Бублик Володимир Васильович

Об'єктно-орієнтоване програмування

Частина 1. Об'єктне програмування.

Лекція 2. Копіювання об'єктів

Лекції для студентів 2 курсу

Повторення

- Що має бути в класі
- class T
- {
- private:
- // Тут розміщують атрибути
- public:
- // Конструктори
- T(T1,…,Tn);
- // Деструктор
- ~T(); // далі селектори, модифікатори, …
- };

Приклад класу. String

```
class String {
private:
  size_t _len;
  char* _allocator;
public:
  String();
  String(const char*);
  String(const char);
  ~String();
  size_t length() const {return _len;}
  bool empty() const {return _len==0;}
  void clear() {*this=String();}
};
```

Констуктори

- Для чого у класі три різних конструктори?
- class String {
- private:
- size_t _len;
- char* _allocator;
- public:
- String();
- String(const char*);
- String(const char);
- .....
- };

Властивість інкапсуляції

- Відокремлення реалізації класу від його визначення

```
String::String (const char c):
 _allocator( new char [2]),
 _len(1)
{
 _allocator [0]=c;
 _allocator [1]='\0';
 return;
}
```

Питання

- Чому у визначенні класу розміщені реалізації?
- class String
- {
- private:
- size_t _len;
- char* _allocator;
- public:.....
- size_t length() const {return _len;}
- bool empty() const {return _len==0;}
- void clear() {*this=String();}
- };

Питання

- Чи коректний параметр замовчування? — Ні. Чому?

```
class String
{
private:
 size_t _len;
 char* _allocator;
public:
 String ();
 String (const char* ps=0);
 String (const char);
 ~String ();
};
```

Конструктор копіювання

```
class T
{
 T(T1,...,Tn);
 ~T();
//  конструктор копіювання
//  створює новий об'єкт, ідентичний
//  переданому параметром
 T(const T&);
//  Можливий варіант: T(T&);
//  але не T(T)
};
```

Використання

Конструктор копіювання викликається **кожного** разу, коли параметр або результат передаються значеннями

```
T1 f(T2 x)
{
 T1 y;
//     тіло f...
 return y;
}
```

```
a=f(b);//   T2 x(b); T1 y; тіло f... ; a = T1(y);
```

Облік об'єктів (Off top)

Інвентаризація об'єктів

```
class Point
{
 static int _freeID;
 const int _pointID;
 double _x;
 double _y;
public:
 Point (double x=0, double y=0);
 Point (const Point &);
 ~Point();
};
```

Конструктор Point

```
Point::Point (double x, double y):
        _x (x),
        _y (y),
        pointID (++_freeID)
{
#ifdef NDEBUG
 cout<<pointID<<": created "<<*this<<endl;
#endif
 return;
};
```

// Де коректно розмістити замовчування параметру?

Копіювальний конструктор Point

```
Point::Point (const Point & u):
        _x (u._x),
        _y (u._y),
        pointID(++_freeID)
{
#ifdef NDEBUG
 cout<<pointID<<": copied  "<<*this<<endl;
#endif
 return;
};
```

// Чи може копіювання мати замовчуваний параметр?

Замовчування у копіювальному конструкторі

```
class Foo;
int main() {
 Foo f1(10);  // Створення нового об'єкту
 Foo f2(f1);  // Копіювання існуючого об'єкту
 Foo f3;   // Це що? Наперед невідомо
}
```

Перший варіант

```
class Foo {
private:
 int _k;
 static int _freeid;
 const int _id;
public:
 Foo(int k=0):_k(k), _id(++_freeid){
    cout<<"New Foo id="<<_id<<endl;}
 Foo(const Foo& foo=0):_k(foo._k), _id(++_freeid){
    cout<<"New copy Foo id="<<_id<<endl;}
}
```

- warning C4520: 'Foo' : multiple default constructors specified
- Чому все-таки це дозволено? `Foo& foo=0` проігноровано

Другий варіант

```
class Foo {
private:
 int _k;
 static int _freeid;
 const int _id;
 static Foo _static_foo;
public:
Foo(int k=0):_k(k), _id(++_freeid){
 cout<<"New Foo id="<<_id<<", _k="<<_k<<endl;}
Foo(const Foo& foo=_static_foo):_k(foo._k),_id(++_freeid){
 cout<<"New copy Foo id="<<_id<<", _k="<<_k<<endl;}
};
```

- Тепер ігнорується `int k=0`

Експеримент

```
int Foo::_freeid = 0;
Foo Foo::_static_foo(100);
int main() {
 cout<<"START"<<endl;
 Foo f1(10);
 Foo f2; // Копія об'єкту _static_foo
}
```

Зверніть увагу на порядок виконання дій

New Foo id=1, _k=100

START

New Foo id=2, _k=10

New copy Foo id=3, _k=100

Інший експеримент

```
int Foo::_freeid = 0;

int main() {
 cout<<"START"<<endl;
 Foo f1(10);
 Foo f2; // Копія об'єкту _static_foo
}
Foo Foo::_static_foo(100);

// Щось зміниться, якщо визначення Foo::_static_foo
// перенести до іншого місця?
```

Деструктор Point

```
Point::~Point()
{
#ifdef NDEBUG
 cout<<pointID<<": removed "<<*this<<endl;
#endif
 return;
};
```

Передача об'єктів параметрами

Значенням

```
Point operator+ (Point u, Point v)
{
 Point res(u.x()+v.x(), u.y()+v.y());
 return res;
}
```

Відсилками

```
ostream& operator<<(ostream &os, const Point& u)
{
 os<<'('<<u.x()<<','<<u.y()<<')';
 return os;
}
```

```
int main()
{
 Point a(1,2);
 Point b(5);
 a+b;
 return 0;
}
```

Протокол

```
1: created (1,2)   //a
2: created (5,0)   //b
3: copied  (5,0)   //v
4: copied  (1,2)   //u
5: created (6,2)   //res
6: copied  (6,2)   //return
5: removed (6,2) //res
4: removed (1,2) //u
3: removed (5,0) //v
6: removed (6,2) //returned
2: removed (5,0) //b
1: removed (1,2) //a
```

Вправа до передачі об'єктів параметрами

Що зміниться в протоколі, якщо у виводі забрати сталу відсилку?

```
ostream& operator<<(ostream &os, Point u)
{
 os<<'('<<u.x()<<','<<u.y()<<')';
 return os;
}
```

Без локальної змінної

```
Point operator+ (Point u, Point v)
{
/*   Замість
 Point res(u.x()+v.x(), u.y()+v.y());
 return res;
*/
 return Point ( u.x()+v.x(), u.y()+v.y() );
}
```

Протокол 2

```
int main()
{
 Point a(1,2);
 Point b(5);
 a+b;
 return 0;
}
```

```
1: created (1,2)   //a
2: created (5,0)   //b
3: copied  (5,0)   //v
4: copied  (1,2)   //u
5: created (6,2)   //return
4: removed (1,2) //u
3: removed (5,0) //v
5: removed (6,2) //returned
2: removed (5,0) //b
1: removed (1,2) //a
```

Сталі відсилки

```
Point operator+ (const Point & u, const Point & v)
{
 return Point ( u.x()+v.x(), u.y()+v.y() );
}
```

Протокол 3

```
int main()
{
 Point a(1,2);
 Point b(1);
 a+b;
 return 0;
}
```

- 1: created (1,2) //a
- 2: created (5,0) //b
- 3: created (6,2) //return
- 3: removed (6,2) //returned
- 2: removed (5,0) //b
- 1: removed (1,2) //a

Урок передачі параметрів

- Передаючи параметр і одержуючи результат, усвідомлюйте, з чим маєте справу: з оригіналом чи копією

Копіювання агрегатів

```
class WrappedVector
{
private:
 static const size_t n;
 double * v;
public:
 WrappedVector();
 WrappedVector(const WrappedVector&);
 ~WrappedVector();
};
```

Копіювальний конструктор вектора

```
WrappedVector::
 WrappedVector (const WrappedVector& vec):
 _v (new double[_n])
{
 for (size_t i=0; i<_n; i++)
      _v[i] = vec._v[i];
 return;
}
```

// Як бути з нестачею пам'яті?

Копіювальний конструктор за замовчуванням

```
WrappedVector::
 WrappedVector (const WrappedVector& vec):
 _v (vec._v)
{ };

//   Чим закінчиться виконання програми?
int main()
{
 WrappedVector u, v(u);
 return 64; //  катастрофою!!!
}
```

Копіювання присвоєнням

```
class WrappedVector
{
private:
 static const size_t n;
 double * v;
public:
 WrappedVector();
 WrappedVector(const WrappedVector&);
 ~WrappedVector();
 WrappedVector& operator= (const WrappedVector&);
};
```

Реалізація копіювального присвоєння

```
WrappedVector&
 WrappedVector::operator=
      (const WrappedVector& vec)
{
//Нам поталанило: vec і this мають одну й ту ж довжину

 for (size_t i=0; i<n; i++)
      v[i] = vec.v[i];
 return *this;
}
```

Присвоєння за замовчуванням

```
WrappedVector& WrappedVector :: operator=
      (const WrappedVector& vec)
{
 v = vec;
 return *this;
}
//   Чим закінчиться виконання програми?
int main()
{
 WrappedVector u, v;
 u=v;
 return 64;//  знову катастрофою!!!
}
```

Вектори різної довжини

```
class DissimilarVector
{
private:
 size_t _n; //non static, non const(?)
 double * _v;
public:
 DissimilarVector(int);
 DissimilarVector(const DissimilarVector&);
 ~DissimilarVector();
 DissimilarVector& operator=(const DissimilarVector&);
};
```

Конструктор вектора

```
DissimilarVector::
 DissimilarVector (size_t len) :
       _n (len),
       _v (new double[n])
{
 for (size_t i=0; i<_n; i++)
       _v[i] = 0;
 return;
}
```

Копіювальний конструктор

```
DissimilarVector::
 DissimilarVector (const DissimilarVector& vec):
        _n (vec._n),
        _v (new double[vec._n])
{
 for (size_t i=0; i<_n; i++)
        _v[i] = vec._v[i];
 return;
}
```

Чому атрибут _n не може бути сталим?

1. Спробуйте присвоєння за замовчуванням
2. Навіть копіювальне присвоєння, взагалі кажучи, не працюватиме

Копіювальне присвоєння

```
DissimilarVector& DissimilarVector::operator=
                (const DissimilarVector& vec)
{
//1. Видалити старий об'єкт
  if (this==&vec)
        return *this;
  delete [] _v;
//2. Створити новий об'єкт
  _n = vec._n
  _v = new double[_n];
//3. Скопіювати значення
  for (size_t i=0; i<_n; i++)
        _v[i] = vec._v[i];
  return *this;
}
```

_n = vec._n

Рядки з копіюванням

```
class String
{
private:
 size_t _len;
 char* _allocator;
public:
 String();
 String(const char*);
 String(const char);
 String (const String & s);
 ~String();
};
```

Копіювальний конструктор рядка

```
String::String(String& s)
 _len( s._len),
 _allocator( new char[_len+1])
{
 strcpy(_allocator, s._allocator);
 return;
};
```

Редагування оригіналу (без const)

```
class String
{
private:
  size_t _len;
  char* _allocator;
  int _amountOfCopies;
public:
  String();
  String(const char*);
  String(const char);
  String (String & s);
  ~String();
};
```

Копіювання з редагуванням

```
String::String(String& s)
 _amountOfCopies (0),
 _len( s._len),
 _allocator( new char[_len+1])
{
//    Кількість копій, зроблених з оригіналу
//    збільшується на одиницю
 s._amountOfCopies++;
 strcpy(_allocator, s._allocator);
 return;
};
```

Мультиконструктор копіювання

```
class String
{
public:
 String();
 String(const char*);
 String(const char);

 String(const String & s, int multiplayer=1);
 ~String();
};
```

Реалізація мультиконструктора копіювання

```
String:: String(const String & s, int multiplayer):
 _len (s._len*multiplayer),
 _allocator (new char [_len+1])
{
 char * target = _allocator;
 for (int i=0; i<multiplayer; i++)
 {
     strcpy(target, s._allocator);
     target+=s._len;
 }
 return;
};
```

Застосування копіювання

- // Common constructor
- String s(p);
- // Copy version of multiplication constructor
- String ss(s);
- // Multiplication constructor
- String s10(s,10);

Проблема замовчуваного параметру

- Що станеться, якщо замовчуваний параметр перенести до реалізації? — Катастофа

- class String
- {
- public:
- String(const String & s, int multiplayer);
- };

- String:: String(const String & s, int multiplayer=1):…{…;}

Чому?

Некоректне копіювання

- #include “String.h”

- // Common constructor
- String s(p);
- // **Default** copy constructor
- String ss(s);
- // Multiplication constructor
- String s10(s,10);

Сигнатури присвоєнь

Якій з сигнатур віддати перевагу?

1. `void operator=( T&);`
2. `T operator=( T&);`
3. `T& operator=( T );`
4. `T operator=( T );`
5. `T& operator=( T&);`

Сигнатури присвоєнь

Якій з сигнатур віддати перевагу?

```
1.  void operator=( T&);    //  Як бути з x=y=z;
2.  T    operator=( T&);    //  чим копіювати результат?
3.  T&   operator=( T  );    //  чим копіювати параметр?
4.  T    operator=( T  );    //  див 2 і 3 разом
5.  T&   operator=( T&);    //  ОК!!!
```

Сигнатури присвоєнь

Якій з сигнатур віддати перевагу?

1. `void operator=( T&);`
2. `T operator=( T&);`
3. `T& operator=( T );`
4. `T operator=( T );`
5. `T& operator=( T&);`

`T& operator=(const T&);`

Що таке this?

```
class T
{
public:
 T(T1,...,Tn);
 ~T();
 T(const T&);
 T& operator= (const T&);
};
```

this має тип T * const

Чому * const?

- **this** не можна перемістити на інший об'єкт

- this = anything; не коректно

Повернення значення в присвоєнні

```
Point& Point::operator=(const Point & u)
{
 this ->_x = u._x;
 *this._y   = u._y;
 return *this;
}
```

Рядки з присвоєнням

```
class String
{
public:
 String();
 String(const char*);
 String(const char);

 String(const String & s, int multiplayer=1);
 String& operator=(const String&);
 ~String();
};
```

Присвоєння рядків

```
String& String::operator=(const String& s)
{
 if (this==&s)
      return *this;
 delete [] _allocator;
 _len = s._len;
 _allocator = new char[_len+1];
 strcpy(_allocator, s._allocator);
 return *this;
}
```

Висновки

- Конструктор копіювання створює новий об'єкт
- Присвоєння звичайно замінює існуючий об'єкт іншим об'єктом (навіть якщо не доводиться видаляти попередні значення)
- Присвоєння **не можна** визначити поза класом
- Присвоєння в класі T має тип T& (чому?)
- Присвоєння повертає *this, конструктори не повертають нічого (чому?)