Билеты. Технология программирования.

1. Введение в технологии программирования. Стили и парадигмы программирования.

Технология программирования - совокупность методов средств и процедур используемых в процессе разработки ПО. При этом все существующие средства разработки можно разделить на две основные категории: языки программирования высокого уровня и низкого.

При разработке ПО определяют парадигму программирования.

Парадигма программирования - способ создания программ с помощью определенных принципов и подходящего языка, позволяющего писать ясные программы:

Структурное - методология разработки ПО, в основе лежит представление программы в виде иерархической структуры . Вся программа разбивается на отдельные блоки, элементарными блоками являются : последовательность, ветвление, циклический блок.

Функциональное - в которой любой процесс вычисления трактуется как вычисление значения функции в ее математическом понимании. Вычисление ведется от исходных данных, которые в свою очередь могут Являться результатами работы предедыщих функций. Функция будет запущена тогда и только тогда, когда на ее входе будет иметься полный набор исходных данных. Порядок выполнения функции определяется порядком прихода аргументов. Представитель: lisp.

Логическое программирование - парадигма, основанная на автоматическом доказательстве теорем на базе заданных фактов и правил вывода. Оно основано на аппарате математической логики.

Язык- пролог.

Автоматное программирование - при использовании которой программа или ее фрагмент определяется как модель формального автомата. В зависимости от сложности задачи в автоматном программирование могут использоваться конечные автоматы или автоматы более сложной структуры

ООП - основные компоненты :объект и класс, а программа представляет собой процесс взаимодействия объектов, функционирующих на основе имеющихся у них методов, обладающих поведением и возможностью обмена сообщениями. На базе ООП может быть построено проттотипное программирование. В этом случае отсутствует понятие класса, а наследование определяется как механизм

клонирования или создания прототипов, то есть полной копии уже существующего в программе объекта. Программирование на основе шаблонов

СОП - при которой выполнение проги определяется срабатыванием некоторых событий, генерируемых пользователем либо другими элементами системы. События представляют собой потоки, организующие взаимодействие между компонентами программы с помощью исполнительных механизмов.

Агентно ориентированное программирование - парадигма, в которой основная концепция - работа параллельных независимых агентов. Для каждого агента определяется поведение, которое зависит от среды, в которую агент помещается в текущий момент времени. Агент воспринимает среду через набор датчиков и регулирует собственное поведение в зависимости от полученное информации за счет собственных исполнительных организмов.

Не существует средство программирования, которое придерживалось бы одной парадигмы. Чаще всего это комплексные средства. VS поддерживает структурное, ООП, событийное.

2. Технология .NET. Основные положения.

Технология NET предложена компанией майкрософт и обладает улучшеной функциональной совместимостью, в основе которой лежит использование открытых стандартов. NET технология повышает устойчивость интерфейса и предоставляет программное платформу и набор инструментов с использованием языка разметок. Состав NET платоформы входят следуйте составляющие:

- Обще языковая среда выполнения CLR - организация безопасной работа элементами системы, обеспечивает корректной доступ к динамической памяти и сборку мусора в реальном времени, обеспечивает поддержку многоязыковых приложений.
- Средства разработки приложения на наборе языков программирования.
- Библиотеку классов NET. Framework для создания приложений различного рода. Содержит много кратные используемые коды на любом из поддерживающих языков программирования.
- Поддержку сетевой инфраструктуры на основе открытых стандартов NET. Для создания устойчивых приложений.
- Поддержку нового промышеленного стандарта, а именно технологии веб - служб.
- Модель безопасности, может использовать в приложениях для организации доступа к памяти и прочим ресурсам

- Мощные инстурменталтны средства разработки для визуальной разработки приложений

Структура платформы NET

Особенность - разработка приложений на основе каркаса. Понятие каркаса появилось начиная с 4 версии студии, где каркас представлялся статическими компонентами. В настоящее время каркас объединяет следующий набор элементов:

1) Базовые классы среды - библиотека, которая содержит основные функциональные возможности среды разработки(функции работы с различными типами данных, массивами)

2) Интерфейс пользователя- библиотека элементов, обеспечивающих графическое взаимодействие с пользователем, в том числе элементы форм

3) Классы для работы с данными и XML-библиотека, поддерживающий расширенный язык разметки, что позволяет разрабатывать web - приложения

4) Web- службы - набор служб поддержки интернет

5) Классы удаленной обработки - случайный набор классов поддержки сетевого сервиса, для создания серверных приложений

6) Общая система типов - определяет набор типов данных, подлежащих обработке средой с возможностью контроля организации доступа к ним

В состав каркаса включены встроенные примитивные типы - библиотека FCL. Примитивные типы - типы, встроенные в язык программирования. Эти типы монтируются в каркас для «умного» управления ими, т е определение возможности сопоставления типов для много язычных приложений.

С появлением данного компонента процесс написания программ облегчаемся за счет объединения разноязычных модулей. Платформа NET является надстройкой над ОС, реализующей функции пользователь их приложений. Наличие общеязыковой исполнительной среды предполагает расширение возможности разработки за счет введения следующих элементов:

Двухэтажная компиляция - предполагает наличие в программе двух понятий : управляемый модуль и управляемый код:

- Модуль - исходный модуль программы, написанный на перенос моим языке при компиляции котором создается перенос мой исполняемый файл. Этот файл состоит из двух частей - методанные и MSLчасть. Методанные определяют структуру элементов исполняемого модуля и хранят информацию, требуемую как к среде так и к конечному пользователю. Часть хранит

информацию связанную с языком разработки, позволяющую правильно интерпретировать записанную в место данные структуру.

- На втором этапе компиляции создается управляемый код где предполагается что код исполняемого модуля выраженный в терминах MSAL интерпретируется общеязыковой исполнительной средой.

Виртуальная машина

Так как конечным этапом компиляции является создание кода, он должен быть интерпретирован для пользователя. Это осуществляет виртуальная машина, которая преобразует команды управляемого кода в машине команды текущего аппаратной.

Наличие дизассемблера

Среда снабжена дополнительным механизмом, позволяющий осуществить асемб разбор управляемого кода. Этот разбор позволяет обеспечить связи команд и методанных приложений. Он позволяет анализировать разобранные данные и создавать управленцы код

Встроенный сборщик мусора - среда берет на себя функции связанные с освобождении памяти и уменьшении фрагментации. Нужным требованием для подключения является использование умных указателей, то есть указатели библиотеки CLR. При этом среда берет на себя обязанности по управлению данными.

Инструментальной средства студи. NET

1) поддержка много языковой среды CLR

2) возможность создания проекта на наиболее подходящем языке внутри среды

3) доступно то всех средств NET для каждого из широкого спектра языков программирования

4) возможность облеченной разработки транслятора для любого языка

При использовании технологии .NET разработчики доступны следующие элементы:

aSP.NET- данная модель содержит службы для построения удаленных приложений предприятия

netCF- работе на небольших устройствах и позволяет создавать приложения для мобильных средств

Silver light - независимый от браузера элемент, позволяющий проектировать, поставлять интерфейса с поддержкой мультимедиа а также много

функциональные инет приложения. Анимации оные и интерактивные приложения

Компоненты среды, позволяющие создавать приложения взаимодействующие с компонентами Office

WinForms - библиотека содержащая набор компонент для создания окон

Система для построения клиентских приложения под WinDows . Создавать автономные и размещаемые в браузерах приложения; векторная система визуализации.

xNA- технология позволяющая создавать игровые программы, под управлением винды.

3. Основы C++. Определение времени жизни и области видимости переменных. Пространство имен.

Достоинства языка :

1) гибкость и компактность языка (часть операторов языка может вбить и в полной и в сокращенной форме);

2) эффективность, основанную на том, что семантика языка отражает архитектуру компьютера;

3) доступность (для любой ОС имеется компилятор с языка в машин но - зависимые коды);

4) переносимость (степень сложности переноса с 1 платформы на другую относительно проста по сравнению с другими языками);

Структура программы:

#include "stdafx.h" - файл для подключения заголовочных файлов,используемых в проекте; директивы предпроцессора, определяющий набор файлов, используемых в проекте. Заголовочные файлы содержат библиотеки, которые могу быть вызваны. Местоположение определяется по названию.

float tax (float) -предварительное объявление функций(нет понятия процедуры);

int main - главная функция программы

return 0; - обязателен

float tax (float amount) - описание вспомогательной функции (ее тело)

Void - произвольный невозвращаемый тип

Cin - входной поток, перемещающий из потока в/в в указанную переменную

Count - в поток вывода

Нужно учитывать время жизни и области видимости переменных

Время жизни определяется по следующим правилам:

1) переменные объявленные на внесшем уровне, всегда имеют глобальное время жизни;

2) переменные, объявленные на внутреннем уровне, имеют локальному время жизни. Можно обеспечить глобальное время жизни для перемен ной внутри блока, задавая ей класс памяти static при ее объявлении. Такая переменная будет сохранять сове значение на протяжении всего времени продолжения программы, но видима будет только внутри блоков, где объявлена.

Видимость переменной в программе определяется по правилам:

1) переменные, объявленные или определенные на внешнем уровне, видимы от точки объявления или определения до конца исходного файла;

2) переменные, объявленные или определенные на внутреннем уровне, видимы от точки объявления или определения до конца блока;

3) переменные из объемлющих блоков, включая переменные, объявленные на внешнем уровне, видимы во внутренних блоках.

Управлять характерами переменных можно используя модификаторы памяти:

- auto — *автоматическая* переменная. Память выделяется в стеке и при необходимости инициализируется каждый раз при выполнении оператора, содержащего ее определение. Освобождение памяти - при выходе из блока
- extern — переменная определяется в другом месте программы.
- static — *статическая* переменная. Время жизни — постоянное. Инициализируется один раз при первом выполнении оператора, содержащего определение переменной. В зависимости от расположения оператора описания статические переменные могут быть *глобальными и локальными*.
- register — аналогично auto, но память выделяется по возможности в регистрах процессора.

Пространство имен - логически объединяет классы с близкой функциональностью, предназначено для разрешения конфликтов между именами в разных сборках (в различных участках приложения);

Не рекомендуется помещать пространства имен внутрь блоков, например циклов.

```
namespace Jack
        { void  fetch( ) ;
        int  pal;}
namespace Jill
        {double fetch ;
        int pal ;}
Jack :: pal = 12 ;
```

Объявления using и директивы using

Механизмы позволяют упростить обращение к пространстве имен;

Механизм Объявления обеспечивают доступ к определенным идентификаторам внутри пространства имен.

Директива делает доступными пространство имен в целом

:: - двойное разноименование;

```
namespace Jill
 { int pal;
 double fetch ;
 } ;
har  fetch ;
int main ( )
{
using Jill :: fetch ;
double fetch ;
cin >> fetch ;
cin > > :: fetch ; }
```

```
-------------------------------------
using namespace System;
std::cout<<“HELLO WORD”<<std::endl
using namespace std;
cout<<“HELLO WORD”<<endl
-------------------------------------
Jack :: pal = 3 ;
Jill :: pal = 10 ;
using Jack :: pal ;
using Jill :: pal ;
pal = 4 ; / /  Конфликт имен
```

4. Основы C++. Управляющие конструкции. Функции. Параметры по умолчанию. Функции с неизвестным числом параметров. Перегрузки. Шаблоны.

Функция - фрагмент программы, который может вызываться много раз из каждого места программы. => уменьшается избыточность кода и повышается структурированность.

Выделяют: описание (объявление и определение) и вызов

Определение - содержит темы (тело ??) функции и спецификаторы ее параметров в заголовке.

Вызов - обращение по ее имени с указанием списка фактических параметров.

При вызове могут передаваться различные параметры.

Механизмы передачи :

а) передача по значению

б) передача по ссылке или указателю (передается адрес параметра)

в) обращение к постоянному параметру по ссылке или адресу

г) использование функции с параметрами по умолчанию

При вызове заданные в заголовке параметры могут быть опущены. В качестве параметров по умолчанию - только последние параметры.

```
int sum (int a, int b=10)
sum (c,d); или  sum (x);
```

д) можно организовать функции с переменным числом параметров.Пример:

```
int sum(int x, ...);
int main() {
   std::cout << sum(2, 20, 30) << std::endl;     // 50
   std::cout << sum(3, 1, 2, 3) << std::endl;    // 6
```

```
    std::cout << sum(4, 1, 2, 3, 4) << std::endl; // 10
    return 0;
  }

  int sum(int x, ...) {
    int result = 0;
    int *p = &x;     // Получаем адрес последнего параметра
    for(int i=0; i<x; ++i)
{
   ++p;          // Перемещаем указатель на следующий параметр
   result += *p; // Прибавляем очередное число
 }
 return result;
}
```

Передача параметров в стек в обратном порядке. На вершине стека - всегда 1 параметр, указанный в заголовке.

Перечисленные параметры должны иметь возможность интерпретации или признак окончания.

Перегрузка функции

Разные функции =>разные имена, однако на практике удобно использовать одноименные функции для однотипных операций для различных параметров.

Перегруженные функции - одинаковые имена, но разные сигнатуры. При компиляции создается декор => выбор функции при вызове осуществляется на основе декора.

```
int sum (int a, int b) {return a+b;}
                              // public @sum#qii
double sum (double a, double b) {return a+b;}
                              // public @sum#qdd
```

При использовании может быть неоднозначность. Возникает при:

1)преобразовании типа;

Тип фактических параметров могут быть преобразования неявным образом.

```
float sum(float x, float y);
double sum(double x, double y);
std::cout << sum(10.5, 20.4) << std::endl; // Нормально
std::cout << sum(10, 20) << std::endl;     // Неоднозначность
```

2) использовании параметров-ссылок;

Так как ссылка интерпретируется как целое число

```
void print(char *str);
void print(char str[]);
print("String");
```

3) использовании аргументов по умолчанию

```
int sum(int x);
int sum(int x, int y=2);
...std::cout << sum(10, 20) << std::endl; // Нормально
std::cout << sum(10) << std::endl;     // Неоднозначность
```

Правила описания перегруженных функций.

- Перегруженные функции должны находиться *в одной области видимости;*
- Перегруженные функции могут иметь *параметры по умолчанию*, при этом значения одного и того же параметра в разных функциях должны совпадать. В различных вариантах перегруженных функций может быть различное количество параметров по умолчанию;
- Функции не могут быть перегружены, если описание их параметров отличается только *модификатором* const *или использованием ссылки* (например, int и const int или int и int&).

Шаблонные функции

Шаблон - обобщенное определение функции из которой компилятор автоматически создает представление функции для заданного пользователем типа данных или типов. Создается порожденная функция.

template <параметры> заголовок
{ /* тело функции */ }

Каждый параметр шаблонной функции должен быть объявлен с ключевым словом class или с типом. Создание производится в момент вызова. Порождается функция, соответствующая типу фактических параметров.

```
template <class T>
void printarray (T *array, const int count)
        {for (int i=0; i<count; i++)
                cout<<array[i]<<” ”;
                cout<<endl;
        }
int main()
        {const  int account=5, bcount=7;
        int a[account]={1,2,3,4,5};
        float b[bcount]={1.1,1.2,1.3,1.4,1.5,1.6};
        printarray (a, account);
        printarray (b, bcount);
        return 0 ;
        }
```

При работе:

а) все объявленные шаблонные типы должны быть использованы в списке формальных параметров.
б) имена шаблонных переменных не могут совпадать
в) шаблонная функция может быть объявлена как внешняя, статическая или встроенная.

Спецификатор в заголовке шаблона.

При использовании шаблонов - явная спецификация типов. При этом от компилятора надо сконкретизовать шаблонную функцию с указанным типом.

```
template <class T>
void FuncName(T) { . . . } ;
void AnotherFunc(char ch)
{
//Требуем сгенерировать
//конкретизацию шаблонной функции
FuncName<int>(ch) ;}
```

Шаблонные функции могут использоваться при конкретизации в различных точках. Принцип конкретизации определяется моделью.

1. Модель с включением - определение шаблона включается в каждый файл в котором он конкретизируется
2. Модель с разделением - объявление шаблона в отдельный заголовочный файл. Определение самого шаблона в одноименном исходном файле.

В ряде случаев для обращения удобно использовать ссылку : используют процедурные типы - определение указателя на функцию с заданием типа возвращаемого значения и перенес параметров.

```
void err (char * p) // определение функции
{...}
void (*p_funct) (char *); // указатель на функцию
...
Void main()
{p_funct = &err;     // получение адреса функции
(*p_funct)("data"); // вызов функции}
```

Функция может быть использована как встроенная. В общем случае вместо вызова - адрес вызываемого участка кода и нужны дополнительные затраты.

Если функция невелика, то затраты неоправданны. Удобно объявить как встроенную. Компилятор вставляет код вызываемой функции в точку вызова.

```
inline int sum(int a, int b) {return (a+b);}
```

5. Указатели. Устройство виртуальной памяти. Функции C++ для работы с динамической памятью.

Понятие указателя

Любой объект программы, будь то переменная базового или же производного типа, занимает в памяти определенную область.

Местоположение объекта в памяти определяется его адресом. При объявлении переменной для нее резервируется место в памяти, размер которого зависит от типа данной переменной, а для доступа к содержимому объекта служит его имя (идентификатор). Для того чтобы узнать адрес конкретной переменной, служит унарная операция взятия адреса (&). Мощным средством разработчика программного обеспечения на C++ является возможность осуществления непосредственного доступа к памяти. Для этой цели предусматривается указатели.

В стандартизируемом варианте языка C++ предполагается, что все указатели одинаковы, т.е. внутреннее представление адресов всегда одно и тоже. Однако, практически все компиляторы языка C учитывают архитектуру процессора и включают дополнительные возможности для ее эффективного использования.

Основная память ЭВМ - это память с произвольным доступом. К каждому элементу которого можно обратиться по средством сегментированного адреса. Любые два смежных байта образуют 16-тиразрядное слово. Адресом слова является младший из адресов байтов. Понятие слово относительно, т.к. один и тот же байт может входить в два смежных слова.

В общем случае оперативная память, с которой работает программа, подразделяется на три вида: статическую, автоматическую и динамическую.

Статическая память — это область памяти, *выделяемая при запуске программы* до вызова функции *main* из свободной оперативной памяти для размещения глобальных и статических объектов, а также объектов, определённых в пространствах имён.

Автоматическая память — это специальный регион памяти, *резервируемый при запуске программы* до вызова функции *main* из свободной оперативной памяти и *используемый в дальнейшем* для размещения локальных объектов: объектов, определяемых в теле функций и получаемых функциями через параметры в момент вызова. Автоматическую память часто называют **стеком.**

Динамическая память — это совокупность блоков памяти, выделяемых из доступной свободной оперативной памяти непосредственно *во время выполнения программы* под размещение конкретных объектов.

Процессы и адресное пространство

Прежде чем изучать управление памятью в Windows, надо понять, что такое *процесс* (process). Программа — это EXE-файл, который можно запустить из Windows. После запуска программа называется процессом. У процесса есть собственные память, описатели файлов и другие системные ресурсы. Запустив две копии одной программы, вы получите два отдельных процесса.

Важно знать, что процессу выделяется свое частное 4-гигабайтное виртуальное адресное пространство, Программа может обращаться к любому байту этого адресного пространства, используя один-единственный 32-разрядный линейный адрес. При этом в адресном пространстве каждого процесса содержится масса самых разных элементов:

•образ EXE-файла программы;

•все несистемные DLL, загруженные вашей программой;

•глобальные данные программы (как доступные для чтения и записи, так и предназначенные только для чтения);

•стек программы;

•динамически выделяемая память, в том числе куча Windows и куча *библиотеки С периода выполнения* (C runtime library, CRT);

• файлы, спроецированные в память;

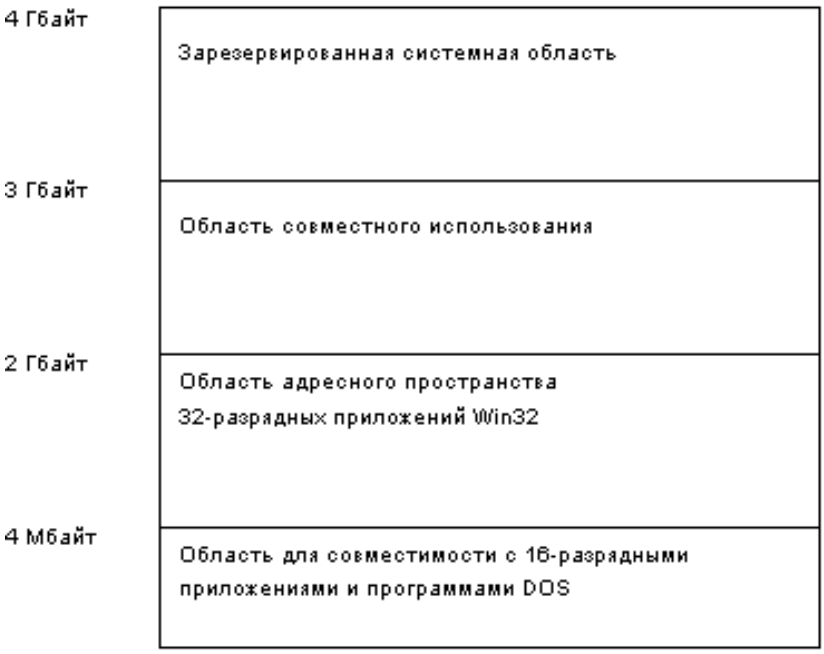
• блоки памяти, совместно используемые несколькими процессами;

• локальная память отдельных выполняемых потоков;

• всевозможные особые системные блоки памяти, в том числе таблицы виртуальной памяти;

• ядро, исполнительная система и DLL-компоненты Windows,

Адресное пространство процесса в Windows 95/98



В Windows 95 только нижние 2 Гб адресного пространства (0 — Ox7FFFFFFF) по-настоящему закрыты, причем доступ к нижним 4 Мб этих 2 Гб запрещен. Стек, кучи и глобальная память, доступная для чтения и записи, проецируются на нижние 2 Гб, как, впрочем, и EXE- с DLL-файлами приложения.

Верхние 2 Гб совместно используются всеми процессами.

Ядро Windows 95, драйверы виртуальных устройств (VxD), код файловой системы, а также таблицы страниц располагаются в верхнем гигабайте адресного пространства (OxCOOOOOOO — OxFFFFFFFF). Динамически подключаемые библиотеки и спроецированные в память файлы расположены в диапазоне 0x80000000 — OxBFFFFFFF.

Посторонний процесс практически не в состоянии перезаписать стек, глобальные данные или память кучи другого процесса, потому что вся память в нижних 2 Гб виртуального адресного пространства, принадлежит одному процессу. Весь код EXE- и DLL-файлов помечен как доступный только для чтения, поэтому нет никакой проблемы в том, что он спроецирован в несколько процессов.

Однако верхний гигабайт адресного пространства весьма уязвим, поскольку в него спроецированы важные данные Windows, доступные для чтения и записи. При ошибке программа может уничтожить важные системные таблицы, расположенные в этой области. А какой-нибудь процесс может повредить содержимое спроецированных в память (0x80000000 — OxBFFFFFFF) файлов, потому что эту область совместно используют все процессы.

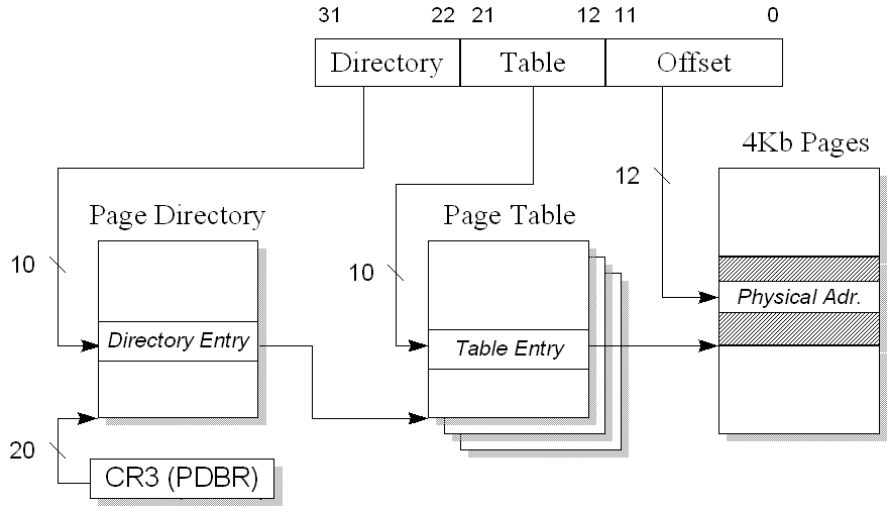
Адресное пространство Windows NT/2000/XP

Процесс в Windows NT/2000/XP имеет доступ только к нижним 2 Гб своего адресного пространства, причем самые нижние и самые верхние 64 кб этого диапазона недоступны. Исполняемый файл, DLL приложения и Windows, а также спроецированные в память файлы располагаются в диапазоне Ox00010000-Ox7FFEFFFF.

Ядро, исполнительная система и драйверы устройств Windows NT располагаются в верхних 2 Гб, где они полностью защищены от искажения со стороны неисправной программы. Файлы, спроецированные в память, также реализованы надежнее: никакой процесс не получит доступа к спроецированному в память файлу другого процесса, если только он не знает имени файла и не создал проекцию явно.

Устройство виртуальной памяти

4-гигабайтное адресное пространство процесса экономно используется небольшими фрагментами. Программы и элементы данных разбросаны по адресному пространству блоками по 4 кб, выровненными по границам, кратным 4 кб. Каждый такой блок называется *страницей* (page) и содержит либо код, либо данные. Естественно, используемая страница занимает какой-то участок физической памяти. Микропроцессор фирмы Intel эффективно преобразует 32-разрядный виртуальный адрес в номер физической страницы и смещение внутри нее, пользуясь двухуровневыми таблицами 4-килобайтных страниц. Заметьте: отдельные страницы можно помечать либо как только для чтения, либо как для чтения и записи. Кроме того, у каждого процесса свой набор таблиц страниц. Регистр CR3 процессора содержит указатель на страницу каталога, поэтому при переключении с одного процесса на другой Windows просто обновляет этот регистр.



Запись таблицы страниц содержит бит присутствия в физической памяти (present), который сообщает, находится ли сейчас в физической памяти данная

4-килобайтная страница. При попытке обращения к странице, отсутствующей в памяти, инициируется прерывание, и Windows анализирует ситуацию, просматривая свои внутренние таблицы. Если ссылка на область памяти оказывается ошибочной, мы получим ненавистное сообщение об ошибке страницы (page fault), и программа завершится. В противном случае Windows считает в оперативную память нужную страницу из дискового файла и обновит таблицу страниц, записав в нее физический адрес и установив бит присутствия в физической памяти. Так работает виртуальная память в Win32.

Диспетчер виртуальной памяти Windows, стремясь достичь максимума производительности, определяет моменты чтения и записи страниц. Если какой-то процесс не использовал страницу в течение определенного периода, а другому нужна память, страница выгружается из памяти, а вместо нее загружается страница нового процесса. Обычно программа об этом не уведомляется. Все процессы совместно используют один большой общесистемный страничный файл (swap file) (ранее он назывался файл подкачки), куда помещаются (при необходимости) все виды данных для чтения и записи и некоторые виды данных только для чтения. (Windows NT/200/XP способна одновременно поддерживать несколько страничных файлов,) Windows определяет размер страничного файла в зависимости от объема ОЗУ и свободного дискового пространства, но есть способы тонкой настройки размера и физического расположения этого файла.

Диспетчер виртуальной памяти, однако, использует не только страничный файл. Записывать в него страницы кода резона нет. Вместо этого Windows проецирует содержимое EXE- и DLL-модулей прямо в их дисковые файлы. Поскольку страницы кода помечены «только для чтения», необходимости в их записи обратно на диск не возникает.

Если два процесса используют один и тот же EXE-файл, последний отображается на адресные пространства обоих процессов. Код и константы никогда не изменяются во время выполнения программы, поэтому можно проецировать файл на одну и ту же область физической памяти. Однако два процесса не могут совместно обращаться к глобальным данным. С ними Windows 95/98 и Windows NT/200/XP поступают по-разному. Windows 95/98 проецируют в каждый процесс отдельную копию глобальных данных, а вот в Windows NT/200/XP оба процесса используют одну копию глобальных данных до тех пор, пока один из процессов не попытается что-либо записать в страницу. В этот момент страница копируется, а затем у каждого процесса появляется собственная копия, находящаяся по одинаковому виртуальному адресу.

Язык C++ предоставляет программисту большой выбор функций для работы с динамической памятью. К функциям работы с дин. памятью относятся: malloc,

calloc, farcalloc, farmalloc, realloc, free, new, delete. Функции XXXalloc и new используется для выделения блока динамической памяти, а free и delete - для его освобождения.

Рассмотрим особенности работы с функцией malloc. Она имеет следующий формат: void *malloc(size_t size).

Функция возвращает указатель на блок динамической памяти, заданного размера. Поскольку функция возвращает указатель типа void, то требуется явное преобразование типа.

Пример:

```
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

free(count);
```

Теоретически в приведенном выше примере запрашивается и расходуется 5 + 10*2 = 25 байт ОП.

Внимание! Как правило, компилятор настроен на автоматическое выравнивание адреса по границе параграфа (16 байт). Поэтому в нашем примере будет выделен 1 параграф под str и 2 параграфа под count. Т.о. всего будет занято 48 байт памяти.

Место, где будет выделяться блок памяти зависит от используемой модели памяти. Но в любом случае память выделяется из области дополнительной памяти (кучи).

Если имеется свободный последовательный участок памяти, достаточный для размещения необходимого числа байт, то функция malloc возвращает указатель на такой блок, иначе возвращается NULL. Содержимое выделенного блока не меняется. При size = 0, malloc возвращает NULL.

Перед завершением программы следует освободить вся запрошенную динамическую память с помощью функций освобождения. В паре с malloc используется функция free.

Функция calloc.

Аналогична функции malloc, но имеет несколько другой набор параметров: void *calloc(size_t nitems, size_t size).

Nitems - количество отводимых блоков, каждый из которых имеет размер size.

Пример:

char * str;

int * count;

str = (char *) calloc (10, sizeof(char));

count = (int *) calloc (5, sizeof(int));

При выделении памяти каждый байт устанавливается в 0. Данная функция доступна только в 16-разрядных приложениях. Также как и предыдущая функция позволяет получить память в пределах 64 Кбайт (1 сегмента).

Для получения памяти за пределами 64Кбайт используются модификации функций: farmalloc и farcalloc.

Пример:

char far *fptr;

fptr = (char far *) farmalloc(10);

В качестве аргумента должно выступать unsigned long.

Память, отведенная функциями farmalloc и farcalloc освобождается функцией farfree.

Иногда возникает ситуация, когда отведенной ранее памяти недостаточно для работы. В этом случае требуется перераспределение памяти. Для этого используется функция realloc:

void *realloc(void *block, size_t size);

Пытается обрезать или расширить ранее выделенный блок памяти *block до размера size байт. Если значение size равно 0, то память отведенная под *block освобождается.

Если *block указывает на NULL, то функция работает как malloc. При необходимости, недостаточно памяти для размещения нового блока, начиная с адреса block, старое содержимое block копируется по новому адресу.

Возвращает адрес нового блока, который может не совпадать со старым адресом block. При нехватке памяти возвращает NULL.

Все рассмотренные функции могут выделять память размером не более не более 4Г в 32-х разрядных моделях памяти.

При работе с динамической памятью следует иметь в виду, что в каждом выделенном блоке несколько байт отводится на служебную информацию.

В отличие от функций работы с динамической памятью malloc, calloc и free, заимствованных в C++ из стандарта ANSI C для совместимости, новые операторы гибкого распределения памяти new и delete обладают дополнительными возможностями, перечисленными ниже.

- Функции malloc и calloc возвращают пустой указатель, который в дальнейшем требуется приводить к заданному типу. Оператор new возвращает указатель на тип, для которого выделялась память, и дополнительных преобразований уже не требуется.
- Операция new предполагает возможность использования совместно с библиотечной функцией set_new__handler, позволяющей пользователю определить свою собственную процедуру обработки ошибки при выделении памяти.
- Операторы new и delete могут быть перегружены с тем, чтобы они, например, могли принимать дополни- тельные параметры или выполняли специфические действия с учетом конкретной ситуации работы с памятью.

Операторы new и delete имеют две формы:

- управление динамическим размещением в памяти единичного объекта;
- динамическое размещение массива объектов.

Синтаксис при работе с единичными объектами следующий:

тип_объекта *имя = new тип_объекта;

delete имя;

При управлении жизненным циклом массива объектов синтаксис обоих операторов имеет вид:

тип__объекта *имя = new тип_объекта[число];

delete[] имя;

Здесь число в операторе new[] характеризует количество объектов типа тип_объекта, для которых производится выделение области памяти. В случае успешного резервирования памяти переменная-указатель имя ссылается на начало выделенной области. При удалении массива его размер указывать не нужно.

Форма оператора delete должна обязательно соответствовать форме оператора new для данного объекта.

Пример:

```
int * p;

p = new int;
```

Можно запросить сразу линейный массив в динамической памяти:

```
p = new int [10]; // резервируется место под 10 int.
```

При резервировании места для двумерных массивов нужно отдельно резервировать место под указатели на начало строк и на сами строки:

Пример:

```
int ** d;
  int i;
  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];
```

Для доступа к элементам массива используется обычное обращение:

```
// инициализация массива
  for ( i=0; i<n; i++)
        for ( j=0; j<m; j++)
        {d[i][j]=i+j;}
```

При выходе из программы требуется освобождать память с помощью оператора delete:

```
for (int i = 0; i < n; i++)
delete[] d[i]; // освобождаются столбцы
delete[] d;   // освобождаются указатели на строки
```

6.Алгоритмы для управления области памяти

Все алгоритмы для управления областями памяти можно отнести к нескольким категориям, по типам объектов, с которыми они оперируют:

1) Объекты одного типа. Для выделения памяти под эти объекты нецелесообразно использовать malloc. Так как размер таких объектов постоянен, лучше использовать свой менеджер объектов, либо так называемый slab аллокатор.

2) Объекты определенного размера (например, страницы физической памяти). Для таких объектов наиболее подходят метод битовых масок или алгоритм "близнецов".

3) Объекты произвольного размера и типа (malloc). Для таких целей, как правило, применяется метод граничных маркеров.

В современных системах используется двухуровневый подход. На нижнем уровне идет выделение страниц памяти, а на верхнем уровне происходит выделение кусков произвольной длины. Верхний уровень запрашивает память у системы с помощью диспетчера страниц. Существуют также системы, которые не нуждаются в нижнем уровне (диспетчере страниц), это, например, встроенные системы с ограниченными ресурсами, выполняющие статический набор процессов. В таких случаях диспетчер памяти может захватить при инициализации отведенное для него пространство и больше никогда не обращаться к диспетчеру страниц. Также это часто применяется в системах реального времени, где жертвуют предсказуемостью за счет удобства.

Менеджер объектов

Данный метод подходит для выделения и освобождения большого количества одинаковых объектов, ограничение на количество заданно на момент компиляции системы.

Под объекты выделяется массив памяти из максимального количества объектов данного типа, и содержащиеся в нем объекты связываются некоторой структурой данных, например списком свободных объектов. Тогда выделение объекта - это всего лишь возвращение ссылки на голову списка, а освобождение объекта - это помещение ссылки на него в конец очереди свободных объектов.

Данный метод может применяться также в системах реального времени, поскольку известно что поиск подходящего куска памяти произвольной длины более трудоемкая задача и требует больших временных затрат.

В качестве доказательства достаточно привести временные результаты следующего теста. В первом случае происходит выделение с помощью системных функций malloc или new множества однотипных объектов. Во втором случае однократно выделяется пул для всех объектов сразу, и с помощью например списка создается список этих объектов. После этого в программе объекты выделяются из этого списка.

Во втором случае затраченное время будет существенно ниже

Slab менеджер

Менеджер объектов обладает хорошими временными характеристиками, но достаточно накладно писать для каждого объекта свой менеджер. Чтобы избежать дублирования кода, но сохранить предсказуемость и скорость применяют slab allocator. Основная суть этого метода заключается в частом выделении однотипных объектов, то есть ситуация описанная в разделе **Менеджер объектов**. Но в отличие него в системе вводится единый интерфейс для выделения и освобождения различных однотипных объектов. Для каждого из этих объектов создается свой пул, а в системе создается кеш подобных пулов для различных зарегистрированных объектов.

Алгоритмом близнецов используется когда различные размеры являются степенями числа 2, как 512 байт, 1Кбайт, 2Кбайта и т.д. Он состоит в том, что мы ищем блок требуемого размера в соответствующем списке. Если этот список пуст, мы берем список блоков вдвое большего размера. Получив блок большего размера, мы делим его пополам. Ненужную половину мы помещаем в соответствующий список свободных блоков. Одно из преимуществ этого метода состоит в простоте объединения блоков при их освобождении. Действительно, адрес блока-близнеца получается простым инвертированием соответствующего бита в адресе нашего блока. Нужно только проверить, свободен ли этот близнец. Если он свободен, то мы объединяем братьев в блок вдвое большего размера, и т.д.

Алгоритм близнецов значительно снижает фрагментацию памяти и резко ускоряет поиск блоков. Наиболее важным преимуществом этого подхода является то, что даже в наихудшем случае время поиска не превышает. Это делает алгоритм близнецов труднозаменимым для ситуаций, когда необходимо гарантированное время реакции - например, для задач реального времени. Часто этот алгоритм или его варианты используются для выделения памяти внутри ядра ОС. Например, функция kmalloc, используемая в ядре ОС *Linux*, основана именно на алгоритме близнецов.

Bitmap

Для представления памяти используется битовая карта, в которой каждому блоку соответствует один бит. Предназначен для выделения блоков памяти, выровненных по адресу и длине, например, страниц физической памяти.

Метод граничных маркеров (Boundary markers)

В основе метода - двусвязный список всех свободных и занятых блоков памяти. Как правило, элементами списка являются сами блоки памяти, точнее их "шапки", называемые маркерами. Модификация списка происходит в несколько простых операций. Существует несколько алгоритмов поиска свободного блока памяти заданного размера. Среди них - алгоритм первого соответствия, следующего соответствия, наилучшего и наихудшего соответствия. Кроме того,

можно поддерживать списки блоков для часто используемых запросов (т.н. алгоритм быстрого соответствия). Предназначен для выделения блоков памяти произвольной длины. Используется в аллокаторах общего назначения.

Алгоритм граничных маркеров хорошо подходит под выделение объектов произвольной длины, для сегмента же кода обычно используют другие алгоритмы. Алгоритм описан в книге Дэвида Кнута "Искусство программирование часть 1".

Для работы данного алгоритма в каждый свободный блок добавляется два дескриптора (расположенные в начале и в конце каждого блока) и два указателя (расположенные в начале блока) на предыдущий и последующий свободные блоки. Таким образом все свободные блоки соединяются в двунаправленный список, по которому и происходит поиск подходящего блока для выделения.

При выделении блока памяти помимо запрашиваемой памяти выделяется память под дескрипторы блоков, точно так же, один дескриптор в начале блока, другой - в конце. Таким образом выделяемый блок не может быть меньше, чем два указателя, которые могут понадобиться при освобождении блока под ссылки на предыдущий и последующий свободные блоки.

Дескриптор блока представляет из себя размер блока и флаг (свободен или занят) в терминах Кнута (Тэга блока). При использовании структуры, описанной выше, процессы выделения и освобождения блока приобретают следующий вид:

Выделение

Берем голову списка и, итерируясь по указателям, ищем подходящий блок. Могут быть различные стратегии:

- первый подходящий
- наилучший подходящий
- наихудший подходящий - в этом случае берется наибольший блок и от него вырезается требуемый размер.

Первый вариант быстрее, но второй и третий приводят к меньшей фрагментации памяти.

Освобождение

Основное преимущество данного алгоритма - в скорости освобождения памяти. Поскольку при освобождении мы знаем адрес освобождаемого блока, на предыдущем адресе лежит дескриптор блока. Кроме того, если освобождаемый блок не является крайним, непосредственно слева и справа от него расположены дескрипторы соседних блоков, содержащую информацию о

занятости и размере. Это значительно ускоряет процесс объединения соседних свободных от блоков, образовавшихся при освобождении данного. Таким образом, при освобождении блока проверяется соседний младший блок (с младшими адресами), и, если он свободен, эти блоки соединяются, а затем проверяется блок, следующий за освобождаемым, и, если он свободен, блоки тоже объединяются.

Последней операцией является корректировка указателей в списке. Все склеенные блоки (если есть) удаляются из списка свободных. Так как применяется двунаправленный список, операция удаления тривиальна. После этого освобождаемый блок добавляется в список свободных.

7. Утечка памяти. Сборка мусора.

Утечки памяти, определяемые как сбой при освобождении ранее выделенной памяти, — это одна из наиболее трудно обнаруживаемых ошибок в приложениях C/C++. Небольшая утечка памяти сначала может остаться незамеченной, но постепенно нарастающая утечка памяти может приводить к различным симптомам, от снижения производительности до аварийного завершения приложения из-за нехватки памяти. Более того, приложение, в котором происходит утечка памяти, может использовать всю доступную память и привести к аварийному завершению другого приложения, в результате чего может быть непонятно, какое приложение отвечает за сбой. Даже безобидная на первый взгляд утечка памяти может быть признаком других проблем, требующих устранения.

Отладчик Visual Studio и библиотеки времени выполнения C (CRT) предоставляют средства для обнаружения утечек памяти.

Основным средством для обнаружения утечек памяти является отладчик и отладочные функции кучи библиотеки времени выполнения C (CRT).

Чтобы включить отладочные функции кучи, вставьте в программу следующие операторы:

```
#define _CRTDBG_MAP_ALLOC

#include <stdlib.h>

#include <crtdbg.h>
```

Для правильной работы функций CRT операторы **#include** должны следовать в приведенном здесь порядке.

Включение заголовочного файла crtdbg.h сопоставляет функции **malloc** и free с их отладочными версиями, _malloc_dbg и **free**, которые отслеживают

выделение и освобождение памяти. Это сопоставление используется только в отладочных построениях, в которых определен **_DEBUG.** В окончательных построениях используются первоначальные функции **malloc** и **free.**

Оператор **#define** сопоставляет базовые версии функций кучи CRT соответствующим отладочным версиям. Если оператор **#define** не используется, дамп утечки памяти будет менее подробным.

После того как с помощью этих операторов будут включены отладочные функции кучи, можно поместить вызов **_CrtDumpMemoryLeaks** перед точкой выхода приложения для отображения отчета об утечке памяти перед завершением работы приложения:

_CrtDumpMemoryLeaks();

для обнаружения утечек памяти включает получение "снимков" состояния памяти приложения в ключевых точках. Чтобы получить снимок состояния памяти в заданной точке приложения, создайте структуру **_CrtMemState** и передайте ее функции **_CrtMemCheckpoint.** Функция поместит в структуру снимок текущего состояния памяти:

_CrtMemState s1;

_CrtMemCheckpoint(&s1);

Функция **_CrtMemCheckpoint** поместит в структуру снимок текущего состояния памяти.

Чтобы вывести содержимое структуры **_CrtMemState**, передайте ее функции **_ CrtMemDumpStatistics**:

_CrtMemDumpStatistics(&s1);

Функция **_ CrtMemDumpStatistics** выводит дамп состояния памяти

8. Препроцессор. Директивы препроцессора

Отличительной чертой языка Си++ является выполнение препроцессирования перед выполнением компиляции. Суть препроцессирования заключается в просмотре исходного файла с подготовкой его к компиляции. Программист может управлять процессом с помощью специальных директив препроцессора. Особенностью директив является то, что они действуют от места, где они описаны, до конца исходного файла. Директива может быть вставлена в любом месте программы.

Согласно стандарту языка, директива препроцессора начинается с символа # после которого через пробел (или несколько пробелов) следует ключевое слово. Рассмотрим назначение некоторых из них.

Директива include включить в исходный файл содержимого другого файла. Как правило, это библиотечные файлы с расширением .h или hpp. Синтаксис директивы:

include "имя-файла"

include <имя-файла>

include macro__identifier

В первом случае файл ищется в рабочем каталоге. Во втором - в каталоге, определенном в среде программирования. Как правило, это стандартные файлы в каталоге bc\include\. В третьей версии предполагается, что существует макроопределение, которое раскрывается в допустимое имя заголовочного файла.

Препроцессор удаляет директиву #include из текста файла с исходным кодом и направляет содержимое указанного в ней файла для обработки компилятором.

Директива define позволяет описать макро и имеет формат:

define *имя-макро* <*тело-макро*> или

define *имя-макро* (*формальные-аргументы*)<*тело-макро*>

Когда на этапе препроцессирования в тексте программы встречается *имя-макро,* то оно заменяется на последовательность операторов, указанных в *теле-макроса*. При использовании макроса с параметрами, они перечисляются в списке формальных аргументов.

Например:

```
# define Pi 3.1456

#define Mul(a,b) ((a)*(b))

#define Min(a,b) ((a)<(b)) ? (a) : (b)

int main ()

{ int x=10;

  int y=12;

  int z;

cout << Mul(x,y);
```

```
cout << Min(x,y);

return 0;}
```

При использовании макросов следует быть предельно внимательными и помнить, что на этапе препроцессирования тело макроса заменяет имя макроса в исходном файле. При создании макросов следует особенно внимательно следить за наличием скобок.

Примеры ошибок:

```
#define P =3.145  // вместо числа будет подставляться символьное // выражение =3.145

#define Mul(a,b) a * b

c=Mul(x+3, y+2); // при подстановке макроса будет выполняться

// неправильная последовательность арифметических операций.
```

Директива undef используется для отмены действия объявленного ранее макроса с текущей точки программы.

Группа директив ifdef, ifndef, elif, else, endif используется для управления формированием входного файла для компилятора.

```
#ifdef идентификатор1

текст программы

#elif идентификатор2          // может быть несколько

текст программы

#elif идентификатор3

текст программы

#else

текст программы

#endif
```

Если *идентификатор1* определен, то текст программы, следующий за директивой включается в входной файл компилятора, иначе он опускается и проверяется *идентификатор2.* В случае, если ни один из идентификаторов не объявлен, компилируется тест программы, идущий после директивы #else. Директива ifdef (if) всегда должна заканчиваться директивой endif.

Препроцессорные операции # и ##. Над формальными параметрами макро может быть выполнена одна из двух препроцессорных операций: операция образования строки (#) или операция объединения лексем (##).

Если знак образования строки предшествует формальному параметру в макросе, то соответствующий ему фактический параметр будет заключаться в кавычки и считаться строковым литералом. Все пробелы предшествующие первой лексеме и последующий за последней игнорируются. Каждая последовательность пробелов между лексемами заменяется на один пробел.

Пример:

```
#include <stdio.h>

#define stroka(x) printf(#x "\n")

int main()

{

stroka( Данные на   экран );

return 0;

}
```

Знак операции объединения лексем (##) позволяет объединить несколько лексем в одну. Как правило эти операции полезны на этапах отладки программного продукта.

Пример:

```
#define deb_session(x) printf("îòëàäî÷íûé ðåæèì: %d \n", data##x)

int data2=10;

deb_session(2);
```

Директива error позволяет направить в стандартный поток ошибок сообщение об ошибке. Используется для управления процессом компиляции на этапе отладки. Формат директивы:

#error *сообщение*

После вывода сообщения компиляция прекращается, а программисту выводиться строка, содержащая в себе указанное *сообщение*.

Обычно ее используют внутри условных директив. Вот типичный пример ее использования:

```
#if !defined(MYNAME)
```

#error Должна быть определена константа MYNAME

#endif

Директива pragma описывается с указанием имени "прагмы". Перед выполнение препроцессирования следует ознакомиться с набором прагм, поддерживаемых компилятором. В отличие от всех остальных директив, данная директива учитывается уже на этапе компиляции. Формат директивы:

#pragma *имя-прагмы*

Рассмотрим данную директиву на примере прагм startup и exit.

#pragma startup function-name <*приоритет*>

#pragma exit function-name <*приоритет* >

Данные прагмы определяют функции, которые будут выполняться при запуске программы (до передачи управления процедуре main) - startup и по завершении программы (перед вызовом функции ОС _exit) - exit. *Приоритет* указывает приоритетность вызова процедур (их может быть объявлено несколько). Диапазон приоритета может быть от 64 до 255 для процедур пользователя (наивысший приоритет - 0). Приоритеты 0-64 используются библиотеками Си++.

Вызываемая функция не должна возвращать значений и не может принимать параметров, поэтому объявляется как:

void function-name(void)

Ввод-вывод

Библиотеки языка Си поддерживают три уровня ввода-вывода: уровень потоков, ввод-вывод для консоли и порта и ввод-вывод нижнего уровня.

При организации ввода-вывода на уровне потоков все данные рассматриваются как поток отдельных байт. Для пользователя поток - это файл или устройство. Функции ввода-вывода для потоков позволяют обрабатывать данных различных размеров и форматов с буферизацией форматизованного или неформатизованного ввода-вывода.

Ввод-вывод потока позволяет:

- открывать и закрывать потоки;
- создавать и удалять временные потоки;
- читать и записывать символ;
- читать и записывать строки;

– читать и записывать форматизованные данные;

– читать и записывать неформатизованные данные;

– анализировать ошибки ввода-вывода потока и условия конца потока;

– управлять буферизацией и размером буфера;

– выгружать буфера, связанные с потоком;

– получать и устанавливать указатель текущей позиции в потоке.

Для работы с потоком необходимо определить указатель на структуру предопределенного типа FILE.

Рассмотрим работу с файлами на примере:

```
#include <stdio.h>
int main(void)
{
  FILE *in, *out;
  if ((in = fopen("\\AUTOEXEC.BAT", "rt"))
     == NULL)
  {  fprintf(stderr, "Cannot open input file.\n");
    return 1;
  }
  if ((out = fopen("\\AUTOEXEC.BAK", "wt"))
     == NULL)
  {  fprintf(stderr, "Cannot open output file.\n");
    return 1;
  }
  while (!feof(in))
    fputc(fgetc(in), out);
  fclose(in);
  fclose(out);
  return 0;
```

```
}
```

Все потоковые операции ввода-вывода буферизированны. Т.е. вместо того, чтобы по команде пользователя типа fputc() выполнять операцию записи байта в файл на диске, вывод осуществляется в буфер обмена. Буфер выгружается в следующих случаях:

– буфер заполняется;

– закрывается связанный с ним поток;

– принудительная выгрузка буфера с помощью специальных операторов;

– при успешном завершении программы вызывается специальное прерывание DOS, по которому буфер выгружается.

Обычно вывод выполняется через стандартные буферы ОС, которые не доступны пользователю, но Си++ позволяет назначать потоку буфера отличные от стандартных. Для этого используются функции типа setbuf.

Пример:

```
#include <stdio.h>

/* BUFSIZ is defined in stdio.h */

char own_buf[BUFSIZ];

FILE *fp;

int main(void)

{ fp=fopen(...);

...

  setbuf(fp, own_buf);

...

   fclose(...);

   return 0;

}
```

Если вместо указателя на массив в функцию setbuf передать null, то поток будет считаться небуферезированным и ввод-вывод будет осуществляться напрямую с диском.

Кроме ввода-вывода на уровне потоков имеется возможность управлять вводом-выводом на уровне дескрипторов файлов, т.е. на уровне ОС. Однако, на

этом уровне теряется полная совместимость со стандартами ANSI. Использование функций ОС для работы с файлами и устройствами ввода-вывода на уровне дескрипторов могут увеличить скорость выполнения программы, но повлекут за собой непереносимость исходного кода на другие платформы.

9. Механизм исключений. Обработка исключительных ситуаций.

Исключения - некоторое событие, которое является неожиданным или прерывает нормальный процесс выполнения программы.

Исключительные ситуации можно разделить на 2 основных типа:

1) синхронные - могут возникать только в определенных, заранее известных точках программах.
2) асинхронные - возникают в любой момент времени и не зависят от программной реализации

Для обработки используются 2 механизма

1) с возвратом
2) без возврата

В любом случае программа должно определить аварийную точку, выявить причину ошибки и передать управление соответствующему обработчику ошибки.

1) - обработчик устраняет и приводит программу к состоянию, в котором она продолжает работу по основному алгоритму
2) - после устранения - управление передается в заранее заданное место программы.

Существует 2 варианта подключения обработчика исключительных ситуаций к программе: структурная и неструктурная обработка.

Неструктурная – реализуется в виде механизма регистрации функции для каждого возникшего типа исключения. Системные библиотеки предоставляют процедуры: регистрация + удаление.
Независимо от того какая программа работает, вызывается последний зарегистрированный обработчик. С помощью такой обработки – анализ асинхронных исключений, для синхронных – она неудобна.

Структурная обработка – требует обязательной поддержки со стороны языка программирования.

Языковая конструкция содержит 2 блока:

1) защищаемый – блок, где может возникнуть ошибка.
2) Обработчик – операции при возникновении ошибки.

Если ошибка входит в исключения, то системная ошибка не генерируется. Обработчики исключений могут описываться по разному, но работают одинаково. При ошибке находится 1 подходящий обработчик, работа контролируемого блока завершается, и управление передается обработчику.
Try – начало защищаемого блока
Except – определяет выражение – фильтр, после которого реализуется блок обработчки исключения.

Внутри защищаемого блока исключение может быть специально вызвано. Для этого используется вызов leave - передает управление в конец защищаемого блока, прекращает дальнейшее выполнение основного алгоритма. При выполнении блока finally можно проанализировать код завершения защищаемого блока. Если защищаемый блок завершился без завершения исключения, то возвращается результат переменной AbnormalTerminantion. Если результат отличен от нуля, блок завершился без ошибок. Блок finally будет выполняться всегда кроме тех случаев, когда завершение программы вызвали системные функции: append и TerminateProccess. Использование SEH:

Плюсы:

1) позволяет отлавливать широкий спектр исключений – деление на ноль, переполнение стека

2) обработка исключений ведется на уровне ядра ОС

3) возможно использование исключений без подключения внутренних механизмов C++, что позволяет сократить размер программы и использовать функции ядра.

Минусы:

1) плохая совместимость с C++, т.к. данный механизм реализован на уровне ядра он не воспринимает внутренних структур C++: не видит созданных классов и не освобождает выделенную им память

2) SEH не вызывает деструктуры пользовательских классов

3) невозможно использовать одновременно механизм SEH и стандартное исключение Си.

Стандартное исключение C++ реализовано через механизм CRTL. Данный механизм принято считать окончательное моделью управления. После того, как исключение произошло, обработчик исключений не может потребовать, чтобы произошел возврат управления в точку возникновения ошибки. Это приводит к тому, что с помощью CRTL могут обрабатываться только синхронные исключения – исключения, происходящие внутри программы, аппаратные и асинхронные исключения не поддерживаются. В языке C++ практически любое состояние достижимое в ходе выполнения вычислительного процесса может быть определенно как исключение. Они заданны в языке для того чтобы дать возможность программисту динамически отрабатывать возникшие ситуации. Анализ и обработка исключений переносится из точки возникновения в специально предназначенные для обработки блоки, при этом возможна передача обработчику набора необходимых параметров, который характеризуют возникшую ситуацию. При CRTL выделяются 2 блока:

1) Защищенный блок после try
2) Блок обработчика после catch

Внутри блока обработчика может задаваться повторная генерация исключения с помощью вызова throw. При возникновении ошибки в любом из операторов защищаемого блока управление передается соответствующему обработчику блока catch при этом генерируется исключение конкретного типа и кода. После выполнения соответствующего обработчика управление может быть передано в завершающую точку программы. В случае если исключение не происходит управление передается оператору расположенному за блоком catch. Если внутри блока catch используется операция throw то происходит повторное возбуждение исключения. Текущая функция завершается и исключение передается обработчику верхнего уровня. Т.О. используется структурная вложенная обработка исключений, которая позволяет для каждого исключения, возникшего внутри контролируемого блока, найти обработчик либо внутри

текущей функции либо выше по списку вызовов. В наихудшем случае обработку исключения выполнит ОС.

В C++ используется несколько форм обработчика исключений:

1. Полная форма, когда после ключевого слова catch (тип имя) {обработчик}
   Данная форма используется для обработки исключений, если известно состояние переменных в момент генерации исключений.
2. Catch (тип) {обработчик}
   Используется для обработки исключений, для которых известен только факт возникновения исключений (деление на 0)
3. Catch (…) {обработчик}
   Перехватывает любое исключение возникающее в программе.

Одновременно в рамках одной функции может использоваться несколько обработчиков catch, при этом выбор соответствующего обработчика осуществляется путем последовательного просмотра. Следовательно, наиболее детализированные обработчики должны определяться раньше обработчика без параметров.

Генерация исключения может быть выполнена в любой точке программы. Для этого используют ключевое слово throw, после которого определяют тип возбуждаемого исключения. В этом случае исключение формируется как статический объект, значение которого определенно выражением генерации. Копия объекта передается за пределы контролируемого блока и инициирует переменную, используемую в спецификации обработчика исключений. Копия объекта сохраняется до тех пор, пока не будет начата обработка данного исключения. В ряде случаев используется вложение контролируемых блоков, тогда преимущественно обработка исключений должна выполняться на внешнем уровне. При этом желательно использовать ретрансляцию исключений (catch (…)).

Иногда в программе приходится отслеживать непредвиденно или не специфицированное исключение. В этом случае используют функцию unexpected. ??? Вызов terminate приводит к корректному завершению работы программы. Функция unexpected с помощью функции установки может определить собственный обработчик, который будет вызываться, если программа выбрасывает не специфицированные исключения. В качестве

параметра функция set определяет адрес функции обработчика. Если функция unexpected не определенна любое непредвиденное исключение будет передано в ОС.

10. Основы языка Assembler. 11. Интерфейс Assembler с языками высокого уровня.

Базовая архитектура процессора, определяющая его программную модель, состоит:

1) из 8 регистров общего назначения – используются для хранения данных и указателей;
2) сегментные регистры – хранят 6 селекторных сегментов (SS – сегмент стека, DS – сегмент данных, CS – сегмент кода, ES,FS,GS расширенные регистры);
3) регистр счетчик команд – определяет адрес выполняемой команды программы;
4) регистр управления и контроля за состоянием программы и процессора (регистр флагов);
5) режим адресации данных и команд.

Любой из регистров общего назначения может быть использован для хранения операндов при выполнении арифметических и логических операций, а также для вычисления адресов и в качестве указателей на переменные в памяти. Однако необходимо учитывать специфику некоторых регистров:

- ESP служит для хранения вершины указателя стека при вызовах и возвратах подпрограмм.
- EAX является регистром аккумулятором и при выполнении ряда арифметических операций содержит результат.
- EBX указывает на данные находящиеся в сегменте данных и доступные через селектор DS.
- ECX регистр счетчик циклов используется для автоматического подсчета количества итераций программы.
- EDX является указателем на порты устройства ввода-вывода.

- EDI и ESI используются для организации индексной адресации к данным доступным через сегмент DS. EBP используется в качестве базы для выполнения базовой адресации и содержит указатель на данные находящиеся в стеке.

Регистр флага – 32 разрядный регистр, каждый разряд которого может быть проанализирован в ходе работы программы.

- CF инициирует перенос или знак при выполнении арифметических операций, а так же служит индикатором ошибки при обращении с системным функциям.
- PF устанавливается в 1, если младшие 8 бит результата содержат четное число 1.
- AF флаг вспомогательного переноса используется в операциях над упакованными 2-10 операндами, инициирует перенос в старшую тетраду или заем старшей тетрады.
- ZF устанавливается в 1, если результат операции равен 0.
- SF показывает знак результата операции.
- DF флаг направления используется особой группой команд предназначенных для изменения направления чтения адресов от младших к старшим D=1, наоборот D=0. OF фиксирует переполнение в 1, если результат выходит за пределы разрядной сетки.

Регистр счетчик команд хранит номер текущей выполняемой команды и инкрементируется автоматически при выполнении текущей операции. Принудительно изменение счетчика команд выполняется с помощью команд переходов (условных или безусловных) команд вызовов подпрограмм и команд возврата подпрограмм. Для получения операнда используемого внутри операции необходимо определить его эффективный адрес. В самом общем случае эффективный адрес операнда определяется как адрес базы + индексная адреса* размер текущего операнда + смещение внутри сегмента данных.

В зависимости от используемых элементов данной конструкции определяют различные способы адресации. В настоящее время известно более 20 способов адресации.

Все способы адресации можно разделить на 2 класса:

1. Прямая когда получают непосредственный операнд

2. Непрямая когда для получения доступа к операнду необходимо выполнить вычислительные операции.

Прямая адресация

1. Непосредственные операции предполагают, что в качестве одного из операндов указывается непосредственное значение.
2. Строгая адресация предполагает, что операнды используемые в операции находятся в указанных регистрах.
3. Неявная адресация предполагает, что операнд команды хранится по адресу, определенному кодом самой команды. Абсолютна адресация в качестве операнда – непосредственный адрес памяти.

Непрямая адресация

1. Регистровая в указанном в качестве операнда регистре хранится адрес требуемого операнда.
2. Косвенная предполагает, что один из операндов является адресом места расположения операнда
3. Базовая адресация предполагает, что адрес одного из операндов команды формируется база+смещение. База обычно определена через базовый регистр BP.
4. Индексная адресация используется для организации перемещения по элементам массива. Работает по принципу база+индекс*множитель. Сдвиг зависит от длины текущего операнда. SI хранится адрес относительно которого выполняется индексный сдвиг.
5. Стековая адресация:

Команды передачи данных: move(операнд приемник, операнд источник). В качестве операнда приемника не может быть использован косвенная адресация, в случае если косвенно определен операнд источник. При выполнении операции передачи данных необходимо отслеживать соблюдение размерности.

Команды с портами ввода-вывода: in позволяет взять информацию из указанного порта и записать в указанный регистр, out записывает информацию в указанный порт ввода-вывода. Может быть связан напрямую регистр DX или указан непосредственный адрес.

Команды выполнения арифметических операций: арифметические команды:

1) двухоперандные командные команды + и –(ADD и Sub) складывают (вычитают) второй операнд с первым с записью результата первый операнд при этом необходимо отслеживать соответствие размерности операндов
2) одноадресные команды Inc и Dec увеличивают или уменьшая операнд на 1, сохраняя результат
3) целочисленное умножение MUL и деление без знака DIV, результат операции в зависимости от размера записывается в регистр AX, если операнд является байтом, регистровую пару DX,AX, если операнд является словом, или а регистровую пару EX,DX, если операнд занимает 32 бита, при выполнении деления остаток записывается в регистр DX. При делении если операнд является байтом, то частное записывается следующим образом: целая часть в регистр AL,а остаток от деления в регистр AH
   умножение и деление целых чисел со знаком IMUL и IDIV выполняет умножение и деление целых чисел с записью результата, команда IMUL может работать с одним, двумя или тремя операндами. В случае работы с одним в качестве второго – аккумулятор или регистровая пара EAX,EDX.

Команды выполнения логических операций: AND – выполняет логическое умножение операнда первого на операнд второй с сохранением результата в операнд первый, OR – операция логического сложения операнда первого на операнд второй с сохранением результата в операнд первый, XOR, NOT – побитовое отрицание.

Команды передачи управления: 1) условный переход

- по равенству (J)

- по неравенству (JN)

Выполняется анализ соответствующих флагов регистра флагов.

JZ переход если флаг Z установлен в состояние 1

JC переход если флаг CF установлен в состояние 1

JS переход если флаг SF установлен в состояние 1

JO переход если флаг OF установлен в состояние 1

Возможно??? после их предварительного сравнения. Эквивалентные переходы по флагам. Переход JE если операнды равны соответствует JZ, JNE – если операнды не равны = JNZ, JA – переход если первый операнд больше второго =JNBE переход если не меньше и равно, JB переход если первый операнд меньше второго = JNAE, JNA- переход если не больше = JBE, JNB – переход если не меньше = JAE.

CMP - занесение во флаговый регистр без занесения результатов операций.

Команда перехода по счетчику LOOP выполняет переход по указанному адресу в качестве которого выступает метка с декрементом значения содержимого CX, если CX=0, то переход осуществляется на следующую за LOOP команду.

Команда CALL ADDR осуществляет передачу управления подпрограмме, имя которой указанно в качестве адреса.

RET осуществляет возврат из подпрограммы в точку вызова.

3) безусловный переход JMP – осуществляет передачу управления по указанному адресу, в качестве адреса определяется метка программы (L1:mov bx,ax)
4) команды вызова процедур

Команды обработки стека:

1) PUSH – записывает в стек указанный операнд
2) POP – извлекает из стека указанный операнд

Команды загрузки эффективного адреса LEA осуществляет вычисление эффективного адреса второго операнда и записывает его в первый операнд.

Команды сдвига:

1) арифметический SAR(SAL) первый – само число, второе – количество сдвиговых разрядов
2) логический SHR(SHL) два операнда: беззнаковое число, подлежащее сдвигу, второй- количество сдвиговых знаков
3) циклический RCR(RCL) осуществляет сдвиг через флаг переноса ROR(ROL) сдвиг первого операнда на заданное количество разрядов с записью выдвинутого бита в флаг CF

В ряде случаев необходимо выполнять временное изменение типа переменной, для этого используется конструкция type Ptr. Параметр TYPE может иметь одно из следующих значений: byte(1), word(2), dword(4), qword(8), tbyte(10) – позволяет взять значение по указанному адресу приведенное к соответствующему типу.

Для определения констант в программах используют псевдокоманды DB – byte, dw – слово, dd – двойное слово.

Сравнение операндов TEST выполняет поразрядное сравнение операндов без сохранения результата с изменением флага.

Каркас программы

приложение написанное под win32 выполняется в защищенном режиме при этом ОС запускает каждое приложение в отдельном виртуальном адресном пространстве, это означает что каждому приложению будет выделено собственные 4 Гб памяти. Тогда нет необходимости беспокоиться о модели памяти и используемых сегментах. Т.к. всегда будет использоваться плоская модель памяти FLAT.Сегментные регистры можно использовать для адресации к любой точке выделенной памяти. Существует правило о внутреннем использовании регистров ESI,EDI,EBP, поэтому в случае применения их внутри программы необходимо сохранение их предыдущего значения в стек с последующим восстановлением. Каркас программы на ASSEMBLER состоит из следующих частей:

1)заголовок MODEL.FLAT.STDCALL

2)

3)директива правила вызова

4)секция описания данных

.data – содержит набор инициализируемых данных

.data? – определяет список используемых переменных без их предварительной инициализации

.const – содержит определение процедур или констант программы.

5) блок кода .code<метка>proc<код><метка>endp

Метка может иметь произвольное значение. Она устанавливает границу кода.

При вызове ассемблерных процедур из программ написанных на языках высокого уровня существует ряд общих правил и соглашений. Эти правила и соглашения принято называть «Соглашениями вызова».

1. расположение входных параметров подпрограммы и возвращаемых подпрограммой значений
   а) параметры и значения в регистра общего назначения
   б) в стеке
   в) в динамической памяти

2. Порядок передачи параметров. При использовании для параметров стека определяется в каком порядке передаваемое значение помещается в стек. При использовании регистра определяется порядок псопоставления регистров и параметров.

а) при прямом порядке – параметры размещаются в стек в том же порядке, в котором они перечислены при описании подпрограммы

+ единообразие написания кода

б) обратный порядок – передаются в порядке противоположном представленным в заголовке

+ адрес возврата всегда находится на вершине стека, за которым следует первый парметр передаваемый подпрограмм. Позволяет использовать с неизвестным числом параметров и параметрами по умолчанию

4. Определяется, кто восстанавливает стек по окончанию вызываемой подпрограммы.

В случае если стек восстанавливает вызываемая подпрограмма сокращается объем команд, т.к. команды восстановления стека записываются только один раз в конце подпрограммы. Если за восстановление стека отвечает вызывающая подпрограмма – вызов затрудняется,но облегчается использование подпрограммы с неизвестным типом и числом параметров.

3. Содержимое каких регистров процессора подпрограмма обязана восстановить перед возвратом.

Соглашение вызова зависят от архитектуры целевой машины и компилятора. Способ передачи параметров может быть изменен путем внесения соответсвующих директив определяющих соглашение вызова и способы синхронизации стека при использовании подпрограмм.

Существует 4 основных директивы:

1. fastcall определяет передачу параметра слева направо, за очистку стека – вызываемая подпрограмма, восстановлению подлежат eax, edx, ecx(pascal); ecx, edx (C++) регистры.
2. pascal – передача параметров слева направо, за очистку стека – вызываемая подпрограмма, доп.регистры не восстанавливаются.
3. cdecl – передача справа налево, за восстановление стека отвечает вызывающая программа, доп.регистры не восстанавливаются.
4. stdcall – определяет передачу справа налево, за восстановление стек – вызываемая подпрограмма, доп.регистры не восстанавливаются.

Для возврата значений

Если возвращаемое подпрограммой значение не превышает одного байта, оно возвращается через регистр al, до 2х байт через регистр-аккумулятор ax, до 4х байт eax, в случае если возвращаемое значение превышает 4 байта, возврат через участок динамической памяти, адрес начала которого сохраняется в регистровой паре eax,edx.

Пусть имеется некоторая процедура myproc, для которой определена директива вызова stdcall. В этом случае при входе в подпрограмму стек распределится следующим образом: на дне параметр 2, далее 1 и адрес возврата. Тогда при выходе из подпрограммы необходимо восстановить стек. Выход из подпрограммы определяется командой ret. Адрес возврата извлекается автоматически при выполнении команды ret если указать целочисленное значение после команды ret, оно воспримется как число байт, на которое нужно сдвинуть указатель стека в сторону старших адресов. Тогда используя stdcall вызов процедуры раскладывается на 3 команды: занесение в стек параметров и непосредственный вызов процедуры. В случае если не указана директива stdcall используется стандартный вызов в C++ cdecl.

При организации вызовов программ написанных на языку ассемблере из программ написанных на языках высокого уровня используют понятие

стекового фрейма. Он создается всякий раз при вызове подпрограммы. Для создания стекового фрейма необходимо выполнить следующие действия:

1) поместить аргументы в стек;
2) вызвать процедуру с помощью команды call в результате чего в стек помещается адрес возврата вызывающую подпрограмму;
3) сохранить в стек регистр ebp, т.к. в дальнейшем он должен использоваться для организации доступа к параметрам вызываемой подпрограммы;
4) загрузить в регистр ebp текущий указатель вершины стека из регистра esp.

В общем случае стековым фреймом или записью активизации называют область памяти в стеке, расположенную за адресом возврата из подпрограммы. Предназначен для хранения параметров заранее сохраненных в регистр. На структуру оказывает влияние модель фрейма и соглашение вызова.

Пусть необходимо организовать вызов функций, вычисляющих разность 2х целочисленных значений, передаваемый в качестве параметров и возвращающей целочисленный результат. Вызов будет осуществлен из программы, написанной на C++. Т.к. вызываемая подпрограмма является внешней по отношению в вызываемой подпрограмме необходимо ее предварительное объявление с ключевым словом extern. Наличие директивы «C» определяет, что в качестве соглашения вызовов определен cdecl.

```
.MODEL FLAT,C

.CODE

razn proc a:adword, b:dword

…..
```

В случае использования среды visualstudio при организации вызова из ассемблерных процедур среда автоматически осуществляет подготовку стекового фрейма.

Пусть необходимо создать процедуры суммирования двух чисел с возвратом результата через третий параметр. Процедура является внешней и должна быть предварительно описана директивой extern.

В ряде случае необходимо предусмотреть возможность временного хранения локальных значений вызываемой подпрограммы в стеке.

12. Основы программирования для Windows. Особенности ОС Windows. Типы многозадачности.

ОС Windows является многопользовательской системой, что позволяет одновременно выполнять несколько программ => при написании программ следует учитывать наличие нескольких активных копий одной и той же программы.

К особенностям ОС Windows относят:

1) особенность построения файловой системы. С Windows95 реализована файловая система, использующая таблицу размещения файлов защищенного режима. Применяются длинные имена. Для обеспечения совместимости с короткими имена применяются псевдонимы. Программа может создать их отображение на виртуальное пространство размером до 1 Гб. После этого с содержимым можно работать как с объектом ОП, используя указатели. Файл может быть доступным нескольким приложениям => возможен обмен между приложениями.
2) особенность управления программами. Программы имеют возможность влиять друг на друга и обмениваться информацией посредством многозадачности.

Многозадачность - свойство ОС обеспечивать возможность параллельной или псевдопараллельной обработки нескольких процессов.

Истинная многозадачность возможна только в распределенных. ??????

Примитивная многозадачность обеспечивается путем чистого распределения ресурсов; за каждой задачей закрепляется участок памяти;

Развитые многозадачности распределенных ресурсов - динамические, то есть в момент когда задача стартует - выделяется память, заканчивает - очищается.

Такая среда характеризуется особенностями:

1) каждой задаче присваивает со собственный приоритет, на основании которого выделяется процессорное время и память;
2) система организует очередь задач, так чтобы каждая задача получила процессорное время доя выполнения
3) по окончании времени ядро ОС временно переводит активную задачу из состояния выполнения в состояние готовности; ресурсы передаются другим задачам;

1) Нехватка памяти - страницы невыполненных задач вытесняются на диск, а потом восстанавливаются.
4) Система распознаёт сбои и зависания отдельных задач и может прекратить их;
5) система решает конфликты доступа к ресурсам и устройствам, не допуская тупиковых состояний, приводящих к блокировке системы;
6) система гарантирует что каждой задаче будет выделено время;

В настоящее время известно 3 типа многозадачности:

Невытесняющая - при которой ОС одновременно загружает в память 2 или более приложений, но процессорное время выделяется 1 из них. Другие задачи переводятся в фоновый режим. По истечении кванта времени следующая фоновая задача захватывает квант времени

Совместная - многозадачность, при которой следующая задача выполняется только после того как текущая задача явно объявит себя готовой отдать процессорное время. При ней приложение фактически захватывает столько времени, сколько нужно;

+ отсутствует необходимость защищать разделяемые структуры памяти

- в случае ошибки в активное приложении - вероятность сбоя системы в целом

Вытесняющая - при которой ОС самостоятельно передает управление между отдельными задачами. При данном виде процессор переключается с выполнения 1 задачи на другую без извещения задачи. Распределение планировщиком на основании приоритетов.

+ надежность системы. Возможность полного использования многопроцессорного и многоядерного режима

- нужны объекты синхронизации

14.DLL- библиотеки

Библиотеки динамической компановки являются основным компонентотм многих приложений windows. Сама ОС, ее драйверы и ряд расширений также оформлены в виде dll библиотек. Основными причинами применения dll являются:

1. расширение функциональных возможностей приложений. Динамическая библиотека загружается в адресное пространство процесса по необходимости что позволяет экономить ресурсы.

2. возможность использования разных языков программирования при создании приложения. dll библиотекам подключается как готовый откомпилированный файл, представляющей лишь интерфейсы для собственного использования, что не накладывает ограничений на язык разработки самой библиотеки и использующих ее приложений.
3. более простое управление проектами. если в процессе разработки проекта участвует группа разработчиков то каждый из модулей может быть оформлен в виде динамической библиотеки. Это по требует минимальных знаний для объединения модулей. В частности определения межмодульного интерфейса.
4. применение dll позволяет экономить память засчет возможности использования единственного экземпляра загруженной библиотеки несколькими запущенными приложениями.
5. разделение ресурсов dll могут содержать шаблоны диалоговых окон, строки и прочие общие ресурсы.
6. упрощение локализации. Например, приложение содержашее код без компонентов пользовательского интерфейса может подгружать библиотеки с интерфейсными элементами.
7. расширение специфических возможностей. Расширение функциональных возможностей windows осуществляется с помощью com объектов которые в основном оформляются в виде динамических библиотек, это же относится к web компонентам, avtive x.

Создание исполняемого файла состоит из последовательности шагов:

1. подготовка текста на выбранном языке программирования.
2. трансляция модулей программы и формирование объектных модулей.
3. компоновка модулей приложения с формированием единого исполняемого exe модуля.

На этапе компоновки могут быть включены библиотечные модули при этом выделяют 2 вида компоновки:

статическую: внедрение функций определенных в библиотеке и используемых основным приложением осуществляется в исполняемый модуль непосредственно на этапе компоновки. Т.о. библиотечная функция встраивается в каждое из использующих ее приложений, что приводит в не эффективному использованию ресурсов, в частности памяти. Однако

статическая компоновка имеет ряд преимуществ, а именно не требует дополнительной информации об используемых и подключаемых модулях при передаче исполняемого файла заказчику. Для много задачных ОС характерная ситуация когда в памяти размещается только одна копия функции, а все «параллельно работающие приложения» обращаются к ней. такая задача решается с помощью динамической компоновки.

динамическую: код нескольких функций объединяется в отдельный файл, который загружается в ОП по запросу какого либо приложения. Каждое следующее приложение проверяет наличие загруженной библиотеки и, если та существует в ОП, непосредственно использует требуемые функции. В этом случае в исполняемый файл приложения встраивается лишь ссылка на вызываемую функцию.

Для того чтобы приложение могло вызвать функцию, содержащуюся в библиотеке, образ библиотечного файла должен быть спроецирован в адресное пространство вызывающего процесса. Это достигается засчет динамического неявного связывания во время загрузки программы или раннее связывания. Второй вариант использования явного динамического связывания во время выполнения программы или позднего связывания. Как только dll спроецировано в адресное пространство основного процесса ее функции становятся доступными всем потокам данного процесса. Для потоков код и данные dll являются дополнительной информацией принадлежащей родительском процессу. Когда поток вызывает из dll функцию она считывает параметры из стека потока и размещает их в собственной области локальных переменных. В этой связи говорят, что библиотечная функция работает в контексте вызвавшего ее потока.

Структура любого приложения состоит из следующих блоков:

каждое приложение при запуске размещается в ОП, где создается копия приложения или процесса. Каждая копия приложения имеет собственную область статических и динамических данных, собственный стек и собственную очередь сообщений. Процесс является контекстом, в котором выполняется модуль приложения, содержащий непосредственный код приложения и ресурсы. ОС хранит БД модулей приложений, в которой содержится информация о ссылках на экспортируемые функции, на расположение кода и ресурсов. Т.к. модуль приложения не изменяются то

все копии приложения или процессы могут им пользоваться. Для каждого процесса создается виртуальный процессор состоящий из регистров и областей для размещения данных стека и очереди сообщений. В отличие от приложения dll библиотека является только модулем размещаемым в памяти однократно. Она включает в себя код, ресурсы и сегмент локальных данных, не имеет стека и очереди сообщений. Так же как и модуль приложений библиотека загружается в память однократно.

Неявное связывание это динамическое связывание во время загрузки программы, когда код приложения ссылается на идентификаторы, содрежащиеся в dll и тем самым заставляет загрузчик загружать нужную dll при запуске приложеиня. Для упрощения будем считать что исполнячемый модуль импоритрует функции из dll, а dll библиотека экспоритрует их. Собирая исполняемый модуль который импортирует функции из dll нужно создать dll библиотеку для этого осуществляется следующий порядок действий:

1. заголовочные модули с прототипами функций струтурами и идентификаторами экспортируемых данных оформляются и присоединяются к исходному тексту приложений.
2. модули исходного текста вызывают функции определенные в библиотечном модуле.
3. осуществляется компиляция исходного кода библиотеки в объектный модуль.
4. осуществляется компоновка в единый загрузочный модуль, в котором помещается в конечном итоге двоичный код и переменная относящиеся к dll. Скомпонованный модуль используется при создании исполняемого модуля приложения.
5. если компоновщик обнаруживает, что dll экспортирует хотя бы одну переменную ил функцию, то создается отдельный lib-файл. Он содержит список символьных имен функций переменных экспортируемых из dll.
6. все модули исходного кода, где есть ссылки на внешние функции и структуры подключают ранее созданные заголовочные файлы библиотек.
7. осуществляется компиляция исходных модулей с подключенными библиотеками.
8. осуществляется преобразования исходных модулей в объектные модули.

9. компоновщик собирает все объектные модули ТВ единый загрузочный при этом для каждой dll указывается на какие символьные имена ссылается двоичный код исполняемого модуля.
10. загрузчик ОС создается виртуальное адресное пространство для процесса и проецирует на него исполняемый модуль.
11. анализируется раздел импорта, определяются необходимые библиотечные модули, которые также проецируются в пространство процесса. После того как проецирование завершено, начинается выполнение работы процесса.

Раздел импорта – это автоматически создаваемая область, куда помещаются относительные виртуальные адреса каждого идентификатора размещенного внутри модуля dll. При этом каждый идентификатор получает однозначное имя hint, по которому может быть осуществлено обращение к данному элементу библиотеки. При создании исходного исполняемого модуля определяют внешнюю функцию, которая будет загружаться из библиотеки. компоновщик создает exe модуль и создает в нем раздел импорта. Осуществляется вызов функций.

Явное связывание: требуемая dll явно должна быть загружена при запуске того или иного потока в процессе. Поток загружает библиотеке в собственной адресное пространство, получает адрес библиотеки, адрес функцию внутри библиотеке. Вызывает функцию уже по адресу это библиотеки. Происходит в работающем процессе

1. формируется заголовочный файл с экспортируемыми структурами.
2. создаются исходные файлы в которых реализуются экспортируемые элементы.
3. компилятор для каждого исходного фала создает объектный модуль.
4. на основе модулей компоновщик создает единый модуль библиотеки.
5. параллельно создается lib модуль который при явном связывании не используется.
6. определяется заголовочный файл в исходном модуле приложения для связи с экспортируемыми элементами.
7. создается исходные модули приложения, не имеющий ссылок на импортируемый элемент.
8. для каждого модуля создается объектный модуль.

9. осуществляется сборка исполняемого модуля на основании набора объектных.
10. загрузчик ОС создает виртуальное адресное пространство для процесса и начинает выполнение процесса.
11. поток в рамках процесса осуществляет загрузку требуемой функции dll в свое адресное пространство и обращается к этой функции.

В случае использования явного связывания приложение должно загрузить библиотеку, содержащую импортируемую функцию.. Это осуществляется с помощью функции loadlibrary. В качестве параметра функции loadlibrary передается полное имя загружаемой библиотеки. В качестве результат функция loadlibrary возвращать идентификатор загруженной библиотеки если она существует и может быть размещена в ОП. Если библиотека не найден в рамках ОП и по указанному адресу возвращает значение null. Загруженная библиотека проецируется в адресное пространство процесса и получает виртуальный адрес. Для обращения к требуемой функции необходимо получить адрес функции в рамках требуемой библиотеки. Это осуществляется с помощью функции getprocaddress. Она передает в качестве аргумента имя загруженной библиотеки и имя или номер вызываемой функции. Если функция в рамках библиотеки обнаружена то возвращается точка ее кода, в противном случае возвращается значение null.

dll библиотека по структуре напоминает модуль приложения. Основной функцией dll библиотеки является функция dllmain. Она определяет точку входа в библиотеку. Вызывается каждый раз когда выполняется инициализация процесса обращающегося к функциям библиотеки при неявном связывании или осуществляется явная загрузка библиотеки с помощью loadlibrtary.

hinstancedll – идентификатор библиотеке, пир обращении к ресурсам расположенным в файле библиотеки.

dword – код причины вызова функции. данный параметр может быть определен следующими константами: загрузка библиотеки, аварийный ил не аварийный выход из процесса использующего данную библиотек, резервный (null, кроме 2х случаев: если причиной запуска dll является аварийное завершение задачи и функция dll вызвана по причине выгрузки библиотеки из памяти/).

Для каждого библиотечного файла необходимо выполнять согласование импортируемых элементов с экспортируемыми. Согласование осуществляется

на основе таблицы экспорта. В ряде случаев вручную может быть создан дополнительный файл определений влияющий на таблицу экспорта. В ФАЙЛЕ ОПРЕДЕЛЕНИЕЙ ЗАДАЕТСЯ НАБОР ЭКСПОРТИРУЕМЫХ БИБЛИОТЕКОЙ структур после ключевого слова экспорт. предложение файла определений имеет следующий формат

имя точки входа, т.е. имя под который экспортируемая структура будет доступна для вызова

внутренне имя – имя с которым данный элемент определен внутри dll библиотеки.

@номер – номер который присваивается данному элементу в библиотеке экспорта.

директива noname – данный элемент осуществляет обращение к себе только по порядковому номеру, имя экспортируемого элемента становится невидимым.

constant – позволяет экспортировать из dll не только функции но и данные при этом имя точки входа задает имя экспортируемой глобальной переменной.

15. Основы ООП. Объектная модель.

Парадигма программирования это способ создания программ с помощью определенных принципов и подходящего языка позволяющего писать ясные программы.

- структурное программирование – это методология разработки ПО, в основе которой лежит представление программы в виде иерархической структуры блоков, каждый блок является элементом последовательности ветвления или цикла.

- функциональное программирование – парадигма программирования в которой процесс вычисления трактуется как получение значений некоторой функции в математическом ее понимании. Любая функция считается вычислимой, если на ее входе имеются все необходимые для полного определения данные. Никаких дополнительных условий срабатывания не требуется, выходные данные зависят только от входных. Это позволяет при выполнении программ на функциональных языках кешировать ряд результатов и вызывать в порядке неопределенным алгоритмом – срабатывание по готовности.

- логическое программирование – основано на логическом доказательстве теорем и правил вывода, на математической логике. Чаще всего факты описаны в логике высказываний или предикатов и объединены в виде правил «ЕСЛИ А, ТО В», где А и В факты присутствующие в системе (Пролог – метод резолюций).

- автоматное программирование – парадигма программирования, при использовании которой программа или ее фрагмент представляется как модель формального автомата, в зависимости от конкретной задачи в автоматном программировании может использоваться как конечный автомат, так и автомат более сложной структуры.

- объектно-ориентированное программирование – основные понятия: объект и класс. Программа представляет собой взаимодействие активных объектов, каждый объект принадлежит к ранее заявленному типу.

- событийно-ориентированное программирование – выполнение программы определяется событиями. Под событием понимается действие пользователя, сообщение других программ или потоков, команды ОС. При этом программа представляет собой набор обработчиков известных событий.

- агентно-ориентированное программирование - основные концепции: агент и его поведение. Агентом является всё, что может воспринимать среду через набор реальных или откатных датчиков, а также 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

В качестве основных конструктиврованных элементов используют не алгоритмы, а объекты. Каждый объект – экземпляр определенного класса. Классы образуют иерархию на основе принципа наследования, т.е. дочерние классы включают в себя все элементы присущие родительским, дополняя и расширяя собственную функциональность. Программа считается объектно-ориентированное тогда и только тогда, когда выполнены все три приведенные ранее условия. Основой объектно-ориентированного стиля является – объектная модель.

Состоит:

- абстракция – необходимо выявлять существенные характеристики некоторого объекта, отличающие от других видов объектов и таки образом четко описывающие концептуальную модель объекта с точки зрения конкретного наблюдателя. Различают процедурную абстракция (объект, являющийся полезной моделью некоторой сущности предметной области; предполагает, что объект состоит из множества операций, каждая из которых выполняет требуемые функции) и абстракцию данных.

- инкапсуляция – представляет собой процесс разделения элементов абстракции определяющий ее структуру и поведение. Предназначена для изоляции конкретных обстоятельств абстракции от их реализации.

- модульность – свойство системы, разложенной на слабо связанные элементы.

- иерархия – ранжирование или упорядочивание абстракции, реализованное на основе наследования. Наиболее удаленные от исходной точки абстракции имеют более сложную структуру. Элементы иерархии связаны между собой отношением «родительский элемент является встроенным элементом для дочернего».

- контроль типов – это правило использования объектов, не допускающее взаимную замену объектов разных классов.

- параллелизм – свойство отличающее активные объекты от пассивных. Активные объекты могут работать независимо друг от друга обмениваясь некоторыми данными или сообщениями. Пассивные объекты способны только принимать сообщения не оказывая влияния на другие абстракции системы.

- персистентность – способность объекта преодолевать временные рамки, т.е. продолжать свое существование после исчезновения своего создателя или выходить за пределы адресного пространства.

Преимущества:

1. использование объектной модели стимулирует повторное применение не только кода, но и проектных решений.
2. использование объектной модели приводит к созданию систем с устойчивыми промежуточными формами, что упрощает их изменение. Это позволяет со временем улучшить систему, не прибегая к ее полной переработке.

3. уменьшаются риски с проектированием сложных систем, т.к. сложная система может проектироваться путем наращивания функционала более простых систем.
4. объектная модель учитывает особенности процесса познания человека (от более простого к более сложному).

Основным элементом объектно-ориентированного программирования является объект. Объект можно представить как некоторую сущность имеющую хорошо понятное поведение. В этой связи любой объект характеризуется состоянием, поведение и индивидуальностью.

Состояние объекта определяет набор его статических свойств, имеющих динамические значения, через которое определяется индивидуальность объекта.

Поведение – это набор функций, определяющих состояние объекта и возможность передачи сообщения другим объектам.

Индивидуальность – это элемент, однозначно определяющий объект.

Класс – это множество объектов, имеющих общую структуру и общее поведение.

Объект – это конкретная реализация элемента общего класса (имеет идентификатор).

С точки зрения языка C++ всякий класс определяется как структура, содержащая набор данных и набор функций, предназначенных для операций над этими данными. При этом данные принято называть член-данные, а операции – член-функции.

3 раздела, определяющие области видимости элементов класса:

1. public – все описанные член-данные и член-функции являются глобальными
2. protected – доступны только внутренним операциям класса и операциям его прямых наследников.
3. private – доступны только элементам данного класса.

Прародителем классового типа является структурный тип, который также может содержать набор данных и операций, но в отличие от типа класс все

элементы структуры являются обще доступными. Т.о. в структуре отсутствует принцип инкапсуляции.

Считается правилом хорошего тона объявлять все член-данные класса как private. Это обеспечивает:

1. при необходимости корректировки данных в определенных классах, изменения нужно внести только в член-функциях этого класса.
2. обеспечивает защита член-данных от несанкционированного изменения и несанкционированного доступа.

При объявлении класса вводится новый тип данных. При объявлении объекта выделяется память под размещение структуры объекта, причем память выделяется только под член данные объекта для заполнения которых необходимо выполнять инициализацию объекта. Инициализация объекта выполняется через вызов специальной функции, называемой конструктором. Он имеет имя совпадающее с именем класса на основе которого создается объект, при этом возвращаемым значением является указатель на созданную в памяти структуру. По окончании использования объекта необходимо выполнить деинициализацию с освобождение памяти. Деструктор имеет имя соответствующего класса с предшествующей ~.

Свойства и правила использования конструктора:

1. имеет тоже имя и класс, в котором он объявлен.
2. не возвращает значения (не описывается возвращаемое значение).
3. не наследуется в производных классах, для каждого класса собственный конструктор.
4. может иметь параметры заданные по умолчанию.
5. является функцией, но не может быть виртуальным.
6. в программе невозможно получить адрес конструктора.
7. если конструктор не задан в программе он будет автоматически сгенерирован средой. Все конструкторы сгенерированные средой имеют атрибут public. конструктор вызывается автоматически только при описании объекта.
8. объект, содержащий конструктор нельзя включить в объединение union.
9. конструктор класса X не может иметь параметр типа X, но может иметь параметр-ссылку на объект типа X, такой конструктор называется

конструктором копирования (не производится выделение памяти под структура объекта, а только ссылка на него).

Свойства и правила использования деструктора:

1. имеет имя соответствующего класса с предшествующей ~.
2. не возвращает значение.
3. не наследуется в производных классах.
4. производный класс может вызвать деструкторы своих базовых классов.
5. не имеет параметров.
6. класс может иметь только один деструктор.
7. может быть виртуальной.
8. в программе невозможно получить адрес.
9. если не задан, автоматически генерируется средой.
10. можно вызывать как обычную функцию.
11. вызывается автоматически при разрушении объекта.

Каждый класс определяет некоторую область видимости, имена членов видимости локальны в данном классе, внутри класса область видимости не зависит от точки объявления членов класса. Любой член класса виден внутри всего класса.

Иногда необходимо чтобы все объекты класса имели доступ к некоторой общей переменной, а не к его копии. Такие член-данные следует объявлять внутри класса со спецификатором static. Благодаря которому данные будут хранится в заданной единственной ячейке памяти и любая их модификация будет иметь последствия для всех объектов данного класса. Изменение статических данных обычно производят через полное обращение к имени класса. Функции класса могут быть объявлены так что им разрешается только чтение значение данных этого класса, но не разрешается изменение. Такие функции должны быть объявлены со спецификатором const. при попытке изменения константного члена компилятор выдает ошибку. Константным может быть объявлен и объект в целом. К константному объекту могут быть применены только константные члены функции. В некоторых функциях необходимо выполнить обращение к компонентам объекта без указания имени. Внутри каждой функции формируется неявный указатель на создаваемый объект класса. Когда объект создается, под него выделяется память, в которой имеется специализированное поле, содержащее скрытый указатель в адресуемый первый байт. Получить

значение скрытого указателя можно получить через слово this. Случаем для использования явных указателей является использования связанных структур.

1. каждый вновь создаваемый объект получает скрытый указатель.
2. this указывает на начало текущего объекта в памяти
3. this не требует дополнительного объявления
4. this передается в качестве ??? во все не статические член-функции данного объекта.
5. this – локальная и недоступна за пределами объекта.

В языке C++ одна и та же функция не может быть компонентом двух различных классов, однако с некоторых ситуациях необходимо иметь доступ из одной функции к локальным компонентам различных классов, объявленным в области видимости private или public.

Окружность и прямоугольник. Необходимо разработать функцию, осуществляющую сравнение цветов двух графических объектов. В этом случае функция должна иметь доступ к локальному компоненту объекта класса окружность и к локальному объекту класса прямоугольник. Одновременно объектом двух классов функция сравнения быть не может, поэтому ее можно объявить как дружественную функцию. Если некоторая функция объявлена как дружественная некоторого класса X, то она не является член-функцией класса X и имеет доступ ко всем компонентам этого класса.

Функция со спецификатором friend является обычными глобальными функциями. Единственным их отличием является возможность доступа к локальным компонентам класса. Поскольку функция friend не является компонентом класса в ней нельзя использовать спецификатор this. Объявление friend функции может быть помещено в любую секцию класса: приватную, общедоступную или защищенную. Т.к. friend функция объявляется в классе X то она может рассматриваться как часть интерфейса класса X. И использоваться для осуществления взаимодействия другими классами программы. Член-функция одного класса может быть объявлена со спецификаторами friend для другого класса. В некоторых случаях можно сделать дружественным класс целиком. При этом все член-функции класса Y имеют спецификатор friend для всех компонент класса Z и получают доступ к ним.

Основные свойства и правила использования friend

1. не являются компонентами класса но получают доступ ко всем компонентам независимо от их области видимости
2. если friend-функция одного класса не является компонентом другого класса то она вызывается как обычная глобальная функция без использования . и ->
3. если friend функция не является компонентом другого класса, то она не имеет неявного указателя this.
4. не наследуются в производных классах.
5. отношения friend не является транзитивным.

Создание и разрушение объектов.

Если класс имеет конструктор то этот конструктор вызывается всякий раз когда создается объект этого класса. Если класс имеет деструктор то он вызывается всегда когда объект класса уничтожается при этом каждый объект имеет некоторое время существования, которое принято называть временем жизни объекта. Время жизни объекта определяется тем, где создан объект и где создан. Объекты могут быть созданы в следующих формах:

1. глобальные объекты создаются вначале исполняемой программы и разрушаются при ее завершении.
2. автоматические объекты создаются в момент их объявления в исполняемой программе и разрушаются при вызове деструктора и освобождении памяти.
3. статические объекты создаются однократно при запуске программы, к которой они относятся и однократно разрушаются при завершении программы.
4. объекты динамически выделяемой области памяти.
5. объекты-компоненты классов создаются при построении объекта класса, в котором они описаны и разрушаются при удалении объекта класса, в котором они описаны.

16. Наследование. Множественное наследование.

Наследование – это организация связей между абстрактными типами данных при котором имеет возможность на основании существующих типов порождать новые типы при этом новый тип способен наследовать все данные и функции типа родителя. Это позволяет создавать иерархию типов разделяющих между

собой одни и те же функции и данные. При этом вновь создаваемый класса называется производным классом, класс-родитель – базовым классом, класс являющийся прародителем множества класса – супер классом.

В С++: простое наследование (когда каждый производный класс имеет одного прямого родителя), множественное наследование (когда производный класс порожден от нескольких прямых родителей).

При создании нового класса предполагается:

1. возможность добавления новых член-данные и член-функции.
2. замена существующих родителей существующих член-функции новыми одноименными функциями класса.

Производные классы являются эффективным средством для расширения функциональных возможностей существующих классов без их перепрограммирования.

Основные правила использования базовых и производных классов:

Пусть функция F принадлежит базовому классу Б. Тогда в производном классе П можно:

1) полностью заменить функцию F (старая Б::F и новая П::F);

2) доопределить (частично изменить) функцию F;

3) использовать функцию Б::F без изменения.

- Если объявить указатель рБ на базовый класс, то ему можно присвоить значение указателя на объект производного класса;

- указателю рП на производный класс нельзя присвоить значение указателя на объект базового класса;

- регулирование доступа к компонентам базового и производного классов осуществляется с помощью атрибутов private, public и protected;

- производный класс может быть в свою очередь базовым. Множество классов, связанных отношением наследования базовый - производный, называется иерархией классов.

Фактически атрибут наследования определяет верхнюю допустимую границу для атрибутов компонентов базового класса к которым осуществляется доступ из производного класса. Иногда возникает ситуация когда требуется private порожденных классов получить доступ к членам базового класса объявленным как public или protect. в этом случае осуществляется обращение к компонентам базового класса по их полному имени. При этом если выполняется обращение к член-функции определяется ссылка на имеющуюся функцию.

Множественное наследование - когда производный класс порожден от двух или более базовых классов. Вероятность использования одноименных член-данные и член-функции внутри родительских функций, тогда для устранения несогласования в дочернем классе необходимо использовать полное имя член-данные или член-функции. Возможность использования общего прародительского класса при множественном наследовании приводит к неэффективному использованию памяти за счет многократного размещения компонентов общего прародителя при создании производного класса. Для устранения данного недостатка используют виртуальное наследование. При этом отличительной особенностью виртуального наследования является то, что класс инициализируется не непосредственно порожденным классом а более отдаленным порождением. Конструкторы и деструкторы при виртуальном наследовании выполняются в следующем порядке:

1. конструкторы виртуальных базовых классов выполняются до не виртуальных вне зависимости от порядка задания в списке порождения.
2. если класс имеет несколько виртуальных базовых классов, т.е. конструкторы вызываются в соответствие со списком порождения.
3. деструкторы виртуальных базовых классов всегда выполняются после деструкторов не виртуальных.

При использовании виртуальных базовых классов функции на разных уровнях иерархии могут иметь одинаковые имена. При этом при вызове будет доминировать функция, принадлежащая более далекому от основания базовому классу.

17. Полиморфизм.

Принцип полиморфизма – подразумевает возможность перегрузки имен операции и позволяет использовать одни и те же функции для решения

различных задач. Различают полиморфные (поддерживают принцип полиморфизма в теории типов) и мономорфные (предполагают, что процедуры и операторы имеют уникальный тип) языки программирования. При наследовании часто возникают ситуации когда необходимо чтобы поведение базовой функции было модифицировано в зависимости от того объектом какого класса эта функция вызвана. Простое переопределение функции может привести к конфликту имен.

При определении полиморфных функций следует различать 2 понятия:

1) перегрузка функций предполагает использование одноименных методов в рамках иерархии отличающихся типами или количеством параметров
2) перекрытие – сигнатуры родительских и дочерних методов должны полностью совпадать. Особенность ярко иллюстрируется при вызове внутри переопределенного метода другого переопределенного метода, при этом возникает вопрос о выборе в качестве вызываемого метода родительского или дочернего компонента. Для того чтобы изменить данную ситуацию необходимо определять перекрытые методы как виртуальные при этом процесс вызова метода будет изменен. Каждый объект при своем создании запрашивает выделение памяти под свою структуру. Память выделяется под член данные объекта и указатели на 2 таблицы:
   1) таблица статических методов (STM)
   2) таблица виртуальных методов (VMT) - содержит адреса функций, расположенных в порядке иерархии создания. При вызове виртуальной функции осуществляется переход по адресу, указанному в таблице виртуальных методов со смещением присущим данному классу объектов.

Правила описания и использования виртуальных функций:

1. Виртуальная функция может являться только методом класса и не может являться внешней функцией программы.
2. Любую перегружаемую операцию можно сделать виртуальной.
3. Виртуальные функции, как и сама виртуальность, наследуются.
4. Виртуальная функция может быть const.
5. Если в базовом классе впервые объявлена виртуальная функция, то она должна быть, либо чистой (virtual int f(void)=0;), либо у нее должно быть

задано определение, т.е. тело. Если в базовом классе объявлена чистая виртуальная функция, то в порожденных классах нужно дать ей определение или определить снова как чистую.

6. Если в базовом классе определена виртуальная функция, то метод производного класса с таким же именем автоматически, то будет являться виртуальным.
7. Конструкторы не могут быть виртуальными.
8. Методы, объявленные как static, не могут быть виртуальными.
9. Деструкторы чаще всего должны быть виртуальными.
10. Если некоторая функция вызывается с использованием в виде полного имени, то виртуальный механизм игнорируется.

Выделяют 2 типа полиморфизма (отличаются моментом связывания объекта с вызываемым методом):

- статический (времени компиляции) – привязка метода к объекту осуществляется на этапе компиляции программы, следовательно статический полиморфизм реализуется через механизм перегрузки или шаблона.

- динамический (времени выполнения) – привязка метода к объекту осуществляется на этапе выполнения программы, что связано с изъятием адреса используемого метода из таблицы метода данного объекта.

Реализуется через механизм виртуальных функций. Класс, включающий виртуальную функцию, называется полиморфным. Если в классе присутствует чистая виртуальная функция, то такой класс называется абстрактным. Нельзя создать объект абстрактного класса. Абстрактные классы используются только для дальнейшего наследования. Необходимость введения виртуальных функций определяется следующими обстоятельствами:

1. вся тяжесть по определению типа объекта ложится на программиста, а не на компилятор.
2. программа, написанная в статическом стиле, привязана к деталям иерархии приложения, при изменении иерархии необходимо менять текст приложения.
3. при изменении программы сторонним разработчиком в случае статического программирования требуются знания подробности иерархии наследования.

Вызов виртуальных функций может являться не виртуальным в следующих случаях:

1. если осуществляется не через указатель или ссылку при этом вызов становится статическим.
2. если вызов осуществляется через указатель или ссылку, но с уточнением имени класса.
3. если он осуществляется внутри конструктора или деструктора базового класса.

18. Многопоточное программирование. Понятие потока и процесса. Создание и уничтожение потока.

В любой многозадачной системе используется концепция процессов.

Процесс (задача) - абстракция описывающая выполняющуюся программу, представляет собой единицу работы, заявку на потребление системных ресурсов.

Для ОС процесс представляет собой единицу работы или заявку на потреблении системных ресурсов. Архитектура ОС при этом должна удовлетворять:

1) Возможность чередования выполнения нескольких процессов с целью повышения степени использования процессов при разумной обеспечении времени отклика
2) Возможность распределения ресурсов между процессами в соответствии с заданной стратегией.
3) Возможность создания новых пользователь их процессов.

Основная работы ОС связана с управлением процессами. В любой момент времени этапы выполнения каждого процесса характеризует одно из состоянии процесса.

Любой процесс может находиться в одном из трех состояний: готовность, выполнение, блокировка.

Выполнение - при котором процессу выделяется процессорное время для осуществления алгоритма его работы.

Блокировка - при котором процесс останавливает свое выполнение в связи с невозможность получения доступа к требуемому ресурсу, процессорное время есть у него.

Готовность - готовый к выполнению процесс, в ожидании выделения процессорного времени.

В много поточных системах традиционная концепция процесса разделена на 2 части:

- Первая связана с принадлежностью ресурсов - процесс
- Вторая с выполнени км машинах команд - поток

Потоки возникли как средство распараллеливания

Использовании нескольких поток в увеличивает производительность приложения.

Данное утверждение не полностью корректно: ОС управляет имеющимися процессами и распределением процессорного времени между ними, занимается созданием и уничтожением. Для этой работы в памяти поддерживается специальные структуры, в которых указываются какие ресурсы выделены для использования. Ресурсы могут быть назначены в монопольное или совместное использование. Ресурсы могут быть прописаны процессу на все время его жизни, либо на определенный период. В ОС, где есть понятия процесс и поток, процесс рассматривается как заявка на потреблении всех видов ресурсом кроме процессорного времени. Процесс контейнер ресурсов для потока, процессорное время потребляет поток. Время распределения между потоками, каждый поток-последователь сеть команд. Обычно процесс включает в себя следующие элементы:

- Закрытое виртуальное адресное пространство
- Исполняемую программу, то естьначальныйкод и данные, проецируемые в вирт адресное пространство процесса
- Список открытых описателей различных системных ресурсов.
- Объект защиты- маркер доступа, также идентификации пользователя, определяющий правило получения доступа к процессу
- Каждый процесс имеет идентификатор процесса

Поток - объект ядра ОС, получающий процессорное время для выполнения и существующий в контексте конкретного процесса.

Включает в себя:

1) Два стека

2) Набор регистров процессора

3) Закрытая область памяти-локальному память потока

4) Уникальный индетификатор, по которому он распознается внутри процесса.

**********Создание**********

Каждый поток начинает свое выполнение с некоторой входной функции. Она называется функцией потока и имеет как минимум один параметр. Функция потока может содержать любые операции, связанные с выполнением работы потока. По окончании ее поток автоматически завершается. При завершении потока система выполняет действия:

1) Останавливает поток, определенный данной функцией потока
2) Освобожден стек область, выделенную текущему потоку
3) Уменьшает счетчик пользователя на 1 для объекта ядра потока

В случае когда счетчик обнуляется система удаляет поток. Таким образом объект ядра имеет большее время жизни, чем сам поток.

Такая конструкция позволяет получить информацию о статусе завершения потока вне потока основному потоку или внешней проге. Функция потока всегда должна возвращать некоторое значение.

Правилом хорошего тона считается использование внутри функци локальных переменных, так как обращение к глобальным переменным либо указателям на внешние участки памяти требует использования дополнительных механизмов, в частности механизма синхронизации с целью обеспечения корректного доступа к общим данным. Создание потока осуществляется путем вызова функции

////// CreateThread /////

Она создает новый поток в процессе и возвращает идентификатор потока в случае успешного создания. В качестве параметра функция создания потока использует :

1) Указатель на структуру атрибутов безопасности. psa
2) Параметр определяющий размер стека, выделяемый для потока: если указано не нулевое значение, то система сравнивает это значение со значением по умолчанию и всегда выделяет стек большего размера. Он резервирует адресное пространство физической памяти которое выделяется процессу по необходимости. cbStack
3) Параметр указатель на потоковую функцию. pfnStartAddr
4) Параметр потоковой функции: может иметь произвольное значение и передается в функцию потока с целью дальнейшей обработки внутри тела потока. pvParam
5) Дополнительный параметр, используемый при создании потока; может принимать нулевое значение, если поток должен быть запущен немедленно; или может быть указана задержка запуска потока: выполняет ининциализацию

потока, но не запускает его, его запуск - отдельной системной функцией. tdwCreate

6) Параметр- идентификатор потока- указатель на перемен ную, которая определяет идентификатор потока. может быть назначен вручную и затем использован для общения к потоку. pdwThreadID

**********Завершение потока**********

Поток может завершится в одном из след случаев:

1) Поток самоуничтожается, путем вызова системной функции (она не рекомендуется, так как вызов может привести к блокировке других потоков в случае захвата синхронизации) ExitThread
2) Функция потока возвращает управление
3) Один из потоков данного или стороннего процесса вызывает функцию TerminateThread - нежелательный, прерывание, ничего нельзя выяснить что там было и что сохранилось
4) Завершается процесс, создавший данный поток - нежелательно, неизвестно состояние потоков

Функции потока следует проектировать так чтобы поток всегда завершался только после того как она возвращает управление. При этом:

- Любые объекты, созданные данным потоком, уничтожаются соответствующими деструкторами
- Система корректно освобождает память, которую занимал стек потока
- Система устанавливает код завершения данного потока
- (поддерживаемый объектом ядра)
- Счетчик пользователей данного объекта ядра "поток" уменьшается на 1

При работе в средах, связанных с CLR, можно использовать операции другого рода. Имеется возможность совместимости потоков и стандартной библиотеки.

Endtread позволяет корректно завершить поток с освобождении памяти и ресурсов

19. Многопоточное программирование. Синхронизация потоков. Объекты синхронизации.

Основная сложность состоит в организации процесса синхронизации пока.

Синхронизация - работа по организации не конфликтного доступ к общим ресурсам системы: физические устройства (принтер), глобальные переменные, общие участки памяти.

В языках программирования есть специализированные операции, помогавшие поддерживать механизмы синхронизации для многопоточных приложений

Приостановка работы потока:

- sleep - остановить поток на данное количество милсекунд
- WaitForSingleObject - данная функция заставляет поток перейти в состояние ожидания до прихода сигнала от другого потока, а сброс потока каждое количество милсекунд
- WaitForMultipleObjects - можно перевести поток в состоянии ожидания сигналов от множества потоков
- Infifnite. Бесконечное ожидание

Объекты синхронизации:

- Критическая секция - Часть кода, принадлежащая некоторому потоку, которая выполняется без передачи управления между потоками.
- мьюекс - объект ядра, который создается для остановки около критчисекого ресурса. Два состояния: занятое и свободное. Любой поток может посмотреть его состояние и в зависимости от его состоянии принять решение и возможности или невозможности захвата ресурсов .
- Семафоры - объект ядра, позволяющий останавливать доступ к заданному критическому ресурсу. Он может быть захвачен или отпущен каким либо потоком. Имеет счетчик, что позволяет получить доступ к ресурсу одновременно нескольким ограниченным потокам. При освобождении инкремент счетчика. Если счетчик на нуле то все потоки останавливаются в ожидании захвата.

Событие - объект синхронизации, состон которого может быть установлено путем вызова спец функций. Установлено и сброшено- два состояния. Позволяет приостановить поток прежде критические ресурсовс до совершения конкретного события.?????

Более подробный разбор каждого

Мьютекс - объект ядра, который создается с помощью функции CreateMutex. Мьютекс может находиться в 1 из двух состояний: занятом или свободном. С его помощью хорошо защищать единичный ресурс от одновременного обращения к нему разными потоками. В один момент времени только один поток владеет мьютексом. Мьютекс может быть известен одновременно нескольким потокам. Мьютексы с одинаковыми именами можно задать в

разных потоках. При вызове функции CreateMutex только первый поток создает мьютекс, а все остальные получают идентификатор уже существующего объекта. Это дает возможность нескольким процессам управлять одним и тем же мьютексом, избавляя программиста отслеживать место создания мьютекса и момент освобождения памяти, выделенного под объект. Нужно устанавливать флаг bIntiaal???? Если мьютекс создан в родительском потоке, то его идентификатор является глобальным и доступным для всех ниже созданных потоков. Освобождение мьютекса осуществляется с помощью Release после чего мьютекс может быть захвачен другим потоком.

Пример:

В случае если синхронизация будет нестабильной, функция print будет выводить значение массива в произвольное порядке.При правильной синхронизации только после заполнения массива произойдет его вывод.

Критической секция позволяет организовать синхронизацию в пользовательском режиме, создавая неделимый блок. Критическая секция в заданный момент времени может использоваться только одним потоком. Она не является объектом ядра и представляет собой структуру, содержащую набор флагов. При входе в эту секцию проверяются флаги и если выясняется, что критическая секция с данным именем уже задана другим потоком, текущий поток переводится в состояние ожидания. Она характеризуется тем, что для проверки занятости программа не переходит в режим ядра, что не накладывает дополнительных задача на ОС. Такая синхронизация более быстра. Критической секция имеет индентификатор, который может быть доступен всем потокам, созданного процесса, если о создании критической секции объявлено до создания потока, в противном случае секция является монопольной для потока. Создается с помощью функции. ??? В качестве параметра ссылка на идентификатор. enter для открытия критической секции. После нее начинается неделимы код, который завершается ???. Она монопольное: либо для каждогоп откоа своя или глобальная.

Событие это объект синхронизации, состояние которого может быть установлено в сигнальное. ???петух вызова функции SetEvent. ???Либо функции PulseEvent. Имеет два состояние: установленное и сброшенное. Существует два типа событий:

Событие с ручным сбросом - объект, сигнальное состояние которого сохраняется до ручного сброса с помощью функции ReverseEvent. Как только состояние объект установлено ??в сигнальное??, все находящиеся в цикле ожидания потоки, получают данное событие и продолжают свое выполнение.

Событие с автоматические сбросом - объект сигнал не сотские которого сохраняется до тех пор, пока не будет освобожден некоторый поток, после чего

ОС автоматически установит несигнальное состояние события. Если не существует потоков, ожидающий данного события, объект остается в сигнальном состоянии. Предположил, несколько потоков ожидают одного и того же события и оно срабоатло. Если оно в авто сбросом, то онон позволит продолжить работу только одному потоку , остальные останутся в режиме ожидания, если ручного сброса то все потоки будут освобождены, а событие останется в состоянии сигнальном до ручного вызова функции Reverse. События полезны в тех случаях, когда необходимо послать сообщение некоторому потоку, сигнализирующее потоку о возникновении некоторого события. При организации асинхронных операций ввода вывода на одном физическом устройстве. В общем случае поток может исп. звать множество различных событий при организации собственной работы. Поток может самостоятельно создавать события, путем вызова функции CreateEvernt. Данная функция создает объект событие и возвращает его идентификатор. Создающее событие устанавливает его в начальное состояние. Если не связанные потоки вызывают операцию создания события, используя общий идентификатор, то лишь один поток - создатель события, остальные получают его копию. Любой поток может использовать функцию PulseEvent для установки состояния события в сигнальном, а затем сбрасывать его в несигнальное.

Семафоры объект синхронизации, являющийся объектом ядра. В отличие от мьютекса семафор имеет счетчик. Семафор открыт, если счетчик больше нуля и закрыт если нуль. Обычно используются для ограничения достау нескольим потокам некоторому общему ресурсу, возможность получения доступа n потокам к этому ресурсу. Функция создания- атрибут безопасности мин и Макс значени счетчика, глобальное имя. Для того чтобы захватить семафор, необходимо выполнить операцию Wait при этом происходит дикремент семафор оного счетчика и его проверка на нулевое значение. Если значени счетчика более ноля, то разрешается доступ к ресурсу, иначе полк в состоянии ожидания. Для освобожедения семафора: Release. При это инкриминирует на указанное значение счетчика семафора и освобождение критическому ресурса. Использование заданных значений при искренен те и создании семафора позволяет регулировать количество потоков, которым позволено зайти за семафор.

Пример: 2 чителя и один писатель. Для писателя два wait for signal. Пускает или не пускает за себя.