ООП 2021
Лекция 11

Статические переменные
Динамическая информация о типах

Семантика перемещения

oopCpp@yandex.ru

Статические переменные

Статические данные относятся ко всем объектам класса. Такие данные используются, если

- требуется контроль общего количества объектов класса;
- требуется одновременный доступ ко всем объектам или части их;
- требуется разделение объектами общих ресурсов.

В этом случае в определение класса могут быть введены статические члены.

Статические члены описываются с помощью ключевого слова **static**, которое может использоваться при объявлении членов-данных и членов-функций класса.

Такие члены классов называются статическими, и независимо от количества объектов данного класса, существует только одна копия статического элемента.

Обращение к статическому элементу осуществляется с помощью оператора разрешения контекста и имени класса:
ИмяКласса :: ИмяЭлемента
Если **x** – статическое член-данное класса **Type**, то к нему можно обращаться как **Type :: x**
При этом не имеет значения количество объектов класса **Type**.

Статический метод класса это функция, которой требуется доступ к членам класса, но не требуется, чтобы она вызывалась для конкретного объекта класса.

Поэтому статические функции не могут обращаться к нестатическим членам класса.

Статические члены-данные и члены-функции

```
struct A{
 int a;
 A(){a=10;}
 static int f();
};

int A::f(){return a;} // error C2597: illegal reference to non-static member 'A::a'

int _tmain(int argc, _TCHAR* argv[ ])
{
A a;
return 0;
}
```

```
//  делаем a - статическим
struct A{
 static  int a;
 A(){a=10;}
 static int f();
};

int A::f(){return a;}

int _tmain(int argc, _TCHAR* argv[])
{
A a;
return 0;
}
// Получаем ошибку линковщика:
 error LNK2001: unresolved external symbol "public: static int A::a" (?a@A@@2HA)
```

```
//  статический член- данных класса
//  нужно обязательно где-то  в программе инициализировать:
struct A{
 static  int a;
 A(){a=10;}
 static int f();
};

 int A::a=99; // инициализация

int A::f(){return a;}

int _tmain(int argc, _TCHAR* argv[ ])
{
A a;
return 0;
} //  Теперь работает!
```

Singleton

(по материалам cpp-reference.ru/patterns/creational-patterns/singleton/)

Часто в системе могут существовать сущности **только** в единственном экземпляре, например, система ведения системного журнала сообщений или драйвер дисплея. В таких случаях необходимо уметь создавать единственный экземпляр некоторого типа, предоставлять к нему доступ извне и **запрещать** создание нескольких экземпляров того же типа.

Паттерн Singleton предоставляет такие возможности.

Архитектура паттерна Singleton основана на идее использования глобальной переменной, имеющей следующие важные свойства:

- переменная доступна на всем протяжении времени выполнения программы.
- переменная предоставляет глобальный доступ, то есть доступна из любой части программы.

Однако, использовать глобальную переменную некоторого типа непосредственно невозможно, так как существует проблема обеспечения **единственности** экземпляра, а именно, возможно создание нескольких переменных того же самого типа.

Для решения этой проблемы паттерн Singleton возлагает контроль над созданием единственного объекта на сам класс.

Доступ к этому объекту осуществляется через **статическую** функцию-член класса, которая возвращает указатель или ссылку на него.

Этот объект будет создан только при первом обращении к методу, а все последующие вызовы лишь возвращают его адрес. Для обеспечения уникальности объекта, конструкторы и оператор присваивания объявляются закрытыми.

Классическая реализация.

```
class Singleton {   // хидер  "Singleton.h"

  private:
    static Singleton * p_instance;
    Singleton() ;
    Singleton( const Singleton& );
    Singleton& operator=( Singleton& );
  public:
    static Singleton * getInstance() {
        if( ! p_instance)
            p_instance = new Singleton();
        return p_instance;
    }
};

// Singleton.cpp
#include "Singleton.h"

Singleton* Singleton::p_instance = 0;
```

Реализация С.Мейерса.

```
// Singleton.h
class Singleton
{
private:
    Singleton() ;
    Singleton( const Singleton&);
    Singleton& operator=( Singleton& );
public:
    static Singleton& getInstance() {
        static Singleton  instance;
        return instance;
    }
};
```

Недостатки:

- сложности создания объектов производных классов
- невозможность безопасного доступа нескольких клиентов к единственному объекту в многопоточной среде.

```
// Усовершенствованная реализация.
class Singleton;  // опережающее объявление

class SingletonDestroyer {
  private:
    Singleton* p_instance;
  public:
    ~SingletonDestroyer();
    void initialize( Singleton* p );
};
class Singleton {
  private:
    static Singleton* p_instance;
    static SingletonDestroyer  destroyer;
  protected:
    Singleton() { }
    Singleton( const Singleton& );
    Singleton& operator=( Singleton& );
   ~Singleton() { }
    friend class SingletonDestroyer;
  public:
    static Singleton& getInstance();
};
```

```
  // Singleton.cpp
#include "Singleton.h"

Singleton * Singleton::p_instance = 0;
SingletonDestroyer  Singleton::destroyer;

SingletonDestroyer::~SingletonDestroyer() {
    delete p_instance;
}
void SingletonDestroyer::initialize( Singleton* p ) {
    p_instance = p;
}
Singleton& Singleton::getInstance() {
    if( !p_instance)     {
        p_instance = new Singleton();
        destroyer.initialize( p_instance);
    }
    return *p_instance;
}
```

Достоинства паттерна Singleton:

- Класс сам контролирует процесс создания единственного экземпляра.
- Паттерн легко адаптировать для создания нужного числа экземпляров.
- Возможность создания объектов классов, производных от Singleton.

Недостатки Singleton:

- В случае использования нескольких взаимозависимых одиночек их реализация может резко усложниться.

Динамическая информация о типах

Еще раз возвращаюсь к этому вопросу.

Существует возможность узнавать информацию о классе объекта и даже изменять этот класс прямо во время выполнения программы.

Мы вкратце рассмотрим два механизма, которые служат для этого: операция **dynamic_cast** и оператор **typeid**. Их можно рассматривать как дополнительные средства языка.

Эти возможности используются, когда базовый класс имеет ряд порожденных классов, созданных порой довольно непростым путем.

Для того чтобы использовать динамический подход, базовый класс обязан быть полиморфным, то есть он должен содержать по крайней мере одну виртуальную функцию.

Повторный пример

```
#include <typeinfo>       ( из книги Лафоре)
#include <iostream>
using namespace std;

class Base {
     virtual void vertFunc() {  }
};
class Derv1 : public Base { };
class Derv2 : public Base { };

bool  isDerv1 ( Base* pUnknown)
{          // неизвестный подкласс базового
    Derv1* pDerv1;
if( pDerv1 = dynamic_cast<Derv1*> (pUnknown))
    return true;
else
    return false;
}
```

```
int main()
{
    Derv1* d1 = new Derv1;
    Derv2* d2 = new Derv2;
if( isDerv1( d1) )
     cout << "d1 - компонент класса Derv1\n";
else
     cout << "d1 - не компонент класса Derv1\n";
if(isDerv1( d2) )
     cout << "d2 - компонент класса Derv1\n";
else
     cout << "d2 - не компонент класса Derv1\n";
   return 0;
}
```

Добавляю в него новые элементы

```
#include <typeinfo>
#include <iostream>
using namespace std;

class Base {
   virtual void vertFunc() {  }
};
class Derv1 : public Base { };
class Derv2 : public Base   { };
class Derv3 : public  Derv2   { };  // еще один  класс

bool  isDerv1(Base* pUnknown) {
        // неизвестный подкласс базового
   Derv1* pDerv1;
   if (pDerv1 = dynamic_cast<Derv1*> (pUnknown))
      return true;
   else
      return false;
}
```

```
template<typename T>
bool  isType( Base* pUnknown) // проверочная функция - шаблонная
{
    T* t_obj;
    if ( t_obj = dynamic_cast<T*> (pUnknown))
        return true;
    else
        return false;
}

int main() {
      setlocale(LC_ALL, "ru");
    Derv1* d1 = new Derv1;
    Derv2* d2 = new Derv2;
    if (isDerv1(d1))
        cout << "d1 - компонент класса Derv1\n";
    else
        cout << "d1 - не компонент класса Derv1\n";
    if (isDerv1(d2))
        cout << "d2 - компонент класса Derv1\n";
    else
        cout << "d2 - не компонент класса Derv1\n\n";
```

```
if (isType<Derv1>(d1))
      cout << "d1 - компонент класса Derv1\n";
   else
      cout << "d1 - не компонент класса Derv1\n";
   if (isType<Derv1> (d2))
      cout << "d2 - компонент класса Derv1\n";
   else
      cout << "d2 - не компонент класса Derv1\n\n";


   if (isType<Derv2>(d1))
      cout << "d1 - компонент класса Derv2\n";
   else
      cout << "d1 - не  компонент класса Derv2\n";
   if (isType<Derv2> (d2))
      cout << "d2 - компонент класса Derv2\n\n";
   else
      cout << "d2 - не компонент класса Derv2\n\n";
```

```
Derv3* d3 = new Derv3;
   if (isType<Derv3>(d3))
      cout << "d3 - компонент класса Derv3\n";
   else
      cout << "d3 - не  компонент класса Derv3\n";
   if (isType<Derv3> (d2))
      cout << "d2 - компонент класса Derv3\n\n";
   else
      cout << "d2 - не компонент класса Derv3\n\n";

   if (isType<Derv2>(d3))
      cout << "d3 - компонент класса Derv2\n";
   else
      cout << "d3 - не  компонент класса Derv2\n";
   if (isType<Derv2>(d2))
      cout << "d2 - компонент класса Derv2\n\n";
   else
      cout << "d2 - не компонент класса Derv2\n\n";

   return 0;
}
```

```
d1 - компонент класса Derv1
d2 - не компонент класса Derv1

d1 - компонент класса Derv1
d2 - не компонент класса Derv1

d1 - не  компонент класса Derv2
d2 - компонент класса Derv2

d3 - компонент класса Derv3
d2 - не компонент класса Derv3

d3 - компонент класса Derv2
d2 - компонент класса Derv2
```

Присваивание

```
      // Общий вид.
class  A {
public:
 A(){ }                  // конструктор по умолчанию
 ~A() { }                // деструктор
 A( const A&  obj) {  … } // конструктор копирования
A&  operator= ( const A&  obj) { // оператор присваивания копированием
    // что-то делает
 return   *this;
}

A( const A&&  obj) {  … } // конструктор перемещения
A&  operator= ( const A&&  obj) { // оператор присваивания перемещением
    // что-то делает
 return   *this;
}

}
```

R-value и L-value ссылки

Rvalue ссылка – это составной тип похожий на традиционную ссылку в C++.

Чтобы различать эти два типа, традиционную C++ ссылку называют lvalue ссылка.

По семантике lvalue ссылка формируется путём помещая & после некоторого типа.

```
A a;
A&    a_ref1 = a;      // это lvalue ссылка
```

А если после некоторого типа поместить &&, то получится rvalue ссылка.

```
A a;
A&&   a_ref2 = a;  // это rvalue ссылка
```

Rvalue ссылка ведет себя точно так же, как и lvalue ссылка, за исключением того, что она может быть связана с временным объектом, тогда как lvalue связать с временным (не константным) объектом нельзя.

```
A&  a_ref3 = A();  // Ошибка!
A&& a_ref4 = A();  // Ok
```

Комбинация rvalue ссылок и lvalue ссылок необходима для реализации семантики перемещения (move semantics). Rvalue ссылка может также использоваться для достижения идеальной передачи (perfect forwarding), что ранее было нерешенной проблемой в C++.

Семантика перемещения

Одной из наиболее значимых возможностей, введенных в C++ 11, была семантика перемещения. Можно использовать ее для оптимизации копирований и присваиваний, перемещая (“эстафетой") внутренние ресурсы из исходного объекта в объект назначения **вместо копирования** этого содержимого. Это может быть выполнено при условии, что исходный объект больше не нуждается в своем внутреннем значении или состоянии (потому что все равно будет отброшен).

Семантика перемещения оказывает значительное влияние на дизайн шаблонов, и для ее поддержки в обобщенном коде были введены специальные правила.

Предположим, что нужно написать обобщенный код, который передает фундаментальные свойства передаваемых аргументов.

Модифицируемый объект должен оставаться модифицируемым.

Константный объект должен быть передан как константный.

Перемещаемый объект (из которого можно «взять» его внутреннее содержимое за ненужностью) должен передаваться как перемещаемый объект.

```
#include <utility>
#include <iostream>

#include <io.h>      // _setmode
#include <fcntl.h>   //  _O_U16TEXT

class X {  };
void g(X&) {
      std::wcout << L"g () для переменной \n";
}
void g(X const&) { }

void g(X&&){
      std::wcout <<L"g() для перемещаемого объекта \n";
}
   // Передача функцией f() аргумента val в g():
void f(X& val){
      g(val); // val – неконстантная  lvalue => вызов g(X&)
}
void f(X const& val){
g(val); // val – константная  lvalue => вызов g(X const&)
}
```

```
void f(X&& val) {
       g(std::move(val)); // val - неконстантное lvalue =>
             // необходим std::move() для вызова g(X &&)
}

int main() {
    _setmode(_fileno(stdout), _O_U16TEXT);
    _setmode(_fileno(stdin),  _O_U16TEXT);
   _setmode(_fileno(stderr), _O_U16TEXT);

X v; // Создание переменной
X const c; // Создание константы
f(v); // Для неконстанты вызов f(X&) => g(X&)
f(c); // Для константы вызов f(X const&) => g(X const&)
f(X()); // Для временного объекта вызов f(X&&) => g(X&&)
f(std::move(v));     // Для перемещаемой переменной
                     // вызов f(X&&) => g(X&&)
}
```

Если мы попытаемся объединить все три случая в обобщенном коде, то столкнемся с проблемой:

```
template<typename T>
void f( T& val) {
      g(val) ;
}
```

Этот код работает для двух первых случаев, но не для третьего, когда передается перемещаемый объект.

По этой причине C++11 вводит **специальные правила** для прямой передачи (идеальной передачи, **perfect forwarding**) параметров. Идиоматическая схема кода имеет следующий вид:

```
template<typename T>
void f (T&& val) {
      g(std::forward<T>(val));   // Прямая передача val в g()
}
```

std: :move () не имеет шаблонного параметра и “запускает” семантику перемещения для передаваемого аргумента, в то время как std::forward < > () “передает” потенциальную семантику перемещения в зависимости от переданного аргумента шаблона

T&& для параметра шаблона T ведет себя иначе, чем X&& для конкретного типа X. Однако синтаксически они выглядят одинаково.

- X&& для конкретного типа X объявляет параметр как r-ссылку (rvalue). Она может быть связана только с перемещаемым объектом (например, временный объект, и объект, переданный с использованием std: :move (); Она всегда изменяема, и вы всегда можете “взять” ее значение.
- T&& для параметра шаблона T объявляет передаваемую ссылку (forwarding reference), именуемую также **универсальной ссылкой** (universal reference).

Термин универсальная ссылка был придуман Скоттом Мейерсом как общий термин, который может означать и Lvalue, и Rvalue. Из-за слишком большой универсальности термина “универсальный” стандарт C++17 вводит термин **передаваемая ссылка** (forwarding reference), так как основной причиной использования такой ссылки является передача объектов.

Сжатие ссылок (reference collapsing

(по материалу из https://habr.com/ru/post/242639/)

Как известно, взятие ссылки на ссылку в C++ не допускается, но это иногда может происходить при реализации шаблонов:

```
template <typename T>
void baz (T t) {
  T& k = t;
}
```

Что случится, если вызвать эту функцию следующим образом:

```
 int ii = 4;
baz<int&>(ii) ;
```

При инстанцировании шаблона T установится равным int&. Какой же тип будет у переменной k внутри функции baz ? Компилятор «увидит» int& & — а так как это запрещенная конструкция, компилятор просто преобразует это в обычную ссылку. Фактически, до C++11 такое поведение не было стандартизированным, но многие компиляторы принимали и преобразовывали такой код, так как он часто встречается в метапрограммировании. После того, как в C++11 были добавлены rvalue-ссылки, стало важным определить поведение при совмещении различных типов ссылок (например, что значит int&& & ?).

Так появилось правило сжатия ссылок. Это правило очень простое – одиночный амперсанд (&) всегда побеждает. Таким образом – (& и &) это (&), также как и (&& и &), и (& и &&). Единственный случай, при котором в результате сжатия получается (&&) — это (&& и &&). Это правило можно сравнить с результатом выполнения логического ИЛИ, в котором & это 1, а && это 0.

Другое дополнение C++, имеющее прямое отношение к рассматриваемой теме – это правила особого вывода типа (special type deduction rules) для rvalue-ссылок в различных случаях. Рассмотрим пример шаблонной функции:

```
template <class T>
void func(T&&   t) {
}
```

Не позволяйте двойному амперсанду обмануть вас – t здесь не является rvalue-ссылкой. При появлении в данной ситуации (когда необходим особый вывод типа), T&& принимает особое значение – когда func инстанцируется, T изменяется в зависимости от переданного типа. Если была передана lvalue типа U, то T становится U&. Если же U это rvalue, то T становится просто U.

Пример:

```
func(4);          // 4 это rvalue: T становится int

double d = 3.14;
func(d);          // d это lvalue; T становится double&

float f() {...}
func(f());        // f() это rvalue; T становится float

int bar(int i) {
  func(i);        // i это lvalue; T становится int&
}
```

Это правило может показаться необычным и даже странным. Оно такое и есть. Но, тем не менее, это правило становится вполне очевидным, когда приходит понимание что это правило помогает решить проблему перемещения.

std::move и std::forward

Функция **move** выполняет простую работу. Её задача: принять либо lvalue, либо rvalue параметр, и вернуть его как rvalue без вызова конструктора копирования:

```
template <class T>
typename   remove_reference<T>::type&&
move(T&& a) {
    return a;
}
```

std::**forward** тоже преобразует lvalue в rvalue ссылку, но только в том случае, если аргумент передан в функцию по rvalue ссылке.

```
template <typename T> // может принять объект и по rvalue, и по lvalue ссылке
void bar(T && v) {
  vector<int> a = v;  // всегда будет вызвано копирование, даже если v - rvalue

  vector<int> a = move (v); // всегда будет вызвано перемещение,
                            //   даже если v - lvalue
  vector<int> a = forward <T>(v); // будет вызвано копирование для lvalue
                                  //  и перемещение для rvalue
}
```

Примерный класс с конструктором перемещения

Из ru.stackoverflow.com/questions/490753/Конструктор-перемещения

```
#include <vector>

class Buffer {
private:
    void destroy()
    {
        if (pBuff)
            delete[ ] pBuff;
    }

    char*      pBuff;
    size_t    buffSize;
```

```
public:
   Buffer(const std::string& buff)
   :   pBuff(nullptr)     ,   buffSize( buff.length() )
   {
       pBuff = new char[buffSize];
       memcpy(pBuff, buff.c_str(), buffSize);
   }

   ~Buffer(){  destroy(); }

  Buffer( const Buffer& other)
   :   pBuff( nullptr )
   ,   buffSize( other.buffSize)  {
       pBuff = new char[buffSize];
       memcpy(pBuff, other.pBuff, buffSize);
   }
```

```
Buffer& operator=(const Buffer& other)
   {
      destroy();
      buffSize = other.buffSize;
      pBuff = new char[buffSize];
      memcpy(pBuff, other.pBuff, buffSize);
      return *this;
   }
```

```
Buffer(Buffer&& tmp)
   :  pBuff(  tmp.pBuff )   ,   buffSize( tmp.buffSize )
  {
      tmp.pBuff = nullptr;
   }

   Buffer& operator=(Buffer&& tmp)
   {
      destroy();
      buffSize = tmp.buffSize;
      pBuff = tmp.pBuff;
      tmp.pBuff = nullptr;
      return *this;
   }

};  //  конец класса
```

```
Buffer CreateBuffer( const std::string& buff)
{
    Buffer retBuff(buff);
    return retBuff;
}

int main()
{
    Buffer buffer1 = CreateBuffer("123");  // срабатывает конструктор
     перемещения
    Buffer buffer2 = buffer1;             // срабатывает конструктор копирования
    buffer2 = CreateBuffer("123");        // срабатывает конструктор
                    //  перемещения, а затем оператор перемещения

    buffer2 = buffer1;      // срабатывает оператор присваивания копии
}
```

Пример для понимания выигрыша перемещения

```
class Intvec {
private:
    void log(const char* msg)
    {
        cout << "[" << this << "] " << msg << "\n";
    }

    size_t m_size;
    int* m_data;

public:
    explicit Intvec(size_t num = 0)  : m_size(num), m_data( new  int[ m_size] )     {
        log("constructor");
    }
```

```
~Intvec()    {
        log("destructor");
        if (m_data) {
            delete[ ] m_data;
            m_data = 0;
        }
    }
    Intvec(const Intvec& other)  : m_size(other.m_size), m_data(new int[m_size] )   {
        log("copy constructor");
        for (size_t i = 0; i < m_size; ++i)
            m_data[i] = other.m_data[i];
    }
    Intvec& operator=(const Intvec& other)     {
        log("copy assignment operator");
              Intvec tmp(other);
        std::swap(m_size, tmp.m_size);
        std::swap(m_data, tmp.m_data);
        return *this;
    }   };
```

```
void copy(){
Intvec v1(20);
Intvec v2;

cout << "assigning lvalue...\n";
    v2 = v1;  // присваивание копированием
cout << "ended assigning lvalue...\n";
}

int _tmain(int argc, _TCHAR* argv[])
{
    copy();
return 0;
}
```

Фрагмент вывода:

```
assigning lvalue...
[008FFAA4] copy assignment operator
[008FF99C] copy constructor
[008FF99C] destructor
ended assigning lvalue...
```

```
   // теперь создадим еще один вариант функции copy
void copy2 (){
Intvec v1(20);
Intvec v2;

cout << "assigning lvalue...\n";
     v2 = Intvec  ( 40) ;  // Intvec  ( 40)  - это rvalue ссылка
cout << "ended assigning lvalue...\n";
}

int _tmain(int argc, _TCHAR* argv[ ])
{
      copy2();
    return 0;
}
```

Фрагмент вывода:

assigning lvalue...
[004FF914] constructor
[004FF9E4] copy assignment operator
[004FF8C4] copy constructor
[004FF8C4] destructor
[004FF914] destructor
ended assigning lvalue...

Видим лишний вызов конструктора и деструктора

// тогда добавляем в класс оператор копирования перемещением

```
Intvec& operator= (Intvec&&  other)
{
    log("move assignment operator");
    std::swap(m_size, other.m_size);
    std::swap(m_data, other.m_data);
    return *this;
}
```

Двойной асперсанд — это ссылка на rvalue. Он означает как раз то, что и даёт **ссылку на rvalue**, который будет уничтожен после вызова. Мы можем использовать этот факт, чтобы просто позаимствовать внутренности rvalue.

Фрагмент вывода:

assigning lvalue...
[00F8F8A4] constructor
[00F8F974] move assignment operator
[00F8F8A4] destructor
ended assigning lvalue...

Лишние вызовы исчезли.

Вызывается оператор присваивания перемещением, так как rvalue присваивается v2.

Вызовы конструктора и деструктора всё же необходимы для временного объекта, который создаётся через Intvec(33).

Однако другой временный объект внутри оператора присваивания больше не нужен. Оператор просто меняет внутренний буфер rvalue со своим, и таким образом деструктор rvalue удаляет буфер самого объекта, который больше не будет использоваться.

(по материалам https://habr.com/ru/post/348198/)

Домашнее задание на неделю

Проект 34.

Создать базовый и производный класс полиморфной иерархии **именем своей фамилии и имени**, распределив их по своим модулям h/cpp.

Создать в базовом классе член-данных m_txt нужного типа и инициализировать его прочитанными данными из подготовленного заранее файла с текстом басни И.А.Крылова «Стрекоза и муравей» (прочесть первую половину басни). В производном классе: тоже член-данных m_txt. Его также заполнить данными из второй половины того же файла.

Реализовать отдельный от иерархии **шаблонный** класс базы данных DB<T>, который внутри содержит **vector**.

В функции main создать 6-8 объектов базового и производного типов и положить их в объект типа DB, а затем скопировать в другой объект типа DB.

Далее последовательно удалить через **erase** объекты из второй БД с условием, чтобы остались внутри нее только один объект базового класса и один - производного. К объекту производного типа применить, выводя m_txt :

wcout<< db2;

Должна быть распечатана, таким образом, вся басня.

Контрольная работа

Пусть есть класс Storage, а внутри

```
vector < You_Base_Name*> _db;
```

Реализовать для него конструкторы копирования и перемещения.