Технологии программирования (первый семестр)

Парадигмы программирования

- Процедурное программирование – реши, какие требуются процедуры; используй наилучшие доступные алгоритмы.
- Модульное программирование – реши, какие требуются модули; разбей программу так, чтобы скрыть данные в модулях.
- Использование «пользовательских» типов – реши, какие требуются типы; обеспечь полный набор операций для каждого типа.
- Объектно-ориентированное программирование- реши, какие требуются классы; обеспечь полный набор операций для каждого класса; явно вырази общность через наследование.
- Обобщенное программирование – реши, какие требуются алгоритмы; параметризуй их так, чтобы они могли работать со множеством подходящих типов и структур данных.

Объектно-ориентированное программирование. Структура класса.

- **Структура класса:**

1. Данные классы: **обычные** члены-данные класса, **static** – статические члены классы (одни и те же на все экземпляры классов (сходные с глобальными переменными)), **const** – константные члены класса.
2. Операции класса (методы): **обычные методы** класса, **static** – статические методы класса, **const** – константные методы (не изменяют данных класса). Операции – **виртуальные, переопределенные, обычные, операторные** функции (перегрузка стандартных операторов)
3. Указатель **this** позволяет получить указатель на сам объект из методов объекта.
4. **Конструктор** – функция, вызывающаяся в момент создания объекта.
5. **Деструктор** – функции, вызывающаяся при удалении объекта.
6. Области видимости:

Public	Видимы всем объектам (из любого места программы). **ALL**
Private	Только методам этого класса и дружественным функциям и классам. **THIS + FRIEND**
Protected	Только методам этого класса и дружественным функциям и классам и классам-потомкам. **PRIVATE + CHILD CLASSES**
Не указана	**PRIVATE для class, PUBLIC для structure**

7. **Объект – это конкретный экземпляр класса**

Связи между классами.

- **Наследование** - возможность порождать один класс от другого с сохранением всех свойств и методов класса – предка. Одиночное наследование и множественное наследование.
- **Агрегация** – физическое включение. Отношение «целое/часть» (part of). *(в классе есть член-данные с типом другого класса)*
- **Использование** – один класс пользуется услугами другого. *(Есть ссылка в вызовах функций, непосредственное использование в теле функций и т.д.)*
- **Инстанцирование** – параметризированные классы. Когда определяется экземпляр класса шаблона, нужно в явном виде указать тип шаблона. (*vector <int> v*)
- **Ассоциация** – смысловая связь между классами. *(В классе есть ссылка на другой класс, по которой устанавливается связь между экземплярами этих классов. Часто ассоциация превращается в простую агрегацию)*

Наследование.

- **Наследование** - возможность порождать один класс от другого с сохранением всех свойств и методов класса – предка.
- !!!Наследуются только функции и данные из разделов public и protected.

  Управление доступом при наследовании: **class A: <*public/private/protected*> B**

Public	***B.Public –> A.Public***	***B.Protected->A.Protected***
Private	***B.Public-> A.Private***	***B.Protected->A.Private***
Protected (обычно не используется)	*B.Public-> A.Protected*	*B.Protected->A.Protected*
Не указан = PRIVATE	***B.Public-> A.Private***	***B.Protected->A.Private***

class A { private: int prv; protected: int prt; public: int publ}; class B: A { int x; }

main() {B b; ***b.publ = 0;/*Ошибка «publ» не доступен в классе B*/*** *}*

class A { private: int prv; protected: int prt; public: int publ}; class B: ***public*** *A { int x; }*

*main() {B b; b.publ = 0;/*ошибки нет*/;* ***b.prt = 0; /*Обращение к защищенной переменной*/*** *}*

- При создании объекта (порядок вызова конструкторов) в начале создаются все его родители с самого первого, а затем создается сам объект.
- При удалении объектов – наоборот, в начале деструктор объекта, а потом деструкторы родителей.

Виртуальные функции.

- **Виртуальная функция** – *функция-член класса, которая может быть замещена в классах потомках. Функция однажды объявленная виртуальной в базовом классе, остается таковой во всех классах потомках.*
- Реализация механизма виртуальных функций в компиляторе. Таблица виртуальных функций. Рассмотрим пример:

class A { public: virtual void f(){cout<<“A”};}; class b:public A {public: void f(){cout<<“B”}};

void g(A*e) {e->f}. Main() { A a; g(&a);B b; g(&b)}; на экране: “A””B”. Откуда программа знает, что в первом случае нужно запустить f от класса A, а во втором f от класса B?.

- Для этого компилятор для каждого класса, где есть виртуальные функции создает таблицу виртуальных функций (vtbl), куда помещает идентификаторы (индексы) и адреса этих функций. К каждому объекту классов, содержащих виртуальные функции, добавляется еще один член-данные – ссылка на VTBL. В данном примере объект «a» хранит ссылку на VTBL_A, а «b» на VTBL_B. И когда делается вызов e->f , просто ищется адрес функции f в соответствующей VTBL.

Различия между виртуальной функции и переопределенной функцией

```
class A
{
 public:
    void f() {cout<<“1”;}                virtual void f(){cout<<“1”;}
}
class B:public A
{
   public:
       void f() {cout<<“2”;}                    void f() {cout<<“2”;}
//переопределение OVERRIDE;                          // f – виртуальная функция
}
main()
{
       A *a;
       B *b = new B ;
       b->f(); // print “2”                  // print “2”
       a=b;
       a->f(); // print “1”                  // print “2”  ПОЛИМОРФИЗМ  - один и тот же код
         исполняется по разному в зависимости …..
}
```

Технология обработки исключительных операций

- Обработка исключительных ситуаций языка C++ (стандартные библиотеки языка C++, свои собственные алгоритмы). Не зависят от операционной системы. (блоки **try** {} **catch**(…) генерация **throw**)

  идея:*Когда разработчик определяет ошибку(например a/b, b!=0) , то в случае ее возникновения при работе программы, она должна как-то ее обработать. Можно было бы снабдить каждую процедуру или функцию дополнительным параметром «код ошибки», или использовать глобальную переменную, как это делает C (errno). Но в этом случае, программа состояла бы из ОГРОМНОГО количества IF error THEN ELSE операторов. Чтобы этого избежать в C++ используется следующий механизм: разработчик в нужном месте вызывает throw ExceptionType. Это сообщение «помещается в стек», и «достается из стека» ближайшим блоком try, catch. Это избавляет программиста от лишнего кода и делает код более читабельным, логичным и понятным.*

  важно:*При обработке исключительной ситуации, важно ее именно обработать, а не только сообщить о ней. Например, если ошибка произошла в алгоритме, где выделялась динамическая память, то при выходе по исключительной ситуации, память должна быть корректно освобождена.*

- Обработка исключительных ситуаций, генерируемых операционной системой или компилятором:
  - (SEH - Structure exception handling) система обработки исключительных ситуаций в OS Windows (__try, __exept, __finally)
  - обработка исключительных ситуаций OS средствами вспомогательных классов и компонентов. Например, библиотека VCL Borland генерирует исключения операционной системы в стиле C++. (См. Help разделы «VCL/CLX error handling»)

Этапы проектирования «простой» программы. Учебный пример.

- Построение диаграммы классов.
- Разработка структуры пользовательского интерфейса. (Эскизы экранных форм).
- Составление событийной модели реакции программы на действия пользователя. (Описывается конечным автоматом или машиной состояний).
- Детализация алгоритмов работы отдельных процедур и функций (методов классов). (Диаграммы действий (блок-схемы),конечные автоматы, структурированное описание на естественном языке, описание на псевдокоде).
- В случае сложной объектной структуры или использования активных объектов построение диаграмм взаимодействия объектов.
- Разработка структуры исходных файлов. (Какие будут модули, как будут называться, в каком модуле какие классы будут реализованы, как модули будут с друг другом связываться).
- Разработка структуры выходных файлов (исполняемые файлы, используемые библиотеки, файлы помощи).
- Разработка тестов (тестовых сценариев) для готовой программы.

Программирование под Windows

- Ядро – программный код, реализующий основные базовые функции по исполнению программ на процессоре, распределением процессорного времени между процессами и т.д. Windows – микроядерная OS. Небольшой размер ядра позволяет его хранить полностью в физической оперативной памяти.
- Набор вспомогательных программ (сервисы, системные утилиты и т.д.), написаны с использованием функций, которые «понимает» ядро - Win API.
- WinAPI (Windows application programming interface) - набор служебных функций для работы с OS.
- Все высокоуровневые библиотеки для прикладного программирования (MFC, VCL и т.д.) написаны на WinAPI.
- Есть три типа объектов, которыми можно управлять работой OS и компьютера: Kernel objects (process, thread, job ..), USER (curcos, mouse…), GUI (windows,controls …).
- Windows – МНОГОЗАДАЧНАЯ OS. Естественно, что если у ВАС один процессор, то «многозадачность» это иллюзия, так как реально в один момент времени выполняется только одна программа. Иллюзия «многозадачности» построена на том, что процессорное время квантуется и каждому процессу по очереди выделяется определенный квант. Это делает встроенный в ядро планировщик задач. При выделении процессорного времени учитывается приоритет процесса.
- Из многозадачности следует возможность параллельного выполнения процессов, а значит есть проблема по их синхронизации. Одним из способов передачи данных между процессами является механизм обмена сообщениями.

Схема обработки сообщений в OS Windows

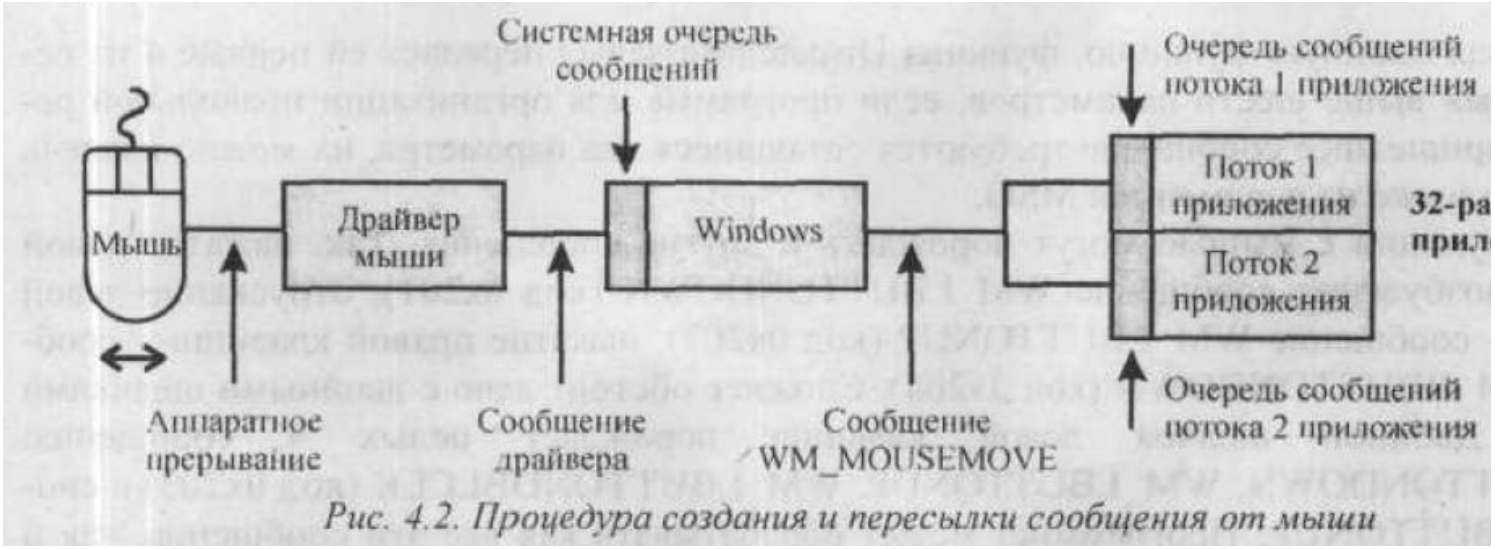

Рис. 4.2. Процедура создания и пересылки сообщения от мыши

- Отдельные сообщения на выходе из системной очереди можно обработать (переопределить процедуру разбора сообщений) с помощью установки HOOK функции.
- Каждый поток, может содержать цикл приема и обработки сообщений.
- Сообщения можно отправлять в очередь другого процесса минуя системную очередь.
- **Идея построения сложной системы по технологии обмена сообщениями, с использованием одной центральной очередью сообщений, управляющей взаимодействием с внешними устройствами (клавиатура, мышь, touchpud …) интересна инженерам. Так как может быть применена в других местах.**

Процесс. Поток. Задание.

Процесс (Process) (user kernel object) – экземпляр выполняемой программы.

Состоит из:

- Объекта ядра, через который OS управляет процессом. Тут же храниться статическая информация о процессе.
- Адресного пространства, в котором находятся код и данные всех EXE, DLL модулей. Именно здесь находятся области памяти под динамические стеки потоков и т.д.

Чтобы процесс что-нибудь выполнил в нем нужно создать **поток**. При создании процесса система автоматически создает его первый поток, называемый главным потоком.

Поток (Thread) – исполняют код процесса.

- Объект ядра, через который OS управляет потоком
- Стек потока, который содержит параметры всех функций, локальные переменные, необходимые потоку для выполнения кода.

Задание (Job) – объект ядра, позволяющий сгруппировать группу процессов и рассматривать ее как единое целое.

Виртуальное адресное пространство процесса. (Win 32) (~4,5 ГБ)

Раздел для выявления нулевых указателей	0x00000000-0x0000FFFF. Резервируется, чтобы программисты могли выявлять нулевые указатели. Недоступен для чтения или записи.
Раздел для кода и данных пользовательского режима (~2Гб)	• Закрытая не разделяемая часть адр. пр-ва процесса. • Ни один процесс не может получить доступ к данным другого процесса, размещенным в этом разделе. • Состоит из: – Раздел кода программы – Стек потока (1 МБ) (Устанавливаются при компиляции) – Куча по умолчанию (1 МБ) (при компиляции, сверх увеличивается динамически) – **Свободной пользовательское адресное пространство процесса**
Закрытый раздел	64 КБ
Раздел для кода и данных режима ядра (~ 2 ГБ)	Код операционной системы, драйвера устройств, код низкоуровневого управления потоками, памятью и т.д. Все что находится здесь, доступно любому процессу. В Windows 2000 полностью защищен.

Трансляция виртуального адреса на физический

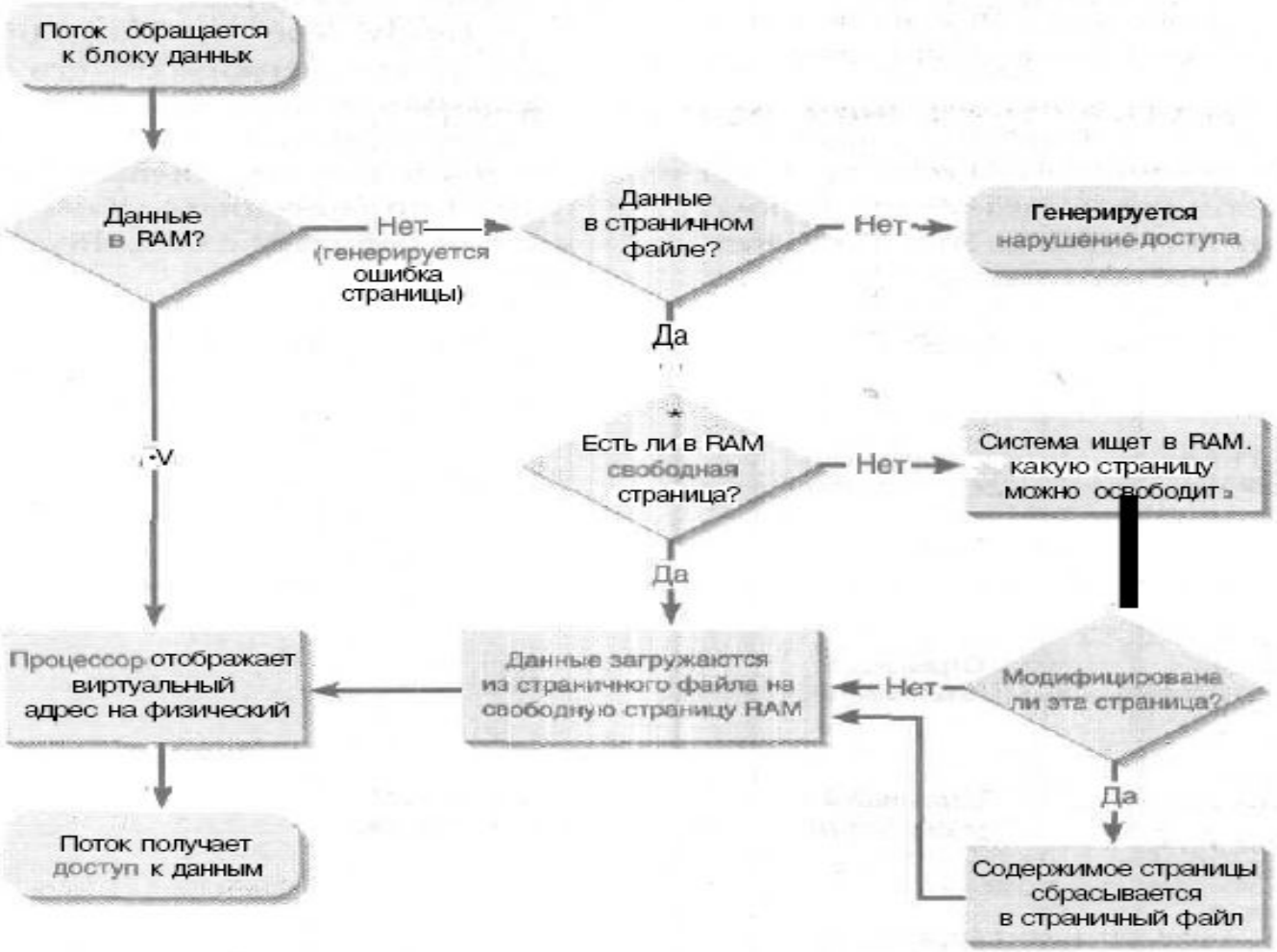

Рис. 13-2. *Трансляция виртуального адреса на физический*

Способы освоения свободного пользовательского адресного пространства процесса в OS Windows.

- Использование «виртуальной памяти»: *резервирование адресов памяти, явное выделение памяти. Наиболее подходящая для операций с большими массивами объектов или структур.*
- Использование кучи (heap) – *наиболее подходящие для работы с множеством малых объектов. (**Оператор new, malloc (calloc,alloc) выделяют память именно из кучи**).*
- Файлы, проецируемые в память – *наиболее подходящие для операций с большими потоками данных (обычно большие файлы) и для совместного использования данных при параллельных процессах.*

- *!!! Понятно, что если физическая оперативная память не позволяет хранить все адресное пространство в памяти, то эти данные должны где-то сохраняться. Где -> в специальных файла на диске. Файлы подкачки (swap files). И операционная система сама подгружает их с диска и на диск по мере необходимости их использования.*

Пример использования виртуальной памяти

- Реализация электронной таблицы (типа Excell): требуется на экране редактировать табличный документ размерами [100][100].
- Логичнее всего реализовать его с помощью массива <T> s[300][300]. Предположим, что sizeof(T) = 100b, тогда на хранение этой структуры понадобиться 100x300x300 = 9000000 b ~ 9MB.
- Существует три способа реализации такого механизма:
  - Через оператор new сразу выделить эту память. Память будет выделена, но заполнена пустыми значениями. Достаточно расточительно, если считать, что мы воспользуемся только парой ячеек в таблице.
  - Использовать связные списки, выделяя память, только под те ячейки, которые редактируются. Но по связным спискам очень тяжело осуществлять навигацию.
  - Но можно сразу зарезервировать адреса в виртуальном адресном пространстве на весь массив. При этом это только резерв адресов, а физическая память еще пока не выделена. Физическую память выделять по мере необходимости при работе с изменяемыми ячейками.

DLL – Dynamic Link Library

Преимущества использования DLL:

- Расширение функциональности приложения
- Более простое управление проектом
- Экономия памяти
- Разделение ресурсов
- Упрощение локализации
- Решение проблем, связанных с особенностями различных платформ
- Реализация специфических возможностей