Лекция 7. Namespace. Переопределение операторов

namespace

- Задает именованное пространство имен
- Позволяет избежать конфликты имен
- Может быть вложенным
- В отличие от классов, может расширяться

```
1.  namespace std
2.  {
3.      void sort(/*...*/){/*...*/}
4.  }
5.
6.  namespace std
7.  {
8.      void for_each(/*...*/){/*...*/}
9.  }
```

Объявление и использование namespace'ов.

```
 1.  namespace std
 2.  {
 3.      void sort(/*...*/);
 4.  }
 5.
 6.  namespace std
 7.  {
 8.      void sort(/*...*/) {/*...*/}
 9.  }
10.  // or even better(why?):
11.  void std::sort(/*...*/){/*...*/}
12.
13.  // usage:
14.  void foo()
15.  {
16.      int a [10] = {1, 2, 3, 4, /*...*/};
17.      std::sort(a, a + 10);
18.  }
```

using директива

- Можно раскрыть пространство имен:

```
1.  namespace std
2.  {
3.      void sort(/*...*/);
4.      struct vector{/*...*/};
5.  }
6.
7.  void process()
8.  {
9.      using namespace std;
10.
11.     vector v;
12.     sort(v.begin(), v.end());
13. }
```

using объявление

- А можно вносить лишь то, что нужно:

```
1.  void process()
2.  {
3.      using std::sort;
4.      using std::vector;
5.
6.      vector v;
7.      sort(v.begin(), v.end());
8.  }
9.
10. ////////////////////////////////////////////////////////////
11. ///////
12. // operations like on sets
13. namespace a { void foo(); /*...*/ }
14. namespace b { void foo(); /*...*/ }
15. namespace c
16. {
17.     using namespace a;
18.     using namespace b;
19.     using a::foo();
20. }
```

Поиск Кёнига (ADL)

- Позволяет найти функцию в пространстве имен одного из ее аргументов (крайне полезно для операторов).

```
1.  namespace long_numbers
2.  {
3.      struct lint{/*...*/};
4.      lint& operator+=(lint&, int);
5.  }
6.
7.  void process()
8.  {
9.      // --- 1 ---
10.     std::vector v;
11.     // don't need to write std::sort
12.     sort(v.begin(), v.end());
13.
14.     // --- 2 ---
15.     long_numbers::lint i;
16.     i+= 5;
17.
18.     //without Argument dependent name lookup:
19.     long_numbers::operator+=(i, 5);
20. }
```

Перегрузка операторов

- Традиционный интерфейс для объектов из предметной области, особенно математики

- Сохраняйте естественную семантику. Для класса `rectangle::operator+(int)` может увеличить его размер, а может покомпонентно сдвинуть – здесь лучше именованная функция.

- Если есть `a @= b`, то программисты ожидают и `a@b` и `b@a`.

Что можно и нельзя?

- Можно перегружать почти все операторы (кроме `::` ; `.*` ; `?:`)
- Нельзя определить новую лексему оператора. Например `a**b` вместо `pow(a, b)`. Может это `a*(*b)`?
- Нельзя изменить приоритет. Если `point::operator^` - векторное умножение, то `p1 ^ p2 + p3` это `p1 ^ (p2 + p3)`

Что можно и нельзя? (part 2)

- Если есть a@b и a=b, то само a@=b не появится. Аналогично наличие оператора a==b не сделает a!=b. Поможет boost::operators
- Нельзя изменить количество операндов
- Нельзя переопределить операторы, в которых участвуют только встроенные типы

Где и как писать операторы?

- Объявление оператора начинается со слова operator
- Можно вызвать как обычную функцию
- Можно объявлять как member, свободную или дружественную функцию

```
1.  T T::operator+(T const& other);
2.
3.  //...
4.  T a, b;
5.  a + b;
6.  a.operator+(b);
```

Примеры объявления операторов

```
    struct lint
 2. {
 3.     lint& operator++();
 4.     lint  operator++(int);
 5.
 6.     lint& operator+=(lint const&);
 7.     lint  operator+ (lint const&);
 8.
 9.     int operator[](size_t index) const;
10. };
11. // or
12.
13. lint& operator++(lint&);
14. lint  operator++(lint&, int);
15.
16. lint& operator+=(lint&, lint const&);
17. lint  operator+ (lint const&, lint const&);
18.
19. int operator[](const lint&, size_t index); // error!
```

Оператор a@b через a@=b

```
1.  struct string
2.  {
3.      string& operator+=(const char*);
4.  };
5.  string operator+(string, const char*);
6.
7.  ////////////////////////////////////////////////////////////
8.  string& string::operator+=(const char* str)
9.  {
10.     //impl ...
11.     return *this;
12. }
13.
14. string operator+(string lhs, const char* rhs)
15. {
16.     return lhs += rhs;
17. }
18. // or even better
19. string operator+(string const& lhs, const char* rhs)
20. {
21.     string tmp;
22.     tmp.reserve(lhs.size() + strlen(rhs));
23.     return (tmp += lhs) += rhs;
24. }
```

Предопределенный смысл операторов

- У операторов =; &; , есть предопределенный смысл. Но и их можно переопределить или даже закрыть

```
 1.  struct file
 2.  {
 3.      // impl ...
 4.  private:
 5.      file(file const&);
 6.      file& operator=(file const&);
 7.  };
 8.
 9.  //
10.  file a(/*...*/), b(/*...*/);
11.  a = b; // error!
```

Явное и неявное преобразование

```
1.  struct my_int
2.  {
3.      explicit my_int(int);
4.      operator int();
5.  };
6.
7.  struct my_double
8.  {
9.      my_double(double);
10.     explicit operator double(); // C++11
11. };
12.
13. void foo(my_int const&);
14. void bar(double);
15.
16. //
17. foo(5);                        // error
18. bar(double(my_double(5.))); // ok
19.
20. // never mix implicit constructor and operator
21. // my_int + my_int
22. // or
23. // int + int ?
24. my_int(5) + 6; // error, ambiguous
```

Смешанная арифметика

- Есть string s, хотим иметь возможность писать `s < "a", "a" < s, s < s`. Как?

```
1.  // way 1 (lazy):
2.  string::string(const char*); // implicit
3.  operator<(string const&, string const&);
4.
5.  // way 2 (optimal):
6.  bool less(const char*, const char*);
7.
8.  operator<(string const& lhs, const char* rhs)
9.  { less(lhs.c_str(), rhs); }
10.
11. operator<(const char*lhs, string const& rhs)
12. { less(lhs, rhs.c_str()); }
13.
14. operator<(string const&, string const&);
15. { less(lhs.c_str(), rhs.c_str()); }
```

Friend функции

- Member-функции:
  1. Доступ к `private`
  2. В области видимости класса
  3. Имеют неявный `this`
- `static` – 1 и 2
- `friend` – только 1
- `friend` должны быть объявлены прежде в обрамляющей класс области видимости или иметь параметр класса
- Объявлять можно и в `public` и в `private`
- Можно сделать `friend` целиком класс (все его функции)

Пример friend operator'a

```
1.  struct lint
2.  {
3.      friend std::ostream& operator<<
4.          (std::ostream& os, lint const& i);
5.
6.  private:
7.      std::vector<int> data_;
8.  };
9.
10. std::ostream& operator<<(std::ostream& os, lint const& i)
11. {
12.     //impl...
13.     return os;
14. }
```

Префиксный и постфиксный инкремент

```
 1. struct iterator
 2. {
 3.     // impl...
 4.     iterator& operator++();
 5.
 6. private:
 7.     T* ptr_;
 8. };
 9.
10. iterator& iterator::operator++()
11. {
12.     ++ptr_;
13.     return *this;
14. }
15.
16. iterator operator++(iterator& it, int)
17. {
18.     iterator tmp(it);
19.     ++it;
20.     return tmp;
21. }
```

Функторы

- Очень удобны для реализации callback'ов и visitor'ов

```
1.  struct rotation
2.  {
3.      rotation(double angle);
4.      void operator()(point& p);
5.  private:
6.      double angle_;
7.  };
8.
9.  void rot_points(vector<point>& vec, double a)
10. {
11.     for_each(vec.begin(), vec.end(), rotation(a));
12. }
```

Операторы для указателя

```
 1. template<class T>
 2. struct smart_ptr
 3. {
 4.     T* operator->()          { return ptr_;   }
 5.     T& operator[](size_t i) { return ptr_[i];}
 6.     T& operator* ()          { return *ptr_;  }
 7.
 8. private:
 9.     T* ptr_;
10. };
11.
12. struct X {void foo();};
13.
14. int main()
15. {
16.     smart_ptr<X> px;
17.     px->foo();
18.
19.     return 0;
20. }
```

Какими делать функции?

- Если функция – один из операторов `=`; `->`; `[]`; `()` делаем их членами (нет выбора)
- Иначе
  - если
    1. функция требует левый аргумент иного типа (`<<`)
    2. требует преобразования типа для левого аргумента,
    3. можно реализовать только через открытый интерфейс класса(!),
  - делайте ее не членом класса (возможно, в (1) и (2) – друзьями)
- Иначе, сделайте ее функцией-членом.

Вопросы?