRead, 0);
      WriteFile(hAppend, buff, dwBytesRead,
          &dwBytesWritten, NULL);
      UnlockFile(hAppend, dwPos, 0, dwPos + dwBytesRead, 0);
  }
} while (dwBytesRead == 4096);
// Close both files.
CloseHandle(hFile);
CloseHandle(hAppend);
```

2.2. Работа с файлами (C# .NET)

Большинство операций ввода-вывода в **.NET Framework** реализованы в пространстве имен **System.IO Namespace**. В **.NET Framework** различаются файлы и потоки. Файл представляет собой поименованные данные, хранящиеся на диске; часто, кроме имени, указан и каталог. При открытии для чтения/записи файл становится потоком. Поток — это сущность, допускающая операции чтения и записи. Но потоки — это не только открытые файлы. Данные, получаемые по сети, являются потоком, кроме того, поток можно создать в памяти. В консольных приложениях, клавиатурный ввод и текстовый вывод — тоже потоки.

Важнейший класс файлового ввода-вывода

Если вы собираетесь изучить только один класс из пространства имен System.IO, это должен быть класс File Stream. Этот базовый класс используется для открытия, чтения, записи и закрытия файлов. FileStream наследуется от абстрактного класса Stream, и множество его свойств и методов являются производными из этого класса.

Для открытия существующего файла или создания нового нужно создать объект класса FileStream.

Для чтения и записи текстовых файлов используется StreamReader и StreamWriter, а для чтения и записи двоичных файлов – BinaryReader и BinaryWriter.

Приведем пример для работы с текстовыми файлами. С другими файлами работа происходит аналогичным образом.

Чтение текстового файла

```
using System;
using System.IO;
public class TextFromFile
{
    private const string FILE_NAME = "MyFile.txt";
    public static void Main(String[] args)
    {
        if (!File.Exists(FILE_NAME))
        {
            Console.WriteLine("{0} does not exist.", FILE_NAME);
            return;
        }
        StreamReader sr = File.OpenText(FILE_NAME);
        String input;
        while ((input=sr.ReadLine())!=null)
        {
            Console.WriteLine(input);
        }
        Console.WriteLine ("The end of the stream has been reached.");
        sr.Close();
    }
}
```

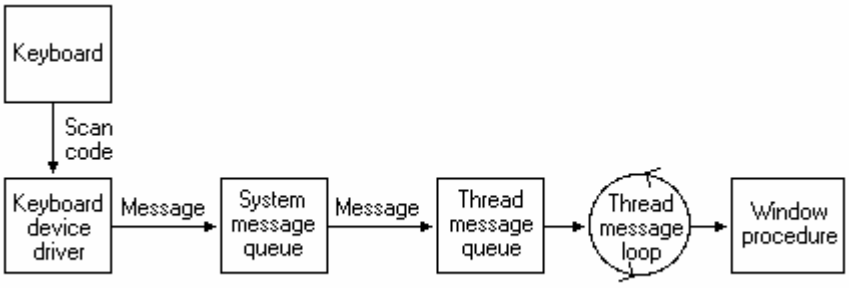
Создание текстового файла и запись в него

```
using System;
using System.IO;
public class TextToFile
{
    private const string FILE_NAME = "MyFile.txt";
    public static void Main(String[] args)
    {
        if (File.Exists(FILE_NAME))
        {
            Console.WriteLine("{0} already exists.", FILE_NAME);
            return;
        }
        StreamWriter sr = File.CreateText(FILE_NAME);
        sr.WriteLine ("This is my file.");
        sr.WriteLine ("I can write ints {0} or floats {1}, and so on.", 1, 4.2);
        sr.Close();
    }
}
```

2.3. Работа с клавиатурой (C Windows NT API)

Окна обеспечивают независимую от устройства поддержку клавиатуры для приложений, устанавливая драйвер устройства клавиатуры, соответствующего текущей клавиатуре. Окна обеспечивают независимую от языка поддержку клавиатуры, используя определенное языковое расположение клавиатуры, в настоящее время выбранное пользователем или приложением. Драйвер устройства клавиатуры получает коды от клавиатуры, которые посланы клавиатуре, переведены в сообщения и отправлены к соответствующим окнам вашего приложения.

Схема взаимодействия клавиатуры:



Клавиатура должна разделяться между всеми приложениями, работающими под Windows. Некоторые приложения могут иметь больше одного окна, и клавиатура должна разделяться между этими окнами в рамках одного и того же приложения. Когда на клавиатуре нажата клавиша, только одна оконная процедура может получить сообщение об этом. Окно, которое получает это сообщение клавиатуры, является окном, имеющем "фокус ввода" (input focus).

Концепция фокуса ввода тесно связана с концепцией "активного окна". Окно, имеющее фокус ввода — это либо активное окно, либо дочернее окно активного окна. Определить активное окно обычно достаточно просто.

Если у активного окна имеется панель заголовка, то Windows выделяет ее. Если у активного окна вместо панели заголовка имеется рамка диалога (это наиболее часто встречается в окнах диалога), то Windows выделяет ее. Если активное окно минимизировано, то Windows выделяет текст заголовка в панели задач.

Наиболее часто дочерними окнами являются кнопки, переключатели, флажки, полосы прокрутки и списки, которые обычно присутствуют в окне диалога. Сами по себе дочерние окна никогда не могут быть активными.

Если фокус ввода находится в дочернем окне, то активным является родительское окно этого дочернего окна. То, что фокус ввода находится в дочерних окнах, обычно показывается посредством мигающего курсора или каретки (caret).

Если активное окно минимизировано, то окна с фокусом ввода нет. Windows продолжает слать программе сообщения клавиатуры, но эти сообщения выглядят иначе, чем сообщения, направленные активным и еще не минимизированным окнам.

Обрабатывая сообщения WM_SETFOCUS и WM_KILLFOCUS, оконная процедура может определить, когда окно имеет фокус ввода. WM_SETFOCUS показывает, что окно получило фокус ввода, а WM_KILLFOCUS, что окно потеряло его.

Аппаратные и символьные сообщения

Сообщения, которые приложение получает от Windows о событиях, относящихся к клавиатуре, различаются на "аппаратные" (keystrokes) и "символьные" (characters). Такое положение соответствует двум представлениям о клавиатуре. Во-первых, вы можете считать клавиатуру набором клавиш. В клавиатуре имеется только одна клавиша <A>. Нажатие на эту клавишу является аппаратным событием. Отпускание этой клавиши является аппаратным событием. Но клавиатура также является устройством ввода, генерирующем отображаемые символы.

Клавиша <A>, в зависимости от состояния клавиш <Ctrl>, <Shift> и <CapsLock>, может стать источником нескольких символов. Обычно, этим символом является строчное ‘a’. Если нажата клавиша <Shift> или установлен режим Caps Lock, то этим символом является прописное <A>. Если нажата клавиша <Ctrl>, этим символом является <Ctrl>+<A>. На клавиатуре, поддерживающей иностранные языки, аппаратному событию ‘A’ может предшествовать либо специальная клавиша, либо <Shift>, либо <Ctrl>, либо <Alt>, либо их различные сочетания.

Эти сочетания могут стать источником вывода строчного ‘a’ или прописного ‘A’ с символом ударения.

Для сочетаний аппаратных событий, которые генерируют отображаемые символы, Windows посылает программе и оба аппаратных и символьное сообщения. Некоторые клавиши не генерируют символов. Это такие клавиши, как клавиши переключения, функциональные клавиши, клавиши управления курсором и специальные клавиши, такие как Insert и Delete. Для таких клавиш Windows вырабатывает только аппаратные сообщения.

Аппаратные сообщения

Когда вы нажимаете клавишу, Windows помещает либо сообщение WM_KEYDOWN, либо сообщение WM_SYSKEYDOWN в очередь сообщений окна, имеющего фокус ввода. Когда вы отпускаете клавишу, Windows помещает либо сообщение WM_KEYUP, либо сообщение WM_SYSKEYUP в очередь сообщений.

	Клавиша нажата	Клавиша отпущена
Несистемные аппаратные сообщения	WM_KEYDOWN	WM_KEYUP
Системные аппаратные сообщения	WM_SYSKEYDOWN	WM_SYSKEYUP

Обычно сообщения о "нажатии" (down) и "отпускании" (up) появляются парами. Однако, если вы оставите клавишу нажатой так, чтобы включился автоповтор, то Windows посылает оконной процедуре серию сообщений WM_KEYDOWN (или WM_SYSKEYDOWN) и одно сообщение WM_KEYUP (или WM_SYSKEYUP), когда в конце концов клавиша будет отпущена. Также как и все синхронные сообщения, аппаратные сообщения клавиатуры также становятся в очередь. Вы можете с помощью функции *GetMessageTime* получить время нажатия и отпускания клавиши относительно старта системы.

Пример получения сообщений от клавиатуры:

```
case WM_KEYDOWN:
          switch (wParam) {
              case VK_LEFT:
                  .
                  . /* Process the LEFT ARROW key. */
                  .
                  break;
              case VK_RIGHT:
                  .
                  . /* Process the RIGHT ARROW key. */
                  .
                  break;
              case VK_UP:
                  .
                  . /* Process the UP ARROW key. */
                  .
                  break;
              case VK_DOWN:
                  .
                  . /* Process the DOWN ARROW key. */
                  .
                  break;
              case VK_HOME:
                  .
                  . /* Process the HOME key. */
                  .
                  break;
              case VK_END:
                  .
                  . /* Process the END key. */
                  .
                  break;
              case VK_INSERT:
                  .
                  . /* Process the INS key. */
                  .
                  break;
              case VK_DELETE:
                  .
                  . /* Process the DEL key. */
                  .
                  break;
              case VK_F2:
                  .
                  . /* Process the F2 key. */
                  .
                  break;
              .
              . /* Process other noncharacter keystrokes. */
              .
              default:
                  break;
          }
```

2.3. Работа с клавиатурой (C# .NET)

Клавиатура в Windows — разделяемый ресурс. Вводимую информацию все приложения получают с одной и той же клавиатуры, хотя у некоторых клавиш должен быть единственный получатель. В случае программ Windows Forms этим получателем должен быть объект типа Control (класса, в котором реализована обработка клавиатуры) или его потомок, например Form.

Объект, получающий событие клавиатуры, обладает фокусом ввода (input focus). Это понятие тесно связано с понятием активной формы (active form). Обычно активную форму легко определить. Часто это расположенная поверх остальных на рабочем столе форма с подсвеченным заголовком (если таковой есть). Если вместо заголовка у активной формы рамка диалогового окна, то подсвечивается она. Если активная форма в данный момент свернута, ее окно на панели задач отображается как нажатая кнопка.

Активная форма доступна из единственного статического свойства, реализованного в Form.

Клавиатура всегда генерирует некоторые числовые коды, но ее можно рассматривать по-разному:

- как совокупность отдельных физических клавиш;
- как средство генерации кодов символов.

В первом случае любой генерируемый ею код должен идентифицировать клавишу и указывать, нажата она или отпущена. Во втором — код, генерируемый при нажатии некоторой клавиши, идентифицирует уникальный символ из набора символов. Традиционно этим набором символов был ASCII, но в среде Windows Forms поддерживается набор символов Unicode.

У многих клавиш современного компьютера нет кодов символов. Ни функциональные клавиши, ни клавиши управления курсором их не генерируют. Поэтому программам, использующим ввод информации с клавиатуры нетривиальным способом, обычно приходится иметь дело с клавиатурой как с совокупностью клавиш, и как с генератором символом одновременно.

Клавиши можно разделить на четыре общие группы:

- **Клавиши-переключатели** — Caps Lock, Num Lock, Scroll Lock и в некоторых случаях Insert. При нажатии такой клавиши включается ее состояние, а при повторном нажатии — выключается. Состояние клавиш Caps Lock, Num Lock и Scroll Lock является общесистемным, т. е. если на компьютере работает несколько программ одновременно, не может быть, чтобы клавиша Caps Lock была включена для одной программы и в то же время отключена для другой. Как правило, на клавиатуре есть световые индикаторы состояния клавиш -переключателей
- **Клавиши управления регистром** — Shift, Ctrl и Alt. Будучи нажатой, такая клавиша меняет интерпретацию нажатия других клавиш. В библиотеке классов Windows Forms они называются модификаторами.
- **Клавиши, не имеющие символов** — функциональные клавиши, клавиши управления курсором, Pause, Delete и в некоторых случаях Insert.
- **Символьные клавиши** — буквы, цифры и другие символы, пробел, Tab, Backspace, Esc и Enter (клавиши Tab, Backspace, Esc и Enter также можно рассматривать как клавиши без символов).

Значительная часть функциональности клавиатуры реализована в классе Control. Каждый элемент формы при необходимости может обрабатывать события от мыши. Рассмотрим на примере основной формы. Двойным нажатием на событии KeyDown в инспекторе объектов создадим обработчик клавиатуры для формы:

```
private void Form1_KeyDown(object sender, System.Windows.Forms.KeyEventArgs e)
            {
                  if (e.KeyCode==Keys.PageDown)
                  {
                        Close();
                  }
            }
```

При нажатии на главной форме кнопки PageDown программа завершает свою работу. Рассмотрим более сложный пример:

```
private bool nonNumberEntered = false;
private void Form1_KeyDown(object sender, System.Windows.Forms.KeyEventArgs e)
{
    nonNumberEntered = false;
    if (e.KeyCode < Keys.D0 || e.KeyCode > Keys.D9)
    {
        if (e.KeyCode < Keys.NumPad0 || e.KeyCode > Keys.NumPad9)
        {
            if(e.KeyCode != Keys.Back)
            {
                nonNumberEntered = true;
            }
        }
    }
}
```

Как видно, обработка сообщений от клавиатуры не доставляет много хлопот. Поэтому можно считать, что платформа .NET удобно реализует данный механизм. Таким образом, можно отследить нажатие и отпускание любой клавиши при нажатых управляющих клавишах на любом визуальном компоненте формы.

2.4. Работа с таймером (С Windows NT API)

Таймер в Windows является устройством ввода информации, которое периодически извещает приложение о том, что истек заданный интервал времени. Ваша программа задает Windows интервал, как бы говоря системе: "Подталкивай меня каждые 10 секунд". Тогда Windows посылает вашей программе периодические сообщения **WM_TIMER**, сигнализируя об истечении интервала времени.

Применения таймера в Windows, некоторые из которых, может быть, не столь очевидны:

- Многозадачность — хотя Windows NT является вытесняющей многозадачной средой, иногда самое эффективное решение для программы — как можно быстрее вернуть управление Windows. Если программа должна выполнять большой объем работы, она может разделить задачу на части и отрабатывать каждую часть при получении сообщения WM_TIMER.
- Поддержка обновления информации о состоянии — программа может использовать таймер для вывода на экран обновляемой в "реальном времени" (real-time), постоянно меняющейся информации, связанной либо с системными ресурсами, либо с процессом выполнения определенной задачи.
- Реализация "Автосохранение" — таймер может предложить программе для Windows сохранять работу пользователя на диске всегда, когда истекает заданный интервал времени.
- Завершение демонстрационных версий программ — некоторые демонстрационные версии программ рассчитаны на свое завершение, скажем, через 30 минут после запуска. Таймер может сигнализировать таким приложениям, когда их время истекает.
- Задание темпа изменения — графические объекты в играх или окна с результатами в обучающих программах могут нуждаться в задании установленного темпа изменения. Использование таймера устраняет неритмичность, которая могла бы возникнуть из-за разницы в скоростях работы различных микропроцессоров.
- Мультимедиа — программы, которые проигрывают аудиодиски, звук или музыку, часто допускают воспроизведение звуковых данных в фоновом режиме. Программа может использовать таймер для периодической проверки объема воспроизведенной информации и координации его с информацией, выводимой на экран.

Основы использования таймера

Вы можете присоединить таймер к своей программе при помощи вызова функции ***SetTimer***. Функция *SetTimer* содержит целый параметр, задающий интервал, который может находиться в пределах (теоретически) от 1 до 4 294 967 295 миллисекунд, что составляет около 50 дней. Это значение определяет темп, с которым Windows посылает вашей программе сообщения WM_TIMER. Например, интервал в 1000 миллисекунд заставит Windows каждую секунду посылать вашей программе сообщение.

Если в вашей программе есть таймер, то она вызывает функцию ***KillTimer*** для остановки потока сообщений от таймера. Вы можете запрограммировать "однократный" таймер, вызывая функцию *KillTimer* при обработке сообщения WM_TIMER. Вызов функции *KillTimer* очищает очередь

сообщений от всех необработанных сообщений WM_TIMER. После вызова функции *KillTimer* ваша программа никогда не получит случайного сообщения WM_TIMER.

Также можно сделать, чтобы через определенные интервалы времени выполнялись процедура обработки таймера. Покажем это на примере – здесь через одну секунду динамик подает сигнал:

```
...
#define IDT_MOUSETRAP 2000
...
UINT uResult;               // SetTimer's return value
...
VOID CALLBACK MyTimerProc(
    HWND hwnd,         // handle to window for timer messages
    UINT message,      // WM_TIMER message
    UINT idTimer,      // timer identifier
    DWORD dwTime);     // current system time
int APIENTRY _tWinMain(HINSTANCE hInstance,
                       HINSTANCE hPrevInstance,
                       LPTSTR    lpCmdLine,
                       int       nCmdShow)
...
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
      ...
      case WM_CREATE:
            uResult = SetTimer(hWnd,            // handle to main window
                               IDT_MOUSETRAP,   // timer identifier
                               1000,            // 1-second interval
                               (TIMERPROC) MyTimerProc);  // timer callback
      ...
      case WM_DESTROY:
            KillTimer(hWnd, IDT_MOUSETRAP);
}
...
VOID CALLBACK MyTimerProc(
    HWND hwnd,        // handle to window for timer messages
    UINT message,     // WM_TIMER message
    UINT idTimer,     // timer identifier
    DWORD dwTime)     // current system time
{
  Beep(55,55);
}
```

2.4. Работа с таймером (C# .NET)

В пространствах имен System.Timers, System.Threading и System.Windows.Forms определены три разных класса с именем Timer.

Таймер можно рассматривать и как гарант, обеспечивающий передачу управления программе. Если после выполнения конструктора или обработчика события программа отдает управление, обычно нельзя определить, когда произойдет следующее событие. В этом смысле событие таймера более определенно.

Я сказал более, потому что таймер не обладает ритмичностью метронома. События, вырабатываемые классом Timer, синхронизированы с другими событиями.

Иначе говоря, событие таймера никогда не прервет процедуру обработки другого события, выполняющуюся в том же потоке. Если обработка события идет слишком долго, она задержит событие таймера.

Рассмотрим на примере работу таймера. Расположим на форме компонент Timer. В инспекторе объектов зададим необходимые свойства, например, интервал срабатывания таймера. Перейдем на вкладку событий и в свойстве Elapsed укажем процедуру обработки таймера (timer1_Elapsed). Код:

```
private void timer1_Elapsed(object sender, System.Timers.ElapsedEventArgs e)
          {
                pictureBox1.Left++;
          }
```

Теперь объект pictureBox1 будет двигаться вправо через заданные промежутки времени. Это пример автоматического создания конструкторов, т.е. среда за нас все сделала сама. Приведем пример ручного создания объекта и процедуры обработки:

```
public class Timer2
 {
     public static void Main()
     {
       System.Timers.Timer aTimer = new System.Timers.Timer(10000);
       aTimer.Elapsed+=new ElapsedEventHandler(OnTimedEvent);
       aTimer.AutoReset = false;
       aTimer.Enabled = true;
       Console.WriteLine("Press \'q\' to quit the sample.");
       while(Console.Read()!='q');
     }
     private static void OnTimedEvent(object source, ElapsedEventArgs e)
     {
       Console.WriteLine("Hello World!");
     }
 }
```

2.5. Работа с мышью (С Windows NT API)

Сообщения мыши: оконная процедура получает сообщения мыши и когда мышь проходит через окно и при щелчке внутри окна, даже если окно неактивно или не имеет фокуса ввода. В Windows для мыши определен набор из 21 сообщения. Однако, 11 из этих сообщений не относятся к рабочей области, и программы для Windows обычно игнорируют их.

Если мышь перемещается по рабочей области окна, оконная процедура получает сообщение **WM_MOUSEMOVE.**

Если кнопка мыши нажимается или отпускается внутри рабочей области окна, оконная процедура получает следующие сообщения:

Кнопка	Нажатие	Отпускание	Нажатие (Второй щелчок)
Левая	WM_LBUTTONDOWN	WM_LBUTTONUP	WM_LBUTTONDBLCLK
Средняя	WM_MBUTTONDOWN	WM_MBUTTONUP	WM_MBUTTONDBLCLK
Правая	WM_RBUTTONDOWN	WM_RBUTTONUP	WM_RBUTTONDBLCLK

Для всех этих сообщений значение параметра ***lParam*** содержит положение мыши. Младшее слово — это координата *x*, а старшее слово — координата *y* относительно верхнего левого угла рабочей области окна.

Вы можете извлечь координаты *x* и *y* из параметра *lParam* с помощью макросов LOWORD и HIWORD, определенных в заголовочных файлах Windows.

Значение параметра ***wParam*** показывает состояние кнопок мыши и клавиш <Shift> и <Ctrl>. Вы можете проверить параметр *wParam* с помощью битовых масок, определенных в заголовочных файлах. Префикс MK означает "клавиша мыши" (mouse key).

MK_LBUTTON	Левая кнопка нажата
MK_MBUTTON	Средняя кнопка нажата
MK_RBUTTON	Правая кнопка нажата
MK_SHIFT	Клавиша <Shift> нажата
MK_CONTROL	Клавиша <Ctrl> нажата

Пример рисования линий мышью:

```
LRESULT APIENTRY MainWndProc(hwndMain, uMsg, wParam, lParam)
HWND hwndMain;
UINT uMsg;
WPARAM wParam;
LPARAM lParam;
{
    HDC hdc;
    RECT rcClient;
    POINT ptClientUL;
    POINT ptClientLR;
    static POINTS ptsBegin;
    static POINTS ptsEnd;
    static POINTS ptsPrevEnd;
    static BOOL fPrevLine = FALSE;
    switch (uMsg) {
        case WM_LBUTTONDOWN:
            SetCapture(hwndMain);
            GetClientRect(hwndMain, &rcClient);
```

```
            ptClientUL.x = rcClient.left;
            ptClientUL.y = rcClient.top;
            ptClientLR.x = rcClient.right + 1;
            ptClientLR.y = rcClient.bottom + 1;
            ClientToScreen(hwndMain, &ptClientUL);
            ClientToScreen(hwndMain, &ptClientLR);
            SetRect(&rcClient, ptClientUL.x, ptClientUL.y,
                ptClientLR.x, ptClientLR.y);
            ClipCursor(&rcClient);
            ptsBegin = MAKEPOINTS(lParam);
            return 0;
        case WM_MOUSEMOVE:
            if (wParam & MK_LBUTTON) {
                hdc = GetDC(hwndMain);
                SetROP2(hdc, R2_NOTXORPEN);
                if (fPrevLine) {
                    MoveToEx(hdc, ptsBegin.x, ptsBegin.y,
                        (LPPOINT) NULL);
                    LineTo(hdc, ptsPrevEnd.x, ptsPrevEnd.y);
                }
                ptsEnd = MAKEPOINTS(lParam);
                MoveToEx(hdc, ptsBegin.x, ptsBegin.y,(LPPOINT) NULL);
                LineTo(hdc, ptsEnd.x, ptsEnd.y);
                fPrevLine = TRUE;
                ptsPrevEnd = ptsEnd;
                ReleaseDC(hwndMain, hdc);
            }
            break;
        case WM_LBUTTONUP:
            fPrevLine = FALSE;
            ClipCursor(NULL);
            ReleaseCapture();
            return 0;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        ...
}
```

2.5. Работа с мышью (C# .NET)

Есть множество событий, связанных с мышью. Каждое из этих событий имеет обработчик, для которого вы можете написать код в приложениях. Эти события включают MouseDown, MouseUp, MouseMove, MouseEnter, MouseLeave, MouseHover. MouseDown, MouseUp, и MouseMove получают аргумент MouseEventArgs содержащий данные, связанные с их событиями. MouseEnter, MouseLeave, и MouseHover получает аргумент типа EventArgs содержащий данные, связанные с их событиями. Эти события обрабатываются точно так же как любые другие события, которые происходят на формах.

Дополнительно вы можете использовать курсор мыши во время работы. Эта способность может быть соединена с MouseEnter и MouseLeave событиями, чтобы и обеспечить обратную связь пользователю. Например, если Вы показываете диалог, чтобы указать, что файлы копируются, общепринято заменить курсор на песочные часы.

Для обработки сообщений от мыши у формы существуют следующие события:

MouseDown — обработка нажатия какой-либо из кнопок вниз;
MouseEnter — вызывается при попадании указателя мыши в область формы;
MouseHover — вызывается при зависании указателя мыши в окне формы;
MouseLeave — вызывается при покидании курсора мыши области формы;
MouseMove — вызывается при движении мыши в области формы;
MouseUp — вызывается при отпускании кнопки мыши.

Пример работы для обработки события от мыши при нажатии на форме:

```
private void Form1_MouseDown(object sender, System.Windows.Forms.MouseEventArgs e)
           {
             if (e.Button == MouseButtons.Left)
               textBox1.Text =  e.X.ToString();
             if (e.Button == MouseButtons.Right)
               textBox1.Text =  e.Y.ToString();
           }
```

2.6. Буфер обмена (С Windows NT API)

Буфер обмена (**clipboard**) в Windows дает возможность передавать данные от одной программы к другой. Это относительно простой механизм, не требующий больших добавлений ни к программе, которая помещает данные в буфер обмена, ни к программе, которая в дальнейшем их получает. В большинстве программ, имеющих дело с документами или другими данными, имеется меню Edit с опциями Cut, Copy и Paste. Если пользователь выбирает опции Cut или Copy, то программа передает данные в буфер обмена. Эти данные находятся в специальном формате, например в виде текста, битового образа или метафайла. Если пользователь выбирает в меню опцию Paste, программа проверяет имеются ли в буфере обмена данные в том формате, который программа может использовать, и если да, то эти данные передаются из буфера обмена в программу.

Программы не должны передавать данные в буфер обмена и получать их оттуда без совершенно точно определенной инструкции пользователя. Например, пользователь, выполняющий в одной программе операцию Cut или Copy, должен быть уверен, что эти данные будут оставаться в буфере обмена вплоть до следующей операции Cut или Copy.

Простое использование буфера обмена

Мы начнем с анализа программы для передачи данных в буфер обмена (Cut и Copy) и получения данных из буфера обмена (Paste).

Стандартные форматы данных буфера обмена Windows поддерживает стандартные форматы данных буфера обмена, идентификаторы которых находятся в заголовочных файлах Windows. Наиболее часто используемыми из них являются:

- CF_TEXT — оканчивающаяся нулем группа символов из набора символов ASCII, в конце каждой строки которой имеются символы возврата каретки и перевода строки. Это простейший формат данных буфера обмена. Передаваемые в буфер обмена данные хранятся в области оперативной памяти, а передаются они с помощью описателя этой области памяти. Эта область памяти становится неотъемлемой частью буфера обмена, и программа, создавшая этот блок памяти, больше не должна его использовать.
- CF_BITMAP — зависящий от устройства битовый образ. Битовый образ передается в буфер обмена с помощью описателя битового образа. И в этом случае программа, после передачи битового образа в буфер обмена, не должна его больше использовать.
- CF_METAFILEPICT — "картинка метафайла" (metafile picture). Это не совсем то же самое, что метафайл. Скорее это метафайл, содержащий дополнительную информацию в виде небольшой структуры типа METAFILEPICT. Программа передает картинку метафайла в буфер обмена с помощью описателя области памяти, содержащего эту структуру. В структуре METAFILEPICT имеется четыре поля: mm (LONG), режим отображения метафайла; xExt (LONG) и yExt (LONG), ширина и высота образа метафайла; hMF (HMETAFILE), описатель метафайла. После того, как программа передаст в буфер обмена картинку метафайла, она не должна продолжать использовать ни область памяти, содержащую структуру METAFILEPICT, ни описатель метафайла, поскольку и тем, и другим теперь будет управлять Windows.

● CF_ENHMETAFILE — описатель расширенного метафайла. В этом формате структура METAFILEPICT не используется.

● CF_SYLK — блок памяти, содержащий данные в формате Microsoft Symbolic Link (SYLK). Этот формат используется для обмена данными между программами Multiplan, Chart и Excel. Это ASCII-формат, в котором каждая строка завершается символами возврата каретки и перевода строки.

● CF_DIF — блок памяти, содержащий данные в формате Data Interchange Format (DIF). Этот формат предложен компанией Software Arts для передачи данных в программу электронных таблиц VisiCalc. Теперь этот формат находится под управлением корпорации Lotus. Это также ASCII-формат, в котором каждая строка завершается символами возврата каретки и перевода строки. Форматы CF_SYLK и CF_DIF концептуально похожи на формат CF_TEXT. Однако, символьные строки в форматах SYLK или DIF необязательно заканчиваются нулевым символом, поскольку эти форматы определяют конец данных.

● CF_TIFF — блок памяти, содержащий данные в формате Tag Image File Format (TIFF). Этот формат предложен корпорациями Microsoft, Aldus и компанией Hewlett-Packard в сотрудничестве с некоторыми производителями аппаратуры. Этот формат (описывающий данные битовых изображений) поддерживается компанией Hewlett-Packard.

● CF_OEMTEXT — блок памяти, содержащий текстовые данные (как в формате CF_TEXT), но из набора символов OEM.

● CF_DIB — блок памяти, определяющий независящий от устройства битовый образ. Блок памяти начинается со структуры BITMAPINFO, за которой следуют биты битового образа.

● CF_PALETTE — описатель цветовой палитры. Обычно используется в сочетании с CF_DIB для определения цветовой палитры, используемой битовым образом.

Передача текста в буфер обмена

Предположим, что в буфер обмена необходимо передать символьную строку, и, что имеется указатель (pString) на эту строку. Передать необходимо iLength байт этой строки.

Во-первых, используя функцию GlobalAlloc, выделяем перемещаемый блок памяти размером iLength, резервируя место для символа окончания NULL:

```
hGlobalMemory = GlobalAlloc(GHND, iLength + 1);
```

Значение hGlobalMemory будет равно NULL, если память выделена быть не может. Если выделение памяти прошло удачно, то для получения указателя на выделенный блок необходимо его зафиксировать:

```
pGlobalMemory = GlobalLock(hGlobalMemory);
```

Теперь копируем символьную строку в зафиксированную область памяти:

```
for(i = 0; i < iLength; i ++)
*pGlobalMemory++ = *pString++;
```

В конец строки NULL-символ помещать не надо, поскольку установка флага GHND в функции GlobalAlloc обнуляет всю выделяемую область памяти. После этого снимаем фиксацию области памяти:

```
GlobalUnlock(hGlobalMemory);
```

Теперь у нас имеется описатель области памяти, в которой находится оканчивающийся нулевым символом текст.

Для помещения его в буфер обмена открываем и очищаем ее:

```
OpenClipboard(hwnd);
EmptyClipboard();
```

Используя идентификатор CF_TEXT, передаем буферу обмена описатель области памяти и закрываем буфер обмена:

```
SetClipboardData(CF_TEXT, hGlobalMemory);
CloseClipboard();
```

Готово.

Ниже приведено несколько правил, регулирующих этот процесс:

- Вызовы функций OpenClipboard и CloseClipboard делаются во время обработки одного сообщения. Буфер обмена не оставляют открытой дольше, чем это необходимо.
- Буферу обмена не передается описатель зафиксированной области памяти.
- После вызова функции SetClipboardData область памяти использовать нельзя. Она больше не принадлежит программе, и описатель следует считать недействительным. Если доступ к данным по-прежнему необходим, нужно сделать их копию или считать их из буфера обмена (как это описано в следующем разделе). Между вызовами функций SetClipboardData и CloseClipboard можно ссылаться на выделенную область памяти, но при этом нужно использовать описатель, возвращаемый функцией SetClipboardData.

Перед вызовом функции CloseClipboard необходимо освободить этот описатель.

Получение текста из буфера обмена

Получение текста из буфера обмена лишь чуть-чуть сложнее передачи текста в нее. Сначала необходимо определить, действительно ли в буфере обмена содержатся данные в формате CF_TEXT. Один из простейших способов сделать это заключается в вызове функции IsClipboardFormatAvailable:

```
bAvailable = IsClipboardFormatAvailable(CF_TEXT);
```

Функция возвращает TRUE (ненулевое значение), если в буфере обмена находятся данные в формате CF_TEXT. Функция IsClipboardFormatAvailable — это одна из нескольких функций буфера обмена, которую можно использовать без предварительного открытия буфера обмена. Однако, если позже для получения этого текста открыть буфер обмена, необходимо снова проверить (используя эту же функцию или другие способы) по-прежнему ли в буфере обмена находятся данные в формате CF_TEXT.

Для получения данных из буфера обмена сначала открываем ее:

```
OpenClipboard(hwnd);
```

Затем получаем описатель области памяти, содержащей текст:

```
hClipMemory = GetClipboardData(CF_TEXT);
```

Этот описатель будет равен NULL, если в буфере обмена отсутствуют данные в формате CF_TEXT. Это второй способ определения наличия текста в буфере обмена. Если возвращаемое значение функции GetClipboardData равно NULL, буфер обмена следует закрыть, ничего с ней не делая.

Описатель, возвращаемый функцией GetClipboardData, не принадлежит вашей программе — он принадлежит буферу обмена. Описателем можно

пользоваться только между вызовами функций GetClipboardData и CloseClipboard. Нельзя освободить этот описатель или изменить данные, на которые он ссылается. Если доступ к этим данным необходим, то нужно сделать копию этой области памяти.

Далее предлагается один из способов копирования данных в программу. Сначала необходимо выделить область памяти точно такого же размера, как область памяти буфера обмена:

```
pMyCopy =(char *) malloc(GlobalSize(hClipMemory));
```

Теперь, для получения указателя на область памяти буфера обмена, необходимо зафиксировать описатель области памяти буфера обмена:

```
pClipMemory = GlobalLock(hClipMemory);
```

Теперь можно копировать данные:

```
strcpy(pMyCopy, pClipMemory);
```

Или можно использовать какой-нибудь простой код на языке С:

```
while(*pMyCopy++ = *pClipMemory++);
```

Перед закрытием буфера обмена необходимо снять фиксацию области памяти:

```
GlobalUnlock(hClipMemory);
CloseClipboard();
```

Итак, получен указатель pMyCopy, который позже можно зафиксировать для получения доступа к этим данным.

Открытие и закрытие буфера обмена

В любой момент времени только одна программа может держать буфер обмена открытой. Цель вызова функции OpenClipboard состоит в предотвращении возможности изменения содержимого буфера обмена в то время, пока программа ее использует. Типом возвращаемого значения функции OpenClipboard является BOOL, показывающее, была ли буфер обмена открыта успешно. Она не будет открыта, если в другом приложении она не закрыта. Если в каждой программе буфер обмена будет аккуратно открываться и закрываться настолько быстро, насколько это возможно, то вы никогда не столкнетесь с проблемой невозможности ее открыть.

Однако, в мире неаккуратных программ и вытесняющей многозадачности такие проблемы могут возникать. Даже если программа не теряет фокус ввода между временем помещения чего-нибудь в буфер обмена и тем временем, когда пользователь выбирает опцию Paste, не надейтесь, что в буфере обмена находится то, что было туда помещено. В это время доступ к буферу обмена мог иметь фоновый процесс.

Кроме этого, опасайтесь более хитрых проблем, связанных с окнами сообщений: если нельзя выделить достаточно памяти, чтобы скопировать что-нибудь в буфер обмена, то можно вывести на экран окно сообщения. Однако, если это окно сообщения не является системным модальным окном, пользователь может переключиться на другое приложение, пока окно сообщений остается на экране. Необходимо либо делать окно сообщения системы модальным, либо закрывать буфер обмена перед выводом на экран окна сообщения.

Также могут возникнуть проблемы, если оставить буфер обмена открытой при выводе на экран окна диалога. Поля редактирования в окне диалога используют буфер обмена для вырезания и вставки текста.

Пример получения текста из буфера обмена. При выборе в меню пункта происходит вывод в окно содержимого буфера – текста.

```
#include "stdafx.h"
#include “string.h”
#include "CLIP.h"
#define MAX_LOADSTRING 100
char * pMyCopy;
HINSTANCE hInst;
TCHAR szTitle[MAX_LOADSTRING];
TCHAR szWindowClass[MAX_LOADSTRING];
ATOM  MyRegisterClass(HINSTANCE hInstance);
BOOL  InitInstance(HINSTANCE, int);
LRESULT CALLBACK  WndProc(HWND, UINT, WPARAM, LPARAM);
int APIENTRY _tWinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,
LPTSTR lpCmdLine, int nCmdShow)
{...}
ATOM MyRegisterClass(HINSTANCE hInstance)
{...}
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{...}
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
      int wmId, wmEvent;
      PAINTSTRUCT ps;
      HDC hdc;
      switch (message)
      {
      case WM_COMMAND:
            wmId    = LOWORD(wParam);
            wmEvent = HIWORD(wParam);
            switch (wmId)
            {
             case IDM_ABOUT:
              bool bAvailable;
              HANDLE hClipMemory;
              char *  pClipMemory;
              bAvailable = IsClipboardFormatAvailable(CF_TEXT);
              if (bAvailable)
              {
                OpenClipboard(hWnd);
                hClipMemory = GetClipboardData(CF_TEXT);
                pMyCopy =(char *) malloc(GlobalSize(hClipMemory));
                pClipMemory = (char *) GlobalLock(hClipMemory);
                strcpy(pMyCopy, pClipMemory);
                GlobalUnlock(hClipMemory);
                CloseClipboard();
                InvalidateRect(hWnd,NULL,true);
                UpdateWindow(hWnd);
              }
              break;
            default:
                  return DefWindowProc(hWnd, message, wParam, lParam);
            }
            break;
      case WM_PAINT:
            hdc = BeginPaint(hWnd, &ps);
            TextOut(hdc,1,1,pMyCopy,strlen(pMyCopy));
            EndPaint(hWnd, &ps);
            break;
      case WM_CREATE:
            pMyCopy = "";
            break;
}
```

2.6. Буфер обмена (C# .NET)

Для работы с буфером обмена нужно использовать Clipboard класс. Вызовите **SetDataObject**, чтобы сохранить данные clipboard. Чтобы размещать копию данных clipboard, установите параметр **copy** в true.

Вызовите **GetDataObject**, чтобы восстановить данные clipboard. Данные возвращаются как объект, который осуществляет интерфейс **IDataObject**. Используйте методы, указанные IDataObject и полем в DataFormats, чтобы извлечь данные из объекта. Если вы не знаете формат данных, вы должны вызывать GetFormats метод интерфейса IDataObject, чтобы получить список всех форматов, в которых находятся данные. Тогда вызовите **GetData** метод интерфейса IDataObject, и определите формат, который ваше приложение может использовать.

Пример для работы с буфером обмена:

```
private void button1_Click(object sender, System.EventArgs e)
            {
                  if(textBox1.SelectedText != "")
                        Clipboard.SetDataObject(textBox1.SelectedText);
                  else
                        textBox2.Text = "No text selected in textBox1";
            }
private void button2_Click_1(object sender, System.EventArgs e)
            {
                  IDataObject iData = Clipboard.GetDataObject();
                  if(iData.GetDataPresent(DataFormats.Text))
                  {
                   textBox2.Text = (String)iData.GetData(DataFormats.Text);
                  }
                  else
                  {
                   textBox2.Text = "Could not retrieve data off the clipboard.";
                  }
            }
```

2.7. Многопоточность (С Windows NT API)

Каждый поток начинает выполнение с некоей входной функции. В первичном потоке таковой является ***main, wmain, WinMain*** или ***wWinMain.*** Если Вы хотите создать вторичный поток, в нем тоже должна быть входная функция, которая выглядит примерно так:

```
DWORD WINAPI ThreadFunc(PVOID pvParam)
{
DWORD rtwResult = 0;
return(dwResult);
}
```

Функция потока может выполнять любые задачи. Рано или поздно она закончит свою работу и вернет управление. В этот момент ваш поток остановится, память, от веденная под его стек, будет освобождена, а счетчик пользователей его объекта ядра "поток" уменьшится на 1. Когда счетчик обнулится, этот объект ядра будет разрушен. Но, как и объект ядра "процесс", он может жить гораздо дольше, чем сопоставленный с ним поток.

А теперь поговорим о самых важных вещах, касающихся функций потоков:

- В отличие от входной функции первичного потока, у которой должно быть одно из четырех имен: ***main, wmain, WinMain*** или ***wWinMain,*** — функцию потока можно назвать как угодно. Однако, если в программе несколько функций потоков, вы должны присвоить им разные имена, иначе компилятор или компоновщик решит, что вы создаете несколько реализаций единственной функции.
- Поскольку входным функциям первичного потока передаются строковые параметры, они существуют в ANSI- и Unicode-версиях: ***main - wmain*** и ***WinMain — wWinMain.*** Но функциям потоков передается единственный параметр, смысл которого определяется вами, а не операционной системой. Поэтому здесь нет проблем с ANSI/Unicode.
- Функция потока должна возвращать значение, которое будет использоваться как код завершения потока. Здесь полная аналогия с библиотекой C/C++: код завершения первичного потока становится кодом завершения процесса.
- Функции потоков (да и все ваши функции) должны по мере возможности обходиться своими параметрами и локальными переменными. Так как к статической или глобальной переменной могут одновременно обратиться несколько потоков, есть риск повредить ее содержимое. Однако параметры и локальные переменные создаются в стеке потока, поэтому они в гораздо меньшей степени подвержены влиянию другого потока.

Вот вы и узнали, как должна быть реализована функция потока. Теперь рассмотрим, как заставить операционную систему создать поток, который выполнит эту функцию.

Если Вы хотите создать дополнительные потоки, нужно вызывать из первичного потока функцию ***CreateThread:***

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    DWORD dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId
   );
```

Завершение потока

Поток можно завершить четырьмя способами:

- функция потока возвращает управление (рекомендуемый способ),
- поток самоуничтожается вызовом функции ***ExitThread*** (нежелательный способ);
- один из потоков данного или стороннего процесса вызывает функцию ***TerminateThread*** (нежелательный способ);
- завершается процесс, содержащий данный поток (тоже нежелательно).

Возврат управления функцией потока

Функцию потока следует проектировать так, чтобы поток завершался только после того, как она возвращает управление. Это единственный способ, гарантирующий корректную очистку всех ресурсов, принадлежавших Вашему потоку. При этом:

- любые C++ объекты, созданные данным потоком, уничтожаются соответствующими деструкторами;
- система корректно освобождает память, которую занимал стек потока;
- система устанавливает код завершения данного потока (поддерживаемый объектом ядра "поток») – его и возвращает Ваша функция потока;
- счетчик пользователей данного объекта ядра "поток" уменьшается на 1

Что происходит при завершении потока

- Освобождаются все описатели User-объектов, принадлежавших потоку. В Windows большинство объектов принадлежит процессу, содержащему поток, из которого они были созданы. Сам поток владеет только двумя User-объектами, окнами и ловушками (hooks). Когда поток, создавший такие объекты, завершается, система уничтожает их автоматически. Прочие объекты разрушаются, только когда завершается владевший ими процесс.
- Код завершения потока меняется со STILL_ACTIVE на код, переданный в функцию *ExitThread* или *TerminateThread.*
- Объект ядра "поток" переводится в свободное состояние.
- Если данный поток является последним активным потоком в процессе, завершается и сам процесс.
- Счетчик пользователей объекта ядра "поток" уменьшается на 1.

Пример работы с потоками:

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#define MM_NEWWIN       8001
typedef struct _PTHREADLIST
{
   HANDLE hThread;
   LPVOID lpvNext;
} THREADLIST, *PTHREADLIST;
HANDLE hModule;            // handle to .EXE file for this process
HWND   hwndMain = NULL;    // handle to main window
BOOL fKillAll = FALSE;     // sets TRUE to terminate all threads
PTHREADLIST pHead = NULL;  // head of thread information linked list
BOOL InitializeApp(VOID);
LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK ChildWndProc(HWND, UINT, WPARAM, LPARAM);
DWORD ThreadFunc(HWND);
VOID AddThreadToList(HANDLE);
VOID ErrorExit(LPSTR);
// Primary thread:  Initialize the application and dispatch messages.
int WINAPI WinMain( HINSTANCE hInst,
   HINSTANCE hPrevInst,
   LPSTR lpszCmdLn,
   int nShowCmd)
{
   MSG msg;
   hModule = GetModuleHandle(NULL);
   if (! InitializeApp())
      ErrorExit("InitializeApp failure!");
   while (GetMessage(&msg, NULL, 0, 0))
   {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
   }
   return 1;
   UNREFERENCED_PARAMETER(hInst);
   UNREFERENCED_PARAMETER(hPrevInst);
   UNREFERENCED_PARAMETER(lpszCmdLn);
   UNREFERENCED_PARAMETER(nShowCmd);
}
// Register window classes and create the main window.
BOOL InitializeApp(VOID)
{
   HMENU hmenuMain, hmenuPopup;
   WNDCLASS wc;
// Register a window class for the main window.
   wc.style            = CS_OWNDC;
   wc.lpfnWndProc      = MainWndProc;
   wc.cbClsExtra       = 0;
   wc.cbWndExtra       = 0;
   wc.hInstance        = hModule;
   wc.hIcon            = LoadIcon(NULL,IDI_APPLICATION);
   wc.hCursor          = LoadCursor(NULL, IDC_ARROW);
   wc.hbrBackground    = (HBRUSH)(COLOR_BACKGROUND+1);
   wc.lpszMenuName     = NULL;
   wc.lpszClassName    = "MainWindowClass";
   if (! RegisterClass(&wc))
      return FALSE;
// Register a window class for child windows.
   wc.lpfnWndProc      = ChildWndProc;
   wc.lpszClassName    = "ThreadWindowClass";
   if (! RegisterClass(&wc))
      return FALSE;
// Create a menu for the main window.
   hmenuMain = CreateMenu();
   hmenuPopup = CreateMenu();
   if (!AppendMenu(hmenuPopup, MF_STRING, MM_NEWWIN, "&New Window"))
      return FALSE;
   if (!AppendMenu(hmenuMain, MF_POPUP, (UINT)hmenuPopup, "&Threads"))
      return FALSE;
// Create the main window.
   hwndMain = CreateWindow("MainWindowClass", "Primary Window",
      WS_OVERLAPPED | WS_CAPTION | WS_BORDER | WS_THICKFRAME |
      WS_MAXIMIZEBOX | WS_MINIMIZEBOX | WS_CLIPCHILDREN |
      WS_VISIBLE | WS_SYSMENU, CW_USEDEFAULT, CW_USEDEFAULT,
      CW_USEDEFAULT, CW_USEDEFAULT, NULL, hmenuMain, hModule,
      NULL);
   if (hwndMain == NULL)
      return FALSE;
```

```
// Set the initial focus.
   SetFocus(hwndMain);
   return TRUE;
}
// Main window procedure: Handle messages for the main window.
LRESULT CALLBACK MainWndProc(HWND hwnd, UINT uiMessage,
   WPARAM wParam, LPARAM lParam)
{
   static HWND hwndClient;
   static DWORD dwCount = 1;
   CLIENTCREATESTRUCT ccsClientCreate;
   HWND hwndChildWnd;
   DWORD IDThread;
   PTHREADLIST pNode;
   switch (uiMessage)
   {
   // Create a client window to receive child window messages.
      case WM_CREATE:
         ccsClientCreate.hWindowMenu = NULL;
         ccsClientCreate.idFirstChild = 1;
         hwndClient = CreateWindow("MDICLIENT", NULL,
            WS_CHILD | WS_CLIPCHILDREN | WS_VISIBLE, 0, 0, 0, 0,
            hwnd, NULL, hModule, (LPVOID)&ccsClientCreate);
         return 0L;
   // Close the main window. First set fKillAll to TRUE to
   // terminate all threads. Then wait for the threads to exit
   // before passing a close message to a default handler.  If you
   // don't wait for threads to terminate, the process terminates
   // with no chance for thread cleanup.
      case WM_CLOSE:
         fKillAll = TRUE;
         pNode = pHead;
         while (pNode)
         {
            DWORD dwRes;
            SetThreadPriority(pNode->hThread,
               THREAD_PRIORITY_HIGHEST);
            dwRes = WaitForSingleObject(pNode->hThread,
               INFINITE);
            pNode = (PTHREADLIST) pNode->lpvNext;
         }
         return DefFrameProc(hwnd, hwndClient, uiMessage,
            wParam, lParam);
   // Terminate the process.
      case WM_DESTROY:
         PostQuitMessage(0);
         return 0L;
   // Handle the menu commands.
      case WM_COMMAND:
         switch (LOWORD(wParam))
         {
         // Create a child window and start a thread for it.
            case MM_NEWWIN:
               HANDLE hThrd;
               MDICREATESTRUCT mdicCreate;
               TCHAR tchTitleBarText[32];
               LONG lPrev;
               sprintf(tchTitleBarText, "Thread Window %d", dwCount);
               mdicCreate.szClass = "ThreadWindowClass";
               mdicCreate.szTitle = tchTitleBarText;
               mdicCreate.hOwner  = hModule;
               mdicCreate.x = mdicCreate.y =
               mdicCreate.cx = mdicCreate.cy = CW_USEDEFAULT;
               mdicCreate.style = mdicCreate.lParam = 0L;
            // Send a "create child window" message to the
            // client window.
               hwndChildWnd = (HWND) SendMessage(hwndClient,
                  WM_MDICREATE, 0L, (LONG)&mdicCreate);
               if (hwndChildWnd == NULL)
                  ErrorExit("Failed in Creating Thread Window!");
            // Window structure used to pass a quit message to
            // the thread.
               lPrev = SetWindowLong(hwndChildWnd, GWL_USERDATA, 0);
            // Create a suspended thread; alter its priority before
            // calling ResumeThread.
               hThrd = CreateThread(NULL,  // no security attributes
                         0,                // use default stack size
                         (LPTHREAD_START_ROUTINE) ThreadFunc,
                         (LPVOID)hwndChildWnd, // param to thread func
```

```
                        CREATE_SUSPENDED, // creation flag
                        &IDThread);       // thread identifier
               if (hThrd == NULL)

                  ErrorExit("CreateThread Failed!");
               AddThreadToList(hThrd);
               dwCount++;
            // Set the priority lower than the primary (input)
            // thread, so the process is responsive to user
            // input.  Then resume the thread.
               if (!SetThreadPriority(hThrd,
                  THREAD_PRIORITY_BELOW_NORMAL))
                  ErrorExit("SetThreadPriority failed!");
               if ((ResumeThread(hThrd)) == -1)

                  ErrorExit("ResumeThread failed!");
               return 0L;
            default:
               return DefFrameProc(hwnd, hwndClient, uiMessage,
                  wParam, lParam);
         }
      default:
         return DefFrameProc(hwnd, hwndClient, uiMessage,
            wParam, lParam);
   }
}
// Process messages for the child windows.
LRESULT CALLBACK ChildWndProc(HWND hwnd, UINT uiMessage, WPARAM
   wParam, LPARAM lParam)

{
   LONG lPrevLong;
   switch (uiMessage)
   {
   // Use a window structure to pass "close" message to thread.
      case WM_CLOSE:
         lPrevLong = SetWindowLong(hwnd, GWL_USERDATA, 1);
         return DefMDIChildProc(hwnd, uiMessage, wParam, lParam);
      case WM_DESTROY:
         return 0L;
      default:
         return DefMDIChildProc(hwnd, uiMessage, wParam, lParam);
   }
}
// Each child window has a thread that can be used to perform tasks
// associated with that window--for example, drawing its contents.
DWORD ThreadFunc(HWND hwnd)
{
   LONG lKillMe = 0L;
   while (TRUE)
   {
      lKillMe = GetWindowLong(hwnd, GWL_USERDATA);
      if (fKillAll || lKillMe) break;
   // Perform tasks.
   }
   // Perform actions needed before thread termination.
   return 0;
}
VOID AddThreadToList(HANDLE hThread)
{
   PTHREADLIST pNode;
   pNode = (PTHREADLIST) LocalAlloc(LPTR, sizeof(PTHREADLIST));
   if (pNode == NULL)
      ErrorExit("malloc Failed!");
   pNode->hThread = hThread;
   pNode->lpvNext = (LPVOID) pHead;
   pHead = pNode;
}
VOID ErrorExit(LPSTR lpszMessage)
{
   MessageBox(hwndMain, lpszMessage, "Error", MB_OK);
   ExitProcess(0);
}
```

2.7. Многопоточность (C# .NET)

Для работы с потоками существует класс Thread. Он имеет методы и свойства. Создает и управляет нитью, устанавливает ее приоритет, и получает ее статус.

Процесс может создавать одну или более нитей, чтобы выполнить часть кода программы, связанного с процессом. Используйте делегат **ThreadStart**, чтобы определить коде программы, выполняемый нитью. Можно определить состояние нити (Sleep, Wait, Join и т.д.), определенных **ThreadState**. Уровень приоритета нити - **ThreadPriority**. **GetHashCode** обеспечивает идентификацию для управляемых нитей.

Данная программа создает, запускает и прерывает нити. На форме расположены два компонента – изображений. При запуске потоков – оба начинают двигаться.

```
Alpha oAlpha,oAlpha1;
Thread tr1,tr2;
private void button1_Click(object sender, System.EventArgs e)
            {
            oAlpha = new Alpha();
            oAlpha.pb=pictureBox1;
            tr1= new Thread(new ThreadStart(oAlpha.Beta));
            tr1.Start();
            oAlpha1 = new Alpha();
            oAlpha1.pb=pictureBox2;
            tr2= new Thread(new ThreadStart(oAlpha1.Beta));
            tr2.Start();
            }
private void button2_Click_1(object sender, System.EventArgs e)
            {
             tr1.Abort();
             tr2.Abort();
            }
public class Alpha
      {
            public PictureBox pb;
            public void Beta()
            {
                  while (true)
                  {
                        pb.Left++;
                        System.Threading.Thread.Sleep(100);
                  }
            }
        };
```

2.8. Синхронизация потоков (C Windows NT API)

2.8.1. Семафоры

Объекты ядра ***«семафор»*** используются для учета ресурсов. Как и все объекты ядра, они содержат счетчик числа пользователей. но, кроме того, поддерживают два 32 битных значения со знаком: одно определяет максимальное число ресурсов (контролируемое семафором), другое используется как счетчик текущего числа ресурсов

Попробуем разобраться, зачем нужны все эти счетчики. Допустим, я разрабатываю серверный процесс, в адресном пространстве которого выделяется буфер для хранения клиентских запросов. Размер этого буфера «зашит» в код программы и рассчитан на хранение максимум пяти клиентских запросов. Если новый клиент пытается связаться с сервером, когда эти пять запросов еще не обработаны, генерируется ошибка, которая сообщает клиенту, что сервер занят и нужно повторить попытку позже. При инициализации мой серверный процесс создает пул из пяти потоков, каждый из которых готов обрабатывать клиентские запросы по мере их поступления.

Изначально, когда запросов от клиентов еще нет, сервер не разрешает выделять процессорное время каким-либо потокам в пуле. Но как только серверу поступает, скажем, три клиентских запроса одновременно, три потока в пуле становятся планируемыми, и система начинает выделять им процессорное время. Для слежения за ресурсами и планированием потоков семафор очень удобен. Максимальное число ресурсов задается равным 5, что соответствует размеру буфера. Счетчик текущего числа ресурсов первоначально получает нулевое значение, так как клиенты еще не выдали ни одного запроса. Этот счетчик увеличивается на 1 в момент приема очередного клиентского запроса и на столько же уменьшается, когда запрос передается на обработку одному из серверных потоков в пуле.

Для семафоров определены следующие ***правила***:

- когда счетчик текущего числа ресурсов становится больше 0, семафор переходит в свободное состояние,
- если этот счетчик равен 0, семафор занят,
- система не допускает присвоения отрицательных значений счетчику текущего числа ресурсов;
- счетчик текущего числа ресурсов не может быть больше максимального числа ресурсов

Не путайте счетчик текущего числа ресурсов со счетчиком числа пользователей объекта-семафора.

Объект ядра «семафор» создается вызовом ***CreateSemaphore:***

```
HANDLE CreateSemaphore(
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes, //указатель на атрибут безопасн.
    LONG lInitialCount, //число ресурсов, которые доступны изначально
    LONG lMaximumCount, //максимальное число ресурсов, обрабат. вашим приложением
    LPCTSTR lpName      //указатель на имя семафора
   );
```

Любой процесс может получить свой («процессо-зависимый») описатель существующего объекта «семафор», вызвав ***OpenSemaphore:***

```
HANDLE OpenSemaphore(
    DWORD dwDesiredAccess,    //флаг доступа
    BOOL bInheritHandle,      //наследуемый флаг
    LPCTSTR lpName            //указатель на имя семафора
   );
```

Поток получает доступ к ресурсу, вызывая одну из ***Wait***-функций и передавая ей описатель семафора, который охраняет этот ресурс. *Wait*-функция проверяет у семафора счетчик текущего числа ресурсов, если его значение больше 0 (семафор свободен), уменьшает значение этого счетчика на 1, и вызывающий поток остается планируемым. Очень важно, что ***семафоры выполняют эту операцию проверки и присвоения на уровне атомарного доступа***; иначе говоря, когда вы запрашиваете у семафора какой-либо ресурс, операционная система проверяет, доступен ли этот ресурс, и, если да, уменьшает счетчик текущего числа ресурсов, не позволяя вмешиваться в эту операцию другому потоку. Только после того как счетчик ресурсов будет уменьшен на 1, доступ к ресурсу сможет запросить другой поток.

Если *Wait*-функция определяет, что счетчик текущего числа ресурсов равен 0 (семафор занят), система переводит вызывающий поток в состояние ожидания. Когда другой поток увеличит значение этого счетчика, система вспомнит о ждущем потоке и снова начнет выделять ему процессорное время (а он, захватив ресурс, уменьшит значение счетчика на 1).

Поток увеличивает значение счетчика текущего числа ресурсов, вызывая функцию ***ReleaseSemaphore:***

```
BOOL ReleaseSemaphore(
    HANDLE hSemaphore, //идентификатор семафора
    LONG lReleaseCount,//количество, добавляемое к текущему счету
    LPLONG lpPreviousCount   //адрес предыдущего счета
   );
```

Она просто складывает величину *lReleaseCount* со значением счетчика текущего числа ресурсов. Обычно в параметре *lReleaseCount* передают 1, но это вовсе не обязательно: часто передают в нем значения, равные или большие 2. Функция возвращает исходное значение счетчика ресурсов в *lpPreviousCount.* Если вас не интересует это значение (а в большинстве программ так оно и есть), передайте в параметре *lpPreviousCount* значение NULL.

Было бы удобнее определять состояние счетчика текущего числа ресурсов, не меняя его значение, но такой функции в Windows нет.

Использование объекта - семафора

В следующем примере, процесс использует объект семафора, чтобы ограничить число окон, которые он создает.

```
HANDLE hSemaphore;
LONG cMax = 10;
LONG cPreviousCount;
hSemaphore = CreateSemaphore(
                              NULL,
                              cMax,
                              cMax,
```

```
                                NULL);
if (hSemaphore == NULL)
{
    //Обработка ошибки
}
```

Прежде, чем любая нить процесса создает новое окно, она использует функцию **WaitForSingleObject**, чтобы определить, разрешает ли текущий счет семафора создание дополнительных окон.

```
DWORD dwWaitResult;
//Попытка входа в семафор.
dwWaitResult = WaitForSingleObject(
                                    hSemaphore,
                                    0L);          //Ждем о секунд
switch (dwWaitResult)
          {
            //Объект семафора был предупрежден
            case WAIT_OBJECT_0:
            // OK - открытие другого окна
            break;
            //Семафор не был предупрежден, т.к. время вышло
            case WAIT_TIMEOUT:
            //Не могу открыть другое окно
            break;
          }
```

Когда нить закрывает окно, она использует функцию ReleaseSemaphore, чтобы увеличить счет семафора.

```
//Увеличить на 1 счет семафора
 if (!ReleaseSemaphore(
        hSemaphore,
        1,           //инкремент
        NULL) )      //не принимает во внимание предыдущий счет
{
    //Обрабатываем ошибки
}
```

2.8.2. Мьютексы

Объекты ядра ***«мьютексы»*** гарантируют потокам взаимоисключающий доступ к единственному ресурсу. Отсюда и произошло название этих объектов (mutual exclusion, mutex). Они содержат счетчик числа пользователей, счетчик рекурсии и переменную, в которой запоминается идентификатор потока. Мьютексы ведут себя точно так же, как и критические секции. Однако если последние являются объектами пользовательского режима, то ***мьютексы — объектами ядра***. Кроме того, единственный объект мьютекс позволяет синхронизировать доступ к ресурсу нескольких потоков из разных процессов; при этом можно задать максимальное время ожидания доступа к ресурсу.

Идентификатор потока определяет, какой поток захватил мьютекс, а ***счетчик рекурсий*** — сколько раз. У мьютексов много применений, и это наиболее часто используемые объекты ядра. Как правило, с их помощью защищают блок памяти, к которому обращается множество потоков. Если бы потоки одновременно использовали какой-то блок памяти, данные в нем были бы повреждены. ***Мьютексы гарантируют, что любой поток получает***

монопольный доступ к блоку памяти, и тем самым обеспечивают целостность данных.

Для мьютексов определены следующие ***правила***:

- если его идентификатор потока равен 0 (у самого потока не может быть такой идентификатор), мьютекс не захвачен ни одним из потоков и находится в свободном состоянии;
- если его идентификатор потока не равен 0, мьютекс захвачен одним из потоков и находится в занятом состоянии;
- в отличие от других объектов ядра мьютексы могут нарушать обычные правила, действующие в операционной системе.

Для использования объекта - мьютекса один из процессов должен сначала создать его вызовом ***CreateMutex***:

```
HANDLE CreateMutex (
  LPSECURITY_ATTRIBUTES lpMutexAttributes, //указатель на атрибут безопасности
  BOOL bInitialOwner,   //флаг начального состояния мьютекса
  LPCTSTR lpName  //указатель на имя мьютекс – объекта
 );

bInitialOwner = false  - мьютекс не принадлежит ни одному из потоков. При этом
                         его идентификатор потока и счетчик рекурсии равны 0.
bInitialOwner = true - идентификатор потока, принадлежащий мьютексу,
                       приравнивается идентификатору вызывающего потока, а
                       счетчик рекурсии получает значение 1. Поскольку теперь
                       идентификатор потока отличен от 0, мьютекс изначально
                       находится в занятом состоянии.
```

Любой процесс может получить свой («процессо-зависимый») описатель существующего объекта «мьютекс», вызвав ***OpenMutex***:

```
HANDLE OpenMutex(
    DWORD dwDesiredAccess,    //флаг доступа
    BOOL bInheritHandle,      //наследуемый флаг
    LPCTSTR lpName      //указатель на имя мьютекс – объекта
   );
```

Поток получает доступ к разделяемому ресурсу, вызывая одну из ***Wait*** - функций и передавая ей описатель мьютекса, который охраняет этот ресурс. Wait-функция проверяет у мьютекса идентификатор потока, если его значение не равно 0, мьютекс свободен, в ином случае оно принимает значение идентификатора вызывающего потока, и этот поток остается планируемым.

Если Wait-функция определяет, что у мьютекса идентификатор потока не равен 0 (мьютекс занят), вызывающий поток переходит в состояние ожидания. Система запоминает это и, когда идентификатор обнуляется, записывает в него идентификатор ждущего потока, а счетчику рекурсии присваивает значение 1, после чего ждущий поток вновь становится планируемым. ***Все проверки и изменения состояния объекта - мьютекса выполняются на уровне атомарного доступа***.

Для мьютексов сделано одно исключение в правилах перехода объектов ядра из одного состояния в другое. Допустим, поток ждет освобождения занятого объекта мьютекса. В этом случае поток обычно засыпает (переходит в состояние ожидания). Однако система проверяет, не совпадает ли идентификатор потока, пытающегося захватить мьютекс, с

аналогичным идентификатором у мьютекса. Если они совпадают, система по-прежнему выделяет потоку процессорное время, хотя мьютекс все еще занят. Подобных особенностей в поведении нет ни у каких других объектов ядра в системе. Всякий раз, когда поток захватывает объект - мьютекс, счетчик рекурсии в этом объекте увеличивается на 1.

Единственная ситуация, в которой значение счетчика рекурсии может быть больше 1, — поток захватывает один и тот же мьютекс несколько раз, пользуясь упомянутым исключением из общих правил.

Когда ожидание мьютекса потоком успешно завершается, последний получает монопольный доступ к защищенному ресурсу. Все остальные потоки, пытающиеся обратиться к этому ресурсу, переходят в состояние ожидания. Когда поток, занимающий ресурс, заканчивает с ним работать, он должен освободить мьютекс вызовом функции ReleaseMutex:

```
BOOL ReleaseMutex(
    HANDLE hMutex       // идентификатор мьютекса
   );
```

Эта функция уменьшает счетчик рекурсии в объекте - мьютексе на 1. Если данный объект передавался во владение потоку неоднократно, поток обязан вызвать ReleaseMutex столько раз, сколько необходимо для обнуления счетчика рекурсии. Как только счетчик станет равен 0, переменная, хранящая идентификатор потока, тоже обнулится, и объект - мьютекс освободится. После этого система проверит, ожидают ли освобождения мьютекса какие-нибудь другие потоки. Если да, система «по-честному» выберет один из ждущих потоков и передаст ему во владение объект - мьютекс.

Отказ от объекта - мьютекса

Объект - мьютекс отличается от остальных объектов ядра тем, что занявшему его потоку передаются права на владение им. Прочие объекты могут быть либо свободны, либо заняты — вот, собственно, и все. А объекты - мьютексы способны еще и запоминать, какому потоку они принадлежат. Если какой-то посторонний поток попытается освободить мьютекс вызовом функции ReleaseMutex, то она, проверив идентификаторы потоков и обнаружив их несовпадение, ничего делать не станет, а просто вернет FALSE. Тут же вызвав ***GetLastError***, вы получите значение ***ERROR_NOT_OWNER***.

Отсюда возникает вопрос, а что будет, если поток, которому принадлежит мьютекс, завершится, не успев его освободить? В таком случае система считает, что произошел отказ от мьютекса, и автоматически переводит его в свободное состояние (сбрасывая при этом все его счетчики в исходное состояние). Если этот мьютекс ждут другие потоки, система, как обычно, выбирает один из потоков и позволяет ему захватить мьютекс. Тогда Wait-функция возвращает потоку ***WAIT_ABANDONED*** вместо ***WAIT_OBJECT_0***, и тот узнает, что мьютекс освобожден некорректно. Данная ситуация, конечно, не самая лучшая. Выяснить, что сделал с защищенными данными завершенный поток — бывший владелец объекта - мьютекса невозможно.

В реальности программы никогда специально не проверяют возвращаемое значение на WAIT_ABANDONED, потому что такое завершение потоков происходит очень редко.

Использование объекта - мьютекса

Вы можете использовать объект mutex, чтобы защитить разделенный ресурс от одновременного доступа множеством потоков или процессов. Каждый поток должен ждать mutex, прежде чем он может его использовать, чтобы получить доступ к разделенному ресурсу. Например, если несколько потоков имеют доступ к базе данных, они могут использовать объект mutex, чтобы разрешить только одному потоку в текущий момент времени заполнять базу данных записями.

В следующем примере, процесс использует функцию CreateMutex, чтобы создать mutex:

```
HANDLE hMutex;
hMutex = CreateMutex(
    NULL,
    FALSE,
    "MutexToProtectDatabase");
if (hMutex == NULL)
{
    //Обработка ошибок
}
```

Функция записи в базу данных:

```
BOOL FunctionToWriteToDatabase(HANDLE hMutex)
{   DWORD dwWaitResult;
    //Запрос к mutex.
    dwWaitResult = WaitForSingleObject(hMutex,
                                       5000L);  //5 секундный интервал
    switch (dwWaitResult)
    {
        //Поток получил мьютекс в свою собственность
        case WAIT_OBJECT_0:
            try { //Запись в базу данных
                }
            finally { //Освобождение мьютекса
                     if (! ReleaseMutex(hMutex))
                        {
                          //Произошла ошибка
                        }
                     break;
                    }
        //Нельзя получить mutex  из-за перерыва.
        case WAIT_TIMEOUT:
            return FALSE;
        // Получил в собственность оставленный объект mutex.
        case WAIT_ABANDONED:
            return FALSE;
    }
    return TRUE;
}
```

2.8.3. Критические секции

Критическая секция (critical section) — это небольшой участок кода, требующий монопольного доступа к каким-то общим данным. Она позволяет сделать так, чтобы единовременно только один поток получал доступ к определенному ресурсу. Естественно, система может в любой момент

вытеснить ваш поток и подключить к процессору другой, но ни один из потоков, которым нужен занятый вами ресурс, не получит процессорное время до тех пор, пока ваш поток не выйдет за границы критической секции.

Если у вас есть ресурс, разделяемый несколькими потоками, вы должны создать экземпляр структуры ***CRITICAL_SECTION***.

Если у Вас есть ресурсы, всегда используемые вместе, вы можете поместить их в одну кабинку — единственная структура CRITICAL_SECTION будет охранять их всех. Но если ресурсы не всегда используются вместе (например, потоки 1 и 2 работают с одним ресурсом, а потоки 1 и 3 — с другим), вам придется создать им по отдельной структуре CRITICAL_SECTION.

Теперь в каждом участке кода, где вы обращаетесь к разделяемому ресурсу, вызывайте ***EnterCriticalSection***, передавая ей адрес структуры CRITICAL_SECTION, которая выделена для этого ресурса. CRITICAL_SECTION - структура, в которую хочет войти поток, а функция EnterCriticalSection — тот инструмент, с помощью которого он узнает, свободна или занята структура.

EnterCriticalSection допустит вызвавший ее поток в структуру, если определит, что та свободна. В ином случае EnterCriticalSection заставит его ждать, пока она не освободится.

Поток, покидая участок кода, где он работал с защищенным ресурсом, должен вызвать функцию ***LeaveCriticalSection***. Тем самым он уведомляет систему о том, что структура с данным ресурсом освободилась. Если Вы забудете это сделать, система будет считать, что ресурс все еще занят, и не позволит обратиться к нему другим ждущим потокам.

Самое сложное — запомнить, что любой участок кода, работающего с разделяемым ресурсом, нужно заключить в вызовы функций EnterCriticalSection и LeaveCriticalSection. Если вы забудете сделать это хотя бы в одном месте, ресурс может быть поврежден.

Применяйте критические секции, если вам не удается решить проблему синхронизации за счет ***Interlocked***-функций. Преимущество критических секций в том, что они просты в использовании и выполняются очень быстро, так как реализованы на основе Interlocked-функций. А главный недостаток — нельзя синхронизировать потоки в разных процессах.

Теперь, когда у вас появилось общее представление о критических секциях (зачем они нужны и как с их помощью можно монопольно распоряжаться разделяемым ресурсом), давайте повнимательнее приглядимся к тому, как они устроены. Начнем со структуры CRITICAL_SECTION.

Хотя CRITICAL_SECTION не относится к недокументированным структурам, Microsoft полагает, что вам незачем знать, как она устроена и это правильно. Для нас она является своего рода черным ящиком - сама структура известна, а ее элементы — нет. Конечно, поскольку CRITICAL_SECTION — не более чем одна из структур, мы можем сказать, из чего она состоит, изучив заголовочные файлы. Но никогда не пишите код, прямо ссылающийся на ее элементы.

Вы работаете со структурой CRITICAL_SECTION исключительно через функции Windows, передавая им адрес соответствующего экземпляра этой структуры. Функции сами знают, как обращаться с ее элементами, и гарантируют, что она всегда будет в согласованном состоянии. Так что теперь мы перейдем к рассмотрению этих функций.

Обычно структуры CRITICAL_SECTION создаются как глобальные переменные, доступные всем потокам процесса. Но ничто не мешает нам создавать их как локальные переменные или переменные, динамически размещаемые в куче. Есть только два условия, которые надо соблюдать. Во-первых, все потоки, которым может понадобиться ресурс, должны знать адрес структуры CRITICAL_SECTION, которая защищает этот ресурс. Вы можете получить ее адрес, используя любой из существующих механизмов. Во-вторых, элементы структуры CRITICAL_SECTION следует инициализировать до обращения какого-либо потока к защищенному ресурсу. Структура инициализируется вызовом:

```
VOID InitializeCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection  //адрес критической секции
   );
```

Поскольку вся работа данной функции заключается в инициализации нескольких переменных-членов, она не дает сбоев и поэтому ничего не возвращает (void). InitializeCriticalSection должна быть вызвана до того, как один из потоков обратится к EnterCriticalSection.

Если вы знаете, что структура CRITICAL_SECTION больше не понадобится ни одному потоку, удалите ее, вызвав ***DeleteCriticalSection***:

```
VOID DeleteCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection //адрес критической секции
   );
```

Она сбрасывает все переменные-члены внутри этой структуры.

Естественно, нельзя удалять критическую секцию в тот момент, когда ею все еще пользуется какой-либо поток.

Участок кода, работающий с разделяемым ресурсом, предваряется вызовом:

```
VOID EnterCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection // адрес критической секции
);
```

Первое, что делает EnterCriticalSection, — исследует значения элементов структуры CRITICAL_SECTION. Если ресурс занят, в них содержатся сведения о том, какой поток пользуется ресурсом. EnterCriticalSection выполняет следующие действия.

- Если ресурс свободен, EnterCriticalSection модифицирует элементы структуры, указывая, что вызывающий поток занимает ресурс, после чего немедленно возвращает управление, и поток продолжает свою работу (получив доступ к ресурсу).
- Если значения элементов структуры свидетельствуют, что ресурс уже захвачен вызывающим потоком, EnterCriticalSection обновляет их, отмечая тем самым, сколько раз подряд этот поток захватил ресурс, и немедленно возвращает управление. Такая ситуация бывает нечасто — лишь тогда, когда поток два раза подряд вызывает EnterCriticalSection без промежуточного вызова LeaveCriticalSection.

● Если значения элементов структуры указывают на то, что ресурс занял другим потоком, EnterCriticalSection переводит вызывающий поток в режим ожидания. Это потрясающее свойство критических секций: поток, пребывая в ожидании, не тратит ни кванта процессорного времени. Система запоминает, что данный поток хочет получить доступ к ресурсу, и - как только поток, занимавший этот ресурс, вызывает LeaveCriticalSection — вновь начинает выделять нашему потоку процессорное время. При этом она передает ему ресурс, автоматически обновляя элементы структуры CRITICAL_SECTION.

Внутреннее устройство EnterCriticalSection не слишком сложно; она выполняет лишь несколько простых операций. Чем она действительно ценна, так это способностью выполнять их на уровне атомарного доступа. Даже если два потока на много процессорной машине одновременно вызовут EnterCriticalSection, функция все равно корректно справится со своей задачей: один поток получит ресурс, другой — перейдет в ожидание.

Поток, переведенный EnterCriticalSection в ожидание, может надолго лишиться доступа к процессору, а в плохо написанной программе — даже вообще не получить его. Когда именно так и происходит, говорят, что поток "голодает".

В действительности потоки, ожидающие освобождения критической секции, никогда не блокируются «навечно». EnterCriticalSection устроена так, что по истечении определенного времени, генерирует исключение. После этого вы можете подключить к своей программе отладчик и посмотреть, что в ней случилось. Длительность времени ожидания функцией EnterCriticalSection определяется значением параметра ***CriticalSectionTimeout***, который хранится в следующем разделе системного реестра:

HKEY_LOCAL_MACHlNE\System\CurrentControlSet\Control\Session Manager

Длительность времени ожидания измеряется в секундах и по умолчанию равна 2 592 000 секунд (что составляет ровно 30 суток). Не устанавливайте слишком малое значение этого параметра (например, менее 3 секунд), так как иначе вы нарушите работу других потоков и приложений, которые обычно ждут освобождения критической секции дольше трех секунд.

Как и EnterCriticalSection, функция LeaveCriticalSection выполняет все действия на уровне атомарного доступа. Однако LeaveCriticalSection никогда не приостанавливает поток, а управление возвращает немедленно.

Вероятность того, что InitializeCriticalSection потерпит неудачу, крайне мала, но все же существует. В свое время Microsoft не учла этого при разработке функции и определила ее возвращаемое значение как VOID, т.е. она ничего не возвращает. Однако функция может потерпеть неудачу, так как выделяет блок памяти для внутрисистемной отладочной информации. Если выделить память не удается, генерируется исключение ***STATUS_NO_MEMORY***. Вы можете перехватить его, используя структурную обработку исключений.

Если потоки все же будут конкурировать за критическую секцию в условиях нехватки памяти, система не сможет создать нужный объект ядра. И тогда EnterCriticalSection возбудит исключение ***EXCEPTION_INVALID_HANDLE***. Большинство разработчиков просто игнорирует вероятность такой ошибки и не предусматривает для нее никакой обработки, поскольку она случается действительно очень редко. Но если

вы хотите заранее подготовиться к такой ситуации, у вас есть две возможности.

Используя критические секции, желательно привыкнуть делать одни вещи и избегать других. Вот несколько полезных приемов, которые пригодятся вам в работе с критическими секциями.

- На каждый разделяемый ресурс используйте отдельную структуру CRITICAL_SECTION .
- Если в Вашей программе имеется несколько независимых структур данных, создавайте для каждой из них отдельный экземпляр структуры CRITICAL_SECTION. Это лучше, чем защищать все разделяемые ресурсы одной критической секцией.
- Не занимайте критические секции надолго. Надолго занимая критическую секцию, ваше приложение может блокировать другие потоки, что отрицательно скажется на его общей производительности.

Использование критической секции

```
CRITICAL_SECTION GlobalCriticalSection;
//Инициализация критической секции
InitializeCriticalSection(&GlobalCriticalSection);
//Начало работы с разделяемыми ресурсами
try
{
    EnterCriticalSection(&GlobalCriticalSection);
    //Получение доступа к разделяемым ресурсам
}
finally
{
    //Выход из критической секции
    LeaveCriticalSection(&GlobalCriticalSection);
}
```

2.8.4. Атомарный доступ: семейство Interlocked-функций

Большая часть синхронизации потоков связана с атомарным доступом (atomic access) — монопольным захватом ресурса, обращающимся к нему потоком.

Данные функции осуществляют способ, гарантирующий приращение значения переменной на уровне атомарного доступа, т.е. без прерывания другими потоками. Большинство разработчиков программного обеспечения недооценивает эти функции, а ведь они невероятно полезны и очень просты для понимания. Все функции из этого семейства манипулируют переменными на уровне атомарного доступа. Взгляните на ***InterlockExchangeAdd***:

```
LONG InterlockedExchangeAdd (
    PLONG Addend,       //адрес ПЗ
    LONG Increment      //прибавляемое значение
   );
```

Что может быть проще? Вы вызываете эту функцию, передавая адрес переменной типа LONG и указываете добавляемое значение InterlockedExchangeAdd гарантирует, что операция будет выполнена атомарно.

Как же работают Interlocked-функции? Ответ зависит от того, какую процессорную платформу вы используете. На компьютерах с процессорами семейства x86 эти функции выдают по шине аппаратный сигнал, не давая другому процессору обратиться по тому же адресу памяти.

Другой важный аспект, связанный с Interlocked-функциями, состоит в том, что они выполняются чрезвычайно быстро. Вызов такой функции обычно требует не более 50 тактов процессора, и при этом не происходит перехода из пользовательского режима в режим ядра (а он отнимает не менее 1000 тактов).

Кстати, InterlockedExchangeAdd позволяет не только увеличить, но и уменьшить значение — просто передайте во втором параметре отрицательную величину. InterlockedExchangeAdd возвращает исходное значение в Addend.

Вот еще функции из этого семейства:

```
LONG InterlockedExchange(
    LPLONG Target,      //адрес 32-битного значения для обмена
    LONG Value          //новое значение
   );
```

InterlockedExchange монопольно заменяют текущее значение переменной типа LONG, адрес которой передается в первом параметре, на значение, передаваемое во втором параметре. В 32-разрядпом приложении обе функции работают с 32-разрядными значениями, но в 64-разрядной программе первая оперирует с 32-разрядными значениями, а вторая — с 64-разрядными. Все функции возвращают исходное значение переменной.

```
PVOID InterlockedCompareExchange(
    PVOID *Destination,
    PVOID Exchange,     //отменяемое значение
    PVOID Comperand     //значение для сравнения
   );
```

Выполняет операцию сравнения и присвоения на уровне атомарного доступа. В 32-разрядном приложении функция работает с 32-разрядными значениями. Функция сравнивает текущее значение переменной типа LONG (на которую указывает параметр Destination) со значением, передаваемым в параметре Comperand. Если значения совпадают, Destination получает значение параметра Exchange; в ином случае Destination остается без изменений. Функция возвращает исходное значение Destination. И не забывайте, что все эти действия выполняются как единая атомарная операция.

Обратите внимание на отсутствие Interlocked-функции, позволяющей просто считывать значение какой-то переменной, не меняя его. Она и не нужна. Если один поток модифицирует переменную с помощью какой-либо Interlocked-функции в тот момент, когда другой читает содержимое той же переменной, ее значение, прочитанное вторым потоком, всегда будет достоверным. Он получит либо исходное, либо измененное значение переменной. Поток, конечно, не знает, какое именно значение он считал, но главное, что оно корректно и не является некоей произвольной величиной. В большинстве приложений этого вполне достаточно.

Использование Interlocked-функций

```
DWORD WINAPI ThreadFunc1(PVOID pvParam)
{
  InterlockedExchangeAdd(&g_x, 1);
  return(0);
}
DWORD WINAPI ThreadFunc2(PVOID pvPararr)
{
  InterlockedExchangeAdd(&g_x, 1);
  return(0);
}
```

2.8. Синхронизация потоков (C# .NET)

Рассмотрим два вида доступа к разделяемым ресурсам в многопоточных приложениях. Будем использовать:

- Мьютекс - класс
- Interlocked – класс (атомарный доступ к разделяемым переменным)

Использование мьютекс - класса

```
using System;
using System.Threading;
class Test
{
    //Создаем новый Mutex
    private static Mutex mut = new Mutex();
    private const int numIterations = 1;
    private const int numThreads = 3;
    static void Main()
    {
        //Создание нитей, которые будут использовать защищенный ресурс
        for(int i = 0; i < numThreads; i++)
        {
            Thread myThread = new Thread(new ThreadStart(MyThreadProc));
            myThread.Name = String.Format("Thread{0}", i + 1);
            myThread.Start();
        }
        //Главный поток существует, но приложение продолжает работать,
        //пока на переднем плане потоки существуют
    }
    private static void MyThreadProc()
    {
        for(int i = 0; i < numIterations; i++)
        {
            UseResource();
        }
    }
    //Этот метод представляет ресурс, который должен быть
    //синхронизирован так, чтобы только одна нить одновременно могла с
    //ним работать
    private static void UseResource()
    {
        //Запрашиваем собственность мьютекса
        mut.WaitOne();
        Console.WriteLine("{0} has entered the protected area",
                          Thread.CurrentThread.Name);

        //Код доступа к разделяемому ресурсу должен быть здесь

        //Моделируем некоторую работу
        Thread.Sleep(500);

        Console.WriteLine("{0} is leaving the protected area\r\n",
                          Thread.CurrentThread.Name);
        //Освобождение мьютекса
        mut.ReleaseMutex();
    }
}
```

Использование Interlocked - класса

Методы этого класса защищают против ошибок, которые могут происходить, когда планировщик переключает контексты, в то время как нить обновляет переменную, к которой могут обращаться другие нити, или когда две нити выполняются одновременно на отдельных процессорах. Члены этого класса не вызывают исключений.

Приращение и методы декремента увеличивают или уменьшают переменную и выполняют все это атомарно. На большинстве компьютеров, incrementing переменной - не атомное действие, а требует следующих шагов:

1. Загрузить значение переменной в регистр.
2. Приращение или декремент значения.
3. Сохранение изменившейся переменной.

Если вы используете приращение и декремент, нить может выгружаться после выполнения первых двух шагов. Другая нить может выполнять все три шага. Когда первая нить возобновляет выполнение, это может привести к неправильному конечному значению, запланированному в программе.

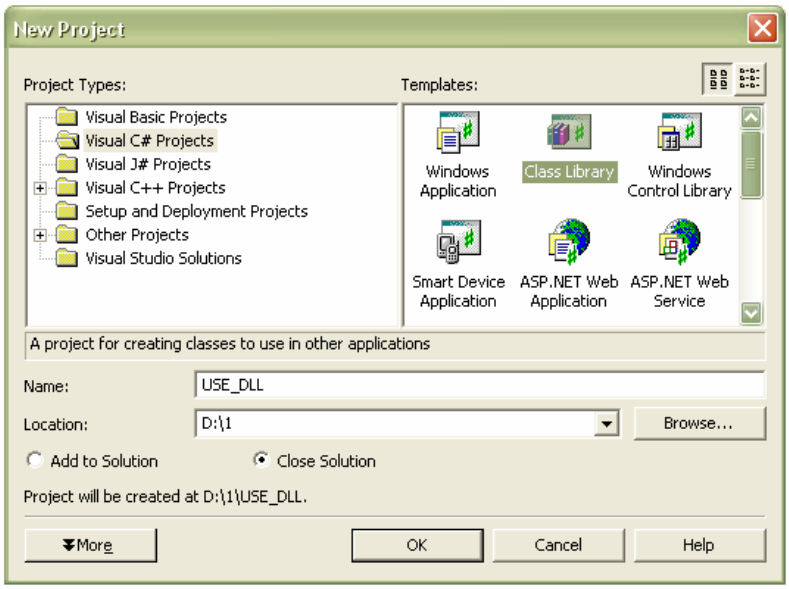
Все действия этого класса выполняются на атомарном уровне.

```
using System;
using System.Threading;
namespace InterlockedExchange_Example
{
    class MyInterlockedExchangeExampleClass
    {
        //0 для false, 1 для true.
        private static int usingResource = 0;
        private static Object currentMso;
        private static Object globalMso = new Object();
        private const int numThreadIterations = 5;
        private const int numThreads = 10;
        static void Main()
        {
            Thread myThread;
            Random rnd = new Random();
            for(int i = 0; i < numThreads; i++)
            {
                myThread = new Thread(new ThreadStart(MyThreadProc));
                myThread.Name = String.Format("Thread{0}", i + 1);
                //Ждите случайное время перед запуском следующего потока
                Thread.Sleep(rnd.Next(0, 1000));
                myThread.Start();
            }
        }
        private static void MyThreadProc()
        {
            for(int i = 0; i < numThreadIterations; i++)
            {
                UseResource();
                //Ждать 1 секунду перед следующей попыткой
                Thread.Sleep(1000);
            }
        }
        static bool UseResource()
        {
            //0 указывает, что метод не используется
            if(0 == Interlocked.Exchange(ref usingResource, 1))
            {
                Console.WriteLine("{0} acquired the lock", Thread.CurrentThread.Name);
                //Код доступа к разделяемому ресурсу
                //Моделирование работы
                Thread.Sleep(500);
                Console.WriteLine("{0} exiting lock", Thread.CurrentThread.Name);
                //Освобождение
                Interlocked.Exchange(ref usingResource, 0);
```

```
                return true;
            }
            else
            {
                Console.WriteLine("   {0} was denied the lock", Thread.CurrentThread.Name);
                return false;
            }
        }
    }
}
```

2.9. Создание и использование DLL (C# .NET)

Для начала создадим саму динамическую библиотеку. Для этого создаем проект:

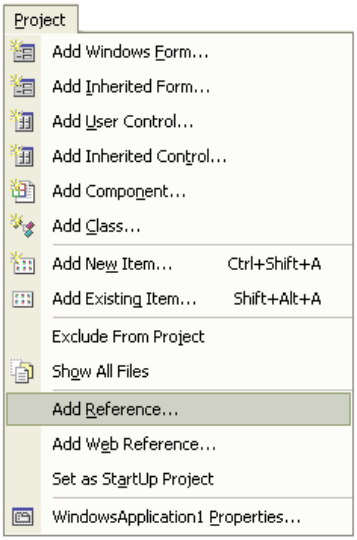


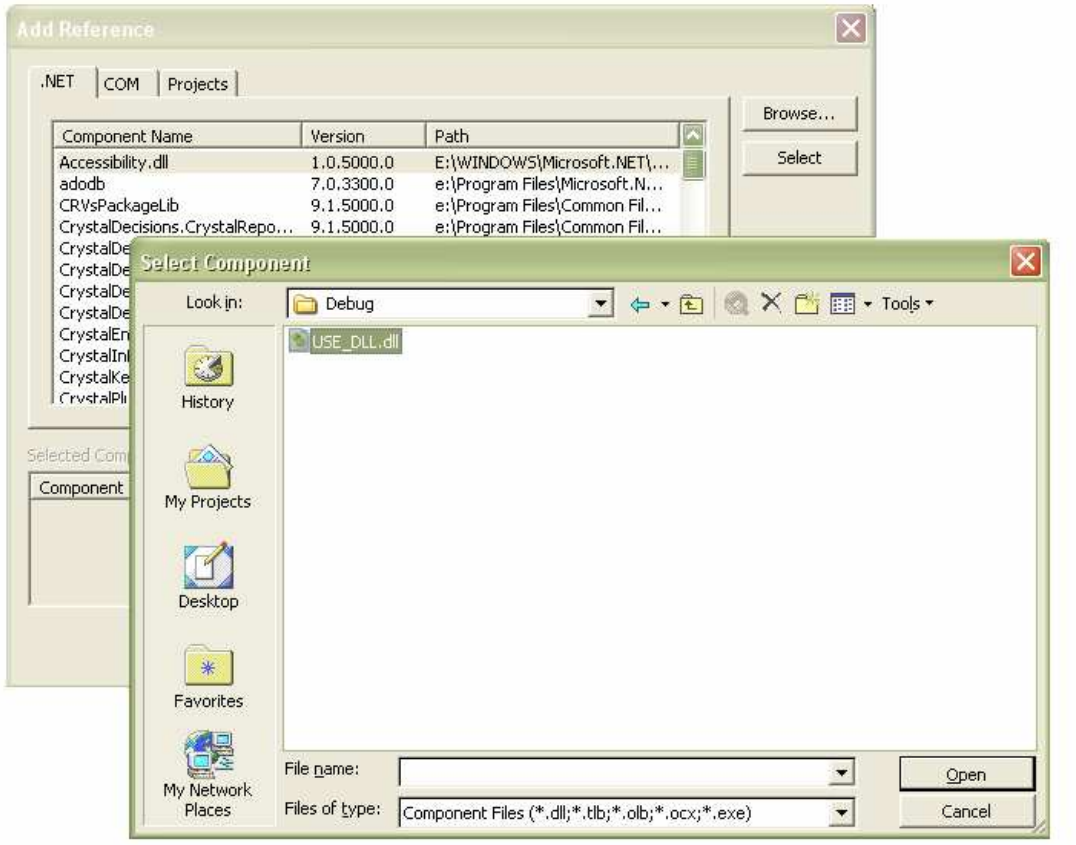
Затем правим код:

```
using System;
namespace USE_DLL
{
      public class DLL_CLASS
      {
      public static long Add(long i, long j)
            {
                  return(i+j);
            }
            public static long Max(long i, long j)
            {
                  if (i>j)
                    return(i);
                  else
                    return(j);
            }
      }
}
```

Компилируем F5 и получаем файл **“USE_DLL.dll”**.

Затем создаем клиентское приложение, использующее данную библиотеку. Создадим проект “Приложение Windows”. Сначала необходимо подключить к проекту dll. Проделаем то, что указано ниже:





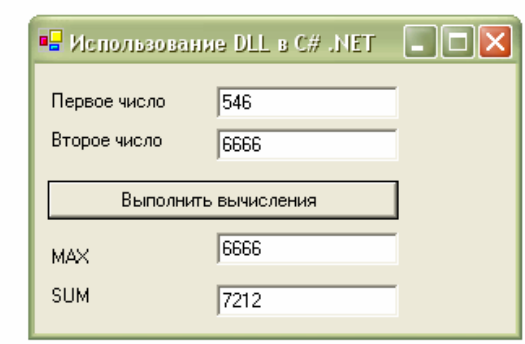
Жмем OK. Теперь можно использовать пространство имен USE_DLL, а вместе с ним и методы библиотеки. Листинг программы:

```
using System;
...
using USE_DLL; //Добавили пространство имен библиотеки USE_DLL.dll
namespace CLIENT_DLL
{
      public class Form1 : System.Windows.Forms.Form
      {
            ...
            [STAThread]
            static void Main()
            {
```

```
                  Application.Run(new Form1());
            }

            private void button1_Click(object sender, System.EventArgs e)
            {
                  long d=Convert.ToInt64(textBox1.Text,10);
                  long d1=Convert.ToInt64(textBox2.Text,10);
                  long M = DLL_CLASS.Max(d,d1);
                  long S = DLL_CLASS.Add(d,d1);
                  textBox3.Text=M.ToString();
                  textBox4.Text=S.ToString();
            }
      }
}
```

Рабочий вид программы:



2.10. Примитивная графика, текст, работа с BMP (C Windows NT API)

Интерфейс графических устройств (**GDI**) обеспечивает функции и родственные структуры, которые приложение может использовать для создания графического вывода данных на дисплее, принтере и других устройствах. Используя функции GDI, вы можете рисовать линии, кривые, замкнутые фигуры, траектории, текст и растровые изображения. Цвет и стиль элементов, которые вы рисуете, зависят от объекта рисования, то есть от ***перьев***, ***кистей*** и ***шрифтов***, которые вы создаете. Вы можете использовать перья, чтобы чертить линии и кривые, кисти, чтобы заполнить внутреннюю часть замкнутых фигур и шрифты, чтобы записать текст.

Прикладные программы направляют вывод данных на заданное устройство, создавая ***контекст устройства*** (DC) для устройства. Контекст устройства - это управляемая GDI структура, содержащая информацию об устройстве, такие как его режимы работы и текущие выбранные элементы. Приложение создает DC при помощи использования функции контекста устройства. GDI возвращает дескриптор контекста устройства, который используется в последующих вызовах, чтобы идентифицировать устройство. Например, используя дескриптор, приложение может извлекать информацию о возможностях устройства, таких, как его технологический тип (дисплей, принтер, или другое устройство) и размеры и разрешающая способность поверхности отображения.

Прикладные программы могут направлять вывод данных (вывод информации) на физическое устройство, типа дисплея или принтера, или на "логическое" устройство, типа ЗУ или метафайл. Логические устройства дают прикладным программам средство сохранения выводимых данных в форме, которая является удобной для передачи впоследствии физическому устройству. После записи выводимых данных в метафайле, приложение может их воспроизводить в этом метафайле любое число раз, отправляя выводимые данные любому числу физических устройств.

Прикладные программы используют функции атрибутов, чтобы установить режимы работы и текущие выбранные элементы для устройства. Режимы работы включают в себя текст и цвета фона, режим смешивания (называемый также бинарной растравой операцией), который устанавливает, как цвета в пере или кисти в комбинации с цветами уже на поверхности отображения, так и в режиме отображения, который определяет, как GDI преобразует координаты, используемые приложением в систему координат устройства. Текущие выбранные элементы определяют, какие объекты рисования использованы, когда рисовались выводимые данные.

Пример позволяет рисовать прямоугольники, окружности, эллипсы на поверхности окна. Пользователь горячими клавишами выбирает что рисовать и цвет фона фигур. Также здесь используется тип HBITMAP, то есть для вывода BMP файлов. Загружая файл BMP функцией **LoadImage** в контекст дисплея, можно показать его пользователю. Здесь лишь приведем пример без загрузки битовой карты:

```
#include "stdafx.h"
#include "math.h "
#include "resource.h"
#include "COMMDLG.H"
#define SCREEN_X 2024   //Для загрузки HBITMAP в hdcmem - только так можно рисовать
#define SCREEN_Y 2784   //Обязательно должно выполняться условие
                        //SCREEN_Y-MAX_PLOSHAD_B>784 (784 - разрешение по вертикали)
#define MAX_LOADSTRING 100
//Виртуальное окно - то окно, которое мы должны скроллингом просмотреть в Windows окне
```

```
#define MAX_PLOSHAD_B 2000 //Максимальная высота виртуального окна
#define MAX_PLOSHAD_A 1000  //Максимальная длина виртуального окна
//////////////////////////////////////////////////////////////////////
//////////////////////////Глобальные переменные///////////////////////////
//////////////////////////////////////////////////////////////////////
int pos_y,pos_x;                        //дискретные положения ползунков
int max_x,max_y;                        //max диапазона прокрутки
int min;                                //min диапазона прокрутки
int x_BUF,y_BUF,a,b,delta_y,delta_x;   //вычисление глобальной нулевой точки
HINSTANCE hInst;                        //Описатель экземпляра самой программы
TCHAR szTitle[MAX_LOADSTRING];          //Заголовок окна
TCHAR szWindowClass[MAX_LOADSTRING];   //Указатель на зарегистрированное имя класса
HWND hWnd; //Глобальный идентификатор окна
bool MOUSE_UP_HDC; //Нужны для рисования мышью
HBITMAP hb,hb2; //Только так можно рисовать в контексте памяти - загрузив hb,hb2
HDC hdcmem,      //Буфер - в котором отображается вид будущей фигуры
    hdcmem2;     //Буфер - в котором юзер рисует фигуры
int xPos,yPos,xTec,yTec; //Координаты при нажатии мыши и текущие координаты
COLORREF CR;          //Цвет, который выбрал пользователь
int FIGURA; //1-прямоугольник,2-окружность,3-эллипс
//////////////////////////////////////////////////////////////////////
//////////////////////Прототипы глобальных функций//////////////////////////
//////////////////////////////////////////////////////////////////////
ATOM                            LAB7_RegisterClass(HINSTANCE hInstance);
BOOL                            InitInstance(HINSTANCE, int);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK About(HWND, UINT, WPARAM, LPARAM);
void PRINT_IMAGE(HDC hd); //Закрашиваем окно юзера
void RECT_HDCMEM();   //Рисуем прямоугольники в памяти
void RGB_USER();      //Цвет фона прямоугольников
//////////////////////////////////////////////////////////////////////
/////////////////Точка входа в программу//////////////////////////////////
//////////////////////////////////////////////////////////////////////
int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR     lpCmdLine,
                     int       nCmdShow)
{
        MSG msg;    //Структура для сообщений
        ///Инициализация глобальных строк//////////////////////////
        LoadString(hInstance,IDS_LAB7_TITLE,szTitle,MAX_LOADSTRING);
        LoadString(hInstance,IDC_LAB7_CL,szWindowClass,MAX_LOADSTRING);
        LAB7_RegisterClass(hInstance); //Регистрируем класс окна
        if (!InitInstance (hInstance, nCmdShow))
        {
                return FALSE;
        }
        RegisterHotKey(hWnd,1,NULL,VK_F1);
        RegisterHotKey(hWnd,2,NULL,VK_F2);
        RegisterHotKey(hWnd,3,NULL,VK_F3);
        RegisterHotKey(hWnd,4,NULL,VK_F4);
        ////////Главный цикл обработки сообщений//////////////////////////////
        while (GetMessage(&msg, NULL, 0, 0))
        {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
        }
        return msg.wParam;
}
//////////////////////////////////////////////////////////////////////
//////////////////////////Функция создания окна//////////////////////////////
//////////////////////////////////////////////////////////////////////
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{

   hInst = hInstance;  //hInst - глобальная переменная
   hWnd = CreateWindow(
                                          szWindowClass,
                                          szTitle,
                                          WS_OVERLAPPEDWINDOW | WS_VSCROLL | WS_HSCROLL,
                                          CW_USEDEFAULT,
                                          0,
                                          CW_USEDEFAULT,
                                          0,
                                          NULL,
                                          NULL,
                                          hInstance,
                                          NULL
                                         );
   if (!hWnd)
   {
           return FALSE;
   }
   ShowWindow(hWnd, nCmdShow);
   return TRUE;
}
//////////////////////////////////////////////////////////////////////
////////////////////Функция регистрации класса окна/////////////////////
//////////////////////////////////////////////////////////////////////
ATOM LAB7_RegisterClass(HINSTANCE hInstance)
```

```
{
        WNDCLASSEX wcex; //Объявляем переменную типа WNDCLASSEX
        wcex.cbSize = sizeof(WNDCLASSEX);
        wcex.style       = CS_HREDRAW | CS_VREDRAW;
        wcex.lpfnWndProc = (WNDPROC)WndProc;
        wcex.cbClsExtra  = 0;
        wcex.cbWndExtra  = 0;
        wcex.hInstance = hInstance;
        wcex.hIcon = LoadIcon(hInstance, (LPCTSTR)IDI_MY100);
        wcex.hCursor = LoadCursor(NULL, IDC_ARROW);
        wcex.hbrBackground = (HBRUSH)(COLOR_3DDKSHADOW+1);
        wcex.lpszMenuName = (LPCSTR)IDC_LAB7_MENU;
        wcex.lpszClassName = szWindowClass;
        wcex.hIconSm = LoadIcon(wcex.hInstance, (LPCTSTR)IDI_SMALL);
        return RegisterClassEx(&wcex);
}
////////////////////////////////////////////////////////////////////////
/////////////////Функция обработки сообщений для окна//////////////////
////////////////////////////////////////////////////////////////////////
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static      int cx,cy,xo,yo;
        int wmId_LAB7;
        wmId_LAB7=LOWORD(wParam); //Идентификатор дочернего окна
        switch (message)
        {
             /////сообщения - посылаются дочерним окнам (меню и т.п.)//
             case WM_COMMAND:
                        switch (wmId_LAB7)
                        {
                                case IDM_ABOUT:
                                   DialogBox(hInst,(LPCTSTR)IDD_ABOUTBOX,hWnd,(DLGPROC)About);
                                   break;
                                case IDM_EXIT:
                                   DestroyWindow(hWnd);
                                    break;
                                case IDM_PASTE:
                                   InvalidateRect(hWnd,NULL,true);
                                    UpdateWindow(hWnd);
                                   break;
                                default:
                                   return DefWindowProc(hWnd, message, wParam, lParam);
                        }
                        break;
               //////Значит программа должна отвечать на прорисовку окна//
               case WM_PAINT:
                  PAINTSTRUCT ps;
                  HDC hdc;
                  hdc = BeginPaint(hWnd, &ps);
                  PRINT_IMAGE(hdc);
                  EndPaint(hWnd, &ps);
                 break;
               ///////Уведомляет окно о том, что оно будет разрушено////
               case WM_DESTROY:
                  //Направляет сообщение wm_Quit обычно в ответ на сообщение wm_Destroy
                  //Если не вызвать эту процедуру - окно закроется, а процесс висит в памяти
                  PostQuitMessage(0);
                  break;
               ////////Сообщение - при создании окна////////////////////
               case WM_CREATE:
                  //////////Устанавливаем диапазон прокрутки////////////
                  max_x=100;
                  max_y=max_x;
                  min=0;
                  SetScrollRange(hWnd,SB_VERT,min,max_y,TRUE);
                  SetScrollRange(hWnd,SB_HORZ,min,max_x,TRUE);
                  SetScrollPos (hWnd,SB_VERT,0,TRUE);
                  //////////Дополнительные переменные/////////////////
                  y_BUF=0;
                  x_BUF=0;
                  a=MAX_PLOSHAD_A;        //длина области прокручивания
                  b=MAX_PLOSHAD_B;        //высота области прокручивания
                  delta_y=10;   //отступ текста сверху
                  delta_x=10;   //отступ текста слева
                  FIGURA=1; //Разрешено вначале рисовать прямоугольники
                  ///////////Буфер для прорисовки//////////////////////
                  hdcmem = CreateCompatibleDC(GetDC(hWnd));
                  hdcmem2 = CreateCompatibleDC(GetDC(hWnd));
                  hb = CreateCompatibleBitmap(GetDC(hWnd),SCREEN_X,SCREEN_Y);
                  hb2 = CreateCompatibleBitmap(GetDC(hWnd),SCREEN_X,SCREEN_Y);
                  SelectObject(hdcmem,hb);
                  SelectObject(hdcmem2,hb2);
                  break;
                ////////////вертикальный скроллинг////////////////////
               case WM_VSCROLL:
                   switch(LOWORD(wParam))
                   {
                    case SB_LINEUP:
                             //////////////Перемещаем ползунок///////////////
                             if (pos_y>min)
```

```
                        {
                          pos_y=pos_y-1;
                          SetScrollPos (hWnd,SB_VERT,pos_y,TRUE);
                          y_BUF=ceil(pos_y*(b/max_y)); //вычисляем глобальную точку (x_BUF,y_BUF)
                        }
                    break;
               case SB_LINEDOWN:
                        if (pos_y<max_y)
                        {
                          pos_y=pos_y+1;
                          SetScrollPos (hWnd,SB_VERT,pos_y,TRUE);
                          y_BUF=ceil(pos_y*(b/max_y)); //вычисляем глобальную точку (x_BUF,y_BUF)
                        }
                        break;
               case SB_THUMBTRACK:  //Пользователь передвигает ползунок
                        pos_y=HIWORD(wParam);
                        SetScrollPos (hWnd,SB_VERT,pos_y,TRUE);
                        y_BUF=ceil(pos_y*(b/max_y));
                       default:
                        break;
            };
            //Заставляем окно обновиться//////////////////////////
           SetWindowOrgEx(hdcmem2,-x_BUF,-y_BUF,NULL);
           InvalidateRect(hWnd,NULL,false); //false - вот главный ключ - мы сами перерисовываем
           UpdateWindow(hWnd);
           break;
         ////////////горизонтальный скролллинг//////////////////
         case WM_HSCROLL:
            switch(LOWORD(wParam))
            {
              case SB_LINEUP:
                  //////////////Перемещаем ползунок///////////////
                     if (pos_x>min)
                       {
                         pos_x=pos_x-1;
                         SetScrollPos (hWnd,SB_HORZ,pos_x,TRUE);
                         x_BUF=ceil(pos_x*(a/max_x)); //вычисляем глобальную точку (x_BUF,y_BUF)
                       }
                       break;
               case SB_LINEDOWN:
                       if (pos_x<max_x)
                       {
                         pos_x=pos_x+1;
                         SetScrollPos (hWnd,SB_HORZ,pos_x,TRUE);
                         x_BUF=ceil(pos_x*(a/max_x)); //вычисляем глобальную точку (x_BUF,y_BUF)
                       }
                       break;
               case SB_THUMBTRACK:  //Пользователь передвигает ползунок
                       pos_x=HIWORD(wParam);
                       SetScrollPos(hWnd,SB_HORZ,pos_x,TRUE);
                        x_BUF=ceil(pos_x*(a/max_x));
               default:
                       break;
            };
         //Заставляем окно обновиться//////////////////////////
        SetWindowOrgEx(hdcmem2,-x_BUF,-y_BUF,NULL);
        InvalidateRect(hWnd,NULL,false); //false - вот главный ключ - мы сами перерисовываем
        UpdateWindow(hWnd);
        break;
         ////////////Горячие клавиши//////////////////////////////////////
         case WM_HOTKEY:
         switch (HIWORD(lParam))
                 {
                   case VK_F1:
                     RGB_USER(); //вызываем процедуру выбора цвета для фигур
                      InvalidateRect(hWnd,NULL,false);
                     UpdateWindow(hWnd);
                      break;
                   case VK_F2:
                          FIGURA=1;
                         break;
                   case VK_F3:
                          FIGURA=2;
                         break;
                   case VK_F4:
                          FIGURA=3;
                         break;
                 };
                 break;
         ////////////////////////////////////////////////////////////////
         case WM_MOUSEMOVE:
            if (wParam & MK_LBUTTON)
            {//Если мы нажали на левую кнопку мыши и водим над окном
            xTec = LOWORD(lParam);
            yTec = HIWORD(lParam);
            RECT_HDCMEM(); //Здесь мы просто отображаем будущий вид фигуры
            }
            else
            {//Не нажимали - просто записываем координаты до нажатия - они будут начальными
              xPos = LOWORD(lParam);
```

```
                    yPos = HIWORD(lParam);
                   }
                   break;
                ////////////////Юзер отпустил кнопку мыши//////////////////////
                case WM_LBUTTONUP:
                    MOUSE_UP_HDC=true;
                    RECT_HDCMEM(); //Здесь закрепляем результат
                  break;
                ////////////////////////////////////////////////////
                default:
                        //Обеспечивает стандартную обработку сообщений для сообщений, которые
                        //явно не обрабатываются прикладной задачей
                        return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
//////////////////////////////////////////////////////////////////////////
//////////////////Обработка сообщений диалога//////////////////////////
//////////////////////////////////////////////////////////////////////////
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
        switch (message)
        {
                case WM_INITDIALOG:
                                return TRUE;
                case WM_COMMAND:
                        if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
                        {
                                EndDialog(hDlg, LOWORD(wParam));
                                return TRUE;
                        }
                        break;
        }
    return FALSE;
}
//////////////////////////////////////////////////////////////////////////
///////Рисуем прямоугольник во все окно - на нем юзер рисует фигуры///////
//////////////////////////////////////////////////////////////////////////
void PRINT_IMAGE(HDC hd)
{
 static bool ONE=false;
 if (ONE==false)
 {////Сюда попадаем в самом начале - здесь рисуется первоначальный фон////
  ////Затем на нем юзер рисует фигуры////////////////////////////////////
  HBRUSH hBr;
  HPEN hp;
  LOGBRUSH lb;
  lb.lbColor=GetSysColor(COLOR_3DDKSHADOW); //Заливаем окно серым цветом
  lb.lbStyle=BS_SOLID;
  lb.lbHatch=NULL;
  ///////////Сначала рисуем в памяти общий вид//////////////////////////
  hBr = CreateBrushIndirect(&lb);
  hp = CreatePen(NULL,0,GetSysColor(COLOR_3DDKSHADOW));
  SelectObject(hdcmem2, hBr);
  SelectObject(hdcmem2, hp);
  Rectangle(hdcmem2,0,0,SCREEN_X,SCREEN_Y);
  DeleteObject(hp);
  DeleteObject(hBr);
  /////Отображаем прямоугольник в окне - на нем все рисуем/////////////////
  BitBlt(GetDC(hWnd), 0, 0, SCREEN_X, SCREEN_Y, hdcmem2, 0,0,SRCCOPY);
  ONE=true;
 }
 else
 {//Здесь происходит прорисовка памяти в контекст окна////////////////////
   BitBlt(GetDC(hWnd), 0, 0, SCREEN_X, SCREEN_Y, hdcmem2, 0,0,SRCCOPY);
 }
}
//////////////////////////////////////////////////////////////////////////
////////Пользователь рисует прямоугольник - в памяти//////////////////////
//////////////////////////////////////////////////////////////////////////
void RECT_HDCMEM()
{
 HBRUSH hBr;
 HPEN hp;
 LOGBRUSH lb;
 lb.lbColor=CR; //Цвет прямоугольника
 lb.lbStyle=BS_SOLID;
 lb.lbHatch=NULL;
 //////////////////////////////////////////////////////////////////////////
 hBr = CreateBrushIndirect(&lb);
 hp = CreatePen(NULL,0,RGB(1,1,1)); //Цвет границы прямоугольника
 SelectObject(hdcmem, hBr);
 SelectObject(hdcmem, hp);
 SelectObject(hdcmem2, hBr);
 SelectObject(hdcmem2, hp);
 if (MOUSE_UP_HDC==true)
 {//////Пользователь отпустил мышь - можно копировать в память////////////
   SelectObject(hdcmem2,hb);
   switch (FIGURA)
   {
```

```
     case 1:
         Rectangle(hdcmem2,xPos,yPos,xTec,yTec);
                 break;
          case 2:
         int D;
                 D=abs(-xTec+xPos);
                 Ellipse(hdcmem2,xPos,yPos,xPos+D,yPos+D);
                 break;
          case 3:
                 Ellipse(hdcmem2,xPos,yPos,xTec,yTec);
                 break;
   };
   BitBlt(GetDC(hWnd), 0, 0, SCREEN_X, SCREEN_Y, hdcmem2, 0,0,SRCCOPY);
   MOUSE_UP_HDC=false;
 }
 else
 {//////////Юзер отображает вид будущей фигуры - копируем hdcmem<=hdcmem2
   BitBlt(hdcmem, 0, 0, SCREEN_X, SCREEN_Y, hdcmem2, 0,0,SRCCOPY);
   switch (FIGURA)
   {
     case 1:
                 Rectangle(hdcmem,xPos,yPos,xTec,yTec);
                 break;
     case 2:
                 int D;
                 D=abs(-xTec+xPos);
                 Ellipse(hdcmem,xPos,yPos,xPos+D,yPos+D);
                 break;
      case 3:
                 Ellipse(hdcmem,xPos,yPos,xTec,yTec);
                 break;
   };
   BitBlt(GetDC(hWnd), 0, 0, SCREEN_X, SCREEN_Y, hdcmem, 0,0,SRCCOPY);
 }
 DeleteObject(hp);
 DeleteObject(hBr);
}
////////////////////////////////////////////////////////////////////////
////////////////////////////Выбор цвета/////////////////////////////////
////////////////////////////////////////////////////////////////////////
void RGB_USER()
{
   ////////////////////////////////////////////////////////////////////
   HBRUSH hbrush;
   CHOOSECOLOR cc;
   static COLORREF acrCustClr[16];
   ////////////////////////////////////////////////////////////////////
   ZeroMemory(&cc, sizeof(CHOOSECOLOR));
   cc.lStructSize = sizeof(CHOOSECOLOR);
   cc.hwndOwner = hWnd;
   cc.lpCustColors = (LPDWORD) acrCustClr;
   cc.rgbResult = CR;
   cc.Flags = CC_FULLOPEN | CC_RGBINIT;
   ////////////////////////////////////////////////////////////////////
   if (ChooseColor(&cc)==TRUE)
   {
    hbrush = CreateSolidBrush(cc.rgbResult);
    CR = cc.rgbResult;
   }
}
```

2.10. Примитивная графика, текст, работа с BMP (C# .NET)

Графический объект **Graphics** обеспечивает методами, чтобы рисовать разнообразные линии и формы. Простые или сложные формы могут быть предоставлены в различных цветах. Линии, кривые, и формы рисуются с использованием объекта **Pen**. Чтобы заполнить область, типа прямоугольника или закрытой кривой, требуется объект **Brush**. Шрифт теста задается объектом **Font**.

Пример вывода графики и текста:

```
private void button1_Click(object sender, System.EventArgs e)
       {
         Pen pn = new Pen(Color.Red);
         pn.Width = 5;
         Graphics gs = this.CreateGraphics();
         gs.DrawRectangle(pn,1,1,67,67);
         gs.DrawLine(pn, 1, 1, 45, 65);
         gs.DrawPolygon(pn, new PointF[] {new PointF(1, 1),
                           new PointF (20, 10), new PointF(5, 4), new PointF(100, 2),
                           new PointF(200, 50), new PointF(39, 45)});
        ////////////////////////////////////////////////////////////////
         System.Drawing.Graphics graphics = this.CreateGraphics();
         System.Drawing.Rectangle rectangle =
                       new System.Drawing.Rectangle(100, 100, 200, 200);
         graphics.DrawEllipse(System.Drawing.Pens.Black, rectangle);
         graphics.DrawRectangle(System.Drawing.Pens.Red, rectangle);
        ////////////////////////////////////////////////////////////////
        Font myFont = new Font("Times New Roman", 24);
         System.Drawing.Drawing2D.LinearGradientBrush myBrush = new
                           System.Drawing.Drawing2D.LinearGradientBrush(ClientRectangle,
                           Color.Red, Color.Yellow, System.Drawing.Drawing2D.
                           LinearGradientMode.Horizontal);
         gs.DrawString("Look at this text!", myFont, myBrush, new
                           RectangleF(10, 10, 100, 200));
       }
```

Для работы с битовыми образами необходим компонент **PictureBox**. Он может работать с графическими файлами:

Тип	Расширение файла
Bitmap	.bmp
Icon	.ico
GIF	.gif
Metafile	.wmf
JPEG	.jpg

Можно загружать файлы, как на этапе построения, так и на этапе работы программы:

```
private void button1_Click(object sender, System.EventArgs e)
  {
    string path = @"e://1.jpg";
    pictureBox1.SizeMode = PictureBoxSizeMode.StretchImage;
    pictureBox1.Image = Image.FromFile(path);
  }
```

2.11. Диалоговые окна и интерфейс SDI и MDI (C Windows NT API)

Многооконный интерфейс (Multiple Document Interface, **MDI**) является спецификацией для приложений, которые обрабатывают документы в Microsoft Windows. Спецификация описывает структуру окон и пользовательский интерфейс, который позволяет пользователю работать с несколькими документами внутри одного приложения. Точно также, как Windows поддерживает несколько окон приложений на одном экране, приложение MDI поддерживает несколько окон документов в одной рабочей области. Первым приложением MDI для Windows была первая версия Microsoft Excel. И Microsoft Word for Windows, и Microsoft Access являются приложениями MDI.

Элементы MDI

Главное окно приложения программы MDI обычно: в нем имеется строка заголовка, меню, рамка изменения размера, значок системного меню и значки свертывания/развертывания. Рабочая область, однако, не используется непосредственно для вывода выходных данных программы. В этой рабочей области может находиться несколько дочерних окон, в каждом из которых отображается какой-то документ.

Эти дочерние окна выглядят совершенно также, как обычные окна приложений. В них имеется строка заголовка, рамка изменения размера, значок системного меню, значки свертывания/развертывания и, возможно, полосы прокрутки. Однако, ни в одном из окон документов нет меню. Меню главного окна приложения относится и к окнам документов.

В каждый конкретный момент времени только одно окно документа активно (об этом говорит выделенная подсветкой строка заголовка) и находится над всеми остальными окнами документов. Все дочерние окна документов находятся только в рабочей области главного окна приложения и никогда не выходят за ее границы.

Поначалу, MDI для Windows-программиста кажется совершенно понятным. Все, что нужно сделать — это создать для каждого документа окно **WS_CHILD**, делая главное окно приложения родительским окном для окна документа. Но при более близком знакомстве с приложением MDI, таким как Microsoft Excel, обнаруживаются определенные трудности, требующие сложного программирования. Например:

- Окно документа MDI может быть свернуто. Соответствующий значок выводится в нижней части рабочей области. (Как правило, в приложении MDI для главного окна приложения и для каждого типа окна документа будут использоваться разные значки.)
- Окно документа MDI может быть развернуто. В этом случае строка заголовка окна документа (которая обычно используется для вывода в окне имени файла документа) исчезает, и имя файла оказывается присоединенным к имени приложения в строке заголовка окна приложения. Значок системного меню окна документа становится первым пунктом строки основного меню окна приложения. Значок для восстановления размера окна документа становится последним пунктом строки основного меню и оказывается крайним справа.
- Системные быстрые клавиши для закрытия окна документа те же, что и для закрытия главного окна, за исключением того, что клавиша <Ctrl> используется вместо клавиши <Alt>. Таким образом, комбинация

<Alt>+<F4> закрывает окно приложения, а комбинация <Ctrl>+<F4> закрывает окно документа. Вдобавок к остальным быстрым клавишам, комбинация <Ctrl>+<F6> позволяет переключаться между дочерними окнами документов активного приложения MDI. Комбинация <Alt>+<Spacebar>, как обычно, вызывает системное меню главного окна. Комбинация <Alt>+<-> (минус) вызывает системное меню активного дочернего окна документа.

- При использовании клавиш управления курсором для перемещения по пунктам меню, обычно происходит переход от системного меню к первому пункту строки меню. В приложении MDI порядок перехода изменяется: от системного меню приложения — к системному меню активного документа — и далее к первому пункту строки меню.
- Если приложение имеет возможность поддерживать несколько типов дочерних окон (например, электронные таблицы и диаграммы в Microsoft Excel), то меню должно отражать операции, ассоциированные с каждым типом документа. Для этого требуется, чтобы программа изменяла меню программы, когда становится активным окно документа другого типа. Кроме этого, при отсутствии окна документа, в меню должны быть представлены только операции, связанные с открытием нового документа.
- В строке основного меню имеется пункт Window. По соглашению, он является последним пунктом строки основного меню, исключая Help. В подменю Window обычно имеются опции для упорядочивания окон документов внутри рабочей области. Окна документов можно расположить "каскадно" (cascaded), начиная от верхнего левого угла, или "мозаично" (tiled) так, что окно каждого документа будет полностью видимо.

Кроме того, в этом подменю имеется перечень всех окон документов. При выборе одного из окон документов, оно выходит на передний план. При знакомстве с поддержкой MDI в Windows требуется кое-какая новая терминология. Окно приложения в целом называется **главным окном** (frame window). Также как в традиционной программе для Windows, это окно имеет стиль **WS_OVERLAPPEDWINDOW**.

Приложение MDI также создает **окно-администратор** (client window) на основе предопределенного класса окна MDICLIENT. Окно-администратор создается с помощью вызова функции ***CreateWindow*** с использованием этого класса окна и стиля WS_CHILD. Последним параметром функции *CreateWindow* является указатель на небольшую структуру типа **CLIENTCREATESTRUCT**. Это окно-администратор охватывает всю рабочую область главного окна и обеспечивает основную поддержку MDI. Цветом окна-администратора является системный цвет **COLOR_APPWORKSPACE**.

Окна документов называются **дочерними окнами** (child windows). Эти окна создаются путем инициализации структуры типа **MDICREATESTRUCT** и посылки окну-администратору сообщения WM_MDICREATE с указателем на эту структуру.

Окна документов являются дочерними окнами окна-администратора, которое, в свою очередь, является дочерним окном главного окна.

Для главного окна и для каждого типа дочерних окон, которые поддерживаются в приложении, необходим класс окна (и оконная процедура). Для окна-администратора оконная процедура не нужна, поскольку ее класс окна предварительно зарегистрирован в системе.

Для поддержки MDI в Windows имеется один класс окна, пять функций, две структуры данных и двенадцать сообщений.

Две из пяти функций заменяют в приложениях MDI функцию ***DefWindowProc***: вместо вызова функции *DefWindowProc* для всех

необрабатываемых сообщений, оконная процедура главного окна вызывает функцию *DefFrameProc*, а оконная процедура дочернего окна вызывает функцию *DefMDIChildProc*. Другая характерная функция MDI ***TranslateMDISysAccel*** используется также, как функция *TranslateAccelerator*. В Windows добавлена функция ***ArrangeIconicWindows***, но одно из специальных сообщений MDI делает ее использование не нужным в программах MDI.

Если в дочерних окнах MDI выполняются какие-то протяженные во времени операции, рассмотрите возможность их запуска в отдельных потоках. Это позволит пользователю покинуть "задумавшееся" дочернее окно и продолжить работу в другом окне, пока первое дочернее окно решает свою задачу в фоновом режиме. В Windows специально для этой цели имеется новая функция ***CreateMDIWindow***. Поток вызывает функцию *CreateMDIWindow* для создания дочернего окна MDI; таким образом окно действует исключительно внутри контекста потока. В программе, имеющей один поток, для создания дочернего окна функция *CreateMDIWindow* не требуется, поскольку то же самое выполняет сообщение **WM_MDICREATE.**

Работу с обычными диалогами рассмотрим на примере ниже. Это модальные диалоги, определенные программистом на этапе проектирования программы, или динамического создания диалога. Диалоговое окно может загружаться из файла ресурсов. Данный механизм очень удобен, так как позволяет создавать диалоги не динамически и на этапе конструирования программы. Рассмотрим диалоги, которые необходимы почти в любой программе – это диалоги открытия файла и т.п.

Библиотека стандартных диалоговых окон обеспечивает создание функций и структур для каждого типа общего диалогового окна. Чтобы использовать стандартных диалоговое окно в самой простой его форме, вы вызываете создающую его функцию и устанавливаете указатель на структуру, содержащую начальные значения и флажки параметров. После инициализации блока диалога, процедура диалогового окна использует структуру, чтобы возвратить информацию о вводе данных пользователем. Вы также можете настроить стандартное диалоговое окно так, чтобы удовлетворить нужды вашей прикладной программы.

Следующая таблица дает краткое описание различных типов стандартных (общих) диалоговых окон и показывает функцию и структуру, используемую с каждым типом.

● **Цвет (Color)** Показывает доступные цвета и позволяет пользователю создавать необязательные дополнительные цвета. Пользователь может выбирать основной или дополнительный цвет. Используется функция **ChooseColor** и структура **CHOOSECOLOR.**

● **Найти (Find)** Показывает диалоговое окно, в котором пользователь может вводить с клавиатуры строку для поиска. Пользователь может также устанавливать параметры поиска, такие как направление поиска и поиск по принципу чувствительности к регистру. Используется функция **FindText** и структура **FINDREPLACE.**

● **Шрифт (Font)** Показывает списки доступных шрифтов, размеров (кеглей) в пунктах, и другие атрибуты шрифта, которые пользователь может выбирать. Используется функция **ChooseFont** и структура **CHOOSEFONT.**

● **Открыть (Open)** Отображает на экране перечни дисков, каталогов и имен файлов с расширением, из которых пользователь может выбрать для показа список имен файлов. Пользователь может ввести с клавиатуры имя файла или выбрать его из списка, который определяет файлы предназначенные

для открытия. Используется функция **GetOpenFileName** и структура **OPENFILENAME.**

● **Печать (Print)** Показывает на экране информацию об установленном принтере и его конфигурации. Пользователь может выбирать параметры задания по выводу на печать, такие как диапазон страниц для печати и число копий и запускать процесс печатания. Используется функция **PrintDlg** и структура **PRINTDLG.**

● **Параметры страницы (Page Setup)** Показывает на экране конфигурацию текущей страницы. Пользователь может выбирать параметры конфигурации страницы, такие как ориентация бумаги, размер, источник и поля. Используется функция **PageSetupDlg** и структура **PAGESETUPDLG.**

● **Заменить (Replace)** Показывает диалоговое окно, в котором пользователь может ввести с клавиатуры строку, которую надо найти и строку замены. Пользователь может устанавливать параметры поиска, такие как, является ли поиск чувствительным к регистру и параметры замены, такие как область замены. Используется функция **ReplaceText** и структура **FINDREPLACE.**

● **Сохранить как (Save As)** Показывает списки дисков, каталогов и имен файлов с расширениями, из которых пользователь может выбрать для показа список имен файлов. Пользователь может ввести с клавиатуры имя файла или выбрать его из списка, чтобы определить имя под которым сохранится файл. Используется функция **GetSaveFileName** и структура **OPENFILENAME.**

Хотя диалоговое окно Параметры печати (**Print Setup**) тоже доступно, оно было заменено диалоговым окном Параметры страницы (**Page Setup**). Новые прикладные программы, написанные для **Windows NT** как правило должны использовать диалоговое окно Параметры страницы (**Page Setup**), а не диалоговое окно Параметры печати (**Print Setup**).

Все стандартные диалоговые окна модальные, за исключением диалоговых окон Найти (**Find**) и Заменить (**Replace**). Модальные диалоговые окна должны быть закрыты пользователем до того, как функция, используемая, чтобы создать диалоговое окно, сможет возвратить значение. Диалоговые окна Найти (**Find**) и Заменить (**Replace**) немодальные; функция возвращает значение перед закрытием блока диалога. Если вы используете диалоговые окна Найти (**Find**) и Заменить (**Replace**), вы должны тоже использовать функцию **IsDialogMessage** в основном цикле обработки сообщений вашей прикладной программы, чтобы гарантировать, что эти диалоговые окна правильно обрабатывают ввод данных с клавиатуры, таких как клавиши **TAB** и **ESC.**

Пример диалога открытия файла BMP (не Windows диалог) и диалога выбора цвета (Windows диалог), который вызывается в процедуре RGB_USER(). В первом диалоге показано, как получить информацию в программу из дочернего окна диалога – текстового окна ввода пути для файла.

```
LRESULT CALLBACK BMP_PROC(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
        switch (message)
        {
                case WM_COMMAND:
                        if (LOWORD(wParam) == IDOK)
                        {
                                GetDlgItemText(hDlg,IDC_EDIT1,PCHAR(S), 255);
                                EndDialog(hDlg, LOWORD(wParam));
                                return TRUE;
                        }
                        if (LOWORD(wParam) == IDCANCEL)
                        {
                                EndDialog(hDlg, LOWORD(wParam));
                                return TRUE;
                        }
                        break;
        }
    return FALSE;
}
void RGB_USER()
{
   ////////////////////////////////////////////////////////////////////
   HBRUSH hbrush;
   CHOOSECOLOR cc;
   static COLORREF acrCustClr[16];
   ////////////////////////////////////////////////////////////////////
   ZeroMemory(&cc, sizeof(CHOOSECOLOR));
   cc.lStructSize = sizeof(CHOOSECOLOR);
   cc.hwndOwner = hWnd;
   cc.lpCustColors = (LPDWORD) acrCustClr;
   cc.rgbResult = CR;
   cc.Flags = CC_FULLOPEN | CC_RGBINIT;
   ////////////////////////////////////////////////////////////////////
   if (ChooseColor(&cc)==TRUE)
   {
    hbrush = CreateSolidBrush(cc.rgbResult);
    CR = cc.rgbResult;
   }
}
```

2.11. Диалоговые окна и интерфейс SDI и MDI (C# .NET)

Многооконный интерфейс документа (MDI) программы позволяет показывать множество окон в одно и тоже время. MDI приложения часто имеют пункт меню с подменю для переключения между окнами или документами.

Обратите внимание, что есть некоторые различия между формами MDI и SDI интерфейсом. Свойство **Opacity** не затрагивает MDI формы. CenterToParent метод не затрагивает MDI формы.

Для того чтобы главное окно было контейнером для дочерних окон, необходимо в свойстве **IsMDIContainer** установить значение в true. Затем необходимо добавить к проекту дополнительную форму (Form2). В обработчике инициализации главного окна необходимо создать процедуру:

```
Form2 newMDIChild = new Form2();
newMDIChild.MdiParent = this;
newMDIChild.Show();
```

Теперь при запуске приложения в главной форме (Form1) появится форма Form2.

Диалоговые окна позволяют пользователю вести диалог с программой. При этом класс **CommonDialog** отвечает за работу со стандартными диалогами. Рассмотрим различные наследуемые классы исходного класса:

- ColorDialog Class
- FontDialog Class
- OpenFileDialog Class
- PageSetupDialog Class
- PrintDialog Class
- PrintPreviewDialog Class
- SaveFileDialog Class

Здесь понятно кто за что отвечает. Приведем пример для работы с диалогами:

```
private void button1_Click(object sender, System.EventArgs e)
            {
                  ColorDialog MyDialog = new ColorDialog();
                  MyDialog.AllowFullOpen = false ;
                  MyDialog.ShowHelp = true ;
                  MyDialog.Color = textBox1.ForeColor ;
                  if (MyDialog.ShowDialog() == DialogResult.OK)
                        textBox1.ForeColor =  MyDialog.Color;
            }
```

Здесь создается и показывается окно выбора цвета текста компонента textBox1.

3. Дополнительные сведения о C# .NET.

3.1. Обзор платформы .NET.

Андерс Хиджисберг, который возглавил в Microsoft работу по созданию языка C#, следующим образом определил стоявшие перед ними цели:

- создать первый компонентно-ориентированный язык программирования семейства C/C++;
- создать объектно-ориентированный язык, в котором любая сущность представляется объектом;
- упростить C++, сохранив его мощь и основные конструкции.

Главное новшество связано с заявленной компонентной ориентированностью языка. Компоненты позволяют решать проблему модульного построения приложений на новом уровне. Построение компонентов обычно определяется не только языком, но и платформой, на которой этот язык реализован.

Платформа .NET — многоязыковая среда, открытая для свободного включения новых языков, создаваемых не только Microsoft, но и другими фирмами. Все языки, включаемые в платформу .NET, должны опираться на единый каркас, роль которого играет .NET Framework. Это серьезное ограничение, одновременно являющееся и важнейшим достоинством.

В основе большинства приложений, создаваемых в среде VC++ 6.0, лежал каркас приложений (Application Framework), ключевую роль в котором играла библиотека классов MFC. Когда создавался новый проект MFC — EXE, ActiveX или DLL, из каркаса приложений выбирались классы, необходимые для построения проекта с заданными свойствами. Выбранные классы определяли каркас конкретного приложения.

Каркас .NET также содержит библиотеку классов (Class Library). Она служит тем же целям, что и любая библиотека классов, входящая в каркас. Библиотека включает множество интерфейсов и классов, объединенных в группы по тематике. Каждая группа задается пространством имен (namespace), корневое пространство имен называется ***System***. Классы библиотеки связаны отношением наследования. Все классы являются наследниками класса ***System.Object***. Для классов библиотеки, равно как и для классов в языке C#, не определено множественное наследование.

Библиотека классов — это статическая составляющая каркаса. В ***.NET Framework*** есть и динамическая составляющая — система, определяющая среду выполнения — ***CLR*** (Common Language Runtime). Роль этой среды весьма велика — в ее функции входит управление памятью, потоками, безопасностью, компиляция из промежуточного байт-кода в машинный код и многое другое. Важный элемент CLR — это мощный механизм «сборки мусора» (garbage collector), управляющий работой с памятью.

Язык C# в полной мере позволяет использовать все возможности CLR. Код, создаваемый на C#, в принципе безопасен. Иными словами, если вы будете придерживаться установленной методики программирования на C#, то вряд ли сможете написать код, способный привести к неправильному обращению с памятью.

В целом, следует отметить следующие моменты:

- среда .NET Framework, задающая единый каркас многоязыковой среды разработки приложений, во многом ориентирована на компонентное программирование. Она оказывает несомненное влияние на все

языки, поддерживающие эту технологию;

- разработчики .NET Framework просто не могли не создать новый язык программирования, который в полной мере отвечал бы всем возможностям .NET Framework, не неся на себе бремени прошлого. Таким языком и стал язык C#;
- главным достоинством языка C# можно назвать его согласованность с возможностями .NET Framework и вытекающую отсюда компонентную ориентированность.

Программа на C# состоит из одного или нескольких файлов. Каждый файл может содержать одно или несколько пространств имен. Каждое пространство имен может содержать вложенные пространства имен и типы, такие как классы, структуры, интерфейсы, перечисления и делегаты — функциональные типы. При создании нового проекта C# в среде Visual Studio выбирается один из 10 возможных типов проектов, в том числе Windows Application, Class Library, Web Control Library, ASP.NET Application и ASP.NET Web Service. На основании сделанного выбора автоматически создается каркас проекта.

3.2. Сборки

Что такое сборки

Выражаясь упрощенно, можно сказать, что сборки представляют собой основные строительные блоки, обеспечивающие развертывание и исполнение приложений .NET Framework. Каждый исполняемый файл, библиотека или файл ресурсов, созданный на платформе .NET, является сборкой — иного не дано.

Сборки — не только фундаментальные компоненты в механизме развертывания приложений. Они представляют собой основную конструкцию, которая определяет систему управления версиями, область действия типов и ресурсов, зависимости, а также управление правами доступа.

Применение сборок привнесло значительные изменения по сравнению с технологиями, использовавшимися до появления .NET, в которых исполняемые или DLL-файлы сохраняли лишь непосредственный код приложения или библиотеки, а метаданные, описывающие содержимое, помещались во внешних файлах. Фактически, внутренняя организация сборки .NET принципиально отличается от той, которая используется в классическом случае исполняемого файла или DLL-библиотеки.

Единственное очевидное сходство между двумя подходами заключается в тех целях, ради которых создаются файлы, а также в том, что эти файлы имеют расширения exe или dll.

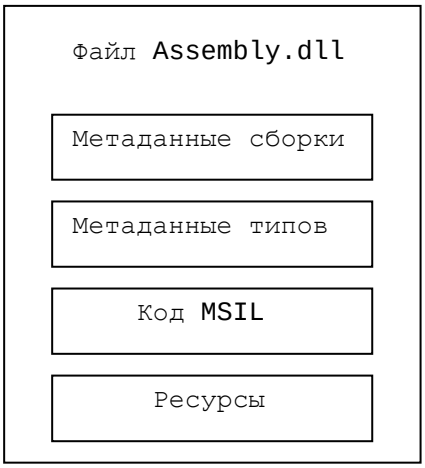
Структура сборки

Сборка представляет собой набор элементов, вместе образующих сегмент приложения. Для формирования всего приложения требуется несколько сборок. Сборки могут состоять из четырех различных типов управляемых элементов, таких как:

- **Декларация (manifest) сборки** — содержит метаданные, описывающие всю сборку. Сюда включается информация обо всех элементах, образующих сборку, и о связях этих элементов друг с другом. Декларация сборки также содержит данные, относящиеся к версиям сборки, безопасности и т.д.
- **Метаданные** - предоставляют информацию о типах, объявленных в сборке, т.е. об именах типов, их областях видимости, базовых классах и реализуемых интерфейсах. Наличие в сборке метаданных и декларации означает, что сборка является полностью самодокументированной.
- **Код приложения** — код, который компилируется в формат **промежуточного языка Microsoft (Microsoft Intermediate Language - MSIL)** и реализует типы, описываемые метаданными.
- **Другие ресурсы**, такие как графические файлы, курсоры и статический текст.

Кроме того, сборка также может содержать неуправляемый код и неуправляемые данные. Существует множество способов группировки четырех перечисленных элементов в сборке. Единственным обязательным элементом сборки является декларация, однако без метаданных, кода приложения или ресурсов от сборки будет немного пользы. Сборка не обязана представлять собой один физический файл. Каждый из указанных нами элементов, кроме декларации, может быть распределен среди

нескольких файлов. Однако чаще всего для этих целей используется единственный файл:



Такой файл называется **переносимым исполняемым** файлом **(Portable Executable - PE).** Чаще всего сборки, состоящие из одного файла, включают лишь декларацию, метаданные и код MSIL.

Если сборка содержит несколько файлов, ее элементы размещаются в отдельных модулях. Модули могут состоять из откомпилированного кода MSIL, файлов с расширениями bmp или jpeg либо других файлов ресурсов. Модули, которые образуют сборку, состоящую из нескольких файлов, хранятся в файловой системе как отдельные файлы, физически никоим образом не связанные друг другом. Вместо этого они связываются логически через декларацию сборки, где содержится детальная информация о файлах, составляющих сборку. При этом отдельные файлы "не знают" о том, что они являются частью сборки. Все файлы должны находиться в одном и том же физическом каталоге, и на их основе можно создать сборку с помощью утилиты **Assembly Linker** (Al.exe). Среда CLR работает с многофайловой сборкой как с единым объектом.

Лишь один из модулей сборки может содержать декларацию. При этом возможна ситуация, когда единственный модуль включает только декларацию. Но чаще всего декларация оказывается упакованной в один из связанных модулей. Ниже приведен пример многофайловой сборки, которая распределена среди четырех файлов. Декларация входит только в файл Assembly.exe.

Многофайловые сборки полезны при создании больших приложений, состоящих из компонентов, разрабатываемых на разных языках. В этом случае можно выделить для каждого конкретного языка свой собственный модуль и поручить разработку отдельных модулей различным разработчикам. Сборки, состоящие из многих файлов, удобно также применять для оптимизации процессов загрузки. В этом случае имеет смысл поместить редко используемые типы и методы в отдельные модули и загружать их только в случае необходимости.

Основные характеристики сборок

Ниже перечислены некоторые из основных характеристик сборок, облегчающие их использование, инсталляцию и распространение:

- Сборки определяют границы системы защиты для приложения. Они задают набор правил доступа, которые не должны нарушаться во время выполнения, чтобы приложение работало корректно.
- Благодаря сборкам предотвращаются конфликты типов, поскольку полное имя создаваемого разработчиком типа включает имя сборки, в которой этот тип определен.
- Декларация сборки содержит информацию, достаточную для работы с внешними ссылками, а также описывает все типы и ресурсы, предоставляемые сборкой.
- Сборка точно определяет, какие из версий других сборок, на которые она ссылается, ей необходимы.
- Сборки поддерживают самоописание. Вся информация, необходимая для исполнения сборки, содержится в ней самой.
- Благодаря возможностям управления версиями сборок две сборки в приложении могут ссылаться на различные версии третьей сборки, не вступая в конфликт друг с другом.
- Преимуществом сборок является отсутствие проблем, связанных с инсталляцией и деинсталляцией. Инсталляция часто сводится к простому перетаскиванию мышью каталога приложения в выбранное нами место. Для автоматической инсталляции оказывается достаточно команды xcopy.

Закрытые и совместно используемые сборки

Сборки бывают двух видов: закрытые и совместно используемые. Эти широко трактуемые понятия достаточно хорошо описывают ситуацию. Оба типа сборок могут состоять из одного или нескольких модулей, но при этом каждый из них имеет лишь одну декларацию. Фактически и тот и другой тип предоставляют одинаковую функциональность. Единственное различие между ними состоит в том, для каких сборок они доступны.

Закрытые сборки (private assemblies) предназначены исключительно для приложения, которое их использует. Они должны находиться в каталоге приложения или в одном из его подкаталогов. Поскольку такие сборки могут использоваться лишь единственным приложением, их именование и задание информации о версиях не создают никаких проблем.

Разработчик может не беспокоиться о конфликтах с другими приложениями. Если он придерживается своих собственных соглашений об именах и делает сборки уникальными в пределах приложения, трудностей не возникнет.

С **совместно используемыми сборками (shared assemblies)** могут одновременно работать несколько приложений. В отличие от закрытых сборок совместно используемые сборки размещаются в **глобальном кэше сборок (Global Assembly Cache - GAC).** GAC представляет собой область дисковой памяти, доступную всем приложениям на машине и резервируемую только для сборок. Мы уже применяли совместно используемые сборки в нашем коде — все сборки, содержащие реализации классов .NET Framework, являются совместно используемыми и размещаются в GAC.

Поскольку такие сборки могут использоваться несколькими приложениями, их именование и нумерация версий для них должны выполняться намного более аккуратно. Например, если два разных разработчика создадут сборки с одним и тем же именем, такая ситуация потенциально грозит конфликтом. Чтобы избежать этого, совместно

используемые сборки должны также иметь строгое **имя** (strong name), которое является гарантированно уникальным.

Вполне реален и такой случай, когда двум приложениям на одной и той же машине потребуются различные версии той же самой сборки. Применение совместно используемых сборок позволяет решить эту проблему. Однако, в отличие от COM, сборки не обязательно должны поддерживать обратную совместимость (хотя это и не помешает). Если приложение получит доступ к некорректной версии сборки, оно, скорее всего, не будет нормально работать.

Глобальный кэш сборок

Как свидетельствует само название, **глобальный кэш сборок (Global Assembly Cache, GAC)** представляет собой кэш, где хранятся сборки. Это звучит достаточно претенциозно, поскольку фактически GAC представляет собой всего лишь каталог (с открытым доступом и фиксированным местоположением), в котором сохраняются совместно используемые сборки. Сборки, помещаемые в GAC, могут иметь три различных формата:

- Машинонезависимый код, т.е. MSIL
- Собственный машинный код
- Кроме того, ASP.NET позволяет загружать сборки так, что они могут использоваться в страницах HTML.

Глобальный кэш сборок представляет собой фиксированный каталог <WINDIR>\assembly. В большинстве систем этот каталог имеет вид: C:\WinNT\assembiy. Поскольку это местоположение гарантировано, CLR не испытывает проблем при поиске совместно используемых сборок, необходимых для исполнения того или иного приложения.

3.3. Обработка сообщений Win32

Для обработки сообщений Api .NET предоставляет удобный механизм, благодаря которому несколько строк кода могут помочь отловить то или иное сообщение. Интеграция Win Api сообщений в код C# очень проста. Для этого рассмотрим пример обработки сообщения OnClick и OnPaint:

```
...
protected override void OnPaint (System.Windows.Forms.PaintEventArgs e)
                    {
                     Pen pn = new Pen(Color.Red);
                     pn.Width = 5;
                     e.Graphics.DrawRectangle(pn,1,1,67,67);
                     }
protected override void OnClick(EventArgs e)
            {
             textBox1.Text="HELLO WIN API";
            }
...
```

Когда главному окну приходят сообщения о перерисовке, то он выполняет перегруженный метод

protected override void OnPaint

который также рисует в окне красный прямоугольник. А при нажатии на форме мышью окну посылается сообщение, которое обрабатывает метод

protected override void OnClick

Он выводит приветствие в текстовом окне.

3.4. Вызов функций Windows NT Api

Используем C# .NET для того чтобы вызвать функцию WIN API GetUserName:

```
BOOL GetUserName(
  LPTSTR lpBuffer,
  LPDWORD nSize
);
```

Чтобы использовать эту функцию в C#, вы должны перевести типы Win32 в C# типы.

Данная функция содержится в Advapi32.dll. Это означает, что код во время выполнения вызывается из Advapi32.dll. Мы помещаем DllImport в наш код:

```
[DllImport("Advapi32.dll",CharSet = CharSet.Auto)]
```

Также мы должны объявить данную функцию как внешнюю:

```
public static extern bool
                  GetUserName(StringBuilder lpB, ref long n);
```

Теперь рассмотрим весь код:

```
using System;
using System.Text;
using System.Runtime.InteropServices;
namespace GETUSERNAME_
{
      class Class1
      {
            [DllImport("Advapi32.dll",CharSet = CharSet.Auto)]
            public static extern bool
                  GetUserName(StringBuilder lpB, ref long n);
            static void Main(string[] args)
            {
                  StringBuilder lpB = new StringBuilder(256);
                  long n=256;
                  GetUserName(lpB,ref n);
                  System.Console.WriteLine(lpB);
                  System.Threading.Thread.Sleep(1000);
            }
      }
}
```

3.5. Делегаты

В большинстве случаев, когда мы вызываем функцию, мы определяем функцию, которую нужно назвать непосредственно. Если бы класс MyClass имел функцию по имени Process, мы вызвали бы:

```
MyClass myClass = new MyClass();
myClass.Process();
```

Это работает хорошо в большинстве ситуаций. Иногда, однако, мы не хотим вызвать функцию непосредственно. Рассмотрите следующий пример:

```
public class MyClass
{
   public void Process()
   {
      Console.WriteLine("Process() begin");
      Console.WriteLine("Process() end");
   }
}
```

Делегат необходим для этого случая; он позволит нам определить то, что функция вызывается без напоминания того, как она функционирует.

Для нашего примера, мы объявим делегат, который имеет один входной параметр и не возвращает значений. Наш класс изменен следующим образом:

```
public class MyClass
{
   public delegate void LogHandler(string message);
   public void Process(LogHandler logHandler)
   {
      if (logHandler != null)
         logHandler("Process() begin");
      if (logHandler != null)
         logHandler ("Process() end");
   }
}
```

Использование делегата - как запрос функции, хотя мы должны добавить проверку, является ли делегат пустым, (то есть не указывающий на функцию) перед запросом функции.

Чтобы вызвать **Process()**, мы должны объявить функцию, которая соответствует делегату.

```
class Test
{
   static void Logger(string s)
   {
      Console.WriteLine(s);
   }
   public static void Main()
   {
      MyClass myClass = new MyClass();
      MyClass.LogHandler lh = new MyClass.LogHandler(Logger);
      myClass.Process(lh);
   }
}
```

Делегат не только указатель функции; он обеспечивает большее количество функциональных возможностей.

Вот - пример того, как обратиться к функции члена:

```
class FileLogger
{
   FileStream fileStream;
   StreamWriter streamWriter;
   public FileLogger(string filename)
   {
      fileStream = new FileStream(filename, FileMode.Create);
      streamWriter = new StreamWriter(fileStream);
   }
   public void Logger(string s)
   {
      streamWriter.WriteLine(s);
   }
   public void Close()
   {
      streamWriter.Close();
      fileStream.Close();
   }
}
class Test
{
   public static void Main()
   {
      FileLogger fl = new FileLogger("process.log");
      MyClass myClass = new MyClass();
      MyClass.LogHandler lh = new MyClass.LogHandler(fl.Logger);
      myClass.Process(lh);
      fl.Close();
   }
}
```

FileLogger класс инкапсулирует файл. Main() изменена так, чтобы делегат вызвал Logger() функцию FileLogger. Когда этот делегат вызван из Process(), функция вызывается, и строка будет применена к соответствующему файлу.

В C# делегаты могут указывать на больше чем одну функцию одновременно. Делегат поддерживает список функций, которые будут вызываться, когда делегат вызван:

```
MyClass.LogHandler lh = (MyClass.LogHandler)
   Delegate.Combine(
                    new Delegate[]
                                {
                                 new MyClass.LogHandler(Logger),
                                 new MyClass.LogHandler(fl.Logger)
                                }
                    );
```

Но C# позволяет и так:

```
MyClass.LogHandler lh = null;
lh += new MyClass.LogHandler(Logger);
lh += new MyClass.LogHandler(fl.Logger);
```

Когда вы вызываете мультиделегат, делегаты в списке обращения вызываются синхронно в порядке, в котором они появляются. Если ошибка происходит в течение этого процесса, процесс прерывается.

Если вы хотите управлять порядком обращения - вы можете получить список обращений от делегата и вызвать функции самостоятельно:

```
foreach (LogHandler logHandler in lh.GetInvocationList())
{
   try
   {
      logHandler(message);
   }
   catch (Exception e)
   {
      // do something with the exception here…
   }
}
```

Код просто обрабатывает каждое обращение так, чтобы возникшее исключение не прервало программу.

3.6. События

Если пользователь нажимает на кнопке, то обрабатывается событие, но нельзя ли сделать так, что бы вызывался делегат.

```
Button.Click = new Button.ClickHandler(ClickFunction);
```

Что означает, что ClickFunction() будет называться, когда нажимают на кнопке. Но в данном коде ошибка – здесь назначен делегат непосредственно. Любой другой делегат, присоединенный к кнопке – теперь не будет вызываться. Необходимо что-то сделать:

```
Button.Click = null;
```

Это удалит всех делегатов. А вот правильный код:

```
public class MyObject
{
   public delegate void ClickHandler(object sender, EventArgs e);
   public event ClickHandler Click;
   protected void OnClick()
   {
      if (Click != null)
         Click(this, null);
   }
}
```

Делегат ClickHandler имеет два параметра. Первый параметр - объект, который послал сообщение, и второй параметр используется, чтобы передать параметры.

Добавление делегата к обработчику нажатия кнопки:

```
class Test
{
   static void ClickFunction(object sender, EventArgs args)
   {
      // process the event here.
   }
   public static void Main()
   {
      MyObject myObject = new MyObject();
      myObject.Click += new MyObject.ClickHandler(ClickFunction);
   }
}
```

Позже можно добавить еще один делегат, который будет обрабатываться при нажатии на кнопку.

3.7. Интерфейсы

Ключ к пониманию интерфейсов лежит в их сравнении с классами. Классы — это объекты, обладающие свойствами и методами, которые на эти свойства воздействуют. Хотя классы проявляют некоторые характеристики, связанные с поведением (методы), они представляют собой предметы, а не действия, присущие интерфейсам. Интерфейсы же позволяют определять характеристики или возможности действий и применять их к классам независимо от иерархии последних. Допустим, у вас есть дистрибьюторское приложение, составляющие которого можно упорядочить. Среди них могут быть классы Customer, Supplier и Invoice. Некоторые другие, скажем, MaintenanceView или Document, упорядочивать не надо. Как упорядочить только выбранные вами классы? Очевидный способ — создать базовый класс с именем типа Serializable. Но у этого подхода есть большой минус: одна ветвь наследования здесь не подходит, так как нам не требуется наследование всех особенностей поведения. C# не поддерживает множественное наследование, так что невозможно произвести данный класс от нескольких классов. А вот интерфейсы позволяют определять набор семантически связанных методов и свойств, способных реализовать избранные классы независимо от их иерархии.

Концептуально интерфейсы представляют собой связки между двумя в корне отличными частями кода. Иначе говоря, при наличии интерфейса и класса, определенного как реализующий данный интерфейс, клиентам класса дается гарантия, что у класса реализованы все методы, определенные в интерфейсе.

Когда вы создаете интерфейс и в определении класса задаете его использование, говорят, что класс реализует интерфейс. ***Интерфейсы*** — это набор характеристик поведения, а класс определяется как реализующий их.

Использование интерфейсов

Чтобы понять, где интерфейсы приносят пользу, рассмотрим традиционную проблему программирования в Windows, когда нужно обеспечить универсальное взаимодействие двух совершенно различных фрагментов кода без использования интерфейсов. Представьте себе, что вы работаете на Microsoft и являетесь ведущим программистом команды по разработке панели управления. Вам надо предоставить универсальные средства, которые дают возможность клиентским апплетам «закрепляться» на панели управления, показывая при этом свой значок и позволяя клиентам выполнять их. Если учесть, что эта функциональность разрабатывалась до появления COM, возникает вопрос: как создать средства интеграции любых будущих приложений с панелью управления? Задуманное решение долгие годы было стандартной частью разработки Windows.

Как ведущий программист по разработке панели управления, вы создаете и документируете функцию (функции), которая должна быть реализована в клиентском приложении, и некоторые правила. В случае апплетов панели управления, Microsoft определила, что для их написания вам нужно создать динамически подключаемую библиотеку, которая реализует и экспортирует функцию CPLApplet. Вам также потребуется добавить к имени этой DLL расширение .cpl и поместить ее в папку

Windows System32 (для Windows ME или Windows 98 это будет Windows\System32, а для Windows 2000 — WINNT\System32). При загрузке панель управления загружает все DLL с расширением .cpl из папки System32 (с помощью функции LoadLibrary), а затем вызывает функцию GetProcAddress для загрузки функции CPLApplet, проверяя таким образом выполнение вами соответствующих правил и возможность корректного взаимодействия с панелью управления.

Как уже говорилось, эта стандартная модель программирования в Windows позволяет выйти из ситуации, когда вы хотите, чтобы ваш код универсальным образом взаимодействовал с кодом, который будет разработан в будущем. Однако это не самое элегантное решение. Главный недостаток этой методики в том, что она вынуждает включать в состав клиента — в данном случае в код панели управления — большие порции проверяющего кода. Например, панель управления не может просто полагаться на допущение, что каждый .cpl-файл в папке является DLL Windows. Панель управления также должна проверить наличие в этой DLL функций коррекции и что эти функции делают именно то, что описано в документации. Здесь-то интерфейсы и вступают в дело. Интерфейсы позволяют создавать такие же средства, связывающие несовместимые фрагменты кода, но при этом они более объектно-ориентированны и гибки. Кроме того, поскольку интерфейсы являются частью языка C#, компилятор гарантирует, что если класс определен как реализующий данный интерфейс, то он выполняет именно те действия, о которых заявляет, что должен их выполнять.

В C# ***интерфейс*** — понятие первостепенной важности, объявляющее ссылочный тип, который включает только объявления методов. Но что значит «понятие первостепенной важности»? Необходимо сказать, что эта встроенная функция является неотъемлемой частью языка. Иначе говоря, это не то, что было добавлено позже, после того как разработка языка была закончена. Давайте подробнее познакомимся с интерфейсами, узнаем, что они собой представляют и как их объявлять.

Объявление интерфейсов

Интерфейсы могут содержать методы, свойства, индексаторы и события, но ни одна из этих сущностей не реализуется в самом интерфейсе.

Рассмотрим их применение на примере. Допустим, вы создаете для вашей компании редактор, содержащий элементы управления Windows. Вы пишете редактор и тестовые программы для проверки элементов управления, размещаемых пользователями на форме редактора. Остальная часть команды пишет элементы управления, которыми будет заполнена форма. Вам почти наверняка понадобятся средства проверки на уровне формы. В определенное время, скажем, когда пользователь явно приказывает форме проверить все элементы управления или при обработке формы, последняя циклически проверяет все прикрепленные к ней элементы управления, или, что более подходяще, заставляет элемент управления проверить самого себя.

Как предоставить такую возможность проверки элемента управления? Именно здесь проявляется превосходство интерфейсов. Вот пример простого интерфейса с единственным методом Validate:

```
interface IValidate
{
  bool Validate();
}
```

Теперь вы можете задокументировать тот факт, что если элемент управления реализует интерфейс IValidate, то этот элемент управления может быть проверен.

Рассмотрим пару аспектов, связанных с приведенным кодом. Вы не должны задавать для метода интерфейса модификатор доступа, такой как public. При указании модификатора доступа перед объявлением метода возникает ошибка периода компиляции. Дело в том, что все методы интерфейса открыты по умолчанию. Кроме методов, интерфейсы могут определять свойства, индексаторы и события:

```
interface IExampleInterface
{
 //Пример объявления свойства
 int testProperty { get; }
 //Пример объявления события,
 event testEvent Changed;
 //Пример объявления индексатора
 string this[int index] { get; set; }
}
```

Создание интерфейсов

Поскольку интерфейс определяет связь между фрагментами кода, любой класс, реализующий интерфейс, должен определять любой и каждый элемент этого интерфейса, иначе код не будет компилироваться.

Интерфейсы позволяют реализовать несколько характеристик поведения в одном классе. На C# можно создавать объекты, производные от одного класса, и в дополнение к этой унаследованной функциональности реализовать столько интерфейсов, сколько нужно для класса.

Интерфейс может унаследовать от одного или более интерфейсов. В следующем примере, интерфейс IMyInterface унаследует от двух интерфейсов IBase1 и IBase2:

```
interface IMyInterface: IBase1, IBase2
{
   void MethodA();
   void MethodB();
}
```

Интерфейсы могут быть осуществлены классами и structs. Идентификатор осуществленного интерфейса появляется в списке базового класса. Например:

```
class Class1: Iface1, Iface2
{
   // class members
}
```

Следующий пример демонстрирует выполнение интерфейса.

```
using System;
interface IPoint
{
   int x
   {
      get;
      set;
   }
   int y
   {
      get;
      set;
   }
}
class MyPoint : IPoint
{
   private int myX;
   private int myY;
   public MyPoint(int x, int y)
   {
      myX = x;
      myY = y;
   }
   public int x
   {
      get
      {
         return myX;
      }
      set
      {
         myX = value;
      }
   }
   public int y
   {
      get
      {
         return myY;
      }
      set
      {
         myY = value;
      }
   }
}
class MainClass
{
   private static void PrintPoint(IPoint p)
   {
      Console.WriteLine("x={0}, y={1}", p.x, p.y);
   }
   public static void Main()
   {
      MyPoint p = new MyPoint(2,3);
      Console.Write("My Point: ");
      PrintPoint(p);
   }
}
```

Комбинирование интерфейсов

Еще одна мощная функция C# — возможность комбинирования двух или более интерфейсов, в результате чего класс должен реализовать только результат комбинирования. Допустим, вы хотите создать новый класс TreeView, реализующий два интерфейса: IDragDrop и ISerializable. Поскольку резонно предполагать, что и другим элементам управления, таким как ListView и ListBox, понадобится скомбинировать эти функции, вам, возможно, захочется скомбинировать интерфейсы IDragDrop и ISerializable в единое целое:

```
public class Control
      {
         public interface IDrugDrop
            {
               void Drag();
               void Drop() ;
            }
         public interface ISerializable
            {
               void Serialize();
            }
         public interface ICombo: IDrugDrop, ISerializable
            {
               //этот интерфейс объединяет в себе
               //два других интерфейса
            }
      }
public class MyTreeView: Control
      {
        public void Drag()
            {
              Console.WriteLine("Вызов метода MyTreeView.Drag");
            }
            public void Drop()
            {
              Console.WriteLine("Вызов метода MyTreeView.Drop");
            }
            public void Serialized()
            {
              Console.WriteLine("Вызов метода MyTreeView.Serialize");
            }
      }
class CombiningApp
      {
         public void Main()
            {
               MyTreeView tree = new MyTreeView ();
               tree.Drag();
               tree.Drop();
               tree.Serialized();
            }
      }
```

Комбинируя интерфейсы, вы не только упростите возможность использование связанных интерфейсов, но при необходимости сможете добавлять к новому «композитному» интерфейсу дополнительные методы.

3.8. Пространство имен namespace

.NET библиотека классов составляет **namespaces**. Каждый namespace содержит типы, которые вы можете использовать в вашей программе: классы, структуры, делегаты, и интерфейсы.

Namespaces обеспечивает возможность двум классам иметь одно и тоже имя, пока они находятся в различном namespaces. Имя namespace - часть полностью квалифицированного имени типа *(namespace.typename)*.

Фундаментальное программирование namespaces в .NET включает:

- **System.Collections**- содержат интерфейсы и классы, которые определяют различные коллекции объектов типа списков, очередей, множеств, хэш таблиц, и словарей.
- **System.IO** - содержит типы, которые позволяют синхронное и асинхронное чтение и запись потоков данных и файлов.
- **System.Text** - содержит классы, представляющие ASCII, Unicode, UTF-7, и UTF-8 коды символов.
- **System.Threading** - обеспечивает классы и интерфейсы, которые позволяют программирование многопоточных приложений. Этот namespace включает **ThreadPool** класс, который управляет группами нитей, класс **Timer**, который позволяет делегату вызываться через указанные промежутки времени, и **Mutex** класс для синхронизирования взаимноблокирующих нитей.

Определяется как:

```
namespace name[.name1] ...]
{
type declarations
}
```

Пример:

```
using System;
namespace SomeNameSpace
{
   public class MyClass
   {
      public static void Main()
      {
         Nested.NestedNameSpaceClass.SayHello();
      }
   }
   namespace Nested
   {
      public class NestedNameSpaceClass
      {
         public static void SayHello()
         {
            Console.WriteLine("Hello");
         }
      }
   }
}
```

3.9. Динамическая компиляция кода

System.CodeDom.Compiler **-** пространство имен, необходимое для заданной задачи.

Пространство имен **System.CodeDom.Compiler** содержит типы для управления генерацией и компиляцией исходного кода в поддерживаемых языках программирования. Каждый генератор кода может создавать исходный код на определенном языке программирования на основе структуры моделей исходного кода модели объектов документов кода (CodeDOM), состоящих из элементов, предоставляемых пространством имен System.CodeDom.

Иерархия пространства имен:

Классы:

Класс	Описание
CodeCompiler	Предоставляет пример реализации интерфейса **ICodeCompiler.**
CodeDomProvider	Предоставляет базовый класс для реализаций **CodeDomProvider**. Это абстрактный класс.
CodeGenerator	Предоставляет пример реализации интерфейса **ICodeGenerator.** Это абстрактный класс.
CodeGeneratorOptions	Представляет набор параметров, используемых генератором кода.
CodeParser	Предоставляет пустую реализацию интерфейса **ICodeParser.**
CompilerError	Представляет ошибку или предупреждение компилятора.
CompilerErrorCollection	Представляет коллекцию объектов **CompilerError.**
CompilerParameters	Представляет параметры, используемые для вызова компилятора.
CompilerResults	Представляет результаты компиляции, возвращаемые компилятором.
Executor	Обеспечивает функции выполнения команд для вызова компиляторов. Этот класс не наследуется.
IndentedTextWriter	Предоставляет текстовый редактор, позволяющий переходить на новую строку с помощью лексемы табуляции.
TempFileCollection	Представляет коллекцию временных файлов.

Интерфейсы

Интерфейс	Описание
ICodeCompiler	Определяет интерфейс для запуска компиляции исходного кода или дерева CodeDOM, используя определенный компилятор.
ICodeGenerator	Определяет интерфейс для создания кода.
ICodeParser	Определяет интерфейс для анализа кода в CodeCompileUnit.

Перечисления

Перечисление	Описание
GeneratorSupport	Задает идентификаторы, определяющие наличие поддержки генератором кода определенных типов элементов кода.
LanguageOptions	

Главная цель динамической компиляции кода - возможность расширения и изменения функциональности приложения без его перекомпиляции.

Нередко встречается ситуация когда, в зависимости от тех или иных факторов, нужно загрузить некоторую сборку (**assembly**) для последующего выполнения содержащегося в ней кода.

Наиболее удобной в использовании технологией динамической компиляции является компиляция исходного кода. Как следует из названия, эта технология позволяет динамически компилировать исходный код. Не правда ли, замечательная возможность – писать макросы для своего приложения на C#?

Но есть и плохая новость – создание сборки происходит не в оперативной памяти, поскольку используется компилятор командной строки (csc.exe). И, хотя у класса **CompilerParameters** есть привлекательное на вид свойство **GenerateInMemory**, присвоив ему значение “true”, вы лишь получите динамическую сборку в виде временного файла. С другой стороны, мыслить нужно позитивно, поэтому можно сказать следующее – .NET Framework предоставляет очень удобную инфраструктуру для динамической компиляции исходного кода (скрывая вызов компилятора командной строки).

Еще одна проблема, которая может возникнуть при частой динамической компиляции исходного кода в рамках одного домена приложения – нехватка памяти. Не забывайте также, что, компилируя дважды одинаковый код, вы получите две сборки. Конечно, не обязательно всегда использовать несколько доменов приложения, иногда вполне допустимо использовать один домен приложения.

Рассмотрим динамическую компиляцию исходного кода на простом примере. Напишем класс, позволяющий запускать «макросы на C#», выдающие информацию пользователю в текстовом виде.

Пример:

```
using System;
using System.Reflection;
using System.CodeDom.Compiler;
using Microsoft.CSharp;
namespace SimpleMacro
  {
   //Чтобы абстрагироваться от способа вывода сообщений (консоль, журнал событий и
   //т.д.) опишем делегат, единственная функция которого – передавать сообщение:
   public delegate void SpeakOut(string message);
   //Чтобы не использовать для запуска макроса рефлексию, опишем интерфейс:
   public interface IMacro
     {
      void Run(SpeakOut speakOut);
     }
   //Теперь, напишем класс, осуществляющий динамическую компиляцию и запуск макроса.
   //Этот класс будет содержать поле типа IMacro и делегат SpeakOut:
   public class Macro {
      readonly SpeakOut speakOut;
      readonly IMacro macro;
      //Основной код, компилирующий макрос, поместим в конструктор класса, параметрами которого являются
      // – исходный код, название класса, реализующего интерфейс IMacro, список подключаемых сборок и
      // делегат SpeakOut:
      public Macro(string code, string className, string [] assemblies, SpeakOut speakOut)
        {
         if( code == null || assemblies == null || speakOut == null )
             throw new ArgumentNullException();
         // запоминаем делегат, для вызова в Run
         this.speakOut = speakOut;
         // создаем экземпляр компилятора
         CSharpCodeProvider codeCompiler = new CSharpCodeProvider();
         // добавляем ссылки на сборки
         CompilerParameters parameters = new CompilerParameters(assemblies);
         // добавляем ссылку на нашу сборку SimpleMacro
         string path = Assembly.GetExecutingAssembly().Location;
         parameters.ReferencedAssemblies.Add(path);
         // компилируем
         CompilerResults results = codeCompiler.CompileAssemblyFromSource(parameters, code);
         // есть ли ошибки?
         if( results.Errors.HasErrors )
          {
           foreach( CompilerError error in results.Errors )
           {
                 speakOut(string.Format("Line:{0:d},Error:{1}\n", error.Line, error.ErrorText));
           }
           throw new ArgumentException("Ошибки при компиляции.");
          }
         // создаем класс
         object objMacro = results.CompiledAssembly.CreateInstance(className);
         if( objMacro == null )
          {
           throw new ArgumentException("Ошибка при создании класса " + className);
          }
         // запоминаем класс как интерфейс
         macro = objMacro as IMacro;
         if( macro == null )
          {
           throw new ArgumentException("Не реализован интерфейс IMacro.");
          }
        }
      //Разумеется, нужно написать метод Run, который будет запускать макрос:
      public void Run()
      {
        macro.Run(speakOut);
      }
      //Теперь осталось написать клиентский код для этого класса. Чтобы не изобретать велосипед для
      //тестового примера, я написал макрос, выводящий на консоль “Hello, world!”:
      const string NAMESPACE_NAME = "Test";
      const string CLASS_NAME = "TestMacro";
      // для нашего случая даже System.dll не нужна
      string [] assemblies = new string[0];
      // формируем исходный код
      string codeString = @"using System;
      using SimpleMacro;
      namespace "+NAMESPACE_NAME+@"
      {
      public class "+CLASS_NAME+@" : IMacro
      {
           public void Run(SpeakOut speakOut)
```

```
            {
                  speakOut("+ "\"Hello, world!\"" +@");
            }
      }
      };
      // компилируем макрос
      Macro macro = new Macro(codeString, NAMESPACE_NAME+"."+CLASS_NAME, assemblies,
                              new SpeakOut(Console.WriteLine) );
      // запускаем макрос
      macro.Run();
}
```

Как видно из примера, динамическая компиляция исходного кода сводится к вызову одного метода с парой параметров. С другой стороны, имеет смысл один раз написать класс, отвечающий за динамическую компиляцию (универсальный или для частного случая) и повторно использовать его в дальнейшем.

Приведенный пример, несмотря на свою простоту, демонстрирует весьма большие возможности, которые открывает для нас .NET Framework.

4. Используемая литература

1. January 2004 Release of the MSDN Library
2. Аравинд Корера, Стивен Фрейзер, Сэм Джентайл, Ниранджан Кумар, Скотт Маклин, Саймон Робинсон, д-р П.Г. Саранг Visual C++ .NET: Пособие для разработчиков C++
3. Лабор В.В. Си Шарп: Создание приложений для Windows.
4. Петцольд Ч. Программирование для Microsoft Windows на C#. В 2-х томах. Пер. с англ. — М.: Издательско-торговый дом «Русская Редакция», 2002.
5. Microsoft Win32 Programmer's Reference
6. Петцольд Ч. Программирование для Microsoft Windows 95 в 2х томах.