ООП 2021
Лекция 3

Память. Размеры объектов. Приведение типов. Глубокое копирование.

oopCpp@yandex.ru

Сгенерированные операции

При определении классов некоторые операции над их объектами всегда существуют:

- Конструктор копирования
- Оператор присваивания копированием
- Деструктор.

Это – правило трех.

Память. Создание объектов.

- **Явное**

  Есть два способа создать объект явно. Во-первых, это можно сделать при объявлении: тогда объект размещается в стеке.

```
DisplayItem item1;
DisplayItem* item2 = new  DisplayItem(Point(100, 100));
DisplayItem* item3 = 0;
```

- **Неявное**

  Часто объекты создаются неявно. Так, передача параметра по значению в C++ создает в стеке **<u>временную копию</u>** объекта. Более того, создание объектов транзитивно: создание объекта тянет за собой создание других объектов, входящих в него. Переопределение семантики копирующего конструктора и оператора присваивания в C++ разрешает явное управление тем, когда части объекта создаются и уничтожаются. К тому же в C++ можно переопределять и оператор **new**, тем самым изменяя политику управления памятью в "куче" для отдельных классов.

Размеры типов данных

sizeof(T) - определяет сколько байт потребуется для хранения в памяти объекта (или типа) T.

```
int x=9;
MyType* obj=0;
int size= sizeof(x);
size= sizeof(int);
size= sizeof(MyType);
size= sizeof( obj );
size= sizeof(MyType* );
```

Память методов класса

Каждый объект имеет собственные независимые поля данных. С другой стороны, все объекты одного класса используют **одни и те же методы**.

Методы класса создаются и помещаются в память компьютера всего **один раз** - при создании первого объекта класса.

```
struct A{
    void f(){   cout<<"aaa"<< endl;   }
};

int _tmain(int argc, _TCHAR* argv[])
{
A *a= new A;
a -> f();
return 0;
}
```

Копирование памяти

memcpy и **memcpy_s**

```
void *memcpy(void *dst, const void *src, size_t n);
```

dst — адрес буфера назначения

src — адрес источника

n — количество байт для копирования

Функция копирует n байт из области памяти, на которую указывает src, в область памяти, на которую указывает dst. Функция возвращает адрес назначения dst.

Перемещение памяти

Области памяти не должны перекрываться , иначе данные могут быть скопированы неправильно.

Для правильного копирования перекрывающихся областей нужно использовать функцию **memmove**().

void * **memmove** (void * destination, const void * source, size_t num);

```
#include <stdio.h>
#include <string.h>

int main ()
{
  char str[ ] = "memmove can be very useful.....";
  memmove (str+20,str+15,11);
  puts (str);
  return 0;
}
```

Output:

memmove can be **very very** useful.

Инициализация памяти

void * **memset** (void * ptr, int val, size_t num);

Функция **memset** заполняет num байтов блока памяти, через указатель ptr. Код заполняемого символа передаётся в функцию через параметр val.

ptr - Указатель на блок памяти для заполнения.

val - val передается в виде целого числа, но функция заполняет блок памяти символом, преобразуя это число в **символ**.

num - Количество байт, которые необходимо заполнить указанным символом.

```
#include <stdio.h>
#include <string.h>
int main (){
  char str[ ] = "almost every programmer should know memset!";
  memset (str,'-',6);
  puts (str);
  return 0;
}
```

Output:

```
------ every programmer should know memset!
```

Адреса памяти

Любой объект - это некоторая область памяти, адрес - выражение,

ссылающееся на объект или функцию. Очевидным примером адреса будет имя объекта. Существуют операции, порождающие адреса, например, если E выражение типа указатель, то *E - адресное выражение,

соответствующее объекту, на который указывает E.

Термин "адрес" ("lvalue" т.е. left value - левая величина) появляется из оператора присваивания E1 = E2, где левый операнд E1 должен "адресовать" изменяемую переменную.

Адрес может изменяться, если он не является именем функции,

именем массива или const.

Указатели

Указатели - необходимы для более эффективного использования языка программирования.

Замечательным примером **важности использования указателей** является создание таких структур данных, как **связные списки** или **бинарные деревья**. Кроме того, некоторые ключевые возможности языка C++, такие, как **виртуальные функции**, операция **new**, указатель **this** , требуют использования указателей.

Идея указателей несложна. Начать нужно с того, что каждый байт памяти компьютера имеет адрес. Адреса это те же числа, которые мы используем для домов на улице. Числа начинаются с 0, а затем возрастают 1, 2, 3 и т. д. Если у нас есть 1 Мбайт памяти, то наибольшим адресом будет число 1 048 575 (хотя обычно памяти много больше) .

Загружаясь в память, наша программа занимает некоторое количество этих адресов. Это означает, что каждая переменная и каждая функция нашей программы начинается с какого-либо конкретного адреса.

Мы можем получить **адрес** переменной, используя операцию получения адреса &:

```
int main(){
int var1 = 11; int var2 = 22; int var3 = 33;
cout << &var1 << endl   // напечатаем адреса этих переменных
<< &var2 << endl << &var3 << endl;
return 0;    }
```

Преобразования указателей

Всюду, где указатели присваиваются, инициализируются, сравниваются или используются иным образом, могут происходить следующие преобразования:

- Константное выражение, которое сводится к нулю, преобразуется в указатель, обычно называемый пустым указателем – **void*** . Гарантируется, что значение такого указателя будет отлично от любого указателя на объект или функцию.

- Указатель на объект любого типа, не являющегося const или volatile, можно преобразовать в void*.

- Указатель на функцию можно преобразовать в void*, при условии, что для void* отводится достаточно памяти, чтобы хранить этот указатель.

- Указатель на данный класс можно преобразовать в указатель на доступный базовый класс данного класса, если такое преобразование не содержит двусмысленность. Базовый класс считается доступным, если доступны его общие члены. Результатом преобразования будет указатель на объект типа базового класса, вложенный в объект типа производного класса. - Пустой указатель (0) преобразуется сам в себя.

- Выражение типа "массив T" может преобразовываться в указатель на начальный элемент массива.

- Выражение типа "функция, возвращающая T" преобразуется в "указатель на функцию, возвращающую T", за исключением тех случаев, когда оно используется как операнд адресной операции & или операции вызова функции ().

Ссылки

Ссылку (reference) можно интерпретировать как **автоматически разыменовываемый постоянный указатель** или альтернативное **имя объекта**. Указатели и ссылки отличаются следующими особенностями.

Присвоение чего-либо указателю изменяет значение указателя, а не объекта, на который он установлен.

Для того чтобы получить указатель, как правило, необходимо использовать оператор new или &.

Для доступа к объекту, на который установлен указатель, используются операторы * и [ ].

Присвоение ссылке нового значения изменяет то, на что она ссылается, а не саму ссылку.

После инициализации ссылку **невозможно** установить на другой объект.

Присвоение ссылок основано на **глубоком копировании** (новое значение присваивается объекту, на который указывает ссылка); присвоение указателей не использует **глубокое копирование** (новое значение присваивается указателю, а не объекту).

Указатели на члены классов

Всюду, где указатели на члены инициализируются, присваиваются, сравниваются или используются иным образом, могут происходить следующие преобразования:

- Константное выражение , которое сводится к нулю, преобразуется в указатель на член. Гарантируется, что его значение будет отлично от любых других указателей на члены.

- Указатель на член данного класса можно преобразовать в указатель на член производного от данного класса, при условии, что допустимо обратное преобразование от указателя на член производного класса в указатель член базового класса, и что оно выполнимо однозначным образом.

- Правило преобразования указателей на члены (т.е. от указателя на член базового класса к указателю на член производного класса) выглядит перевернутым, если сравнивать его с правилом для указателей на объекты (т.е. от указателя на производный объект к указателю на базовый объект). Это необходимо для гарантии надежности типов.

Отметим, что указатель на член **не является** указателем на объект или указателем на функцию и правила преобразований таких указателей не применимы для указателей на члены. В частности указатель на член **нельзя** преобразовать в void*.

Приведение типов

x = dynamic_cast<D*> (p) // Пытается привести указатель p к типу D*

(если приведение не удалось, то результат = 0)

x= dynamic_cast<D&> (*p) // Пытается привести значение *p к типу D& (может генерировать исключение bad cast)

x= static_cast<T> (v) // Приводит тип операнда v к типу T, если тип T можно привести к типу операнда v

x=reinterpret_cast<T> (v) Приводит тип операнда v к типу T, представленному той же самой комбинацией битов

x=const_cast<T> (v) Приводит тип операнда v к типу T, удаляя спецификатор const

x=(T)v Приведение в стиле языка C.

x=T (v) Функциональное приведение.

Динамическое приведение обычно используется для навигации по иерархии классов, если указатель p - указатель на базовый класс, а класс D — производный от базового класса. Если операнд v не относится к типу D*, то эта операция возвращает число 0.

Обнаружение утечек памяти

#define _CRTDBG_MAP_ALLOC

#include <crtdbg.h>

После последней области видимости вызвать функцию:

_CrtDumpMemoryLeaks();

```
int _tmain(int argc, _TCHAR* argv[ ])
{
 int * b = new int;

 //  delete b;

 _CrtDumpMemoryLeaks();
return 0;
}
```

Detected memory leaks!
Dumping objects ->
{187} normal block at **0x00A15730**, 4 bytes long.
Data: < > 98 DC B2 00
Object dump complete.

STL. Строки.

Класс **string** из стандартной библиотеки шаблонов представляет собой <u>специализацию</u> общего шаблонного класса basic_string для символьного типа char; иначе говоря, объект string это последовательность переменных типа char.

Класс **wstring** из стандартной библиотеки шаблонов представляет собой <u>специализацию</u> класса basic_string для типа wchar_t.

typedef basic_string<char, char_traits<char>, allocator<char> > **string**;

typedef basic_string<wchar_t, char_traits<wchar_t>, allocator<wchar_t> > **wstring**;

Операции со строками

s1 = s2 Присвоение строки s2 строке s1; строка s2 может быть объектом класса string или строкой к стиле языка C

s += s1 Добавление объекта s1 в конец строки; объект s1 может быть символом, объектом класса string или строкой в стиле языка C

s[i] Индексация (как у обычного массива)

s = s1+s2 Конкатенация; символы в целевом объекте класса string будут копиями символов их строки s1, за которыми следуют копии символов из строки s2

s1==s2 Сравнение объектов класса string; либо s1, либо s2, но не оба объекта могут быть строкой в стиле языка C.

s1 != s2 Проверка неравенства объектов s1 и s2

s1 < s2 Лексикографическое сравнение объектов класса string; либо s1, либо s2, но не оба объекта могут быть строкой в стиле языка C.

. <=, > и >= Аналогично.

Потоки строк

Объект класса string можно использовать в качестве источника ввода для потока istream или цели вывода для потока ostream. Поток istream, считывающий данные из объекта класса string, называется **istringstream**, а поток ostream, записывающий символы в объект класса string, называется **ostringstream**.

Например, поток istringstream полезен для извлечения числовых значений из строк:

```
stringstream ss;
   ss << "22.84";
    float k = 0;
   ss >> k;
```

Итераторы

Итераторы — это клей, скрепляющий алгоритмы стандартной библиотеки с их данными. Итераторы можно также назвать механизмом, минимизирующим зависимость алгоритмов от структуры данных, которыми они оперируют. (Б. Страуструп)

Итератор — это аналог указателя, в котором реализованы операции **косвенного доступа** (например, оператор * для разыменования) и **перехода к новому элементу** (например, оператор ++ для перехода к следующему элементу).

Последовательность элементов определяется парой итераторов, задающих полуоткрытый диапазон [begin , end).

Здесь итератор begin указывает на первый элемент последовательности, а итератор end — на элемент, следующий за последним элементом последовательности.

Никогда не считывайте и не записывайте значение *end. Для пустой последовательности всегда выполняется условие begin == end. Другими словами, для любого итератора p последовательность [p:p) является пустой

Категории итераторов

input iterator - Можем перемещаться вперед с помощью оператора ++ и считывать каждый элемент только один раз с помощью оператора *. Итераторы можно сравнивать с помощью операторов == и !=. Этот вид итераторов реализован в классе istream

output iterator - Можем перемещаться вперед с помощью оператора ++ и записывать каждый элемент только один раз с помощью оператора *. Этот вид итераторов реализован в классе ostream

forward iterator - Можем перемещаться вперед, применяя оператор ++ повторно, а также считывать и записывать элементы (если они не константные), с помощью оператора *. Если итератор указывает на объект класса, то для доступа к его члену можно использовать оператор ->

bidirectional iterator - Можем перемещаться вперед (используя оператор ++) и назад (используя оператор - -), а также считывать и записывать элементы (если они не константные) с помощью оператора *. Этот вид итераторов реализован в классах list, map и set

randomaccess iterator - Можем перемещаться вперед (с помощью операторов ++ и +=) и назад (с помощью операторов - - и -=), а также считывать и записывать элементы (если они не константные) с помощью оператора * или [ ]. Мы можем применять индексацию, добавлять к итератору произвольного доступа целое число с помощью оператора +, а также вычитать из него целое число с помощью итератора -. Можем вычислить расстояние между двумя итераторами произвольного доступа, установленными на одну и ту же последовательность, вычитая один из другого. Итераторы произвольного доступа можно сравнивать с помощью операторов <, <=, > и >=. Этот вид итераторов реализован в классе vector.

Потоковые классы

Поток — это общее название потока данных. В C++ поток представляет собой объект некоторого класса. Разные потоки предназначены для представления разных видов данных. Например, класс ifstream олицетворяет собой поток данных от входного дискового файла.

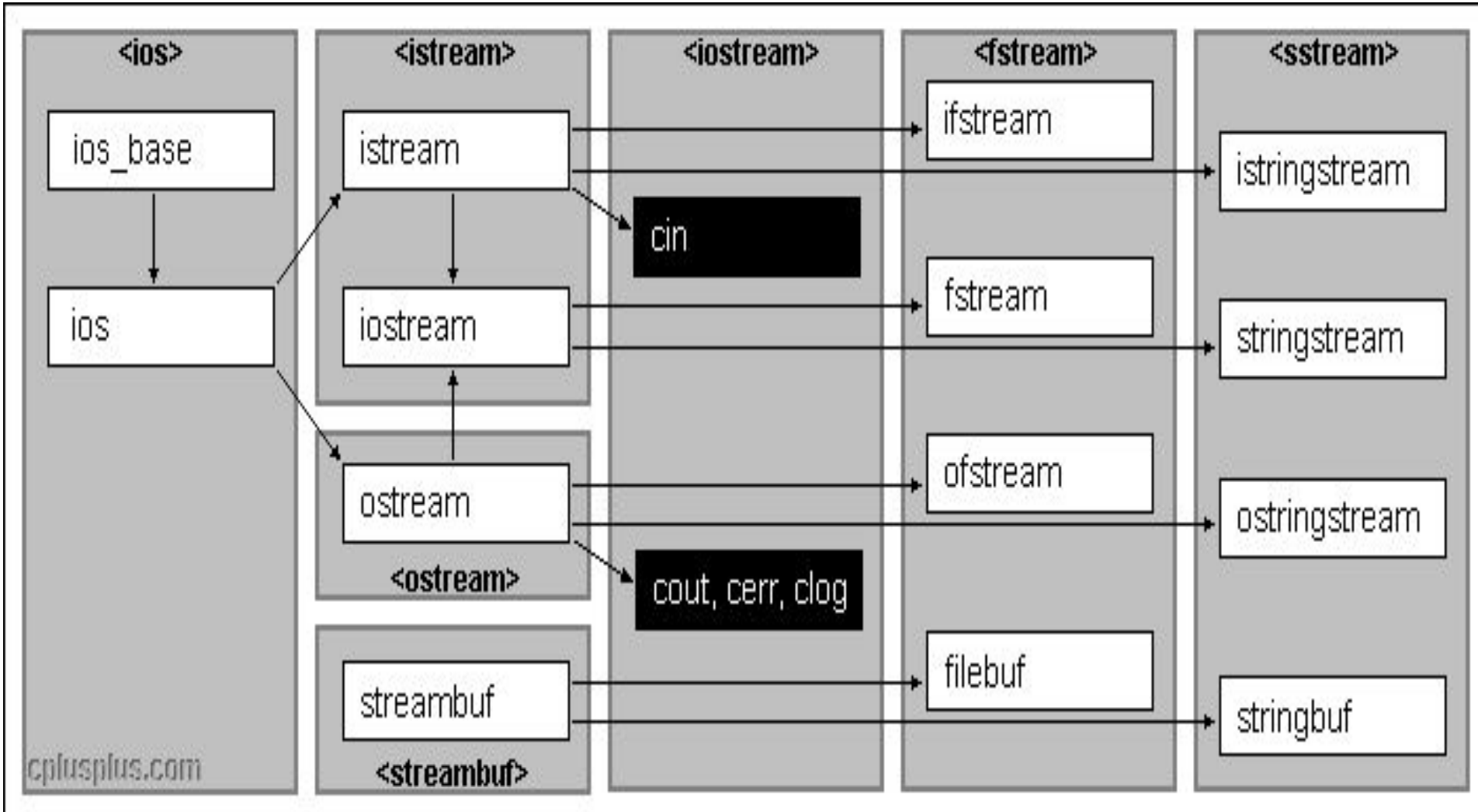
Одним из аргументов в пользу потоков является простота использования.

Если вам приходилось когда-нибудь использовать символ управления форматом %d при форматировании вывода с помощью %d в printf(), вы оцените это. Ничего подобного в потоках вы не встретите, ибо каждый объект **сам знает**, как он должен выглядеть на экране или в файле. Это избавляет программиста от одного из основных источников ошибок.

Другим арументом является то, что можно перегружать стандартные операторы и функции вставки (<<) и извлечения (>>) для работы с создаваемыми классами. Это позволяет работать с собственными классами как со стандартными типами, что, опять же, делает программирование проще и избавляет от множества ошибок, не говоря уж об эстетическом удовлетворении.

Поток данных - лучший способ записывать данные в файл, лучший способ организации данных в памяти для последующего использования при вводе/выводе текста в окошках и других элементах графического интерфейса пользователя (GUI)

Общая структура потоковых классов



Файлы

Обычно мы имеем намного больше данных, чем способна вместить основная память нашего компьютера, поэтому большая часть информации хранится на дисках или других средствах хранения данных высокой емкости. Такие устройства также предотвращают исчезновение данных при выключении компьютера — такие данные являются персистентными.

На самом нижнем уровне файл просто представляет собой последовательность байтов, пронумерованных начиная с нуля.

Файл имеет формат; иначе говоря, набор правил, определяющих смысл байтов. Например, если файл является текстовым, то первые четыре байта представляют собой первые четыре символа.

С другой стороны, если файл хранит бинарное представление целых чисел, то первые четыре байта используются для бинарного представления первого целого числа. Формат по отношению к файлам на диске играет ту же роль, что и типы по отношению к объектам в основной памяти. Мы можем приписать битам, записанным в файле, определенный смысл тогда и только тогда, когда известен его формат. При работе с файлами поток ostream преобразует объекты, хранящиеся в основной памяти, в потоки байтов и записывает их на диск. Поток istream действует наоборот: он считывает поток байтов с диска и составляет из них объект.

Работа с файлами

Для того чтобы **прочитать** файл, мы должны

- знать его имя;
- открыть его (для чтения);
- считать символы;
- закрыть файл (хотя это обычно выполняется неявно).

Для того чтобы **записать** файл, мы должны

- назвать его;
- открыть файл (для записи) или создать новый файл с таким именем;
- записать наши объекты;
- закрыть файл (хотя это обычно выполняется неявно).

std::vector

представляют собой контейнеры последовательностей, представляющие массивы, которые могут меняться по размеру.

Подобно массивам, векторы используют **смежные места хранения** для своих элементов, что означает, что их элементы также могут быть доступны с помощью смещений на обычных указателях к его элементам и так же эффективно, как и в массивах. Но в отличие от массивов их размер может изменяться динамически, при этом хранилище автоматически обрабатывается контейнером.

Внутри векторы используют динамически выделенный массив для хранения своих элементов. Этот массив, возможно, потребуется перераспределить для увеличения размера при вставке новых элементов, что подразумевает выделение нового массива и перемещение в него всех элементов. Это относительно дорогостоящая задача с точки зрения времени обработки, и, следовательно, векторы не перераспределяются каждый раз, когда элемент добавляется в контейнер.

std::list

template < class T, class Alloc = allocator<T> > class list;

Списки представляют собой контейнеры последовательностей, которые позволяют выполнять операции вставки и удаления с постоянным временем в любом месте последовательности и итерации в обоих направлениях.

По сравнению с другими базовыми стандартными контейнерами последовательности (array, vector и deque) list действуют лучше при вставке, извлечении и перемещении элементов в любом положении внутри контейнера, для которого уже был получен итератор.

Основным недостатком list и forward_lists по сравнению с этими другими контейнерами последовательности является то, что они не имеют прямого доступа к элементам по их положению; Например, чтобы получить доступ к шестому элементу в списке, нужно выполнить итерацию из известной позиции (например, начало или конец) в эту позицию, которая занимает линейное время на расстоянии между ними. Они также потребляют некоторую дополнительную память, чтобы связать информацию привязки к каждому элементу (что может быть важным фактором для больших списков малогабаритных элементов).

Глубокое копирование

Копирование называется глубоким, по причине того, что в классе могут находится члены-данных неизвестного размера, и их необходимо правильно скопировать.

Такие члены данных имеют указательный тип и соответственно являются динамическими, память которых выделяется в «куче» через оператор **new**, а освобождается **- delete**.

Правильное, глубокое копирование означает, что будет не просто скопирована память, где лежит указательный объект члена-данных класса, а еще и скопировано содержимое этой памяти.

```
int _tmain ( int argc, _TCHAR* argv[ ]) {
vector <base*> data1;  /// первая База Данных
list <base*> data2;        /// вторая База Данных

data1.insert( data1.end(), new base);
data1.insert( data1.end(), new  der);
      // первая попытка скопировать: попытки не всегда правильные!
for ( vector <base*>::iterator it = data1.begin(); it != data1.end(); ++it ){
  // казалось бы, легко взять и просто по элементам скопировать
  // из первой БД во вторую.
data2.insert( data2.end(), *it);
}
 // Тогда во второй БД будут лежать те же указатели, что и в первой
// И при удалении первой базы эти указатели станут недействительными
// Поэтому этот вариант копирования отпадает.
return 0;
}
```

```
    // Описываем базовый и производный классы
class base {
int *i;
public:
base(){ i=new int; *i=11;}
base( int x){ i=new int; *i=x;}
      // конструктор копирования
base( const base&  obj_for_copy): i( new int) { *I  =  *obj_for_copy.i;  }
virtual ~base(){ delete i;}
};

class der : public base{
public:
 der() { }
 der(int x):base(x){ }
~ der(){ }
      // конструктор копирования
der (const der&  obj_for_copy):base( obj_for_copy){   }
};
```

```
int _tmain ( int argc, _TCHAR* argv[ ]) {
    vector <base*> data1;  /// первая База Данных
    list <base*> data2;        /// вторая База Данных

data1.insert( data1.end(), new base);
data1.insert( data1.end(), new  der);
  //  следующая попытка: попытка непосредственного  копирования
for ( vector <base*>::iterator it = data1.begin(); it != data1.end(); ++it ){
 // В этом случае мы должны явно вызвать конструктор копирования, но мы не
    знаем какого именно класса конструктор надо вызвать – пробуем вызвать
    конструктор базового класса:
    base* new_obj = new  base( *( *it) );
  data2.insert( data2.end(), new_obj);
}
    // Увы , во второй БД будут лежать  лишь объекты базового типа.
return 0;
 }
```

Последняя попытка: предположим, что у нас есть специальная функция (не конструктор) для копирования содержимого одной БД в другую:

```
for ( vector <base*>::iterator it = data1.begin(); it != data1.end(); ++it ){
  base* new_obj= (*it)->copy();
  data2.insert( data2.end(), new_obj);
}
```

// Определим функцию-член copy() в обоих классах, сделав ее **виртуальной**:

```
base* base::copy(){   return new base(*this); }
base* der:: copy(){ return new der(*this); }
```

// А эта функция как раз сможет вызвать конструктор копирования нужного класса и создать новый объект:

```
base:: base( const base&  obj_for_copy): i( new int) { *i  =  *obj_for_copy.i;  }
der:: der (const der& obj_for_copy):base( obj_for_copy){   }
```

Домашнее задание

Проект 27

Представьте, что вы – управляющая компания и у вас в подчинении 5 домов двух видов: деревянный (Wooden) и кирпичный (Brick). Опишите типы этих домов, наследовав от абстрактного класса – House.

В главной функции программы создайте хранилище объектов ваших типов и положите в него 5 объектов-домов. Любых на свой вкус.

Ниже создайте еще одно хранилище, используя std::vector.

Затем скопируйте данные из первого хранилища во второе.

Затем очистите 1 хранилище, а затем тоже самое сделайте со 2-ым.

Проверьте, нет ли утечек памяти. Если есть – исправьте программу.

Контрольная работа 3

- Создать полиморфную иерархию из двух классов, именуя их, используя свою фамилию и имя.
- Создать в **базовом** классе член-данных типа **float* , выделить для него память и** инициализировать затем числом π в конструкторе по умолчанию.

- Создать два глобальных объекта: **vector <Base*>** sklad и **vector <Base***> magazin.
- Также написать глобальную функцию fill, задачей которой будет размещение в динамической памяти 2 объектов полиморфной иерархии (имеющихся типов), а затем добавление их по одному на склад.
- Затем, в главной функции, **перенести информацию со** склада в магазин.
- Не забыть освободить ресурсы !