+Лек1. Вводная лекция

<<Лек1..wma>>

Запись начата: 8:26 8 февраля 2012 г.

6 февраля 2012 г.
7:30

В этом семестре зачёт, в следующем экзамен.
Рейтинг будет и тут.
Есть порог на зачёт. Балл переходит на следующий семестр.

Язык C++, управляемый и неуправляемый.
MS VS2010. Работаем на кластере.
На этот семестр достаточно Professional. В следующем будут командные проекты.
Синтаксис рассказывать не будет, ну то есть в одну лекцию только.
Остальное - ТП.

Долженкову искать в 241 преподской.

Лабы:

1. Перегрузки
2. Динамика
3. Объединение программ на разных языках
4. Создание Win приложений с использованием DLL.

2-3 контрольных работы.

ТП - совокупность методов, средств и процедур, используемых при создании ПО.
СПрограмма - совокупность операторов, представленных в неких кодах для решения поставленной задачи. Для упрощения создания кодов используются языки прог. Языки на две категории делятся:
1) Низкого уровня

1. ЯП высокого уровня.

Низкого уровня:
Семейство языков ассемблера. Данные средства разработки позволяют получать наиболее короткий и эффективный код. Единственным условием оптимальных программ является знание архитектуры системы, для которой предназначена разрабатываемая программа.

Языки высокого уровня позволяют разрабатывать различное ПО без особой опоры на архитектуру . Из-за этого код их не всегда оптимален, зато разработка ПО довольно проста и может быть выполнена в короткие сроки.

В настоящее время применяют **гибридные** подходы, когда прикладная часть ПО разрабатывается с помощью высокоуровневых языков, а вычислительные операции, требующие высокого быстродействия и использующие внутреннюю структуру процессора - с помощью языков ассемблера.
При разработке ПО определяют парадигму программирования. Парадигма программирования - это способ создания программ с помощью определённых принципов и подходящего языка программирования, позволяющего писать простые ясные программы. В настоящее время существует следующие парадигма программирования:

- **Структурное программирование**. Включает в себя методологию разработки ПО, в основе которой лежит представление программы в виде иерархической структуры.

Вся программа разбивается на отдельные блоки, элементарными блоками являются: последовательность, ветвление и циклический блок.

- **Функциональное программирование**. Согласно данной парадигме программирования, процесс вычисления трактуется как определение значения некоторой функции в её математическом понимании. Функциональное программирование сводится к созданию множества функций, срабатывание которых осуществляется не на основании некоторого выбранного алгоритма, а на основании готовности входных данных той или иной функции. Чаще всего выходные данные функции являются выходными данными для следующей функции. Основной особенностью функциональных сред программирования (языков?) является кэширование результатов, так как известно, что одинаковые входные данные на функцию дают одинаковый результат. (важен порядок аргументов)
  Представитель: lisp.
- **Логическое программирование**. Основано на автоматическом доказательстве теорем на основе заданных фактов и правил вывода. Логическое программирование использует аппарат математической логики. Чаще всего факты и правила описываются в виде **предикатов, высказываний** либо **продукционных правил** (**если - то)**.
  Представитель: пролог (про.граммирования лог.ический язык)
- **Автоматное программирование.** При использовании, программа или её фрагмент рассматривается как модель какого-либо формального автомата. В зависимости от сложности задачи может использоваться либо конечный автомат, либо любой автомат более сложной структуры.
- **Объектно-ориентированное программирование.** Основными компонентами программы являются объекты и классы. Любая объектная программа является результатом взаимодействия объектов, функционирующих на основе имеющихся у них методов, обладающих поведением и возможностью обмена сообщениями. Разновидностью ООП является **прототипное программирование**. В отличии от объектного программирования, механизм наследования заменён механизмом клонирования или создания прототипов, то есть полной копии уже существующего в программе объекта. (В ООП есть родительский класс, все его методы наследуются и могут добавляться/изменяться).
- **Событийно-ориентированное программирование.** Ход программы при таком программировании определяется набором событий, генерируемых пользователем либо другими элементами системы. События представляют собой потоки, организующие взаимодействие между компонентами программы с помощью исполнительных механизмов. (сейчас все интерактивные программы - событийно-ориентированные)
- **Агентно-ориентированное программирование.** Основная концепция состоит в том, чтобы поместить в среду действующий объект, называемый агентом. Агент воспринимает состояние среды с помощью "датчиков" и реагирует на их состояние путём реализации функций. Современное.

Для каждого вида задач подходят разные парадигмы.

В настоящее время не существует средства программирования, которое придерживалось бы только одной парадигмы программирования.Чаще всего это комплексные средства. VS поддерживает структурное, ООП есть, событийное есть. Логического и

функционального нет, автоматного в чистом виде нет. Агентно-ориентированного в чистом виде нет, но никто не мешает реализовать.

Лек1. Технология .NET

8 февраля 2012 г.
8:49

Технология .NET предложена компанией Microsoft с целью внедрения разнородных приложений в интернет-сообщества. Технология .NET обладает улучшенной функциональной совместимостью, в основе которой лежит использование открытых стандартов. В рамках данной технологии повышается устойчивость приложений, расширяется набор возможностей за счёт использования языков разметок (типа XML).
В состав платформы входят следующие составляющие:

1. Платформа .NET представляет в первую очередь устойчивую общеязыковую среду выполнения CLR (Common Language Runtime). Эта среда входит в состав платформы и обеспечивает поддержку многоязыковых приложений. Позволяет организовать безопасную работу с элементами системы. Обеспечивает корректный доступ к динамической памяти и сборку мусора в реальном времени.
2. Средства разработки приложений на ряде языков программирования. В платформу .NET включены, VB, VC, C#, и так далее.
3. Библиотека каркасных классов .NET Framework, то есть подготовленная песочница для создания кода. Данная библиотека содержит многократно используемый код, доступный на любом языке программирования, включённом в состав .NET
4. Поддержка сетевой инфраструктуры, построенной на верхнем слое сети интернет. Поддерживается даже сетевого уровня.. Для создания устойчивых приложений.
5. Поддержка промышленного стандарта создания веб-служб. В рамках .NET поддерживается два: SOAP (более новый) и давно применяемый, но модифицированный здесь ASP.
6. Собственная модель безопасности, используемой в приложении, которую можно использовать для доступа к памяти и прочим общим ресурсам.
7. Мощные инструментальные средства для визуальной разработки приложений различного рода.

SOAP позволяет использовать функции приложений в любом месте Internet. С точки зрения программиста, программа разработанная в рамках платформы .NET не отличается по своей структуре в зависимости от того, работает она на локальной машине пользователя или в удалённой среде. Используемая модель программы остаётся неизменной, программа использует встроенные компоненты, которые должны быть установлены на локальной и удалённой машине.

Технология ASP (технология активных страниц) была модифицирована .NET. В основе ASP лежит код сценариев, в который встраиваются команды форматирования некоторого текста. Код сценариев реализуется на одном из языков программирования и обладает ограниченными возможностями. ASP.NET позволяет создавать коды фактически на всех языках, встроенных в .NET платформу, позволяет создавать активные формы, реагирующие на запросы локальных и удалённых устройств.
ASP автоматически поддерживает функциональность многих браузеров.

Если говорить о структуре платформы .NET, то её можно рассматривать как некоторую

надстройку над ОС.
Платформа .NET состоит из следующих частей:

- Средства разработки (VS)
- Сам каркас
  - Статическая часть, включающая библиотеки разработки классов
  - Динамическая часть. Общеязыковая среда выполнения CLR.
- .NET Enterprise Servers
- .NET Building Block Services

Что такое каркас, базовая часть для рутинных операций.) программисту достаточно выбрать приложение для работы с базами данных, и в его программу будет включён весь код работы с этими БД.
Или редактор документов: часть работающая с форматом файлов, и часть отвечающая за просмотр.
В каркасе много лишнего кода.
Каркас снижает производительность.
Каркас жёстко регламентирует назначение (если веб-служба, то и всё тут, никаких других).

Каркас включает в себя статическую часть (библиотеку каркасных классов) **FCL (Framework Component Library).** В состав входит:

- Библиотека базовых классов. Работа со строками, массивами и форматированием.
- Набор классов для передачи данных по сети. (класс удалённой обработки)
- Набор классов диагностики.
- Набор классов обеспечения ввода-вывода и интерфейса пользователя.
- Классы для работы с данными и языком разметки XML
- Классы организации Web-служб и отдельный элемент - общая система типов (CTS). CTS позволяет создавать внутри платформы многоязыковые приложения. С этой целью введена внутренняя типизация. В каждом языке программирования, входящем в состав .NET используется два вида типов: родные типы языка и типы CTS. *Например в Basic есть тип integer, в C# есть тип int, они являются родными, но в рамках .NET можно использовать int32, он будет преобразован в родные типы, этим занимается библиотека, она встроена в класс system.*

Что касается динамической части. Общеязыковая среда выполнения (CLR) является динамическим компонентом платформы. Предполагает следующие особеннсти создания приложений:

1. Наличие CLR определяет двухэтапную компиляцию (сначала в промежуточный).

Сначала код преобразуется в код CLR (состоит из кода MSIL и метаданных). В чём отличие. Если пишем серьёзное приложение, то надо зарегистрировать приложение в реестре и везде. А если .NET сделана для разработки и локальных и удалённых приложений.
При двухэтапной компиляции выделяется два основных понятия:
Управляемый модуль - переносимый исполняемый файл, представленный в виде PE-файла, который состоит:

1. код на промежуточном языке и
2. метаданные, содержащие всю необходимую информацию.

В зависимости от типа проекта, метафайл может быть оформлен как DLLка или как EXE файл.

На втором этапе компиляции говорят о создании управляемого кода. Это код программы на промежуточном языке, выполняемый под управлением CLR. Исполнительная среда работает с промежуточным кодом на основе предложенной спецификации метаданных и создаёт машинный код, адаптируемый к предложенной аппаратной платформе. Такая компиляция называется JIT-компиляцией, или компиляцией на-лету.

Следующая особенность CLR - виртуальная машина. Кхм.. Ради платформонезависимости. В заимствована из Java.

Дизассемблер и ассемблер. В случае наличия PE файла может возникнуть ситуация, когда его необходимо проанализировать. При этом рассматривается как IL-код, так и метаданные. С этой целью в состав среды SDK Framework включены средства ассемблера и дизассемблера. Данные средства позволяют рассматривать внутреннюю структуру файла PE. Представляет программу в виде дерева, в этом дереве описываются все метаданные.

Последний элемент - сборщик мусора - Garbage Collector.
Исполнительная среда берёт на себя часть функций по эффективной работе с памятью. Под сборкой мусора понимают освобождение памяти, занятой объектами, которые не используются в дальнейшей работе приложения. Наличие сборщика мусора обеспечивает нормальное освобождение памяти, если этим не занимается программа. Чтобы не происходила утечка памяти.

Мы под .NET можем использовать управляемый CLR код, а можем неуправляемый, но тогда всю безопасность и прочее обеспечиваем сами.

Основным инструментальным средством является VS.NET:

1. Многоязыковая среда
2. Позволяет создавать компоненты на любом подходящем языке программирования
3. Доступность всех средств .NET для каждого языка программирования
4. Сервисные возможности для разработчиков (анализ кода, отладка …) *Надо будет точки останова, отладка, ога, дамп памяти.*
5. Возможности облегчённой самостоятельной разработки транслятора для любого языка программирования (да для всех).

Какие технологии доступны разработчику под .NET?

- ASP.NET - единая технология для разработки приложений с минимумом кода.
- NetCF- позволяет создавать приложения, работающие на мобильных и встраиваемых устройствах.
- Silverlight - независимая от браузера кроссплатформенная технология, позволяющая проектировать и разрабатывать интерфейсы с поддержкой мультимедиа, а так же многофункциональных приложениях в интернете.
- VTFO - Visual Studio Tools for Office - компоненты для работы с оффисом. Яыляется наследником COM и OLE технологий.
- WinForms - набор технологий для разработки Win-приложений. Формы управления данными, графическими данными..
- WPF - система преднозначенная для построения клиентских приложений с внутренними привелегированными действиаиями пользователя. Как локальные, так и в браузере. Включает систему визуализаций и систему расчётов, может использовать XML в качестве языка разметки.
- XNA - позволяет создавать игры для XBOX в рамках VS.

Лек1. Основы языка C++

8 февраля 2012 г.

9:29

Родился C++ в 1972 году. Автор - Страуструп.
Введена расширенная работа с памятью и ООП.
Си изначально являлся языком системных программистов, постепенно он прешёл к прикладным программистам.

Достоинства

- Гибкость и компактность языка, можно короче описать (лучше пока этим не пользоваться, например для соблюдения читабельности)
- Семантика языка не связана с определённой структурой (только в CLR)
- Доступность под многие системы
- Переносимость (если создали Win приложение, то его довольно легко перенести на Unix платформу с помощью встроенных трансляторов).

Структура

Любая программа на си - это набор функций. Сплошные функции. Сама программа - функция (Main())
Каркас. Указываются (с решётки) директивы компилятора, предназначенные чтобы подготовить файл к компиляции. Include подключает заголовочные и другие файлы.
По-умолчанию подключается stdafx.h.
Далее определяются (объявляются) функции самой программы.
Main, аргументы. Аргумент C (количестко параметров в командной строке) и ссылки (аргумент V, ссылки.. Массив..)
Void - произвольный невозвращаемый тип.

Лабы будут, прийти можно обеими ПГ.

Лаб1. Перегрузки

15 февраля 2012 г.
8:47

Организовать программу содержащую перегруженные шаблоны и функции для сортировки статического массива типа longint и float методом shaker-сортировки.
И то, и другое надо сделать и через шаблон, и через перегрузку.
В выводах оценить, когда удобнее использовать шаблоны, а когда перегруженные функции, всё это будет спрашиваться.
IOstream делают в общем заголовочном файле.

Шаблоны и перегрузки в одну программы
Расчёт перегрузкой, Вывод например шаблоном
Вывод красивый консольный

Лек2.

15 февраля 2012 г.
11:34

Argc - количество параметров, char *argv - список параметров.
Дальше main, void некрасиво делать, надо чтобы что-то возвращала, ошибки.
<<Лек2..wma>>

Запись начата: 11:35 15 февраля 2012 г.

Необходимо определять область видимости и время жизни присутствующих в программе переменных.
Время жизни определяется по следующим правилам:

1. Переменные, объявленные на внешнем уровне имеют глобальное время жизни (то есть существуют с точки входа программы до точки её завершения)
2. Переменные, объявленные на любом внутреннем уровне (внутри функций, внутри циклов, внутри блока) имеют локальное время жизни. Можно обеспечить глобальное время жизни для внутриблочной переменной, для этого при объявлении переменной ей необходимо задать класс памяти static. Замечание: переменная с классом памяти static будет видима в блоке памяти, в котором она определена, и во всех вложенных блоках. Однако такая переменная будет сохранять своё значение на протяжении всего выполнения программы.

Область видимости переменных определяется по следующим правилам:

1. Переменные, объявленные или определённые на внешнем уровне видимы с точки объявления и до конца исходного файла. Переменные, объявленные на внешнем уровне с классификатором или классом памяти static видны только в рамках одного исходного файла.
2. Переменные объявленные или определённые на внутреннем уровне видимы от точки объявления и до конца блока, при выходе из блока они теряют свои значения и видимость.
3. Переменные из объемлющих блоков, включая переменные, объявленные на внешнем уровне, видимы во всех внутренних блоках. Такая видимость называется вложенной. Если переменные внутри блока имеют то же имя, что и переменные внешнего блока, то определение внутренней переменной заменяется более глобальным

Управлять характеристиками переменных можно либо изменяя места их объявления, либо используя модификаторы памяти. В языке C++ используются следующие модификаторы:

1. **Auto** - присваивается переменной автоматически при объявлении, при этом память под переменную выделяется в стеке и при необходимости инициализируется каждый раз при выполнении оператора, где содержится определение этой переменной. Память, занятая автоматической переменной освобождается при выходе их блока, в котором она объявлена.
2. **Extern** - используется для объявления внешних переменных. Компилятор, встретив модификатор extern устанавливает флаг о том, что переменная будет определена в другом месте программы, блоке или файле. Определение предполагает выделение памяти под переменную. Внешние переменные нельзя инициировать при объявлении (то есть инициализация должна происходить отдельно)
3. **Static** - переменная определяется как статическая, имеет постоянное время жизни, инициализируется однократно при выполнении оператора, содержащего определение переменной. Сама статическая переменная в зависимости от размещения может быть глобальной или локальной.
4. **Register** - аналогичен auto, с той разницей, что память выделяется не в стеке, а в регистрах общего назначения процессора. Что позволяет повысить скорость обращения к переменным подобного типа.

По мере усложнения структуры программ возрастает вероятность конфликтов имён. В

языке Си имеется возможность создания именованных пространств имён - namespaces. Пространство имён - это логическое объединение классов с близкой функциональностью. Предназначена для разрешения конфликтов имён в различных участках приложений. Имена переменных, методов, объектов, определённые внутри пространства имён, не конфликтуют с именами, объявленными в других пространствах. При этом существуют механизмы, позволяющие в любой части программы использовать объявленные в пространстве имена.
Namespace Jack
{void fetch(); int pal;}
Jack::fetch
Пространства имён могут находиться как на глобальном, так и на локальном уровне, однако не рекомендуется помещать пространства имён внутрь блоков, например циклов. Кроме пользовательских пространств имён существуют глобальные пространства имён, определённые системой. Обращение к ним происходит подобным же образом.

Чтобы не обращаться *Jack::fetch* введены механизм объявления и директив. Механизм объявления даёт доступ к идентификатору внутри пространства имён. Директива позволяет получить доступ ко всему пространству имён целиком.
Объявляем словом using
Using Jill:fetch
::fetch - из системного.

Есть using namespace System;
Но тогда нужно квалифицированное обращение
Std::cout
А можно простое using namespace std;
Cout

Необходимо использовать пространство имён с осторожностью, так как возможно возобновление конфликта имён.

1021234dec
0815 0151oct
0x15Fhex

[<Спецификатор класса памяти>] [const]<спецификатор типа><идентификатор>[=<начальное значение>];
Auto double first=123.34;
Const float pi=3.14;
Int *x, z;
Float * near y = NULL;
x=&z //операция получения адреса, в x запишется адрес Z

Типы данных
…
Record - это структура
Ещё есть union, класс (объект.).
Перечисление, ссылка..

Ссылка - тип данных, позволяющий определить для переменной некое новое имя.
&another = a
Another++; / переменная a =1

Перечисление - обращение к одному из элементов множества по имени или номеру.

Массивы

Особенностью работы с массивами в Си является то, что к массиву можно обратиться либо по номеру элемента через его индекс, либо по указателю. При объявлении массива должен указываться его размер. Одномерный массив, содержащий два элемента: int mass[2] с номерами 0 и 1.
Двумерный массив: float [6][7] (6 строк, 7 столбцов) располагается в памяти последовательно. Адрес первого элемента является указателем на весь массив целиком.
В случае объявления многомерного массива можно не указывать первую размерность, если будет выполнена инициализация данного массива, char x[][3] = {{9,8,7},{6,5,4},{3,2,1}}
Особенность символьных массивов заключается в том, что последним элементом символьного массива является символ 0. (то есть в массиве есть ещё +1 символ, ноль, символизирующий конец строки). А сразу инициализировать нулём не рекомендуется, не всегда корректно записывается.

Обращение к массиву может быть через указатель.
Char ArrChar …
...
Char * pArr = ArrChar;
Char letter = *parr;
Parr +=3;// указывает на ArrChar[3]
Размер тройки определяется типом указателя, сколько байт занимает там переменная. Вроде само делается.
Передача массивов в функции осуществляется только через указатели, по значению массивы не передаются.

Передача просто имени массива является указателем на первый элемент массива. Обращаться можно, меняя указатель, или по его индексу.

<<Лек3..wma>>

Запись начата: 8:28 22 февраля 2012 г.

Лек3. Продолжение

22 февраля 2012 г.
8:28

Структуры

Что-то типа record в паскале.
Struct
Особое внимание уделяется возможности работы с бинарными структурами.
В случае создания бинарной структуры возможно побитовое распределение информации внутри типа. При этом объём каждого из полей будет определён внутри структуры.
Struct binar{
Unsigned first :2;
}
В случае задания значений каждому из битовых полей определятся младшие разряды поля. Остальные разряды, не вошедшие в размер поля, отбрасываются.

Unioin - смесь (нет в паскале)
Позволяет обращаться к разнотипным данным внутри общего типа, но только к одному из приведённых полей. Память, отводимая под смесь определяется максимальным по размеру полем. При обращении к любому из полей, содержимое других полей теряется.
Union int_or_long { ...}

Операции

Все операции СИ можно разделить на 3 класса:

- Унитарные - выполняют действие над одним операндом
  - Префиксные c=++a; Происходит сначала модификация операнда, а затем использование его значения
  - Постфиксные c=a++; Изменение происходит после выполнения операции присваивания.
- Бинарные. Рассчитаны на обработку двух операндов. *Data+=10; к значению data будет прибавлено 10 и сохранено в ту же переменную*.
- Тернарные. Единственная операция - операция условия. Условие ? Операнд1 : операнд2. *max = (a>b)?5:3*

Правила преобразования

1. Операнды типа char, unsigned char или short преобразуются к int по прпавилам:
   a. Char расширяется нулём или знаком в зависимости от умолчания для char
   b. Unsigned char расширяется нулём, signed char расширяется знаком.
   c. Short, unsigned short и enum при преобразовании не изменяются.
2. Если один из операндов имеет тип long double, то другой преобразуется к типу long double.
3. Если один из операндов является беззнаковым, то и остальные приводятся к беззнаковому значению.
4. Тип результата всегда приводится к типам операндов, участвующих в операции.

Возможно использование явного преобразования:
Int integer=54;
Float
Integer = floating (integer станет флоатом o_O)

Функции

Функция - это фрагмент кода, который можно неоднократно вызвать из любого места програмы, они позволяют умененьшить избыточность програмного кода и повысить его эффективность.

Состоит из описания функции и вызова функции…?

При вызове осуществляется передача локальных параметров, передача через стек, это позволяет их делать локальными внутри функции. Чтобы была возможность глобального доступа, необходимо передавать их через указатели или ссылки.
В некоторых случаях необходимо, чтобы параметры были доступны глобально, но нельзя было изменять. В этом случае параметры передаются с параметром const.

Использование параметров по-умолчанию, то есть часть параметров можно определить до вызова функции при её описании.
Замечание: возможность использовать параметры по-умолчанию определена порядком передачи параметров в стек при вызове. В отличии от паскаля, параметры записываются в стек в порядке, обратном описанию. Следовательно, в качестве параметров по-умолчанию могут быть указаны только конечные параметры вызова. В противном случае компилятор воспримет вызов без полных параметров как ошибочный. (*имхо, только в конце можно ставить поумолчанию определённые параметры)*

Переменное число параметров

На этапе написания функции не всегда можно определить нужное число параметров. При этом можно использовать переменное число параметров.
Описываются известные параметры, а затем ставится многоточие.

Для того, чтобы при вызове и обработке конкретизировать число параметров, подлежащих передаче, среди известных параметров должны быть определены аргументы, определяющие правило вызова.
Через указатели делается, для этого передаём адрес первого (ну то есть последнего в стеке).

Лек3. Перегрузка функций

22 февраля 2012 г.
8:50

Перегрузка функций - это механизм, позволяющий использовать одноимённые функции для работы с различными фактическими аргументами.
Для всех перегруженных функций существует следующее правило использования: имена функций должны обязательно совпадать, а типы и/или количество передаваемых аргументов должны различаться. Во внимание не берутся возвращаемые функцией значения. Компилятор в точке вызова определяет (конкретизирует) вызываемую функцию в зависимости от списка фактических параметров.
С целью ускорения процесса выбора функций, на этапе компиляции выполняется декорирование функции (декорирование). Декор функции определяет её имя и типы формальных параметров. Подстановка функции вместо вызова осуществляется на основе декора на втором этапе компиляции.

Как создаётся декор.
Int sum (int a, int b) {return a+b}
//public @sum#qii
I для int, d для double, zc для char, pi для int* (p для указателя)
Но таким образом (при декоре) не указывается выдаваемый функцией тип данных.
То есть char A и int A функции будут одинаковыми и просто переопределены.

Возможны неоднозначности:

1. В случае выполнения арифметического преобразования типов. Например есть две функции, одна определена как float, другая как double, double длиннее, всегда будет вызвана она.
2. В случае использовании аргументов по умолчанию. Перегруженные функции могут отличаться не только по типу, а и по количеству параметров.
3. В случае использовании параметров-ссылок, так как в качестве передаваемого значения используется не копия значения, а непосредственно адрес аргумента.

Правила описания перегруженных функций

1. Перегруженные функции должны находиться в одной области видимости, иначе вызов будет осуществляться в рамках локальной области видимости.
2. Перегруженные функции могут иметь параметры по умолчанию, при этом значения одного и того же параметра в разных функциях должны совпадать.
3. Функции не могут быть перегружены, если список их параметров отличается только модификатором const или использованием ссылки.

Лек3. Препроцессоры. Директивы препроцессора.

22 февраля 2012 г.
9:21

Отличительной чертой языка C++ является выполнение препроцессирования перед

непосредственно компиляцией.
Суть в просмотре исходного текста, выявлении управляющих последовательностей и подготовки текста компиляции. Управляющие последовательности называют директивами препроцессора. Особенностью директив является то, что каждая директива действует от места её описания до конца исходного файла. Любая директива может быть вставлена в любое место исходного файла. Согласно стандарту, директива препроцессора называется с ключевого символа #, после которого указывается имя директивы, указывающее правило её работы.

Include - включает в исходный файл содержимое другого файла. Как правило используется для подключения библиотечных и загловочных файлов lib и h.
Поиск файлов определяется синтаксисом определения директивы.

1. Если имя файла указано в кавычках, то поиск файла будет осуществляться в рабочем каталоге.
2. Если в треугольных скобках, то файл ищется в каталоге, определённом средой программирования.
3. Предполагается, что существует макроопределение, которое раскрывается в допустимое имя заголовочного файла.

Препроцессор удаляет директиву include из файла с исходным кодом и напрявляет содержиое указанное в include на обработку компилятору.

#define позволяет определять макроопределение, которое затем используется в теле программы.
Формат:
#define имя-макро <тело-макро>
Когда встречается макро, оно заменяется на параметры?.. Указанные в теле макро.
Возможно использование макроопределяния с параметрами.
Для того, чтобы переопределить параметры?.. Используют директиву define
#define pi 3.14
Int x=10*pi

Особенность использования макроопределений - тело макроса заменяет его место в исходном файле. При этом никакой проверки на совпадение типов и порядок на выполнение операций не осуществляется.

Ошибки:
#Define p=3.14
Замена грубая.. Оно просто заменяет и всё, надо иметь ввиду

Директива условной компиляции. Групповая компиляция

Используется для управления..
#ifdef - определяет проверяемое условие

```
#ifdef идентификатор1
        Текст программы
#elif идентификатор2
        Текст программы
#endif
```

и **##** позволяет осуществлять операции над формальными параметрами макро.
это операция образования строки, ## операция образования лексемы
Если операция образования строки предшествует формальному оператору макро, то соответствующий ему формальный параметр будет заключён в кавычки и считаться

строковым литералом?.

Для образования лексем выполняется объединение нескольких лексем в одну (грубо говоря, происходит конкатенация строк)

#error позволяет направить ошибки в стандартный поток ошибки
После вывода сообщения компиляция прекращается.
#if !define(MYNAME)
#error Не определена константа MYNAME
#endif

#pragma
Язык с++ имеет набор прагм, подрреживаемых компиляцией. Прагмы исполняются не на этапе прекомпила, а на этапе компиляции.
Наиболее часто используемые:
#pragma startup имя_функции <*приоритет*> - определяет функции, которые будут выполняться при запуске программы до передачи управления функции main
#pragma exit имя функции <*приоритет*> - определяет функции, которые будут выполняться во время завершения программы перед вызовом функции операционной системы exit.
Приоритет указывает порядок вызова функций. Диапазон приоритетов распределяется в пределах от 64 до 255 (это приоритеты процедуры пользователя). От 0 до 64 используются библиотечными функциями СИ.
Вызываемая в директиве startup и exit функция не должна возвращать значений и не может принимать параметров.

Следующая лекция - динамическая память.

Файлы:
<<Шабунин, Прокофьев. Задачник 10-11.djvu>>
<<Указатели С++2012.doc>>
<<Препроцессор.ppt>>
<<Модели памяти2012.ppt>>
<<лекция22012.ppt>>
<<Лекция 3.docx>>
<<Лабораторная работа 1_2011.doc>>

Лек3. Шаблоны

22 февраля 2012 г.
9:03

Шаблоны - это механизм, позволяющий определить обобщённые функции, из которых компилятор автоматически создаёт представителя функции для заданного пользователя, типа или типов данных.
Когда компилятор создаёт по шаблону конкретного представителя функций, то говорят, что происходит порождение функций. Для определения шаблона используется ключевое слово **template**, за которым следует в скобках набор параметров, определяющих возможные типы пользовательских аргументов, а далее заголовок и тело шаблонной функции.
Template <class t>
Void printarray (t *array, const int count)
…

Компилятор при определении вызова порождает конкретную функцию на основании шаблона, таким образом в исходном коде программы формируется столько реализаций шаблона, сколько разнотипных вызовов будет определено в приложении.
В рамках шаблонной функции может быть определено один или более шаблонных классов.
Ошибочным является определение одноимённых шаблонных классов внутри функции (достаточно объявить один раз класс и его использовать).
В ряде случаев шаблонная функция может быть определена как внешняя - описанная в рамках другого подключаемого модуля. Тогда в вызывающем модуле необходимо выполнить объявление функции с указанием списка параметров. Extern означает, что определена она будет в другом месте, а описать надо здесь.
Шаблонная функция может быть определена как статическая (static), то есть функция, видимая внутри внешних блоков и содержащая статические неизменяемые вызовы.
Статическая функция может быть объявлена как встроенная функция (inline), то есть тело конкретизированной шаблонной функции будет подставлено вместо её вызова.

Допускается явная спецификация типов шаблонных параметров, то есть можно потребовать от компилятора создать желаемую конкретную шаблонную функцию. {Funcname<int>(ch);} Когда типовой шаблон задан явно, выполняется преобразование типов функций к соответствующему типу шаблонных параметров, то есть в нашем примере аргумент ch будет приведён к типу int. Когда компилятор обнаруживает шаблон, он сохраняет внутреннее представление определения функции до момента её вызова.
Принцип конкретизации шаблона определяется моделью компиляции, принятой в системе. Си поддерживает две модели компиляции:

1. **Модель с включением** - определение шаблона включается в каждый файл, где он конкретизируется.
2. **модель с разделением -** определение шаблонов помещается в заголовочный файл *.h, а определение в файл исходного текста типа *.cpp; В этом случае, при определении шаблона необходимо использовать ключевое слово export для помещения заголовка в файл h. Модель с разделением является менее предпочтительной, так как требует дополнительных временных затрат на подготовку конкретизации.

Если алгоритм не меняется, а функции для разных параметров - удобно использовать шаблон. Но есть случаи, когда неуодобно использовать шаблон. *Об этом спросят*.

Возможностью СИ является получение указателя на функцию. Мы получаем адрес функции в скомпилированном файле. Она может быть вызвана как по ??, так и по месту в файле… обычно такое испольуется для подключаемых функций.

В ряде случаев для ускорения работы функции выгоднее вместо вызова функции вставлять в исходный код набор операций для текущих действий.
Ибо:
Прерывание
Вызов в стек
… вызов функции
Прерывание процесса
…
Долго.

Выгоднее определить функцию как inline. Если компилятор сочтёт её достаточно маленькой, он её встроит сразу в текст (чтобы не было вызовов).

Лек3. Работа с указателями, динамическая память

29 февраля 2012 г.
11:27
<<Лек3. Работа с указателями, динамическая память.wma>>

Запись начата: 11:28 29 февраля 2012 г.

Лаб2
создать динамические структуры, дамп памяти…

Любой объект программы занимает в памяти определённую область.
Местоположением объекта в памяти - адрес.
Для объявления памяти рекомендуется резервировать место в памяти. Размер отведённого адресного пространства зависит от типа переменной. Для доступа к выделенной ячейке используется имя переменной. Для того, чтобы определить адрес переменной в памяти достаточно взять её ссылку, что выполняется с помощью унитарной операции взятие адреса (операция &).
В C++ в неуправляемом коде имеется возможность осуществления непосредственного доступа к памяти, для этого используется специализированный тип - указатели. В стандартизованном СИ предполагается, что все указатели одинаковы, то есть определяют одинаковое внутреннее представление адресов. С помощью указателя предполагается возможность получения доступа к любой области RAM. Выделяют следующие разделы RAM:

1. **Статическая память** - память, выделяемая при запуске программы для размещения глобальных статических объектов и объектов, определённых в пространстве имён. Такую память ещё называют стеком процесса.
2. **Автоматическая память** резервируется при запуске программы, используется для размещения только локальных объектов. Такую память называют стеком программы.
3. **Динамическая память (куча, heap)** - выделяется из доступной свободной RAM непосредственно во время выполнения программы для размещения динамических объектов.

Любая программа, запускаемая под операционной системой Windows после запуска создаёт процесс. У каждого процесса имеется собственная память, описатели файлов и ряд необходимых системных ресурсов.
Процессу выделяется 4-гигабайтное пространство виртуальной памяти. Программа может обращаться к любому байту пространства, используя 32-битный адрес. При этом в адресном пространстве каждого процесса содержится:

1. Образ исполняемого файла программы
2. Все несистемные DLL, загруженные программой
3. Глобальные данные программы, доступные как для чтения, так и для RW.
4. Стек программы
5. Выделенная куча или куча runtime CRT.
6. Блок shared-памяти
7. Локальная память отдельных потоков
8. Системные блоки, в том числе таблицы виртуальной памяти
9. Память, выделяемая под ядро, исполнительную систему и компоненты DLL windows.

Пространства 4 гига используются так

1. 4 мегабайта - под 16-битные
2. Остаток от 2 гигабайт - память процесса подо всё

3. Третий гигабайт - shared память
4. Системная область - driver locked

То, что выделено процессу для использования (2 гигабайта) распределяется между данными и кодом.
Любая ячейка виртуальной памяти может находиться в одном из трёх состояний:

1. Она может быть помечена как свободная ячейка - ссылки на данный блок отсутствуют и он может быть использован для выделения процессу
2. Может быть зарезервирован. Доступный для использования процессу, но может использоваться и для другого запроса на выделения. Сохранение данных в этом блоке невозможно, пока он не будет выделен.
3. Выделенный - блок памяти назначен физическому хранилищу, к данному блоку имеются обращения из процессов.

В результате многократных выделений памяти виртуальное пространство может фрагментироваться и даже при наличии суммарного количества свободной памяти, процессу будет отказано в выделении памяти под объект.
Виртуальное адресное пространство обычно имеет страничную организацию.
Страница может содержать либо данные, либо код процесса.
Процессоры Intel эффективно преобразуют 32-разрядный физический адрес в номера страниц и смещение внутри страниц. Для этого используется страничный механизм преобразования. У каждого процесса имеется свой набор таблиц страниц.
Регистр CR3 (PDBR - регистр базы страниц) содержит указатель на страницу каталога, в котором размещена таблица страниц данного процесса. При переключении между процессами операционная система перезагружает значение регистра CR3. Выбирается таблица страниц, по 10-битному адресу (из общего 32-битного адреса vmem) выбирается элемент каталога, следующие 10 бит адреса выбирают элемент таблицы - страницу, следующие 12 бит адреса определяют смещение в странице. 4 килобайта страница.
Каждая запись таблицы страниц содержит бит присутствия в физической памяти, который сообщает, находится ли сейчас в физической памяти данная страница. При попытке обращения к странице, отсутствующей памяти, инициируется прерывание windows и генерируется либо ошибка, либо, при возможности, осуществляется обращение к странице.
Элементы таблиц страницы.. Есть биты (5) защиты… указывает, имеется ли доступ к странице, для чтения или чтения-записи.
При использовании виртуальной памяти возможно использование процесса подкачки. Для разрешения файла подкачки используются 4 бита, определяющих индекс файла подкачки.
Диспетчер виртуальной памяти пытается достичь максимальной производительности и определяет моменты чтения и записи страниц. Если какой-то процесс не использует страницу в течение определённого периода, а другому процессу требуется выделение памяти, то страница предыдущего процесса выгружается, а на её место выгружается страница нового процесса. Таким образом получается, что все процессы совместно используют единый общий системный pagefile или swapfile. Кроме того возможно при необходимо перемещения кучи выделенной памяти необходимому? Процессу. Такое возможно при использовании управляемых данных и кодов CRT. ?
Кроме страничных файлов, диспетчер виртуальной памяти работает с проецируемыми файлами. Если два процесса используют один и тот же EXE файл, последний файл отображается на адресное пространство обоих процессов. Адреса файлов никогда изменяются во время выполнения программ, поэтому они могут проецироваться на одну и ту же физическую область памяти. При этом два процесса не могут одновременно обращаться к одним и тем же глобальным данным. В windows данная задача решается следующими способами
В 16-разрядных windows каждый процесс имеет отдельную глобальную область данных, в которой размещается копия данных процесса. Начиная с NT и дальше несколько

процессов используют одну копию глобальных данных до тех пор, пока один из процессов не попытается выполнить модификацию глобальных данных. В момент попытки модификации создаётся копия страницы данных и далее каждый процесс работает с собственной копией, которая теоретически находится по одному виртуальному адресу. (но в разных CR3) (но те, кто не меняли, продолжают работать с собственной копией)

Как же в Си реализован доступ к динамическим участкам памяти?
Язык Си предоставляет набор функций для работы с динамической памятью.
Часть функций унаследованы из Си, а часть функций - C++.
Функции семейства alloc, а так же функции new, free, delete.
Функции семейства alloc используются для выделения участка динамической памяти, delete для освобождения.

Функция malloc

Void *malloc(size_t size);
В качестве единственного параметра malloc передаёт размер требуемого участка памяти.

```
#include <malloc.h>
void *malloc( size_t size);
char * str;
int * count;
str = (char *) malloc (5); // отводится 5*sizeof(char) байт
count = (int *) malloc (10*sizeof(int)); // отводится 10*sizeof(int) байт
free(str);        // освобождение памяти. Вопрос: каким образом функция free определяет участок памяти? Есть флаги!
free(count);
```

void *calloc(size_t nitem, size_t size); Выделяет участок заданного размера, возвращает нетипизированный указатель на начало участка. Первый параметр - определяет количество выделяемых элементов требуемого размера. Второй параметр - размер каждого выделяемого элемента.

```
char * str;
int * count;
str = (char *) calloc (10, sizeof(char));
count = (int *) calloc (5, sizeof(int));
```

void *realloc(void *block, size_t size); Является функцией перераспределния памяти. Возвращает нетипизированный указатель на новый или тот же участок памяти, используемый для перераспределения. Имеет два параметра: указатель на выделенный блок, и размер выделенного блока.
С помощью функции realloc можно расширить ранее выделенное адресное пространство. Как работает функция realloc?

Рассматривается пространство, принадлежащее процессу, ищутся участки памяти требуемого размера, если такой участок нельзя выделить по прежнему адресу. Если участок памяти заданного размера обнаружен, информация, находящаяся по адресу, указанному в качестве параметра, копируется по новому адресу, адрес начала выделенного блока возвращается в качестве результата работы функции, прежнее выделение не изменяется. Если участок заданного размера не найден, функция возвращает значение nul. С помощью этой функции можно освободить участок памяти, подав значение 0. Указатель есть, но флаг выделенности снимается.

count = (int *) realloc (count, 10*sizeof(int));

Операторы new и delete

Являются функциями для работы с динамической памятью C++. Позволяют выделить память под объект заданного размера, работают **с типизированными указателями**.

управление динамическим размещением в памяти единичного объекта:
int * p= new int;
delete p; тип известен, удаляем ровно сколько надо

С помощью указателей new и delete можно размещать динамические массивы элементов. Для этого в операторе new необходимо определить количество выделяемых объектов заданного типа. Оператор delete должен по синтаксису совпадать с соответствующим оператором new.

динамическое размещение массива объектов:
int * p= new int[100];
delete[] p;

Особенности при работе с динамическим: двумерным массивом необходимо отдельно выделять память под указатели на строки и память под элементы строк.

```
#include <malloc.h>
#include <iostream>
void out(int ** mas);  // вывод массива на экран
void clear_mem(int ** mas);  // очистка
int n=2; // число строк
int m=3; // число столбцов
int  main()
{
  int ** d;
  int i,j;
  d = new int* [n];   // выделяем память под указатели на строки
  for ( i=0; i<n; i++)  // выделяем память под указатели на столбцы
         d[i]=new int [m];
// инициализация массива
  for ( i=0; i<n; i++)
        for ( j=0; j<m; j++)
        {d[i][j]=i+j;} обращаемся к индексам нашего массива уже получается
        out(d);
        clear_mem(d);
return 0;
```

Позже были выделены элементы массива (строчки).
Сначала их освобождаем, потом, когда строчки освобождены, освобождаются ссылки на эти строки.
Здесь побалакали раньше

Алгоритмы для управления областями памяти

1. Алгоритмы для работы с объектами одного типа. Для выделения памяти под подобные объекты нецелесообразно использовать функции класса alloc так как размер выделяемых объектов постоянен и чаще всего типизирован. Для работы с такими объектами принято писать собственный менеджер объектов или stab-аллокатор.
2. Для обработки объектов неопределённого, но одинакового размера. Для таких объектов используется алгоритм битовых масок или алгоритм "близнецов".
3. Для объектов произвольного типа и размера. Обычно используется метод граничных маркеров.

Метод менеджера объектов

Данный метод подходит для выделения и высвобождения большого количества однотипных объектов, причём ограничение на количество не устанавливается. Под объекты выделяется массив памяти из максимально допустимого объёма выделенной области. Все объекты, помещаемые в данную область связываются некоторой структурой данных, например списком свободных объектов. Тогда выделение памяти под объект это возвращение ссылки на голову списка. Освобождение памяти - помещение ссылки на него в конец списка свободных объектов.

Алгоритм битовых масок

В соответствии с данным алгоритмом для представления памяти используется битовая карта, в котором для каждого блока содержится отдельный бит, если блок памяти свободен 0, и 1 если блок занят. При каждом следующем выделении просматривается список объектов по битовой маске и занимается первый свободный блок.

Метод граничных маркеров

В основе данного метода лежит использование двусвязного списка всех свободных и занятых блоков памяти. В соответствии с данным методом структура каждого элемента будет следующая:

- Флаг занятости элемента
- Размер блока
- Ссылка на следующий свободный элемент
- Ссылка на предыдущий занятый элемент
- Далее следует выделенный участок памяти
- Конечный дескриптор (окончание занятого участка памяти).

Как происходит выделение? Находится последний занятый участок, по нему определяется номер следующего свободного участка памяти. Адрес участка, из которого взят адрес для выделения, записывается в ячейку предыдущего занятого участка. (двунаправленный список)
Как высвобождение? Выявить переход по адресу к переду щей занятой, освободить требуемый участок, записать адрес его предшествующего разрешения в поле следующий свободный предыдущего участка памяти.
Если одновременно освобождается несколько последовательных участков, они могут объединяться и составят единый свободный участок.

Stab-аллокатор

Работает подобно менеджеру объектов, но обычно используется универсальный для

разных типов объектов. Основная суть заключается в частом выделении однотипных объектов. В системе вводится единый интерфейс для выделения и освобождения памяти под объекты. Для каждого объекта создаётся собственный pool, а в системе создаётся кэш подобных пулов, в этом кэше записываются адреса, зарезервированные под объекты.

Алгоритм "близнецов"

Используется в случае, если выделяемые участки памяти кратны по размеру степеням двойки. Составляется список доступных блоков, разбитых на группы (4КБ, 16КБ и т.п.). При необходимости выделения памяти происходит поиск среди ближайших участков по размеру. Если необходимых участков в исходном списке нет, осуществляется переход к списку с удвоенным размером. Блок, свободный в списке удвоенного размера извлекается из списка, разбивается на два и помещается в младший список. При освобождении, если возможно объединение последовательных участков, они объединяются и помещаются в старший список.

Утечка памяти

Сбой при ненадлежащем освобождении памяти. Отладочные средства VisualC режима CRT позволяют определить потенциальные места утечки памяти. Основным свойством обнаружения утечки является модуль CRT, который подключается через заголовочный файл CRTDebug.
Затем необходимо включить директиву _CRTDBG_MAP_ALLOC. Эта директива позволит в отладочном режиме просматривать состояние участков используемой памяти.
Функция _CrtDumpMemoryLeaks() устанавливается перед точкой останова в режиме отладки приложения и отображает в окне вывода отчёт об утечке памяти. Данный отчёт представляет собой набор снимков о моментах выделения и освобождения памяти. Кроме того, указывается объём занятой и свободной памяти на окончание приложения.
Чтобы сохранить снимок состояния памяти, можно создать структуру _CrtMemState. Переда эту структуру функции Chekpoint мы помещаем снимок в окно вывода.
Для вывода дампа используется _CrtMemDumpStatistics (&s1);
Вставляем в код, следим, довольные:)
Использование управляемого кода и управляемых данных позволяет не заботиться о правильности освобождения памяти так как в случае использования CRT процесс сборки мусора происходит автоматически. Сборка мусора осуществляется в 3 этапа:

1. **Маркировка**. На данном этапе осуществляется поиск всех неиспользуемых объектов и составляется их список. Неиспользуемым считается объект, на который не имеется ссылки.
2. Этап перемещения, обновляющий ссылки на сжимаемые объхекты.
3. Освобождается пространство, занятое неиспользуемыми объекутами и сжимающий выжившие объекты, не уничтьоженные во время сборки мусора)

Выделяют три поколения:

1. Поколение 0 кратко живущмих объекто
2. Поколение 1, переживших хотя бы один цикл сборки мусора.
3. Поколение 2. Долгоживущие обхекты.

Просматриваются по-очереди. Эти действия происходят независимо от программиста в фоновом режиме. Главное условие - управляемый код и специальные управляемые указатели, начинающиеся с префикса JC.

Сегодня в 17:30 гена Чистяков проводит.

Лек4. Обработка исключительных ситуаций

7 марта 2012 г.
8:27
<<Лек4. Обработка исключительных ситуаций.wma>>

Запись начата: 8:28 7 марта 2012 г.

На следующей лекции контрольная по всему.
Пользователь может ввести что попало.
Способы:

1. Ручной обработчик. Нужно продумать, где и как пользователь может ввести фигню.
2. Встроенные средства обработки исключений.

Исключение - это некоторое событие, возникающее в ходе работы ПО, которое является неожиданным или прерывает нормальный процесс выполнения программы (неправильное выполнение математических операций (деление на 0), выход за пределы ресурсов (конец файла и т.д.))

Исключения делят на два класса:

1. **Синхронные** исключения - возникающие в определённых, заранее известных точках программы. Возбуждаются программными средствами или же отлавливаются.
2. **Асинхронные** исключения могут возникнуть в любой момент времени и не зависят от того, какую конкретную инструкцию выполняет система. Типичным асинхронным исключением можно считать отказ питания. Чаще всего асинхронные исключения возникают за пределами выполняемой программы и не отлавливаются средствами высокоуровневого программирования.

Существуют исключения, которые можно отнести как к синхронным, так и к асинхронным. В частности, на ряде аппаратных платформ ситуация деления на ноль может вызвать конвейерное исключение, которое будет обрабатываться как программными средствами, так и средствами системы ядра.
В отсутствии собственных обработчиков исключительных ситуаций при их возникновении программа будет немедленно остановлена и пользователю будет выдано системное сообщение о некорректном завершении программы. Таким образом, единственным обработчиком непредвиденных исключений является операционная система.
Теоретически, возможно игнорировать возникшие исключения и продолжить выполнение программы. На практике, такая ситуация может привести к "лихорадке" либо более серьёзным сбоям.
Обработка исключительных ситуаций программой заключается в том, что при возникновении исключений, управление передаётся блоку кода, называемому обработчиком, который выполняет действия, необходимые для восстановления процесса работы программы.

Существует два механизма функционирования обработчиков исключений

1. **Обработка с возвратом** - обработчик исключений ликвидирует возникшую проблему и переводит программу в состояние, когда она может работать дальше по основному алгоритму. Типична для обработчиков асинхронных исключений и малопригодна для синхронных исключений.
2. **Обработка без возврата** - после выполнения кода обработчика управление передаётся в заранее заданное место программы, с которого и продолжается выполнение.

Существует два варианта подключения обработчика исключений:

1. **Структурная обработка исключений**. Реализуется в виде механизма регистрации

функций или команд обработчика для каждого возможного типа исключений. Языки программирования или их системные библиотеки предоставляют программисту те стандартные процедуры: регистрация обработчика и разрегистрация. Выход регистрации привязывает обработчик к определённому исключению, разрегистрация - снимает привязку. Таким образом, если исключение происходит, выполнение основного кода программы прерывается и начинает работать обработчик. Независимо от того, какая часть программы в данный момент выполнялась, на определённое исключение всегда реагирует последний зарегистрированный обработчик. В ряде языков программирования зарегистрированный разработчик сохраняется только в пределах одной структуры программы (функции), тогда разрегистрация обработчика не требуется.

2. **Неструктурная обработка -** единственный способ отлова асинхронных исключений. Структурная обработка требует обязательного наличия в языке программирования специальных синтаксических конструкций. Такие конструкции состоят из двух частей:
   a. Конструкция создания контролируемого блока. Включает в себя набор операций, при выполнении которых могут возникнуть исключительные ситуации
   b. Вызов обработчика. Обеспечение реакций на те или иные исключения.

   Блоки обработки исключений могут многократно входить друг в друга как неявно, так и явно. Поэтому если ни один из обработчиков текущего блока не способен обработать исключительную ситуацию заданного типа, исключение передаются в блок верхнего уровня с целью обнаружения требуемого обработчика. В самом крайнем случае обработка будет выполнена системными функциями. Такая операция получила название **ретрансляция исключений**. Это главный плюс обработчика исключений, отличающий его от ручной обработки.

В стандарте языка C++ предусмотрен встроенный механизм обработки исключений. Его принято называть **CRTL** исключениями. Компиляторы VisualC, в том числе и билдера, включают в свой состав механизмы обработки исключений **SEH** (структурные).
В случае использования структурного исключения SEH исключительные ситуации возбуждаются, а в случае Си исключений они выбрасываются. Выбрасывание исключений всегда производится программным путём, а возбуждение может быть как программным, так и аппаратным. Когда возбуждается некоторая исключительная ситуация, задаётся её код, который можно как исключение. Фильтр-обработчик может обрабатывать код исключения по-разному.
SHE (Structured Exception Handling) представляет два механизма:
Обработка завершения termination handling
Обработка исключения exception handling

```
__try
{
Охраняемый код
}
__except (выражение-фильтр в качестве параметра)
{
Код обработки исключения
}
```

Или вместо except _finaly
Выполнится независимо от вознивновения исключения.
Обычно туда помещают очистку памяти

Выражение фильтр указывает на то, как должна выполняться программа после обработки исключения. Выражение-фильтр вычисляется сразу после совершения исключительной ситуации.

Представляет значения:
EXCEPTION_EXECUTE_HANDLE - обработка передаётся обработку исключения
EXCEPTION_CONTINUE_SEARCH - продолжается поиск обработчика исключений
EXCEPTION_CONTINUE_EXECUTION - система передаёт управление в точку прерывания программы

В выражении фильтра возможно использование одной из инструкций
Dword GetExceptionCode(void);
Dword GetExeptionInformation(void);
Предоставляют обычно код, тип и место возникновения исключения.

В ряде случаев бывает необходимо заранее выйти из контролируемого блока.
SEH предполагает использование инструкции __leave
Данная инструкция передаёт управление в конец блока __try.

#include <except.h>

В блоке finaly при обработке исключений SEH возможна проверка корректности завершения контролируемого блока. За это отвечает функция abnormal termination. Данная функция возвращает значение true если в контролируемом блоке возникло исключение. То есть мы можем определить, мы попали в finaly при нормальном завершении или при исключении (это нужно для освобождения памяти).

Плюсы и минусы SEH:
Плюсы:
1. Позволяет отлавливать большой спект икслючений: арифметические, обращение к памяти, обращение к стеку, к файлам и так далее.
2. Обработка SEH исключений ведётся на уровне ядра операционной системы
3. Возможно использование SEH без дополнительных перекомпиляций с включением библиотеки Си исключений, что позволяет уменьшить объём исходного кода, для подключения SEH иногда достаточно загрузки DLLки kernel32.

Минусы:
1. Данное исключение не принято в стандарте C++, а ядро про классы C++ ничего не знает. Это приводит к:
2. невозможно применять SEH исключения при обработке ошибок пользовательских классов, так как не будет автоматически вызываться деструктор класса при завершении исключения.
3. Невозможно в пределах одной функции использовать SEH и Си-исключения одновременно.

В SEH используются функции виднового ядра, им есть аналоги на unix, но в os/2 например нету.

CRTL-исключения или Си-исключения

Используется механизм окончательной модели управления. После того, как исключение произошло обработчик не может потребовать возврата управления в точку возникновения исключительных ситуаций. Си-исключения обрабатывают только синхронные исключения.
Исключения Си перехватывают ситуации, выброшенные некоторой функцией. В языке C++ практически любое состояние, достигаемое в ходе вычислительного процесса, можно определить как исключение. Анализ и обработка исключений переносится из точки его возникновения в блок обработчика. Так же, как в SEH исключениях, выделяется два блока:

Try
{}
Catch
{}

Внутри блока обработчика возможна регенерация обработчика исключения, которая осуществляется с помощью ключевого слова throw. При регенерации исключение ретранслируется на блок более высокого уровня.

Существует три формата CATCH:
Catch (тип имя) {обработчик}
Используется, когда для обработки исключений требуется информация о состоянии переменных на момент возникновения ситуации исключения.
Catch (тип) {обработчик}
Используется, когда для обработки исключения достаточно лишь знать факт появления исключения заданного типа. (типы определяем мы сами, можем даже класс создать)
Catch (...) {обработчик}
Используется для всех исключений, возникших внутри этого блока, без определения типа и состояния их переменных.

В программе может одновременно присутствовать несколько предложений catch, которые будут выполняться последовательно. Отсюда возникает вопрос, если первым указано предложение Catch (…), то он перехватывает и возвращает управление после всех catch. Поэтому его надо писать самым последним.

После возникновения ситуаций, которая должна восприниматься как исключительная, можно вручную выбросить исключение с помощью оператора Throw.
Оператор Throw существует двух форматов: с параметром и без параметра генерации исключений).
В случае параметризованной генерации формируется статический объект, значение которого определяет выражение генерации. Копия созданного объекта передаётся за пределы контролируемого блока и инициирует переменную, используемую в специализации обработчика исключений. Копия объекта сохраняется до момента окончания обработки исключения.
В случае непараметризованной генерации происходит прямая ретрансляция обрабатываемого исключения (только внутри блока).

Существует вероятность выбрасывания во время выполнения программы **непредвиденных (неспецифицированных, unexpected)** исключений. При этом вызывается системная функция unexpected. Эта функция вызывает текущий обработчик неспецифицированных исключений, которым по-умолчанию является функция terminate (завершение программы). Возможно переопределение обработчика неспецифицированных исключений. Для этого используется встроенная функция set__unexpected.
Тип unexpected_handler определяет указатель на обработчик исключений, на функцию, которая должна быть определена в программе как собственный тип. Тогда параметр после unexpected_handler является указателем на функцию, которая является по-умолчанию для данной программы и устанавливается для обработки.

Следующая: основы языка ассемблер.
Зачем - чтобы было не страшно в следующем году,
Для ускорения.

Вторая лаба по использованию динамической памяти (new, malloc, динамические структуры, списки

<<Лек5. ASM.wma>>

Запись начата: 11:31 14 марта 2012 г.
<<ASSEMBLER.ppt>>

Лек5. ASM

14 марта 2012 г.
11:31

Базовая модель процессора Intel Pentium содержит 8 регистров общего назначения. 32-разрядные.
Возможно обращение как к регистрам целиком, так и к его частям. В частности, чтобы обратиться к 32 разрядному регистру, обращаемся по EAX, к 1 битам AH, к 8 битам AL.

Сегментные регистры.

Предназначены для хранения селекторных сегментов
SS - сегмент стека
DS - сегмент данных
CS - сегмент кода
ES,FS,JS - расширенные сегменты.
В случае использования 32-разрядных моделей (и 64-разрядных тоже) сегментные регистры теряют своё основное назначение так как память перестаёт быть сегментированной и представляет собой сплошной линейный участок. Теоретически, все сегментные регистры по-умолчанию имеют одинаковое нулевое значение, но могут быть использованы в программе в служебных целях.

Регистр счётчика команд

Предназначен для организации команд переходов и вызовов процедур. Используется как 16, так и 32 разрядный счётчик команд (EIP 32, IP 16).

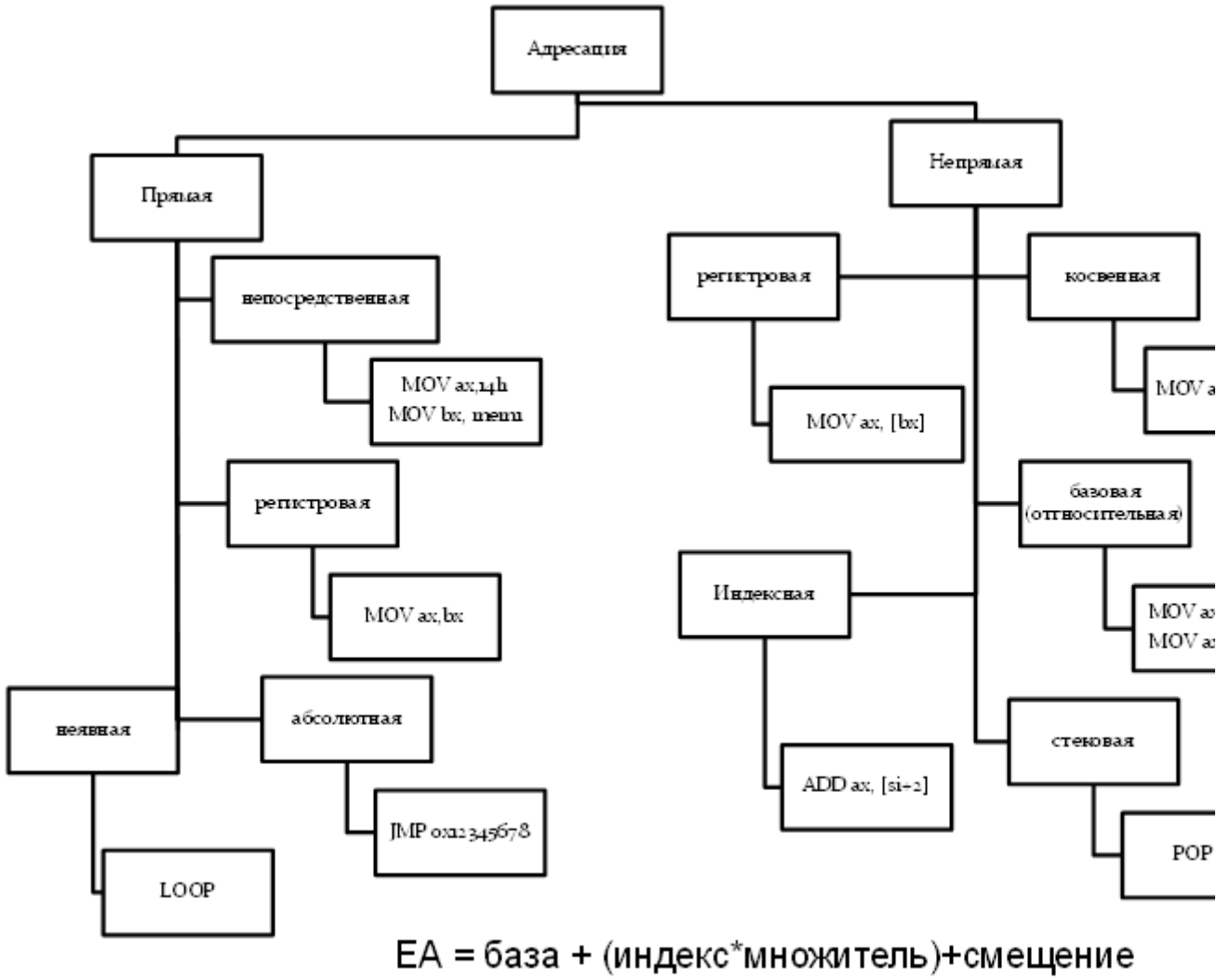
Регистр флагов

32-разрядный регистр, отображает состояние процессора. Изменяет значение определённых бит при переходе из одного состояния в другое. Наиболее часто используются следующие биты регистра флагов:
0 CF - определяет перенос или заём при выполнении арифметических операций. Служит индикатором ошибки при обращении к системным функциям.
2 PF - party flag. 1 если младшие 8 бит результата операции содержат чётное количество двоичных единиц.
4 AF - флаг вспомогательного переноса. Используется в операциях над упакованными двоично-десятичными числами. Определяет перенос в старшую тетраду или заём из старшей тетрады.
6 ZF - устанавливается в 1 если результат операции = 0
7 SF - бит знака. Показывает знак результата операции. Устанавливается в 1, если результат операции отрицательный.
10 DF - флаг направления. Используется особой группой команд, предназначенных для обработки строк. Устанавливается в 0 если строка обрабатывается в прямом направлении (от младших адресов к старшим).
11 OF - флаг переполнения. Фиксирует ПРС.

Ещё в программную модель Intel Pentium входит:
Система команд и
Возможные режимы адресации.

Назначение регистров

Регистры общего назначения могут использоваться для хранения операндов в арифметических и логических операциях, для вычисления адресов и в качестве указателей на ячейки памяти. Однако, следует учитывать специфическое назначение некоторых регистров.
В частности, регистр ESP (SP, Stack Point) предназначен для хранения указателя на вершину стека. Не рекомендуется для произвольного использования в программе.
Регистр AX является регистром аккумулятором, автоматически сохраняет результат после выполнения ряда операций, а так же используется при возврате значения из процедуры в функции.
Регистр ECX (CX) является счётчиком цикла для команд организации цикла. В частности, команда loop выполняет автоматический декремент содержимого CX.
Регистр DX используется для обращения к портам ввода-вывода.
Регистры DI и SI используются для организации индексной адресации.
Регистр BP используется для базовой адресации и доступа к данным, хранящимся в стеке при вызове процедур и функций.
Самыми частыми командами являются использование стека. Захотели попользовать регистр, переместили его в стек, попользовали, вернули.



…в котором хранятся операнды и команды. Способ формирования адреса операнда или метки перехода на другую команду называют режимом или способом адресации. Местоположение операнда называют его эффективным адресом. В самом общем случае, эффективный адрес вычисляется по следующей формуле:
Содержимое базы + смещение + перемещение по массиву
На практике обращение к массиву гораздо проще и определяется способом используемой адресации.
На настоящее время известно более 20 способов адресации и их модификации. Наиболее часто используемые:

- Адресация
  - Прямая - предполагает непосредственное описание операнда или его местоположения в команде
    - Неявная - предполагает, что операнд связан с кодом выполняемой операции. Команда loop использует регистр CX, хотя сам он не указывается.
      - LOOP
    - Непосредственная - в качестве одного из операндов команды указывается значение или имя переменной
      - MOV ax,14h - команда пересылки, данные из операнда второго записываются в операнд первый. 14h - сам операнд.
      - MOV bx, mem1
    - Абсолютная - в качестве операнда указывается адрес ячейки памяти.

        - JMP 0x12345678
    - Регистровая предполагает, что в качестве места хранения операндов определены общедоступные регистры.
        - MOV ax,bx
- Непрямая. Предполагает, что место хранения операндов вычисляется по определённой формуле.
    - Регистровая. Предполагает, что операнд хранится по адресу, указанному в регистре.
        - MOV ax, [bx]
    - Косвенная. Предполагает, что в качестве операнда указан адрес, по которому хранится используемое значение.
        - MOV ax,[1CD5h]
    - базовая (отгносительная) предполагает, что адрес операнда формируется как содержимое некоего базового регистра, плюс смещение относительно данного адреса.
        - MOV ax, [bp]+5 эти записи
        - MOV ax,[bp][5] идентичны. И то и другое приведёт к изменению адреса на 5 байт.
    - Индексная. Используется для организации переходов между элементами массива
        - ADD ax, [si+2] Складываем значение, которое находится в регистре ax, со значением, которое находится по адресу, которое хранится в SI, плюс 2 байта.
    - Стековая. Почему она относится к непрямой - непонятно. Её лучше было бы отнести к неявной.
        - POP bp Переместить из вершины стека в BP.

Это чистые способы адресации. Можно комбинировать способы адресации. Квадратные скобки - это косвенный адрес. То есть, значение, на которое ссылается этот адрес.

```
mov DX, offset mas
      mass db 250 dup (‘*’)
```
непосредственная. Заносим в DX mas, который является массивом из 250 звёздочек.

```
mem1   DD   EC341D7Fh
      mov  ax, word ptr mem1     ;занесётся ax=1d7f
      mov  bx, word ptr mem1+2   ;bx=ec34
```
d?Word ptr - временное преобразование переменной к заданному типу (word)
Смещение на 2 переходит к старшей части

```
mem1  DW  1D7Fh
```
lea bx,mem1 - команда передачи эффективного адреса.

mov ax,[bx] - регистр ax запишем само значение переменной 1d7f

Регистров мало, может использоваться работа прямо с регистром, например DX, а обращение к памяти - в квадратных скобочках.

Младший байт		Старший байт	

7F	1D	34	EC

Команды
MOV
Обеспечивает копирование данных из операнда-источник в операнд-приёмник. При этом размеры операндов должны обязательно совпадать. Кроме того, только один из операндов может быть указан косвенно.
Ptr - временное приведение типов (обрезается).

IN, OUT
Читают или записывают данные из порта ввода-вывода, адрес которого указан в регистре DX. Считанные данные помещаются в регистр AX, EAX либо AL в зависимости от типа. Другие регистры использовать нельзя.

Арифметические команды
ADD o1, o2 Складывает или вычитает..
Sub o1, o2 (из операнда 1 операнд 2) Результат записывается в первый операнд.

INC o1
DEC o1
Увеличивают или уменьшают на единицу значения, указанные в качестве операнда.

Операции целочисленного умножения
MUL sour
И деления
DIV
В качестве приёмника результата используют AL, AX или EX?.. Нее..
Второй операнд указывается в команде. В случае операции целочисленного деления, результат операции записывается в регистр AX или EAX, а остаток от деления в регистр DX или EDX в зависимости от размера.
Команда является одноадресной, второй адрес указывается неявно.
Пример:
```
Mov ax,bx
Mul cx    ;ax=bx*cl

Mov ax,13
Mov cx,3
Div cx    ; ax=4 dx=1
```

Знаковые команды умножения и деления
IMUL
IDIV
Позволяют умножать целые числа с учётом знака. В качестве приёмника результата используется всегда регистровая пара: EDX, EAX. В отличии от MUL и DIV, могут работать с 1, 2 и 3 операндами.

Логические команды
AND, OR, XOR, NOT
Результат записывается по адресу первого операнда

Команды передачи управления
Команды безусловного переход
Передаёт управление в следующую точку.

Переход может выполняться по JE (jump equal)

o1==o2	o1!=o2 o1<>o2	o1>o2	o1<o2	o1<=o2	o1>=o2
JE(JZ)	JNE(JNZ)	JA(JNBE)	JB(JNAE)	JNA(JBE)	JNB(JAE)
Переход если равно	Переход если не равно	Переход если больше	Переход если меньше	Переход если не больше	Переход если не меньше

Команда LOOP предназначена для организации циклических операций. В качестве операнда использует счётчик циклов CX.

```
start:  mov cx,10
for_loop:
        …
         Loop for_loop
final:
```

Команды обработки стека

PUSH o1 заносит в стек
POP o1 извлекает из стека

Команды вызова процедуры и возврата

CALL adr вызова
RET - возврата. Может быть указана с операндом. Операнд предполагает смещение по стеку на заданное количество байт.

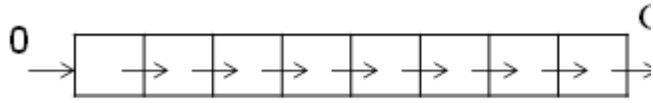
Команда загрузки эффективного адреса

LEA o1,[o2]

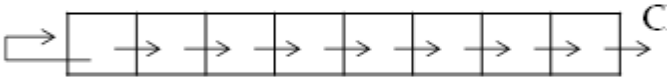
Команды сдвига

Команды сдвига:

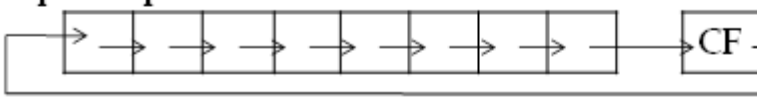
SHR o1 {,size}
SHL o1 {,size} логический сдвиг числа



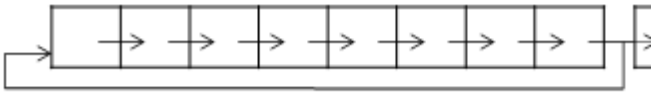
SAR o1 {,size}
SAL o1 {,size} арифметический сдвиг числа



RCR o1 {,size}
RCL o1 {,size} циклический сдвиг через флаг переноса



ROR o1 {,size}
ROL o1 {,size} циклический сдвиг с выносом во флаг переноса



ПРИМЕР: Подсчет количества двоичных единиц в числе

```
        Xor bx,bx
        Mov  cx,16
Repeat:
        Shr  ax,1
        Jnc  not_one
        Inc bx
Not_one:  Loop Repeat
```

Команды сравнения:

CMP o1,o2 С установкой соответствующего знака равенства, изменения операторов не происходит.

Cmp dl,ah

Cmp ax,4

TEST o1,o2 выполняет поразрядное сравнение операндов без сохранения результатов с установкой флага.

Test ax,00000100b

Временное изменение типа переменной:

type Ptr выражение
Mov ax,word ptr [bp+4]
Данные располагаются в перевёрнутом виде.

<<Лек6. ASM.wma>>

Запись начата: 8:28 21 марта 2012 г.

Лек6. ASM

21 марта 2012 г.
8:25

Каркас программы

Программа на ASM состоит из некоторого количества секций. Каждая секция определяет состав и тип хранимой в ней информации. Начало секции определяется директивой, начиняющейся с точки. Окончание секции определяется как начало следующей секции.

.MODEL FLAT, STDCALL

Директива процессора, определяющая модель памяти программы. На современных процессорах - FLAT
STDCALL - директива определяющая порядок и правила восстановления стека после завершения программы.

.DATA
Содержит инициализируемые данные программы.

.DATA?
Определяет неинициализированные данные программы, то есть данные, подлежащие предварительному определению, выделению памяти, инициализации.
Преимущество неинициализированных данных состоит в том, что они не занимают место в исполнительном файле.

.CONST
Определяет набор констант, которые нельзя изменить в программе.

.CODE
Определяет код программы. Может быть разбит на несколько процедур, каждая начинается с имени или метки и ключевого слова proc, далее следует код самой процедуры, заканчивается своим именем и endp (конец процедуры)

.DATA

<инициализируемые данные>

.DATA?

< неинициализируемые данные>

.CONST

< константы>

.CODE

<метка> proc

< код>

<метка> endp

END

Нельзя упускать только две вещи: .MODEL и .CODE

Интерфейс ассемблера с высокоуровневыми языками программирования

При вызове ассемблерных процедур из высокоуровневых программ существует ряд обязательных правил. Такие правила называют соглашениями вызова и они определяют следующие особенности исполнения подпрограммы:

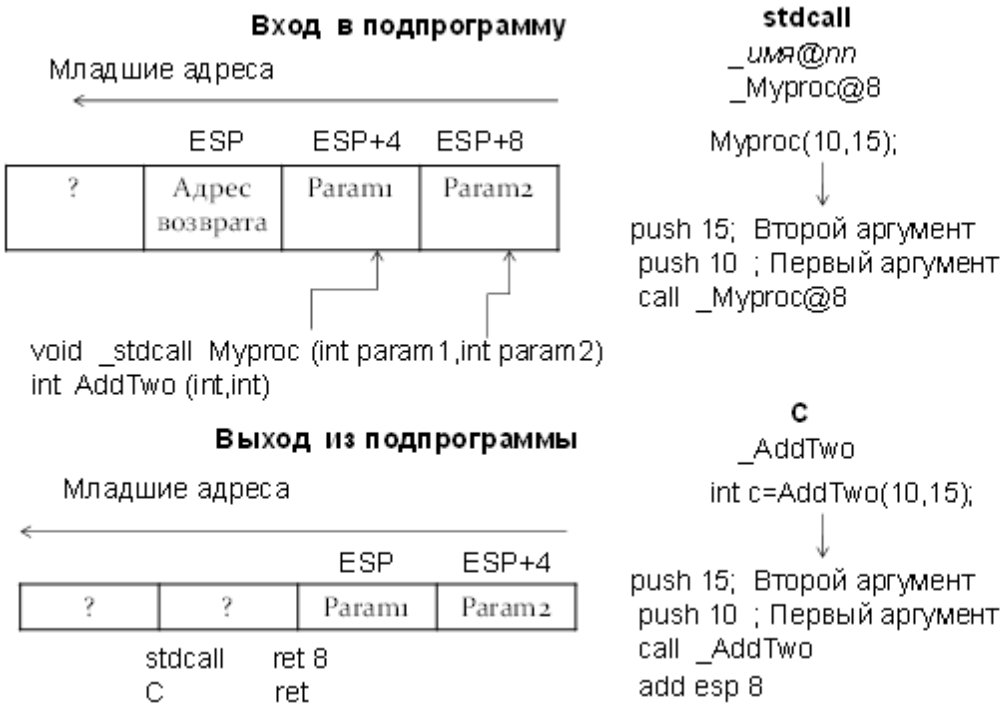
1. Расположение входных параметров подпрограммы и возвращаемых ею значений
    a. Параметры могут находиться в регистре
    b. В стеке
    c. В динамически распределённой памяти
2. Порядок передачи параметра. При использовани для передачи параметра стека определяют, в каком порядке параметры должны помещаться в стек (слева направо или справа налево). При использовании регистров определяется порядок сопоставления параметров и регистров.
    a. Прямой порядок. Параметры размещаются в том же порядке, в котором они перечислены в описании подпрограммы. Достоинства: единообразие кода, ассемблерные и высокоуровневые процедуры.
    b. Обратный порядок. Параметры передаются от конца к началу (на верхнем уровне будет самый первый параметр процедуры). Преимущества: при любом количестве параметров на вершине стека после адреса возврата оказывается первый параметр процедуры, что позволяет использовать функции с неизвестным числом параметров.
2. Кто возвращает указатель стека в исходное состояние (кто восстанавливает стек)
    a. Вызываемая подпрограмма. Преимущества: сокращается объём команд, необходимых для вызова подпрограммы, так как команды восстановления стека записываются один раз в конце подпрограммы.
    b. Вызывающая подпрограмма. При этом вызов усложняется, но облегчается использование подпрограмм с переменным числом и типов параметров.
2. Содержимое каких регистров процессора подпрограмма обязана восстановить перед возвратом. В самом малом случае, это содержимое регистра стека SP, содержимое регистра базы BP и иногда DX, PX (то, что мы будем использовать).

Для организации возврата значений из функции определены следующие соглашения: Если размер возвращаемого значения не превышает 4 байт, то оно передаётся через регистр аккумулятор (AL, AX, EAX). При превышении 4-байтовой границы подпрограмма обязана зарезервировать место в RAM для возвращаемого значения и передать адрес возвращаемого значения через регистр-аккумулятор, либо, если 64-разрядная архитектура, через регистровую пару EDX:EAX.

Директива	Передача параметров	Очистка стека	Использование регистров
fastcall (register)	Слева направо	Процедура (т.е. вызываемая подпрограмма)	eax,edx,ecx (pascal) ecx,edx (VC++)
pascal	Слева направо	Процедура	нет
cdecl	Справа налево	Вызывающая программа	нет
stdcall	Справа налево	Процедура	нет

Fastcall обеспечивает передачу параметров через регистры процессора, причём если мы пишем на pascal и на C - через разные регистры передача параметров.
Pascal - для передачи данных заносится стек, слева направо, вызываемая процедура восстанавливает, регистры не используются.
Cdecl или с предполагает передачу данных или параметров через стек справа налево (в обратном порядке).
Stdcall - параметры записываются в стек..

При организации входа в программу выполняется следующая последовательность действий. Пусть имеется мнемоника вызываемой процедуры:
void _stdcall Myproc (int param1,int param2



В данном случае, так как мы используем stdcall параметры в стек будут помещаться в обратном порядке. Так как параметр 32bit int - сдвигаемся на 4байта. Значение ESP хранит адрес возврата. Перед выходом из подпрограммы и возвратом в основную программу необходимо восстанавливать стек. В случае stdcall при завершении процедуры когда мы удалим всё из стека, стек установится на адерсе возврата, при переходе установится на значении параметра, 8 байт, везде 8, и так и наступает stack owerflow
При выходе нужно сдвинуть указатель стека на количество байт, занимаемых параметром передаваемой процедуры?.. Возвращаемся вправо на сколько-то байт.
Если же с или cdecl, то стек почистит сама вызывающая процедура.
В случае использования директивы stdcall общедоступные имена процедуры при экспортировании заменяются декорированными именами. Декор напоминает тот декор, который был при перегруженных функциях.
_имя@nn (где nn - размер байт, необходимых для передачи параметров)
То есть для Myproc мы получим _Myproc@8

В случае директивы C имя вызываемой процедуры при экспорте не изменяется, добавляется лишь знак подчёркивания. Обращение приводит к добавке аргументов, вызову и после возврата из вызываемой функции осуществляется команда увеличения стека на количество байт на объём передаваемых параметров. Add esp,8

Стековый фрейм

Так как в пределах ОС может быть запущено несколько процессов, для каждого из процессов при выполнении вызова подпрограмм создаётся стековый фрейм, или его ещё называют Запись активаций.
Стековый фрейм - это область в стеке, расположенная за адресом возврата из подпрограммы, в которой размещаются передаваемые параметры, сохранённые регистры и локальные переменные.
Для создания стекового фрейма, программы должна выполнить следующие действия:

1. поместить аргументы вызываемой процедуры в стек;

2. вызвать процедуру командой CALL, в результате чего адрес возврата помещается в стек;
3. в начале выполнения процедуры сохранить в стеке регистр EBP, так как в дальнейшем регистр EBP используется для доступа к аргументу и локальным переменным функции или подпрограммы;
4. загрузить в регистр EBP текущий указатель стека из регистра ESP.

На структуру стек-фрейма оказывает влияние структура памяти и соглашение о передаче параметров.

Пусть имеется вызываемая процедура, вычисляющая разность двух целых чисел и возвращающая результат через регистр-аккумулятор.

Если будем писать в студии - указание параметра обязательно, в борландовском - необязательно.

Пример. Пусть необходимо создать процедуру суммирования двух целых чисел с возвратом значения через третий параметр. В формате C, в качестве возвращаемого значения - ссылку.

Пример вызова процедуры

```
extern "C" void sum(int a, int b, int & c);
void main()
{
int a,b,c;
a=10;
b=20;
sum(a,b,c);
cout << c <<"\n";
}
```

```
.MODEL FLAT,C ;
.CODE
sum proc a:dword, b:dword,cp:ptr dword
        push ebp
        mov ebp,esp
        push eax
        push ebx
        push esi
        mov eax,dword ptr [ebp+8];
        mov ebx,dword ptr [ebp+12];
        add eax,ebx
        mov esi, dword ptr [ebp+16];
        mov [esi],eax ;
        pop esi
        pop ebx
        pop eax
        mov esp,ebp
        pop ebp
        ret
sum     endp
        end
```

Содержимое регистра BP	BP
Адрес возврата в main	BP+4
Значение переменной "a"	BP+8
Значение переменной "b"	BP+12
Адрес переменной "c" в процедуре main	BP+16

В ряде случаев бывает необходимо использовать локальные переменные. Для них необходимо заранее резервировать область в стеке. Сложная функция, нужно где-то

временно хранить сумму, где-то временно хранить разность.

Вызов функции с локальными переменными

$$C=(A+B)+(B-A)^* B^*A$$

```
extern "C" int calc(int a, int b);
void main()
{int a,b,c;
a=10;
b=20;
c = calc(a,b);
cout << c <<"\n";
}
```

Содержимое	
место для 2-ой локальной. переменной	BP-8
место для 1-ой локальной. переменной	BP-4
Содержимое регистра BP	BP
Адрес возврата в main	
Значение переменной "a"	BP+8
Значение переменной "b"	BP+12

```
.MODEL FLAT,C
.CODE
 calc proc  a:dword,  b:dword
 push ebp
 mov ebp,esp
sub esp,8 ;
push ebx
mov eax,dword ptr [ebp+8] ; a
mov ebx,dword ptr [ebp+12] ; b
 add eax,ebx
mov dword ptr [ebp-4],eax  ;  a+b
 mov eax,dword ptr [ebp+8]
sub ebx,eax
mov dword ptr [ebp-8],ebx  ;  b-a
mov ebx,dword ptr [ebp+12]
mul ebx   ; a*b
mul  dword ptr [ebp-8]   ; a*b*(b-a)
add eax,dword ptr [ebp-4]   ; ab(b-a)+(a
pop ebx
mov esp,ebp
pop ebp
ret
calc endp
end
```

Лек7. Основы программирования для Windows

28 марта 2012 г.
11:33

<<Лек7. .wma>>

Запись начата: 11:34 28 марта 2012 г.

<<Основы программирования для Windows.ppt>>

Все, кто 9 и более баллов, контрольная зачтена. Возможно, включит некоторые вопросы в следующие.
Для аттестации: КР и Лаб1.

ОС Windows является многопользовательской многозадачной ОС, что позволяет одновременно выполнять несколько приложений пользователей. Это приводит к тому, что при написании программ под Windows следует учитывать возможность присутствия в системе нескольких активных копий одного и того же приложения.

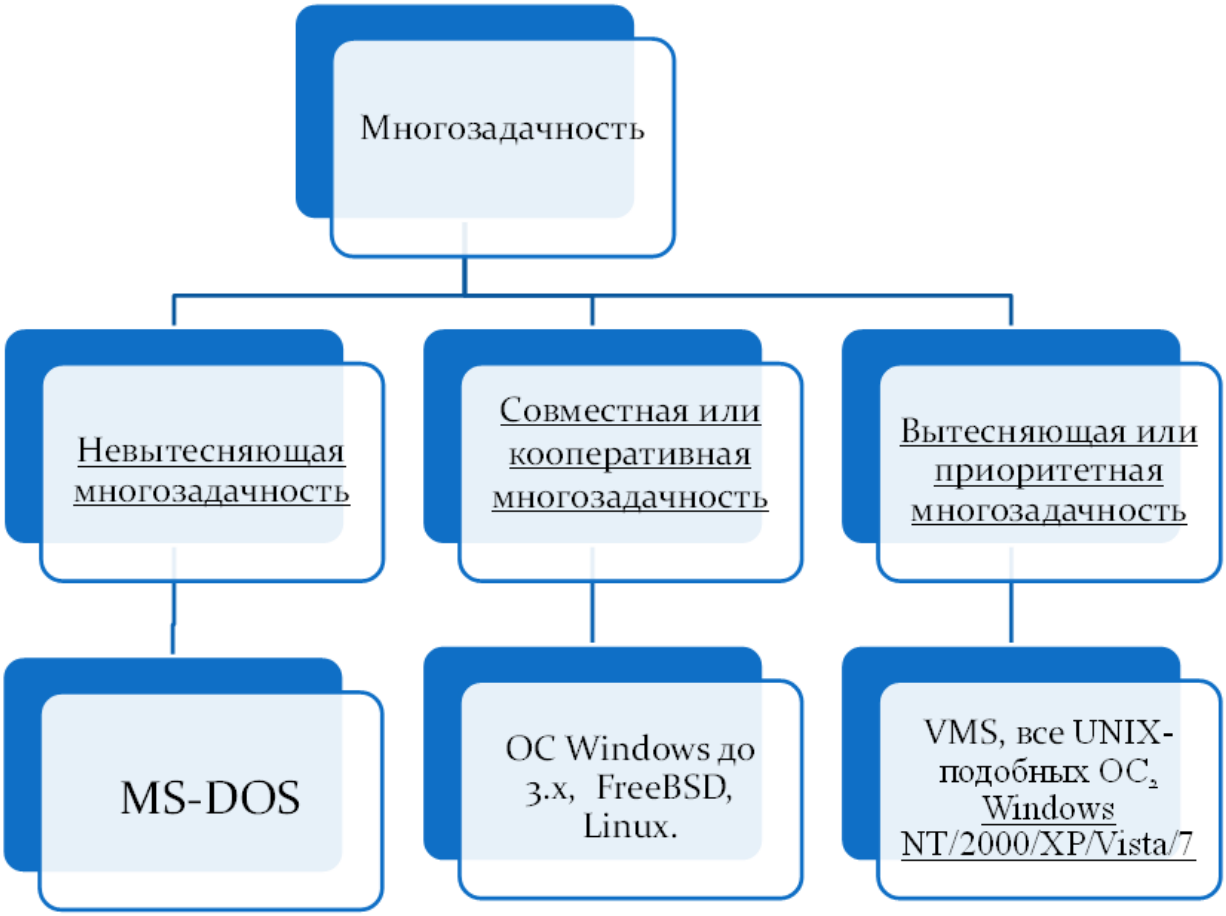
К основным особенностям относят:

- С Win95 файловая система использует таблицу размещения файлов защищенного режима (Protected-mode FAT или VFAT), что позволяет использовать длинные имена файлов. Для программ с короткими именами используется тильда-расширение. При работе с файлом программист может отобразить его на виртуальное адресное пространство, после чего с файлом работают как с участком оперативной памяти. Отображённый файл можно сделать одновременно доступным нескольким приложениям, что обеспечивает связь между приложениями (напрямую между собой контактировать не могут, а вот через файлы могут)
- Управление программы. Windows-приложения могут иметь возможность влиять друг на друга. Данное свойство реализуется за счёт многозадачности. Многозадачность - это свойство операционной системы обеспечить параллельной обработки нескольких процессов. Истинная многозадачность возможна только в распределённых вычислительных системах. В современных Windows операционных системах используется Вытесняющая многозада́чность.
- Несегментированная линейная модель памяти FLAT-модель.
- Возможность использования Библиотек динамической загрузки. Основные функции размещаются внутри программного модуля. Отдельно определяются функции-обработчики прерываний, которые находятся в памяти постоянно и являются общедоступными. Аналогичным образом организован принцип динамической загрузки в системе Windows. Файлы динамических библиотек загружаются в память однократно, находятся в ней до момента последнего использования, возможно обращение к функциям одной библиотеки из нескольких приложений системы. Загрузка библиотеки выполняется при первом обращении к ней из активного приложения. (более подробно на следующей лекции)
- Организация очереди сообщений. Windows преобразует каждое событие, происходящее в системе в сообщение и помещает его в очередь сообщений конкретной программы. Вся работа приложений Windows основана на передаче сообщений. Сообщения - это запись данных, имеющих отношения к происходящему в системе событию. Очередь сообщений - это последовательность сообщений, ожидаемых обработки.
- Регистрационная база данных. Позволяет создавать соответствующие записи об использовании приложений. Чаще всего используется для организации DDE, COM и OLE сервисов.

Многозадачность

Примитивные многозадачные среды обеспечивают чистое разделение ресурсов. То есть за каждой конкретной задачей закрепляется определённый участок памяти или другой ресурс, при этом задача активизируется в строгом интервале времени. Более развитые системы производят разделение ресурсов динамически. Когда задача загружается или покидает память, в зависимости от её приоритета и стратегии, ей выделяется процессорное время, память или другие ресурсы. При этом, у каждой задачи должен быть определён собственный приоритет. Система организует очередь задач так, чтобы все задачи рано или поздно всё-таки ресурс получили. По окончании кванта времени ядро ОС переводит задачу из состояния выполнения в состояние готовности, отдавая ресурсы другим задачам. При нехватке памяти страницы не выполняющихся задач могут быть вытеснены на диск, а потом через некоторое время восстановлены в памяти. При этом, система должна распознавать сбои и зависания отдельных задач с целью их прекращения. Система самостоятельно решает

конфликты доступа к ресурсам и устройствам, не допуская тупиковых ситуаций, связанных с бесконечным ожиданием доступа к тому или иному ресурсу. Различают следующие типы многозадачности:



- Невытесняющая многозадачность - это тип многозадачности, при котором ОС может одновременно загрузить в память 2 или более приложения, при этом процессорное время выделяется только одному приложению. Для выполнения фонового приложения, оно должно быть дополнительно активизировано (то есть система сама не отслеживает, что пора закончить это и дать время другому)
- Совместная или кооперативная многозадачность. Тип многозадачности, при котором следующая задача выполняется только после того, как текущая задача явно объявит себя готовой отдать процессорное время другим задачам. Такую многозадачность называют многозадачностью второй ступени, так как она использует более сложным механизм, чем простое переключение между приложениями. При кооперативной многозадачности приложение может захватить столько процессорного времени, сколько считает нужным. Передача управления осуществляется за счёт анализа периодических управляющих сигналов. Основным преимуществом кооперативной многозадачности является отсутствие необходимости защищать разделяемые структуры данных, что управляет программирование и перенос кода из однозадачных в многозадачные среды. Недостаток - неспособность всех приложений работать в случае ошибки, возникающей в одной из них, если это приложение было активно.
- Вытесняющая многозадачность. Вид многозадачности, при котором ОС сама передаёт управление от одной выполняемой программы другой. Передача

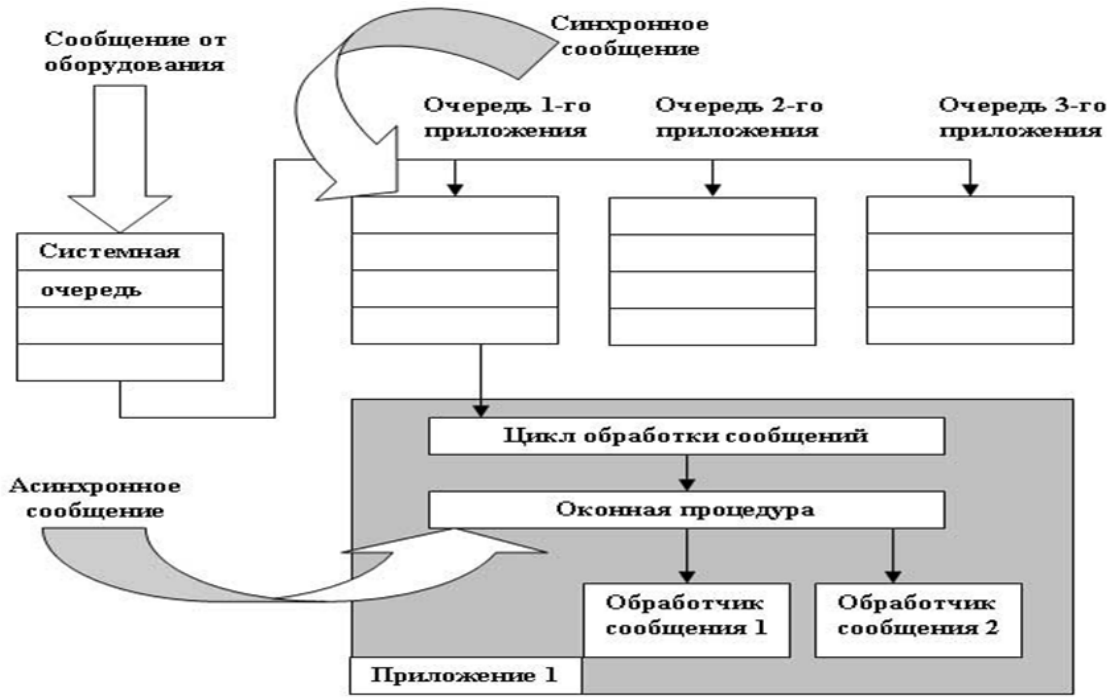
управления осуществляется в случае завершения операции ввода-вывода, при возникновении событий, пришедших от аппаратных устройств, при истечении таймеров и квантов выделенного времени. Либо при поступлении каких-то сигналов от одной программы другой.

Преимущества: возможность полной реализации многозадачности при выполнении операции ввода-вывода, повышение надёжности системы, так как ошибка одной пользовательской программе не приводит к краху всей системы в целом. Возможность использования в многопоточных и многоядерных системах.
Недостаток: необходимы особые дисциплины при написании кода, внимание к защите разделяемых и глобальных данных, организация при необходимости критических секций и элементов синхронизации (семафоров, мьютексов и так далее)

Процесс и поток

Процесс создаётся, когда программа загружается для выполнения. Каждому процессу в монопольное использование выделяется изолированное адресное пространство. Сразу после загрузки процесса создаётся хотя бы одна задача (поток, в российской литературе).
Поток - это фрагмент кода приложения, который может выполняться автономно и независимо от других потоков в рамках одного процесса. При необходимости, поток может взаимодействовать с другими потоками собственной задачи. Это возможно так как они находятся в общем адресном пространстве, выделенном конкретному процессу. (программа может работать как псевдопараллельная, либо однопоточная).
Синхронизация многопоточных задач выполняется через общую память или через семафоры, мьютексы, системные секции и так далее.

Очередь сообщений



Любое сообщение, подлежащее обработке должно попасть в очередь сообщения, способного обработать его приложение.
Выделяют два типа очередей:

- Системная очередь сообщений. В неё поступают сообщения от аппаратных устройств.
- Очередь сообщений приложений. В неё записываются данные из системной очереди, а также внутренние синхронные сообщения, возникающие внутри

приложения. Системная очередь распределяет свои сообщения по очередям приложений в соответствии с идентификаторами приложений. Сообщения разделяются на:

a. Синхронные сообщения - сообщения, приостанавливающие поток до момента обработки данного сообщения.
b. Асинхронные сообщения - сообщения, которые могут возникнуть в ходе обработки синхронных сообщений, записывающих информацию в очередь, но не прерывающие потока с целью ожидания обработки. Передается непосредственно в нужное окно и удаляется потом.

Для обработки сообщений каждое приложение имеет цикл обработки приложений, в котором определяется тип происходящего сообщения и на основании данных оконной процедуры, выбирается тот или иной обработчик сообщений

Структура сообщения

typedef struct MSG

{

HWND hwnd; // определяет дескриптор окна, в котором возникло данное сообщение

UINT message; // определяет идентификатор сообщения

WPARAM wParam; // 32-разрядне дополнительные параметры сообщения, -

LPARAM lParam; // определяющиеся кодом и видом сообщения. Для определения типа обработки сообщения.

DWORD time; //Определяет время, когда возникшее сообщение было помещено в очередь

POINT pt; // определяет положение курсора мыши в момент появления сообщения.

}

Сообщения от мыши передаются окну, над которым она находится. События клавиатуры передаются в окно, на котором фокус.

Регистрационная база данных хранит информацию о системе, о том, что запущено, и параметрах, им передаваемых.

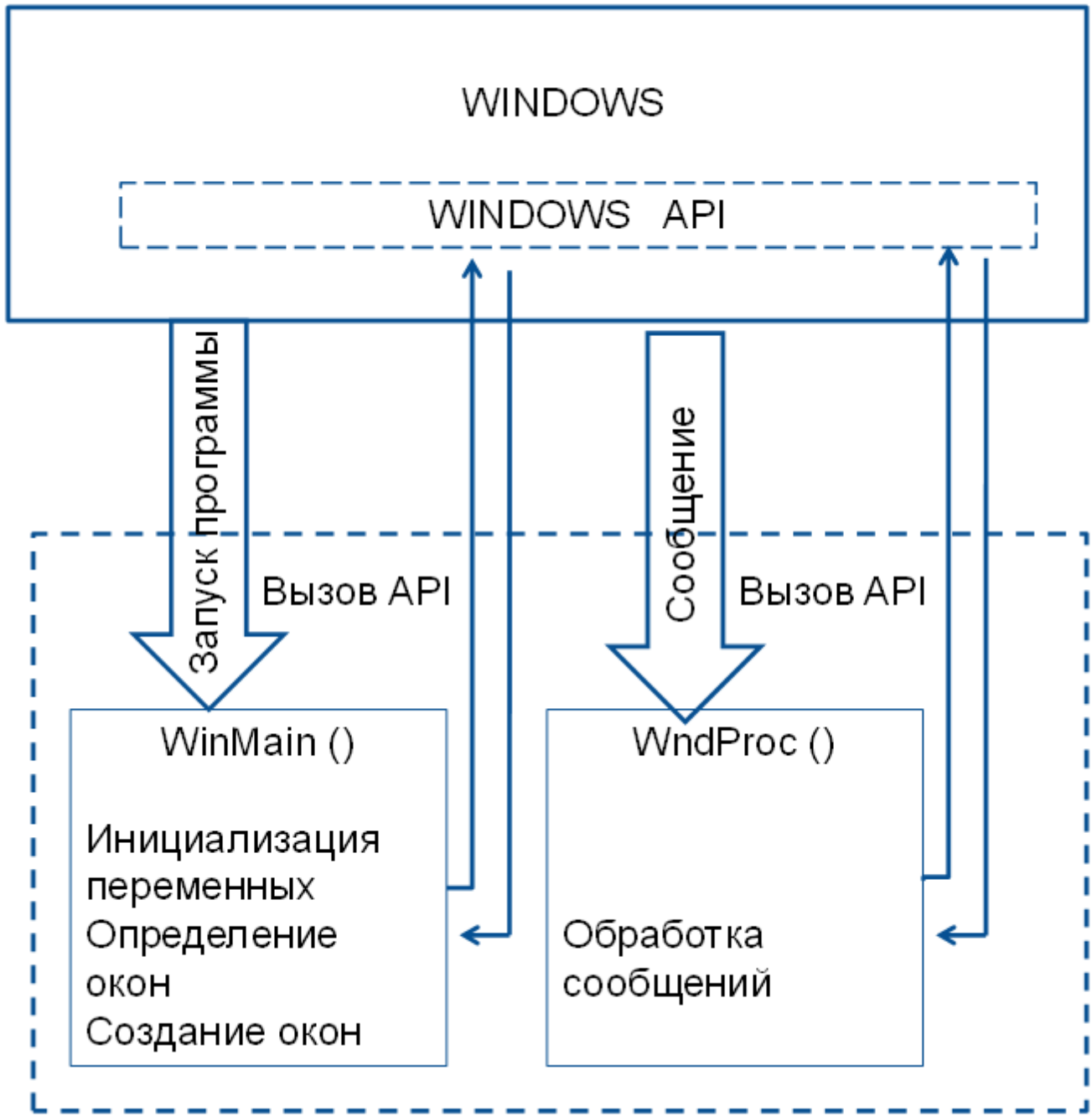
В настоящее время в VS C++ есть три способа создания приложений:

- С использованием интерфейса WIN API - это базовый интерфейс ОС для взаимодействия с приложениями, выполняющимися под её управлением. Является трудоёмким, но максимально информативным. На нём можно на низком уровне писать Win приложения.
- Использование классов MFC - набор классов, написанных на C++, инкапсулирующих в себе функции API интерфейса. Менее трудоёмкая структура, требует знания только классов MFC и его использования (в VS вызываем методы классов)
- Использование Windows Forms, разработка оконных приложений, выполняющихся под управлением CLR, при этом под формой подразумевается любой объект экрана (кнопка, окошко, поле). Максимально облегчённое программирование, за нас всё решает среда.

Показала фильм с TechDay про писание оконной программки. Смесь WinAPI и MFC.

04.04.2012 8:30
<<Лек7. Основы программирования для Windows.wma>>

Запись начата: 8:30 4 апреля 2012 г.



Windows-приложение состоит как минимум из двух основных процедур

1. WinMain(), составляющей основу приложения. Данная функция является точкой входа в приложение. Выполняется при запуске программы, инициализирует все необходимые данные приложения, определяет его окна и запускает главное окно приложения.
2. Оконная процедура WndProc() исполняет роль диспетчера приложений. Само приложение никогда не получает прямого доступа к оконной процедуре. Оконная процедура регистрируется операционной системой и вызывается каждый раз, когда выполняются какие-либо действия над принадлежащей ей окном (находит обработчик для действия, совершенного с окном). Оконная процедура является

функцией обратного вызова. Оконная процедура выполняется через API при возникновении события. Вызывается с помощью множественного выбора case.

(рассказывает, как работает)

Функция WinMain (может быть определена в двух варианта)

int APIENTRY **WinMain** *{определяет точку входа в приложение}*
(HINSTANCE **hInstance** *{определяет уникальный идентификатор текущей копии (запущенного процесса) приложения; создаётся winapi при запуске приложения}*,
HINSTANCE **hPrevInstance** *{в 16-разрядных приложениях определяет идентификатор предыдущей запущенной копии приложения; так как в 32-разрядной архитектуре каждая копия приложения запускается в собственном адресном пространстве, для 32-разрядной архитектуры hPrevInstance=0; потому что в 32-битных приложение запускается в собственном адресном пространстве и ничего знать о других процессах не может}*,

LPSTR lpszCmdLine *{параметр командной строки, определяет набор параметров, заданных при запуске приложения}* ,

int nCmdShow *{параметр, определяющий тип открытия окна, принимает значения SW_MINIMIZE, SW_SHOW }*)

{

- Начальная инициализация приложения (подготовка данных класса окна и его регистрация);
- Создание главного окна приложения
- Запуск цикла обработки сообщений, извлекаемых из очереди

}
//это то, что мы должны написать внутри функции WinMain

WinMain самостоятельно восстанавливает себя после завершения.
Каждое окно, используемое в приложении должно быть создано и зарегистрировано

Регистрация окон

Любые окна, содержащиеся в классах Windows приложений должны относиться к одному из зарегистрированных приложений класса. Класс окна определяет наиболее общие свойства и является шаблоном по которому определяются стили, шрифты, заголовки, положения и прочие свойства окна. Каждый класс имеет собственную структуру параметров. Структура является общедоступной, что позволяет не дублировать данные при создании множества окон по общему шаблону. При этом имя класса должно быть уникальным, чтобы не создавать конфликтов с другими окнами приложений.

WNDCLASSEX wc; //структура класса окна определяется с помощью этого типа
wc.cbSize = sizeof(WNDCLASSEX); //определяет размер класса окна в байтах
wc.hIconSm = 0 ; //позволяет создать значок иконки окна при срёртке, загружаемой из файла ресурсов. Default =0
wc.style = CS_HREDRAW|CS_VREDRAW; //определяет стиль отображения окна
wc.lpfnWndProc = (WNDPROC)WndProc; //определяет имя оконной процедуры, связанной с данным окном.
//дополнительные поля, являющиеся полями-расширениями
wc.cbClsExtra = 0; // определяет число байт, которые нужно запросить дополнительно у ОС под структуру и собственные данные, принадлежащих текущему окну
wc.cbWndExtra = 0; //определяет число байт, дополнительно запрашиваемых у ОС

для размещения структур, присоединённых к данному окну.

wc.hInstance =hInst; //идентификатор копии приложения, которой принадлежит данное окно

wc.hIcon = LoadImage(hInst, MAKEINTRESOURCE(IDI_APPLICATION));

wc.hCursor = LoadCursor(NULL, IDC_ARROW); //определяет тип курсора над окном (может быть загружен из ресурсных файлов)

wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1); //определяет характер подложки окна

wc.lpszMenuName = NULL; //определяет наличие главного меню окна (ссылку на ресурс-меню, связанное с данным окном)

wc.lpszClassName = szAppName; //определяет имя класса, получаемое им при соответствующего окна (является уникальным)

if(!RegisterClassex(&wc)) передается ссылка на регистрируемую структуру

return FALSE;

После регистрации возможен доступ к окну и его запуск.
При успешном создании под окно выделяется память и возвращается целочисленный идентификатор. Если невозможно создать, хорошо createWindow возвращает 0. Об этом ниже.

После выполнения процедуры окна необъодимо выполнить его регистрацию. Регистрация окна выполняется с помощью RegisterClassex. В качестве параметра данная процедура получает ссылку на сформированную структуру, возвращает значение true, если окно зарегистрировано в системе. Регистрация окна (даже успешно завершённая) не свидетельствует о том, что окно создано. Для создания окна используются дополнительные функции. Основной функцией создания окна является функция InitInstance. (обычно выделяется как отедельная функция, но может быть описана и в WinMain).

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)

{ HWND hWnd {определяет тип идентификатора окна};

hWnd = CreateWindow(szWindowClass, //запуск функции createWindow; данная функция создаёт окно в соответствии с заданными параметрами и возвращает при успешном создании идентификатор окна, если окно создать невозможно возвращает нуль?..

szTitle,

WS_OVERLAPPEDWINDOW,//стиль окна

CW_USEDEFAULT, 0, // координата левого верхнего угла

CW_USEDEFAULT, 0, // ширина, высота окна

NULL, // дескриптор родительского окна; если имеет нулевое значение, то это значит, что данное окно является главным окном приложения.

NULL, // дескриптор меню окна

hInstance, // дескриптор экземпляра приложения

NUL) ; // указатель на дополнительны данные окна;

if (!hWnd)

{ return FALSE; }

ShowWindow(hWnd*{идентификатор созданного окна}*, nCmdShow*{тип открытия}*); //необходимо выполнить, чтобы окно было отображено. После этого отображается на экране и начинает действовать.

UpdateWindow(hWnd); //перерисовка окна при его полном или частичном перекрытии другими окнами системы

return TRUE;

}

LRESULT WINAPI WndProc(HWND hWnd,
UINT msg,
WPARAM wParam, LPARAM lParam);
LRESULT CALLBACK WndProc(HWND hWnd, UINT msg,
WPARAM wParam, LPARAM lParam);

это описано ниже <Есть сообщения отвечающие за завершение работы приложения. Генерирует асинхронного сообщение wlquit, которое подлежит немедленному сообщению. В процедуре окна может быть определён обработчик по умолчанию, не имеющий спецификаций в данной процедуре. Это обрабатывается средства ос. Функция окна является функцией WinAPI и имеет список параметров.>

hWnd - идентификатор окна, которому принадлежит данная функция
Msg - сообщение
wParam, lParam - дополнительные параметры, определяющие характеристики оконной процедуры, в частности: идентификатор дочернего окна, идентификаторы органов управления, идентификатор родительского окна.
В некоторых случаях оконная процедура может быть объявлена как CallBack-процедура.

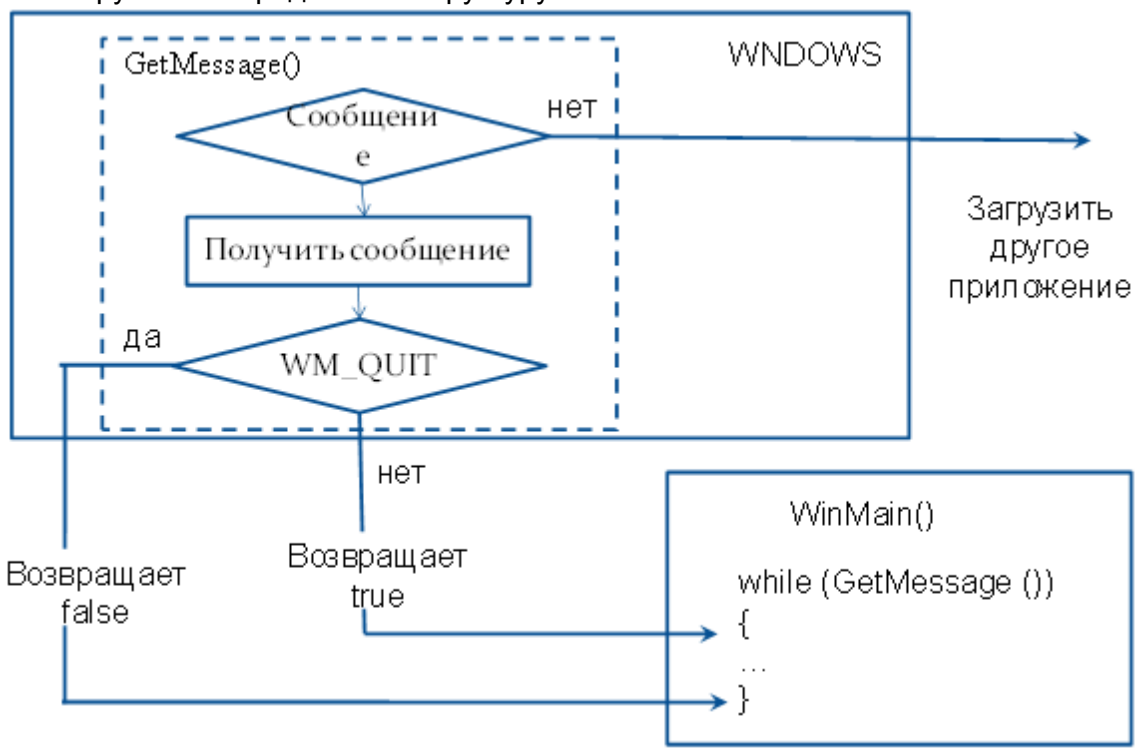
Функция окна

Основным назначением функции окна является обработка поступивших для данного окна событий. Для определения приложения и окна, которому направляется событие, используется понятие Фокус Ввода. Это атрибут, который в текущий момент времени может принадлежать только одному окну. Если окно владеет фокусом ввода, то все сообщения, поступающие от клавиатуры, помещаются в очередь приложения, сформировавшего данное окно, а затем передаются функции окна wndProc. Сообщения от мыши обрабатываются функцией окна, над которым находится курсор.
В Windows определено порядка 200 типов оконных сообщений, все оконные сообщения начинаются с префикса wm. Оконная процедура выбирает сообщение из очереди приложения и идентифицирует их по коду MSG. Определяется тип пришедшего сообщения и обрабатывается внутри оконной процедуры.
Оконная процедура обязана выполнять специфические обработчики сообщений, в частности, обязательно должно быть обработано сообщение WM_DESTROY, данное сообщение приходит при завершении работы приложения стандартным способом. Обработчик WM_DESTROY должен содержать вызов функции PostQuikMessage. Данная функция инициализирует завершение приложения, очищает очередь сообщений и обеспечивает восстановление стека. Вызвать событие WM_DESTROY можно либо автоматически при закрытии приложения, либо путём передачи сообщения WM_QUIT. При появлении этого сообщения, оно ставится в начало очереди сообщений и формирует событие WM_DESTROY.
В случае, если происходит событие, не имеющее обработчика в данной оконной процедуре, должна быть сформирована функция запуска обработчика по-умолчанию DefWindowsProc, который передаёт сообщение операционной системе.
При завершении работы ОС всем активным приложениям в очередь сообщений передаётся системное сообщение WM_QUERYENDSESSION, которое формирует WM_QUIT, которое вызывает WM_DESTROY, которое вызывает PostQuitMessage(0);

```
LRESULT WINAPI WndProc(HWND hWnd, UINT msg,
                WPARAM wParam, LPARAM lParam)
{
switch(msg)
```

```
 {    case WM_LBUTTONDOWN:
       {     MessageBox(NULL,"Нажата левая клавиша мыши«,"Сообщение ",
      MB_OK|MB_ICONINFORMATION);
             return 0;
       }
          }
       case WM_DESTROY:
       {
             PostQuitMessage(0);
             return 0;
       }
             default:  DefWindowProc(hwnd, msg, wParam, lParam);
}
```

Каким образом извлекается сообщение из очереди? Используется стандартный механизм, называемый подкачкой сообщений или циклом сообщений. Для организации сообщений используется функция GetMessage. Определяет наличие сообщений в очереди. Если в очереди сообщений нет, то управление в операционной системе на базе вытесняющей многозадачности может быть передано другому приложению. Если сообщение в очереди есть, оно извлекается и копируется в структуру MSG,... Анализируется и передается в структуру окна.



Через операционную систему вызывается функция обработчика окна.

Цикл обработки сообщений

```
MSG msg
while (GetMessage (&msg,NULL,0,0))
//обязательный вызов двух функций:
{    TranslateMessage(&msg); //обеспечивает преобразования кодов нажатых клавиш
```

(полученного сообщения) в набор символов и передача этого набора в очередь прикладной задачи

DispatchMessage(&msg) //обеспечивает перераспределение (диспетчеризацию) сообщения на основании параметра HWND, то есть передаёт указатель на текущее сообщение операционной системе, которая вызывает соответствующую функцию окна. При этом, если полученное сообщение является сообщением от таймера, то вместо стандартной функции окна можно вызвать другую ранее созданную функцию.

}

В качестве параметров GetMessage получает идентификатор окна (в которое фокус ввода). Если данный параметр равный нулю, то функция окна будет получать все сообщения, принадлежащие этому приложению. Третий и четвёртый параметры определяют минимальный и максимальный идентификатор обрабатываемых сообщений окна.

GetMessage может передать управление другому обработчику сообщений в случае если его собственная очередь сообщений пуста, либо для другого приложения получены сообщения от таймера или WM_PAINT (сообщения перерисовки).

Обрабатываемые чаще всего сообщения:

При создании окна WM_CREATE,
WM_GETMINMAXINFO
WM_NCCREATE

При перерисовке WM_PAINT

При выходе WM_QUIT

При завершении работы Windows WM_QUERYENDSESSION

Для определения (в 32-разрядных средах) копии ранее запущенного приложения может быть запущена функция FindWindow, которая позволяет найти окно верхнего уровня по имени класса или заголовку (имени) окна. Если окно обнаружено, ОС вернет идентификатор класса окна. Иногда достаточно просто инициализировать, а не заново создавать.

```
HWND FindWindow
    (  LPCTSTR lpClassName, //имя класса окна
    LPCTSTR lpWindowName); //имя окна
```

Вместо имени класса окна можно передать параметр типа Atom. Atom создаётся при каждом запуске главного окна приложения и может быть получен от операционной системы в качестве идентификатора окна. Чтобы создать собственный atom при запуске окна используют функцию GlobalAddAtom.

```
BOOL IsIconic  (HWND hwnd);
BOOL SetForegroundWindow  (HWND hwnd);

    LRESULT WINAPI    WndProc(HWND hWnd, UINT msg,
                    WPARAM wParam, LPARAM lParam);
    szAppName[] = "Window";
    char szAppTitle[] = "Window Application";
int  APIENTRY  WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                        LPSTR lpCmdLine, int nCmdShow)
{   WNDCLASSEX wc;
    HWND hWnd;
```

```
    MSG msg;
    hInst = hInstance;
    hWnd = FindWindow(szAppName, NULL);
  if   (hWnd)
    { if   (IsIconic(hWnd))
  ShowWindow(hWnd, SW_RESTORE);
    SetForegroundWindow(hWnd);
  return FALSE;
     }
```

Лек8-9. Создание и использование библиотек динамической загрузки

4 апреля 2012 г.
9:20

Наличие библиотек динамической компоновки является одной из основных особенностей операционной системы Windows. (в принципе сама OS Windows написана как набор DLLек)
Сама DLL является компонентом операционной системы и может использоваться в пользовательских приложениях для обеспечения более эффективной работы.

В самом общем случае создание исполняемого файла приложения состоит из следующих этапов:

1. Подготовка текста приложения на выбранном языке программирования
2. Трансляция текста с получением на выходе объектного файла
3. Компоновка из набора объектных файлов исполняемого загрузочного модуля программы (exe-модуль).

Этап компоновки может выполняться в статическом и динамическом режиме. В чём разница?
При статической компоновке, копии вызываемых программами функций, содержащихся в статической библиотеке, копируются в исполняемый модуль программы.

При динамической компоновке в программный код приложения встраивается лишь указатель на вызываемую функцию. Сама библиотека, содержащая набор функций, загружается в память один раз и является общедоступной для всех использующих её приложений.
Минусы:

1. Нужно нести целый пакет DLLек в кармане
2. Контроль версий
3. Нужно 4 строчки кода, а загружаем DLLку на 4 гига :D

Удобства:

1. DLL загружается в адресное пространство динамически, что позволяет приложению подгружать нужные функции в требуемый момент времени.
2. Возможность использования модулей, написанных на разных языка. Связано это с тем, что DLL показывают только свои интерфейсы (протягивают ручки), а что там

внутри - нас не касается.
3. Упрощение разработки групповых проектов. Если проект пишется разными разработчиками, то можно их попросить оформлять свои части в виде отдельных DLLек.
4. Экономия памяти (многим приложениям нужна одна и та же DLLка)
5. Разделение ресурсов (в одной DLLке куча ресурсов для разных приложений)
6. Упрощение локализации (одна библиотека на функционал, другие на интерфейс)
7. Расширение специфических возможностей (обычно расширение функциональных возможностей Windows возможно только с помощью COM-объектов, а они делаются через DLL, это подобно тому, как ActiveX в браузерах)

При создании DLL библиотеки основное внимание уделяется отображению библиотеки в адресное пространство запущенных процессов. Чтобы приложение могло вызвать функцию, содержащуюся в DLL, образ файла DLL должен быть спроецирован в адресное пространство вызывающего процесса. Это осуществляется за счёт динамического неявного связывания во время загрузки приложения (раннее связывание), либо за счёт динамического явного связывания во время выполнения программы (позднее связывание).

Код DLL отображается в памяти один раз, а данные отображаются столько раз, сколько она вызвана.
Стоит сказать, что DLL не имеет собственного стека и собственной памяти для хранения локальных данных. Поэтому работа DLL осуществляется в рамках вызвавшего её процесса.

Структура приложения

Любое приложение при запуске создаёт процесс. Процесс по сути является копией запущенного приложения. Одновременно в системе может существовать несколько копий приложения. Тогда можно выделить следующие части приложения в системе:
1. Модуль приложения. Загружается в систему однократно, содержит в себе неизменяемую часть приложения (код и ресурсы, то, что не меняется в процессе времени)
2. Процесс. При создании каждого процесса выделяется виртуальное адресное пространство, в котором располагается очередь сообщений приложения, стек данной копии, очередь сообщений.

DLL библиотека при запуске формирует только один модуль, содержащий Код, Ресурсы и небольшую область локальных данных для констант.
Почему не запускается DLLка, потому что при загрузке у неё нет процесса, нет стека, локальных данных. (а точка входа у неё есть).

Принципы явного и неявного связывания

11.04.2012 11:29
<<Лек8. Создание и использование библиотек динамической загрузки.wma>>

Запись начата: 11:29 11 апреля 2012 г.

Существует два типа связывания библиотеки с пользовательским приложением:

Неявное связывание

- это динамическое связывание во время загрузки программы, при этом код)приложения ссылается на идентификаторы, содержащиеся в DLL и тем самым заставляет загрузчик неявно подгрузить нужную библиотеку при запуске приложения. Будем считать, что

исполняемый EXE модуль импортирует функции и переменные из DLL библиотеки. Тогда DLL экспортирует свои элементы в исполняемый модуль.

Порядок неявного связывания следующий

Собрав исполнительный модуль, в который импортируются элементы DLL необходимо создать библиотеку в следующем порядке:

1. Создаются заголовочные файлы (h), содержащие прототипы функций, структуры и идентификаторы, импортируемые из DLL. Заголовочный файл включается в исходный код всех DLL модулей и используется при сборке исполняемого файла.
2. Создаются модули исходного текста, содержащие тела экспортируемых и неэкспортируемых функций библиотек.
3. Выполняется компиляция исходных модулей, при которой исходный код преобразуется в объектный (.obj), таким образом для каждого компонента библиотеки формируется собственный объектный модуль.
4. Посредством компоновщика объекты компоновки последовательно собираются в единый загрузочный модуль DLL библиотеки, в который помещается двоичный код, глобальные и статические переменные, относящиеся к DLL.
5. Если компоновщик обнаруживает, что в объектном модуле имеются экспортируevst функции или структуры, то для модулей библиотеки создаётся специализированный .lib-файл, который содержит список символьных имён функций и переменных, которые экспортируются из DLL.
6. Во все модули исходного кода приложения, имеющие ссылки на внешние функции, переменные или структуры, включается заголовочный файл, предоставленный разработчиком DLL.
7. Формируется исходный код модулей приложения.
8. Компилятор создаёт объектный файл для каждого модуля приложения.
9. Компоновщик собирает все объектные модули приложения в единый загрузочный EXE файл. Кроме того в EXE файле создаётся раздел импорта, где перечисляются имена всех необходимых DLL модулей на основе прикрепляемого .lib-файла.
10. Загрузчик создаёт виртуальное пространство для нового процесса и проецирует на это пространство исполняемый модуль приложения.
11. Анализируя раздел импорта загрузчик находит требуемые DLL библиотеки и так же проецирует их в адресное пространство процесса. После отображения модуля приложения и всех DLL в адресное пространство, первичный поток считается готовым к исполнению и приложение начинает работать.

Преимущество: загружает всё-всё разом, удобно и сразу.
Недостаток: долгая загрузка из-за того, что все DLL и функции предварительно грузятся.

Порядок неявного связывания

В заголовочном файле необходимо определить экспортируемые функции, то есть функции, которые возможно использовать из внешнего окружения.

Раздел export нужен для компоновки из файлов.
Lib-файл.. В разделе экспорта в алфавитном порядке содержится список всех экспортируемых функций DLL библиотеки.
Каждой функции (переменной или структуре) ставится в соответствие относительный виртуальный адрес (RVA), на основании которого функция будет помещаться в адресное пространство процессов. …внутри DLL библиотеки.
Таким образом, обращение к любой функции DLL библиотеки может осуществляться на основании её имени либо порядкового номера, определённого в разделе экспорта.
Замечание:

Порядковый номер функции DLL библиотеки может быть задан вручную при создании раздела экспорта файла определения.

В исходном модуле EXE файла необходимо определить указание на внешнюю используемую функцию. Для этого подключается заголовочный файл библиотеки #include "MyDLL.h" и объявляется внешняя функция. В отличии, от спецификации библиотеки... В результате компоновщик создаст в конечном EXE модуле раздел импорта, в котором будут перечислены DLLки, необходимые этому модулю, и идентификаторы, на которые есть ссылки из используемых DLL. Далее осуществляется прямой вызов функции так, как если это бы была внутренняя функция, а не внешняя.

Таким образом, формируя DLL, компоновщик создаёт дополнительный lib-файл, содержащий имена экспортируемых элементов. Lib-файл добавляется к проекту клиентской программы при сборке клиентской программы. На основе lib-файла создаётся библиотека импорта, что обеспечивает при запуске приложения в рамках операционной системы загрузку нужных DLL библиотек.

Явное связывание

Определяет связь между требуемым DLL и клиентским приложением в процессе работы программы. Поток приложения загружает DLL в адресное пространство процесса, получает виртуальный адрес функций и вызывает эти функции по полученному адресу.

1. Структуры и переменные создаются
2. Создаётся набор файлов? Содержащий экспортируемые и неэкспортируемые?..
3. Компилятор создаёт объектный модуль для каждого исходного модуля библиотеки
4. Компоновщик формирует загрузочный DLL модуль на основе объектных модулей
5. При наличии экспортируемых функций, компоновщик формирует lib-файл, который в дальнейшем не используется
6. Заголовочный файл с импортируемыми прототипами объявляется в исходном модуле приложения
7. Формируются исходные модули клиентского приложения
8. Для каждого исходного модуля компилятор создаёт собственный объектный модуль
9. На основе объектных модулей, компоновщик создаёт загрузочный EXE файл, при этом элементы библиотеки импорта не используются.
10. Загрузчик операционной системы создаёт виртуальное адресное пространство для исполняемого модуля и начинается выполнение первичного процесса приложения.
11. При необходимости, поток вызывает функцию загрузки DLL библиотеки в адресное пространство процесса, что позволяет использовать любую функцию в рамках библиотеки. То есть в любой момент времени приложение может загрузить любую доступную библиотеку путём вызова специальной функции LoadLibrary. Она позволяет загрузить библиотеку по её имени, при этом поиск библиотеки осуществляется путём последовательного просмотра следующих каталогов:
    a. Каталог, из которого запущено приложение
    b. Текущий каталог
    c. 32-разрядный системный каталог Windows (System32)
    d. Каталог, в котором находится сама операционная система (WINDOWS)
    e. Каталоги, перечисленные в переменной описания среды PATH

    Если файл библиотеки найден, то функция LoadLibrary возвращает идентификатор загруженной библиотеки, в противном случае возвращает значение nul. При многократном вызове LoadLibrary различными приложениями, запущенными в системе, инициализация библиотеки осуществляется несколько раз, при этом

формируется код причины вызова. После того, как библиотека загружена, возможно обращение к любой необходимой функции библиотеки с предварительным получением адреса этой функции посредством функции GetProcAddress, которая возвращает адрес RVA, определённый в разделе экспорта, для функции, заданной по имени либо по номеру (можно обращаться по-разному). В качестве первого параметра в функцию GetProcAddress передаётся идентификатор загруженной библиотеки. Возвращает данная процедура указатель на существующую функцию библиотеки. Если функция не найдена, возвращает нулевое значение.

Раз загружаем динамически, то обязаны её и выгрузить. Для выгрузки библиотеки из виртуального пространства текущего процесса используется функция FreeLibrary, которой в качестве параметра передаётся идентификатор библиотеки.
DLL библиотека состоит из набора экспортируемых, неэкспортируемых и интерфейсных функций. В Win32 основной функций DLL библиотеки является функция DLL Main, которая определяет три параметра:
HINSTANCE, который определяет идентификатор модуля DLL.
Функция DLLMain вызывается всякий раз, когда выполняется инициализация процесса, обращающегося к функциям библиотеки, а так же при явной загрузке или выгрузке библиотеки (Load/Free).
Идентификатор модуля DLL может быть использован для обращения к ресурсам, расположенным в модуле DLL библиотек. Код причины вызова функции может принимать одно из 4 значений:
DLL_PROCESS_ATTACH Данный код выполняется, когда библиотека отображается в адресное пространство процесса в результате его запуска либо при вызове функции LoadLibrary.
DLL_THREAD_ATTACH Данный код формируется, когда текущий процесс создаёт новую задачу, после чего система вызывает функцию DLLmain для всех библиотек, подключённых к данному процессу.
DLL_PROCESS_DETACH при завершении программы
DLL_THREAD_DETACH ?
Это использовать в своей программе для того, чтобы понимать, по какой причине была вызвана функция DLLMain.

Вызов функции LoadLibrary

При вызове функции LoadLibrary осуществляется следующий алгоритм:

1. Текущий процесс проверяет наличие спроецированной DLL в собственном адресном пространстве. Если проекция DLL в пространстве процесса не найдена, передаётся сообщение системе о поиске библиотеки с заданным именем. Если заданная библиотека не найдена, возвращается значение 0, в противном случае, библиотека проецируется в адресное пространство процесса.
2. При обнаружении библиотеки в адресном пространстве процесса увеличивается счётчик использования данной библиотеки
3. Вызывается функция DLLMain со значением process_attach. Если функция DllMain возвращает истинное значение, то библиотека считается загруженной и её идентификатор возвращается в качестве результата работы функции. В противном случае, библиотека считается недоступной, что приводит к декременту счётчика использования, извлечения адресного пространства процесса и возвращения нулевого результата загрузки.

FreeLibrary

Проверяется наличие библиотеки в адресном пространстве процесса.
Если библиотека находится в адресном пространстве происходит уменьшение счётчика использования и если счётчик принимает нулевое значение, осуществляется выгрузка библиотеки с кодом причины dll_process_detach. Затем библиотека отключается от

адресного пространства процесса. Если счётчик использования превышает 0, выгрузка библиотеки не происходит.

Вопрос на лабе: если одну DLLку три приложения юзают.
Когда будет лоад? При запуске первого приложения, или при первом обращении.
Когда будет фри - при последнем фри или закрытии последнего.

Файл определения .DEF

Необходимо выполнять согласование импортируемых и экспортируемых значений функций. Данная операция может быть выполнена путём формирования специализированого DEF файла (файл определений).
Файл определений содержит раздел экспорта, в котором описываются все экспортируемые функции, структуры и DLL библиотеки. Формат описания следующий:
ИмяТочкиВхода *(имя DLL библиотеки, по которому функция может быть доступна)*
[=ВнутрИмя *имя по которому, функция будет использоваться внутри библиотеки*]
[@номер *определяет номер данной функции определяет номер данной функции в библиотеке экспорта; если не задали вручную, будет задан автоматически компоновщиком*] если определён параметр NONAME, то возможно обращение к экспортируемой функции только по её номеру. При определении параметра CONSTANT dll-функция не передаётся для дальнейшего использования; возможно использования только данных из DLL библиотеки. При этом имя точки входа определяет имя экспортируемой глобальной переменной.

3 (она же 4) лаба.
Надо будет создать Windows приложение по нами рассматриваемым правилам.
Реализовать две функции.
Одну функцию реализовать через DLL библиотеку с явной загрузкой. Вторую функцию более простую реализовать в виде модуля на ASM.
Отвечать про явное-неявное связывание, что будет если функция или библиотека не найдена. Каким образом изменится программа, если загрузка будет статической (с точки зрения DLL)
Каким образом определяется идентификатор приложения, какое окно является главным, дочерним, запущен.
Как передаются и возвращаются параметры ASM функции. ….
Будет через неделю методичка %) :)

Лек10-11. Объектно-ориентированное программирование

18 апреля 2012 г.
8:26

<<Лек10. Объектно-ориентированное программирование.wma>>

Запись начата: 8:26 18 апреля 2012 г.

- Структурное программирование
- Функциональное программирование (как только данные на входе функции есть, она срабатывает). Лисп, питон.
- Логическое программирование (задача описывается в виде предложений логики предикатов/высказываний, работает не на основе алгоритма, а на основе вывода). Матлог.
- Автоматное программирование. Всё описывается как автомат, всё описывается как модель формального автомата. Входные сигналы - данные. В зависимости от конкретной задачи, в автоматном программировании может использоваться как

конечный автомат, так и автомат более сложной структуры.
- Объектно-ориентированное. Программа представляет собой взаимодействие активных объектов. Каждый объект принадлежит ранее заявленному типу.
- Событийно-ориентированное. Под событием понимается действие пользователя, сообщение других программ или потоков, команды операционной системы, при этом программы представляют собой набор обработчиков известных событий.
- Агентно-ориентированное программирование (агент следит за всем). Парадигма, при которой основными концепциями являются понятия "агент и его поведение". Агентом является всё, что может воспринимать среду через набор реальных или абстрактных датчиков, а так же воздействовать на эту среду с помощью исполнительных механизмов. Поведение агента зависит от текущего состояния среды. Агенты в системе работают асинхронно, выполняя действия, предписанные их поведению.

Объектно-ориентированное программирование - это метод программирования, основанного на представлении программы в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром определённого класса, а классы являются членами определённой иерархии наследования.

Таким образом, ООП:
- использует в качестве логических конструктивных элементов объекты, а не алгоритмы
- Каждый объект является экземпляром определённого класса
- Классы образуют иерархии по принципу наследования, то есть дочерние классы включают в себя все элементы, присущие родительским, и программа расширяет собственную функциональность.

Программа является объектно-ориентированной только тогда, когда выполняются все условия.

Любая программа может считаться объектно-ориентированной, если присутствуют все три перечисленных признака.
ДЗ посмотреть, что такое VB и сказать, является ли он объектно-ориентированным языком.

Основой объектно-ориентированного программирования является объектная модель. Эта модель определяется следующими основными элементами:
- **Абстракция**. Выделяет существенные характеристики некоторого объекта, отличающего его от всех других видов объектов и чётко описывающая его концептуальные границы с точки зрения наблюдателя. Различают
    a. Процедурную абстракцию. Предполагает, что любой объект состоит из обобщённого множества операций, каждая из которых выполняет однотипные требуемые функции.
    b. Абстракцию данных. Предполагает, что объект представляет собой полезную модель некоторой сущности предметной области.
- **Инкапсуляция**. Это процесс разделения элементов абстракции, определяющий структуру и поведение данного предмета. Инкапсуляция предназначена для изоляции конкретных обстоятельств абстракции от её реализации. За счёт инкапсуляции реализуется интерфейс между внутренним содержанием предмета и его внешней средой. *От нас скрыто внутреннее состояние и реализующее лишь интерфейс.*

- **Модульность** - это свойство системы, разложенной на цельные, но слабо связанные между собой модули.
- **Иерархия** - это ранжирование или упорядочивание абстракции. Реализуется через механизм наследования. Механизм наследования основан на отношении is a (является). Наследование означает такое отношение между абстракциями, когда одна абстракция заимствует структуру и поведение одной (простое наследование) или нескольких (множественное наследование) других абстракций. Таким образом, наследование создаёт иерархию абстракций, в которых подклассы заимствуют свойства одного или нескольких суперклассов. *Каждый следующий подкласс берёт всё, что было в суперклассе.*
- **Контроль типов** - это правило использования объектов, не допускающее или ограничивающее взаимную замену объектов разных типов. При этом объект - это понятие, объединяющее две точки зрения:
    a. Каждый объект может представлять собой отдельный поток управления. Такой объект называют активным.
    b. Любая система может быть представлена как совокупность взаимодействующих объектов, часть из которых является активными, а часть независимыми друг от друга.

  Контроль типов предполагает явное совпадение внутреннего представления каждого объекта при его реальной реализации. Если меняется объект, то нужно переделывать всю реализацию.
- **Параллелизм** - это свойство, отличающее активные объекты от пассивных. Только активные объекты способны выполнять множество действий по отношению к другим элементам системы.
- **Персистентность** - это свойство, позволяющее объекту преодолевать временные рамки - продолжать своё существование после исчезновения создателя этого объекта. *Пример: разрабатываем СУБД, с помощью неё создали хранилище, заполнили хранилище, снесли СУБД, база осталась.*

Но это не три принципа ООП, это определения.

Какие преимущества даёт использование объектной модели?

1. Использование объектного подхода стимулирует повторное применение не только кода, но и проектных решений. (паттерны)
2. Использование объектной модели приводит к созданию системы с устойчивыми промежуточными формами, что упрощает их модификацию. Это позволяет улучшить систему в течение короткого промежутка времени, не прибегая к её полной переработке.
3. Объектная модель уменьшает риски, связанные с проектированием сложных систем, так как позволяет разбить сложную систему на набор слабо зависимых подсистем, "живущих собственной жизнью".
4. Объектная модель учитывает особенности процесса познания, на которое способен человек, то есть от простого к более сложному.

Основными кирпичами объектно-ориентированной парадигмы являются

Объекты и классы

Являются базовыми блоками любой объектно-ориентированной программы. Содержательно, любой объект можно представить как некоторую сущность предметной области, имеющей хорошо определённое поведение.

Объект - это часть окружающей нас реальности, существующей во времени и пространстве, имеющей состояние, поведение и определённую индивидуальность. Состояние характеризуется перечнем всех, как правило, статических свойств данного объекта и текущими, как правило, динамическими, значениями каждого из этих свойств. Свойства - это отличительные характеристики качества или особенностей объекта, обеспечивающие его индивидуальность.
Поведение - это действие объекта, выраженное через изменение его состояния и/или передачу сообщений, при этом сообщения - это операция, которую один объект выполняет над другим.
Поведение это набор функций, определяющих действие системы и определяющее взаимодействие с другими элементами. Поведение объекта зависит от выполняемых над ним операций и от его текущего состояния. Причём, некоторые операции могут косвенно изменять состояние объекта. Все операции, выполняемые над объектами, можно разделить на следующие виды:

1. **Модификатор** - это операция, изменяющая состояние объекта.
2. **Селектор** - это операция, имеющая доступ к состоянию объекта, но не изменяющая его состояние.
3. **Итератор** - операция, обеспечивающая доступ ко всем частям объекта в строго определённом порядке.
4. **Конструктор** - операция, создающая объект и/или инициализирующая его состояние.
5. **Деструктор -** операция, стирающая состояние объекта или уничтожающая сам объект.

В примере с лифтом все эти виды применимы. Лифт объект, мы объект.
Индивидуальность - свойство, позволяющая однозначно отличить данный объект ото всех других объектов данного типа. Это динамические значения. Определяется идентификатором.

Класс - это множество объектов, имеющих общую структуру и общее поведение. Реализация одного элемента класса является его экземпляром или объектом.

С точки зрения реализации на C++, основой для создания классов является тип **struct**. Но нарушается инкапсуляция. Чтобы это не нарушать, введён тип Class. По-умолчанию всё внутреннее содержание класса считается скрытым внутри структуры и доступ к элементам класса можно получить только через открытые интерфейсы.

Что есть в class?

Private - то, что доступно только внутри этого класса
Protected - то, что доступно этому объекту и его производным
Public - доступно всем элементам программы, в которой этот объект существует.

Внутри класса применяются ограничения доступа к элементам. По умолчанию элементы класса имеют доступ **private**, то есть данные и функции доступны только через открытые функции данного класса (то есть доступны только элементам данного класса).
Public - все член-данные и член-функции, описанные в этом разделе, являются глобальными и доступны из любого места системы, где имеется представитель данного класса.
Protected - данные функции доступны только внутренним операциям класса и операциям его прямых наследников.

Прародителем классового типа является структурный тип **struct**, который так же может содержать набор данных и операций, но в отличии от типа class все элементы структуры являются общедоступными

или публичными. Таким образом в структуре отсутствует принцип инкапсуляции (вспоминаем, что инкапсуляция используется для сокрытия реализации от внешнего использования, а не только для объединения классов и данных в один контейнер)

Замечание: считается правилом хорошего тона (а всё остальное это плохо) объявлять все данные класса как приватные. (иначе смысла нет, если наши данные может модифицировать кто попало в любой момент). В лабах это надо не забывать.

Такое построение класса обеспечивает:

1. При необходимости корректировки данных в определении класса, изменения производят только в определённой член-функции.
2. Обеспечивается защита внутренних данных (член-данных) объекта от несанкционированного доступа и изменения.

Что происходит, когда мы объявили тип? При объявлении в рамках системы некоторого класса вводится новый абстрактный тип данных. При объявлении переменной данного типа (или в некоторых языках при принудительном вызове конструктора) выделяется память под структуру, соответствующую описанию класса. При этом возможна инициализация некоторых динамических значений состояния класса. Выделение памяти осуществляется только под данные создаваемого объекта, но их нужно инициализировать вместе с объектом. *Почему? Потому что функция перенести стол одинаковая для всех столов и достаточно иметь ссылку на таблицу функций.*
В паскале мы должны были вызвать конструктор Create и только при вызове этого метода переменная инициализировалась и под неё выделялась память. Здесь это не требуется.
При вызове объекта выделяется память на построение этого объекта на основе заданного типа. При построении напрямую или последовательно вызывается операция-конструктор. В C++ операция-конструктор имеет имя, совпадающее с именем класса.
Деструктор имеет такое же имя, но начинается с тильды, деинициализирует данные.

Конструктор может быть определён таким образом, чтобы присваивать по умолчанию значения основным данным объекта. При этом объявление переменной класса однозначно приводит к её инициализации.

Основные свойства и правила конструкторов:

1. Конструктор имеет то же имя, что и класс, в котором он объявляется.
2. Конструктор не возвращает значения (даже типа void)
3. Конструктор не наследуется в производных классах. Если необходимо, конструктор производного класса может вызвать конструктор базового класса. Пояснение: не надо думать, что конструктор это такая функция, которая не подлежит принципу наследования. Нет, речь о том, что конструктор для каждого класса свой. Можем вызвать внутри дочернего конструктора родительский конструктор, но не всегда так делается. В любом случае, делается это вручную.
4. Конструктор может иметь параметры, заданные по умолчанию.
5. Конструктор является функцией, но он не может быть объявлен как виртуальный (чуть позже разберём, что это такое).
6. Невозможно получить в программе указатель на конструктор (его адрес). Это не традиционный вызов функции, тут нет даже void.
7. Если конструктор не задан, то он генерируется системой автоматически на основании структуры описанного объекта. Выполняет такую генерацию компилятор при первом проходе. Все конструкторы, сгенерированные компилятором имеют

атрибут доступа public.

8. Конструктор всегда вызывается автоматически при описании объекта.
9. Объект, содержащий конструктор, нельзя включить в качестве элемента в тип Union (потому что union по-особому выделяет память, надо знать размер). А вот компилятор объём объекта знает и может поставить в union.
10. Конструктор класса X не может иметь параметр типа X, но может иметь параметр - ссылку на объект типа X. Такой конструктор называется конструктором копирования. При вызове конструктора копирования не производится выделение памяти под структуру объекта, а создаётся лишь копия ссылки на него (по сути будем иметь два имени одного и того же объекта)

Основные свойства и правила использования деструкторов

Деструктор - специфическая функция объекта, позволяющая разрушать объект, то есть правильно освобождать память, выделенную под объект при её создании. Основные свойства и правила её использования.

1. Начинается с ~.
2. Деструктор не возвращает никакого значения, ему это и не надо.
3. Деструктор не наследуется в производных классах, а при необходимости может быть вызван вручную.
4. Деструктор не имеет параметров (освобождает всю память, выделенную под объект).
5. Каждый класс может иметь только один деструктор (в отличии от конструкторов, их у класса может быть n)
6. Деструктор - функция, которая может быть (должна быть) виртуальной, в программе нельзя получить адрес деструктора.
7. Если деструктор не задан в программе, то он будет автоматически сгенерирован компилятором. Деструктор не наследуется!
8. Деструктор можно запустить как обычную функцию.
   date *my_day;
   my_day->date::date:~my_day;
9. Деструктор вызвается автоматически при разрушении объекта.
   То есть при использовании Delete.

Области видимости классов

...
только если описаны внутри раздела Public. Внутри класса область видимости не зависит от точки объявления член-класса. Любой член класса виден внутри всего класса.
int x=2;
class Example
{
void f() {x=0.....

Области видимости классов могут быть изменены. Иногда необходимо, чтобы все объекты класса имели доступ к некоторой общей переменной, а не к её копи. Такие член-данные следует объявлять внутри класса со спецификатором **Static**, благодаря которому данные будут храниться в заданной единственной ячейке памяти и любая их модификация будет иметь последствия для всех объектов данного класса.
class Example

```
{ public
static int x;
} p1, p2;
int Example::x==67;
```
Изменение производят через полное обращение к элементу класса.
Функции, которым не разрешается изменение, а только чтение, должны быть объявлены с идентификатором const.
```
const Stack s1;
Stack s2;
```
Константным может быть объявлен и объект в целом. К константному объекту могут быть применены только константные член-функции.
Любая попытка написать изменение в программе константного класа вызовет ошибку при компиляции.

Указатель this

В некоторых функциях необходимо выполнить обращение к компонентам объекта без указания имени объекта.

Внутри каждой функции создаётся указатель на неявный объект класса. Когда объект сздаётся, под него выделяется память, в котором имеется инициализированное поле, куда помещается указатель, в котором скрытый байт, связанный со значением объекта. Получить значение скрытого указателя можно получить через спецификатор this.
```
date today, my_day;
today.out();
my_day.out();
...
```

Неявный указатель this указывает на начало памяти, где размещён конкретный объект. Тогда через обращение к методу через имя объекта, происходит подстановка на место неявного указателя адрса размещения именно этого обхекта.

Основные свойства и правила указателя this

1) каждый созданный объетполучает встроенный указатель this
2) this указывает на начало своего объекта в памяти
3) this не требует дополнительного объявления
4) this перадаётся как скрытый параметр и не можетбыть обявлена как static (потому что статик влияет на всех)
5) this является локальной и недоступна за пределами объекта

08.05.2013
Методы, объявленные как static не могут быть виртуальными.
Для кажого объекта копируется ссылка на обхект памяти.
Правила описания и использования виртуальных функцй:
1. Виртуальная функция может быть только методом класса
Любую перегружаемую операцию-метод класса можно сделать виртуальной
Витуальная функция, как и сама виртуальность, наследуется
Виртуальная фцнкуия может быть константной

Если в базовом класссе впервые бъявлена виртуальная то функция должна быть либо чистой (virtual int f(void)=0;), либо для неё должно быть задано определение.
Если в базовом классе определена виртуальная функция, то метод производного класса с таким же именем и рпототипом автоматически является виртуальными.
Конструкторы не могтут быть виртуальными
СТатистические методы не могут быть виртуальными)
Десттрукторы (чаще должны) быть
Если некоторая функция вызывается с использованием её полного имени, то виртуальный механизм игнорируется.

Полиформизм бывает двух типов:
а) статический полиморфизм - времени компиляции
б) динамический полиморфизм - времени исполнения
Отличаются объектом вызываемого метода. В случае статического метода привяка объекта метода осуществляется на этапе компиляции программы. Статический полиморфизм реализуется через механизм перегрузки и шаблонов. При динамическом полиморфизме привязка метода к объекту осуществляется на этапе выполнения программы, что связано с изъятием адреса используемого метода и с таблицей методов данного объекта. Динамический полиморфизм реализуется через механизм виртуальных функций. Класс, включающий виртуальную функцию, называется полиморфным. Если в классе присутствует чистая виртуальна функция, то такой класс называется абстрактным. Нельзя создать объект абстрактного класса (поттому что в абастрактном классе есть чистая функция). Абстрактные классы используются только для дальнейшего наследования.
Необходимость введения виртуальнх функций определяется следющими обстоятельствами:

1. вся тяжесть по определению типов объектов ложится на программиста, а не на компилятор
2. программа, написанная в статитическом стиле привязана к деталям ииерархии приложения. При изменении иераррхии, необходимо менять текст приложения.
3. При изменении программы сторонним разработчиком в случае статического программирования требуется знание подроббностей иерерахии наследования.

Вызов виртуальной функци может являться не виртуальным в следующих случаях:
1. Если осуществляется не через указатель или ссылку, при этом вызов становится статическим
global object;
object.f();
2. Вызывается через указатель или ссылку, но с уточнением имени калсса:
base * p;

glbal object;
p=&object;
p->f(); виртуальный вызов
p->global::f(); не виртуальный вызов
3. Вызывается в конструкторе или деструкторе базового класса (потому что будет попытка вызова метода удалённого класса).

Задание на 4 лабораторную работу

Разработать графическое приложение, демонстрирующее принципы инкапсуляции, наследования, полиморфизма при наличии следующей иерархии объектов: приложение должно работать с базовым классом и как минимум с двумя классами-наследниками. Каждый класс должен иметь два собственных член-данных. Каждый класс должен иметь два собственных, два перегруженных и наследованных член-функции.
Выполнить свою лабу тавта третьего семестра на си и с использованием объектной парадигмы.

На следующей лекции контрольная работа.
А пока следующая тема.

Немного о неудачах контрольной.
Стек-фрейм образуеся при вызове подпрограммы. Нельзя использовать SP, так как это счётчик. Чтение из стека соответствует удалению. В зависимости от согласования либо вызывающая, либо вызываемая программа восстановит стек.
Если вызываемая, та, которая только что работала, то она должна всё удалить за собой.
Конструктор с параметрами по-умолчанию в классе - конструктор, в котором имеется набор параметров, явно зполняющий член-данные класса. При описани такого конструктора (не путать с объявлением) должно содержать списк обращений, то есть, какой параметр чему будет сопоставлен.

Динамический полиморфизм - возможность использования однимённых функций в разных уровнях иерархии. Причём при вызове функции через переменную определённого класса вызывается функция именно этому классу соответствующая.
Перегруженные функции имеют разное количество параметров и разную сигнатуру, а перекрытые - одинаковую (выбираются по тиму объектов).
Не всегда могут виртуальные функции вызываться виртуальными.
Косвенная адресация в ASM..
Если влезает в 4 байта, то выводим данные через EAX, если 8 байт то EAX+EDX.

Дружественные функци

Лек13-14-15 Многопоточное программирование

16 мая 2012 г.
8:26

<<Лек13 Многопоточное программирование.wma>>

Запись начата: 8:26 16 мая 2012 г.

Мария Львовна рассказывает про кружки.
<<Объектно-ориентированное программирование.ppt>>
<<Многопоточное программирование.pptx>>

Сейчас пока рассматриваем многопоточное программирование без многопроцессорного.

Многопоточное программирование усложняет работу приложения из-за взаимодействия потоков.

Пока говорим о многопоточных программах, взаимодействии потоков. Это не параллельное программирование.
В любой многозадачной системе используется концепция процесса.

Процесс (задача) - есть абстракция, описывающая выполняющуюся программу. Для операционной системы процесс представляет собой единицу работы и заявку на потребление системных ресурсов. При этом архитектура операционной системы должна удовлетворять следующим требованиям:

1. Чередование выполнения нескольких процессов с целью повышения эффективностью использования процессора и обеспечения разумного времени отклика.
2. Распределение ресурсов между процессами в соответствии с заданной стратегией на основе приоритетов, избегая взаимоблокировки.
3. Поддержка обмена информацией между процессами, а так же обеспечение возможности создания новых пользовательских процессов.

Основная работа операционной системы связана с управлением процессами. В любой момент времени каждый процесс может находиться в одном из состояний:

- Готовность - процесс находится в активном ожидании запуска либо доступа к ресурсам;
- Выполнение - процессу выделяется квант времени для выполнения алгоритма его работы;
- Блокировка - процесс останавливает своё выполнение в связи с невозможностью получения доступа к требуемым ресурсам либо захватом процессорного времени другим процессом.

Операционная система отслеживает состояние процессов и управляет их изменением.
В многопоточных системах традиционная концепция процесса разделена на две части
1. Связана с принадлежностью ресурсу и называется процессом
2. Связана с выполнением машинных команды и называется потоком.

Потоки возникли в операционных системах как средство распараллеливания вычислений. В литературе часто утверждается, что использование нескольких потоков увеличивает производительность приложения. Данное утверждение является не полносью корректным, так как в рамках работы приложения операционная система управляет управляет имеющимися процессами и распределяет процессорное рвремя между ними. Кроме того, операционная система занимается созданием и уничтожением процессов, обеспечивает систему необходимыми ресурсами,поддерживает межпроцессное взаимодействие. Для этой работы в паямти поддерживаются специальные информационные структуры, в которых указывается, какие конкретные ресурсы выделены процессу для исполльзования. Ресурсы могут быть назначены процессу в монопольное либо совместное исползование. Часть ресурсов выделяется динамически по запросу процесса во время его выполнения. Ресурсы могут быть приписаны процесс на всё время его жизни, либо только на определённый период. В операционных системах В операционных системах, где существует понятие процесс и поток, процесс рассматривается как заявка на потреблениевсех видов ресурсов, кроме процессорного времени. Процесс является контейнером ресурсов для потока. ПРоцессорное время потребляет поток. Процессорное время распределяется между потоками. Каждый поок распределяет между собой некоторую последовательность команд.
Процесс включает в себя следующие элементы:
1) Закрытое виртуальное адресное пространство, то есть диапазон адресов, которые может использовать данный процесс.
2) Исполняемую программу, то есть начальный код и данные, проецируемые в виртуальное адресное пространство процесса.
3) Список открытых описателей различных системных ресурсов (семафоров, коммутационных портов, файлов и других объектов, доступных всем потока данного процесса.
4) Контекст защиты, который называют маркером доступа. А так же идентификатор пользователя, определяющий права получения доступа к данному процессу и его ресурсам.
5) Кажды

пропуск

Идентифицирующий пользователя, группу безопасности и привилегии, сопоставленные с процессом. Каждый процесс при своём запуске создаёт как минимум один поток.

Поток (или нить) - это объект ядра, являющийся сущностью внутри процесса, которая получает процессорное время для выполнения. Программа процесса может выполняться только в рамках потока.

Поток включает в себя следующий набор элементов:

1. Содержимое набора регистров процессора, отражающее состояние процессора
2. Два стека, один из которых используется потоком при выполнении в режиме ядра, а второй в пользовательском режиме.
3. Закрытая область памяти, образуемая библиотеками исполняющей системы и DLLками.
4. Уникальный идентификатор потока. Генерируется из того же пространства имён, что и идентификатор процесса, породившего данный поток.

Дополнительно каждый поток (так же как и процесс) может иметь маркер доступа. Чаще всего этот маркер наследуется из атрибута безопасности наследуемого процесса, но может быть определён собственный атрибут безопасности.

При создании потока начинает работать специальная функция, называемая потоковой функцией.
Хорошим тоном является разработка потоковой функции, возвращающей некоторое значение, так как это значение может быть получено в рамках верхнего потока или системы и может использоваться для контроля за выполнением потока.

Каждый поток начинается с некоторой функции - функци потока. Функция потока может выполнять любые действия и имеет единственный входной параметр, определяющий ссылку на список значений, инициализирующих процесс при её создании. Может содержать любые операции, связанные с выполнением работы потока. По окончании фукции потока поток автоматически завершается.

Как только завершается функция потока, поток автоматически считается завершённым. В момент завершения потока система выполняет следующие действия:
1. Физически останавливает поток (лишает его кванта времени);
2. Освобождает стек, занятый текущим потоком, в данном случае освобождается стек, связанный с ядром; стек контекста пользователя освобождается автоматически при завершении функции потока;
3. Уменьшает на 1 счётчик пользователя для объекта ядра потока. Когда счётчик объекта ядра обнуляется, система его удаляет. Таким образом, объект ядра может жить дольше, чем поток с ним связанный.

Такая конструкция позволяет получить информацию о статусе завершения потока вне потока: основному потоку либо внешней программе.
Функция потока всегда должна возвращать значение (не может быть void).

При разработке потоковой функции желательно использовать только локальные переменные либо параметры для организации связи с внешним миром. При использовании глобальных переменных и вызове статических методов класса (обращения к в нешним участкам памяти) требуется обеспечивать корректность обращений через механизм синхронизации потоков. В частности, механзма синхронизации с целью обеспечения корректного доступа к общим данным.

Для создания потока используется специализированная функция CreateThread.
Данная функция создаёт новый поток в процессе и возвращает идентификатор потока в случае успешного создания.
В качестве параметров:
1. psa - Указатель на структуру атрибутов безопасности, тот самый маркер доступа. Чаще всего определяется как nul, при этом атрибут безопасности назначается по-умолчанию, наследуясь от атрибутов процесса.
2. cbStack - Размер стека потока. Если указано нулевое зй параметр создания начение, то размер определяется системой по умолчанию. При задании ненулевого значения система выберет большее значение между заданным и дефаултным.
3. pfnStartAddr - Указатель на потоковую функцию (тело потока).
4. pvParam - Указатель на список параметров, передаваемых в потоковую функцию. Может иметь произвольное значение и передаётся в функцию потока. Обычно в качестве параметра передаются либо числовые параметры, либо указатель … потока.

5. tdwCreate - Дополнительный параметр, может принимать два значения: create_suspended либо nul. Если параметр имеет нулевое значение, выполнение потока начинается немедленно, в противном случае по возможности. В случае задержки система выполняет инициализацию потока, но н запускает его. При этом непосредственно запуск потока определяется специальной системной функцией.
6. pdwThreadID - указатель на переменную, которая на выходе будет содержать идентификатор потока. Начиная с Win2000 допустимо использование значения null. Идентификатор потока обязательно указывается, если он требуется для использования вне процесса. В случе невозможности создания потока получается нулевое значение. Идентификатор может быть назначен вручную и может быть использован для обращения к потоку (это удобно при наличии несколькоих потоков с одинаковой потоковой функцией, задать вручную, и как константу использовать в программе).

Вызов функции создания потока (CreateThread) создаёт объект .. Потока и возвращает его дескриптор. Система выделяет память под стек нового потока из адресного пространства процесса, инициализирует структуры данных потока и передаёт управление потоковой функции.
Каждый новый создаваемый поток выполняется в контексте того же процесса, что и родительский поток, следовательно имеет доступ к дескриптору процесса и адресному пространству процесса, что позволяет обеспечить межпотоковое взаимодействие.

Поток может завершаться в следующих случаях:
1. При самоуничтожении с помощью вызова метода **ExitThread**. Такое завершение допустимо, но не рекомендуется, так как может приводить к нарушению синхронизации.
2. Завершение функции потока, когда функция потока возвращает управление основному потоку. Рекомендуемый стандартный способ, обеспечивающий корректность работы многопоточных приложений.
3. Один из потоков данного или стороннего процесса вызывает функцию **TerminateThread** (тоже нежелательный способ, может привести к нарушению синхронизацию). Если это сделала система или другое приложение, то как правило ресурс всё-таки будет отдан в общее пользование обратно, если был зхвачен.
4. Завершается процесс, содержащий поток, убиваются все, но если было общение с внешними потоками, то это может вызывать утечки памяти.

Функцию потока следует проектировать так, чтобы поток завершался только после того, как она возвращает управление. При этом:
1. Любые объекты, созданные данным потоком, уничтожаются соответствующими деструкторами;
2. Система корректно освобождает память, которую занимал стек потока;
3. Система устанавливает код завершения данного потока (поддерживаемый объектом ядра "поток");

ExitThread выполняет те же самые действия кроме выставления кода завершения.

Совместимость библиотеки потоков и стандартной библиотеки CLR.
В VS есть две библиотеки для работы с потоками:
Первая на WinAPI, вторая поддерживаемая стандартом Си.
Для создания потока используем функцию BeginThread (start_address, stack_size, *arglist);
Для корректного завершения потока может использоваться функция EndThread (обычно эта функция используется внутри потока).

Отличия между BeginThread и CreateThread:

1. Обе инициализируют.
2. CreateThread позволяет устанавливать атрибуты безопасности, то есть если мы используем BeginThread, то поток запускается сразу, а при CT можем запустить отложено.

Синхронизация потоков. Пример 1.

При работе в многозадачных и многопоточных системах состоит в организации процесса синхронизации потоков. Синхронизация - это работа по организации корректного бесконфликтного доступа к общим ресурсам системы, при этом общмими ресурсами системы могут я вляться устройства, глобальные переменные, общие участки памяти. Никогда нельзя быть уверенным в том, в какой момент времени какой поток выполняет то или иное действие. В этой связи необходимо предпринимать меры по корректному обращению к общим ресурсам либо обработке общих данных, в противном случае процесс работы программы может быть непредвиденным.
Синхронизация это рабоа, которая позволяет остановить поток, пока другой поток работает с ресурсом.

Как можно? События, блокирующие переменные, критические (неделимые) секции.
Уже пару минут рассказывает о действии процедуры.
Суть в уходе потока в состояние сна, если не положено ему, чтобы отдать процессор другому потоку. Следит за состоянием глобальных переменных.

Как ещё кроме sleep можно приостановить работу потока:
Sleep(dwMilliseconds)
Может быть = 0 (поток откажется от выделенного кванта времени и тут же встанет в очередь на процессорное время).
WaitForSingleObject(hHandle,dwMilliseconds) функция ожидания одиночного объекта
Либо истечёт время ожидания, либо ожидаемый объект перейдёт в сигнальное состояние - состояние при котором объект может быть вновь захвачен. Опрос производится каждое заданное количестко милисекунд.
WaitForMultipleObjects(nCount, *lpHandles, bWaitAll, dw,Milliseconds) функция ожидания множественного объекта. Позволяет ожидать освобождение набора сигнальных объектов. В качестве параметра используется количество элементов множественного объекта.
2)*lpHandles список дескрипторов ожидаемого объекта. bWaitAll (дожидаться всех объектов или только одного); dwMilliseconds.
dwMilliseconds={o.N.INFINITE} процесс ожидания может быть бесконечным.

Пример 2

Программа создаёт два одинаковых потока и ожидает их завершения. Потоки выводят текстовые сообщения, передаваемые им в качестве параметров в момент инициализации. В процессе присутствуют два потока, основной Main и созданный с помощью библиотеки потоков CRT.
Рассказывает минуту-две.
Главный поток (процесс??) не завершится, пока не завершатся потоки.
Возможность приостановки потока не является синхронизацией.

<<Лек13 Многопоточное программирование - 2.wma>>

Запись начата: 9:31 16 мая 2012 г.
В каком порядке выведутся сообщения? А хз. Почему? Потому что синхронизирован главный поток с дочерними. А дочерние между собой борются.

Критическая секция выполняется без передачаи упрвления между потоками.

В обычной ситуации каждый поток

Объекты синхронизации:

Mutex - объект ядра, который может находиться в двух состояниях: занятом и свободном. Если mutex занят одним потоком, он не может быть захвачен другим потоком.
Кретиническая секция (critical section) - создание внутри потоков и функций неделимого блока команд, требующих выполнения до передачи управления другому потоку.
Событие (Event) - объект ядра, способный посылать специфические сигналы потокам, идентифицирующие возможность захвата либо освобождения того или иного ресурса.
Семафор (Semaphore) - объект ядра, обеспечивающий взаимоисключение, но в отличии от мьютекса имеющий счётчик, изменяемый при захвате или освобождении ресурсов.

Мьютексы

23.05.2012 12:04
<<Лек13 Многопоточное программирование - 2.wma>>

Запись начата: 12:04 23 мая 2012 г.

Мьютекс - это объект ядра, который создатся с помощью функции CreateMutex, при этом мьютекс может находиться в одном из двух состояний: занятом либо свободном. Обычно мьютексы используются для защиты единичных ресурсов от одновременного обращения к ним из разных потоков. В любой момент времени только один поток может владеть мьютексом. Чтобы исключать запись двух потоков, например, в общий участок памяти одновременно, для каждого потока создаётся блок ожидания, в котором поток проверяет, возможность захвата мьютекса с последующим доступом к общему ресурсу. Если поток определяет, что мьютекс в текущий момент захвачен, он переходит в режим ожидания и устанавливается в очередь ждущих процессов. Как только мьютекс освобождается, первый процесс из очереди ожидающих, может его вновь захватить. Два или более потока могут создать мьютекс с одним и тем же именем, при этом мьютекс будет создан лишь один раз, а все следующие потоки получат идентификатор уже существующего объекта. Это позволяет нескольким потокам обращаться к одному и тому же мьютексу и освобождать программиста от необходимости знать, кто именно создал мьютекс. Если используется данный подход, то атрибут инициализации владельца bInitialOwner необходимо устанавливать в значение FALSE, в противном случае могут возникнуть трудности по определению создателя мьютекса. Для освобождения мьютекса используется функция ReleaseMutex(hMutex), которая в качестве параметра получает идентификатор освобождаемого объекта. Режим ожидания изменения мьютекса обычно организуется с помощью активного ожидания изменения состояния объекта (waitForSimpleObject, например), реже используется множественное ожидание (так как мьютекс используется для контроля доступа к единичному ресурсу).

При своей работе основной поток создаёт мьютекс, идентификатор которого записывается в hMutex. Чтобы был доступен нескольким владельцам, устанавливается идентификатор False. Внутри основного потока устанавливается бесконечный цикл...)

В случае если синхронизация будет нестабильной функция Print будет выводить значения масива в непредвиденно-произвольном порядке.
Мьютекс либо захыватывается, либо отпускается. Определить, к какому обращаемся мьютексу, можно по идентификатору мьютекса.
Можно синхронизировать доступ к ресурсу множества процессов.

Критическая секция / неделимый блок / синхронизация пользовательского режима.

Критическая секция обеспечивает синхронизацию подобно мьютексу, за исключением того, что объекты, предоставляющие критические секции, доступны только в пределах одного процесса.
Критическая секция может использоваться только одним потоком в текущий момент времени, что позволяет использовать критические секции при организации разграничения доступа к общим ресурсам. При использовании критических секций гарантируется, что ресурс будет удерживаться текущим потоком от начала текущей секции до её окончания независимо от режима переключения между потоками. Передачи управления не происходит ни в каком случае, кроме системного сбоя, например.
При вкходе в критическую секцию проверяются флаги и если выясняется, что критическая сессия с данным именем уже занята другим потоком, текущий поток переводится состояние ожидания. Критическая секция характеризуется тем, что для проверки занятости программа не переходит в режим ядра, что не накладывает дополнительных задач на операционную систему. Не прыгает в режим ядра, находится в пользовательском. При обычном же переключении проц прыгает в режим ядра и в пользовательский.

Критическая секция считается более быстрой по сравнению с мьютексом, так как для проверки, занята критическая секция или нет, потоку не требуется переходить в режим ядра, проверяются лишь флаги занятости секции.

Счатается, что критическая секция является более быстрой.
Критическая секция имеет идентификатор, который может быть доступен всем потокам, если о создании критической секци объявлено до создания потоков. В противном случае, критическая секция является монопольной для потока без переключения контекста. Инициализация выполняется с помощью InitalizeCriticalSection. Для захвата или открытия критических секций используется функция EnterCriticalSection. После этого начинается неделимый код, который завершается LeaveCriticalSection.

Чем отличается от мьютекса? И там, и там мы синхронизируем доступ к одному ресурсу. С мьютексом проверяется в режиме ядра наличие свободного или занятого мьютекса, то здесь просто неделимый блок и переключения контекстов не происходит.

События.

Объект синхронизации, состояние которого может быть установлено в сигнальное путём вызова функции SetEvent. Событие имеет два состояния: установленное и сброшенное. Существует два типа событий:

События с ручным сбросом

Это объект, сигнальное состояние которого которого сохраняется до ручного сброса с помощью функци ReversEvent. Как только состояние объекта установлено в синальное, все находящиеся в цикле ожидания этого объекта потоки, получают данное событие и продолжают своё выполнение.
Событие может передаться сразу нескольким потоком. Последовательно, но теоретически это возможно.

События с автоматическим сбросом

Это объект, сигнальное состояние которого сохраняется до тех пор, пока не будет освобождён единственный поток, после чего система автоматически устанавливает несигнальное состояние события. Если не существует потоков, ожидающих данного собятия, объект останется в сигнальном состоянии.
Предположим, что сразу несколько потоков ожидают одного и того же события. Событие

сработало. В случае, если это событие с автосбросом, оно позволит работать только одному потоку, а остальные потоки останутся в режиме ожидания. Если же событие с ручным сбросом, то все потоки будут освобождены (имеют возможность получить управление), а событие останется в установленном сигнальном состоянии, пока какой-нибудь из потоков не выполнит функцию ReversEvent.
Событие удобно использовать при асинхронных операциях. Один поток может использовать при своей работе множество событий. Поток может самостоятельно создавать объект-событие создаётся путём вызова функции CreateEvent. Данная функция создаёт объект события и возвращает его идентификатор. Создающее событие устанавливает его в начальное состояние. Если не связанные потоки вызывают операцию создания события, используя при этом общий идентификатор, то лишь один является создателем события, а остальные получают его копию.
В качестве параметра к которой получает:

1. ИмяСобытия (обычно используется, если собираемся использовать событие вне данного процесса)
2. Тип события, если установлено false то автоматическое, true ручное
3. Состояние события (true сигнальное, false несигнальное)
4. Дескриптор безопасности, обычно наследуется от потока, в котором создан данный объект.

Уже созданное событие может быть переинициализировано, при этом к ним осуществляется доступ с помощью функции OpenEvent. Для установки события может использоваться SetEvent и PulseEvent событие устанавливается в сигнальное состояние, а затем сбрасывается в несигнальное ыпосле освобождения заданного количества потоков. В случае объекта с автоматическим сбросом, освобождается только единственный поток. В случае с ручным сбросом - все объекты, ожидающие в очереди.

Событие - объект, который устанавливается не в занятое состояние, а в сигнальное состояние, и он может быть доступен сразу нескольким потокам, они могут ждать его сигнального состояния.
До освобождения его потоком... а как освободить? Перевести в сигнальное состояние...
Параметры CreateEvent(Имя события, Автоматическое событие, Сигнальное либо несигнальное состояние, Дескриптор безопасности, разрешающий доступ от родительского объекта или не разрешающий)

В следующий раз закончим с семафорами,
А в остальном следующая лекция - прощённое воскресенье, будет переписывание КР, сдача долгов и прочее.
Зачёт получает тот, у кого лабы сданы без штрафов, все контрольные написаны.

30.05.2012 8:30

Семафоры

Семафоры - объект синхронизации, позволяющий обеспечивать взаимоисключающий доступ к критическому ресурсу. По сути семафор это переменная специального типа, доступная параллельным потокам для проведения над ней двух операций: закрытия и открытия.
Закрытым, если счётчик имеет нулевое значение.
Обычно используется для огрничения доступа к некоторому общему ресурсу в возможностью получения доступа к этому ресурсу n потоков.

Для работы с семафором существует два примитива:

1. Примитив установки
2. Примитив ожидания.

При использовании примитива ожидания поток приостанавливается до тех пор, пока не будет получен сигнал разрешения доступа к ресурсу. В отличии от прочих объектов синхронизации, семафор имеет счётчик, который изменяется при закрытии и открытии семафора. В случае выполнения примитива ожидания уменьшается значение счётчика семафора на единицу, если это возможно. При этом, если значение счётчика становится отрицательным, процесс или поток переходит в состояние ожидания и блокируется. Если значение счётчика семафора положительные, поток продолжает работу. При выполнении примитива устновки, если ни один поток не ждёт данного семафора, увеличивается на единицу значение счётчика семафора. Если есть поток, ожидающий в очереди, освобождается из очереди один поток и продолжается его работа с инструкцией, следующей за вершиной ожидания.
Операция увеличения счётчика семафора является неделимым действием (выборка семафора, изменение и сохранения значений не могут быть прерваны). При выполнении операций доступ к семафору запрещён.
Семафор создаётся с помощью функции CreateSemaphore, где (аттрибут_безопасности, начальное значение в которое устанавливается счётчик семафора, максимальное значение счётчика семафора, имя семафора). Если не предполагается использование семафора вне **процесса** имя обычно присваивают NULL.
Для добавления элемента в очередь используется функция ReleaseSemaphore, эта функция является функцией установки.
Для ожидания устанавливается функция при остановке процесса, в данном случае это WaitForSingleObject.

Захватываем - говорим минус один, освобождаем - говорим плюс один.

Для того, чтобы захватить семафор, нужно поставить WaitForSignalObject. Происходит проверка на нулевое значение. Если значение >0 (положительное), то разрешается доступ к ресурсу. В противном случае поток остаётся в состоянии ожидания.
Для того, чтобы освободить семафор используется операция ReleaseSemaphore, при этом происходит инкремент указанного счётчика семафора и освобождение критического ресурса. Использование заданных значений при инкременте и при создании семафора позволяют регулировать количество потоков, которым позволено зайти за семафор.

Установка значения счётчика семафора и установка значения инкремента после выхода позволяет регулировать, сколько одновременных.
В мьютексе и критической секции всегда один поток запускается.
От события чем отличается: если ручное, то тоже может обращаться несколько потоков, но при этом они должны обращаться к нескольким критическим ресурсам. А семафор осуществляет доступ к критическому ресурсу, не пускает за себя.
Обычно семафор создаётся внутри главного потока, но так же как и у мьютекса происходит копирование названия.

Лекции закончены.

Контрольная работа:

Можно было набрать 45.
Зачесть всем, кто получил более 11 баллов.
У меня 10 баллов, зачтено.)))))))
Сказала: "я подумаю"
Зачёт тем, у кого зачтены все контрольные и все три лабы. Это >300 баллов рейтинга.

У нас борзов, каткова, попов, пушкарёв, речкин, шевяков получают автозачёт.
В 21 группе пляскин.

В понедельник в 10 часов будет зачёт по ТП.

Лек14. Разбор контрольной.

23 мая 2012 г.
11:36

Разбор контрольной

<<Лек14. .wma>>

Запись начата: 11:36 23 мая 2012 г.

Как запускается windows-приложение.

Для чего нужны виртуальные функции?
Если не было слова virtual, а просто вызвать m
Ссылка на метод располагается в таблице статических методов, напрямую произойдёт замена вызова адресом метода, и независимо, вызываем мы метод result для объекта класса money или для объекта класса base_money, доход будет расчитываться так как в классе родителей.
Если функции виртуальные, то записываются в таблциу виртуальны хметодов. И вместо вызова Доход будет установлена ссылка на таблицу виртуальных методов. И для объектов разных классов будет подставляться разная ссылка.
Виртуальные функции нужны для того, чтобы вызов разных методов происходил в зависимости от типа класса данных. Для вызова изнутри статических.

Асм. Возврат 4 байтовых происходит через EAX, возврат 8 байт через EAX+EDX, 2 байта через AH ну и так далее.