

Алгоритмизация и программирование

Лекция 7

Принципы ООП



Инкапсуляция



Инкапсуляция

Без инкапсуляции



С инкапсуляцией



Инкапсуляция

Без инкапсуляции



С инкапсуляцией



		Модификатор члена класса		
		public	protected	private
Место использования	Класс	+	+	+
	Наследник	+	+	-
	Внешний код: • функции • другие классы	+	-	-

Дружественные функции

```
class Accumulator{
private:
    int m_value;
public:
    Accumulator() { m_value = 0; }
    void add(int value) { m_value += value; }

    // Сделаем функцию reset() другом этого класса
    friend void reset(Accumulator &accumulator);
};

// reset() теперь является другом класса Accumulator
void reset(Accumulator &accumulator)
{
    // и может получить доступ к закрытым данным объектов Accumulator
    accumulator.m_value = 0;
}

Accumulator acc;
acc.add(5); // добавляем 5 в накапливающий сумматор
reset(acc); // сбрасываем накапливающий сумматор в 0
```

Специальные функции-члены

Специальные функции-члены

```
MyBaseClass();    // Конструктор  
~MyBaseClass();  // Деструктор  
MyBaseClass(MyBaseClass const &); // Конструктор копирования  
MyBaseClass(MyBaseClass &&);      // Конструктор перемещения  
MyBaseClass operator=(MyBaseClass const &); // Оператор присваивания  
MyBaseClass operator=(MyBaseClass &&);      // Конструктор перемещения
```

Конструкторы

```
MyBaseClass();           // По умолчанию  
MyBaseClass(int a);      // С параметром  
MyBaseClass(int a=0);
```

Делегирующие конструкторы

```
MyBaseClass(int a){  
    // Много полезного кода  
}
```

```
MyBaseClass(int a, int b){  
    MyBaseClass(a); // Что здесь происходит?  
}
```

```
MyBaseClass(int a, int b): MyBaseClass(a){  
}
```

Правило трёх, пяти, нуля

Правило **Трёх** предполагает, что если вам нужно определить что-либо из **конструктора копирования, оператора присваивания копированием** или **деструктора**, то скорее всего вам нужно определить все три. [до C++11]

Правило **Пяти** предполагает, что если вам нужно определить что-либо из **конструктора копирования, оператора присваивания копированием, конструктор перемещения и оператор присваивания перемещением** или **деструктора**, то скорее всего вам нужно определить все три. [начиная с C++11]

Правило **Нуля** заключается в том, что тот сценарий, когда не нужно определять ничего из специальных функций-членов, должен быть предпочтительным:

1. Ваш класс определяет чисто значимый тип, и любое его состояние состоит из чисто значимых типов (например, примитивов).
2. Любые ресурсы, которые приходится задействовать состояниям вашего класса, управляются классами, которые специализируются исключительно на управлении ресурсами (например, умными указателями, файловыми объектами и т. д.).

Перегрузка операторов

Перегрузка операторов

- Во-первых, почти любой существующий оператор в C++ может быть перегружен. Исключениями являются: условный оператор (?:), sizeof, оператор разрешения области видимости (::), оператор выбора члена (.), оператор выбора указателя на член (.*), typeid и операторы приведения типов.
- Во-вторых, вы можете перегрузить только существующие операторы. Вы не можете создавать новые операторы или переименовывать существующие операторы. Например, вы не можете создать оператор ** для возведения в степень.
- В-третьих, по крайней мере, один из операндов в перегруженном операторе должен быть пользовательского типа. Это означает, что вы не можете перегрузить оператор плюс для работы с одним числом `int` и одним числом `double`. Однако вы можете перегрузить оператор плюс для работы с числом `int` и `Mystring`.
- В-четвертых, невозможно изменить количество операндов, поддерживаемых оператором.
- Наконец, все операторы сохраняют свой приоритет и ассоциативность по умолчанию (независимо от того, для чего они используются), и это не может быть изменено

Перегрузка операторов

- Обычная внешняя функция
- Дружественная функция
- Член класса

Перегрузка операторов

```
class Cents{  
private:  
    int m_cents;  
  
public:  
    Cents(int cents) { m_cents = cents; }  
    int getCents() const { return m_cents; }  
};
```

Перегрузка операторов

```
class Cents
{
private:
    int m_cents;

public:
    Cents(int cents) { m_cents = cents; }

    // складываем Cents + Cents с помощью дружественной функции
    friend Cents operator+(const Cents &c1, const Cents &c2);

    int getCents() const { return m_cents; }
};

// обратите внимание: эта функция не является функцией-членом!
Cents operator+(const Cents &c1, const Cents &c2)
{
    return Cents(c1.m_cents + c2.m_cents);
}

int main()
{
    Cents cents1(6);
    Cents cents2(8);
    Cents centsSum = cents1 + cents2;
    std::cout << "I have " << centsSum.getCents() << " cents.\n";
}
```

Перегрузка операторов

```
class Cents
{
private:
    int m_cents;

public:
    Cents(int cents) { m_cents = cents; }

    friend Cents operator+(const Cents &c1, int value);
    friend Cents operator+(int value, const Cents &c1);
    int getCents() const { return m_cents; }
};

Cents operator+(const Cents &c1, int value)
{
    return { c1.m_cents + value };
}

Cents operator+(int value, const Cents &c1)
{
    return { c1.m_cents + value };
}

int main()
{
    Cents c1 = Cents(4) + 6;
    Cents c2 = 6 + Cents(4);

    std::cout << "I have " << c1.getCents() << " cents.\n";
    std::cout << "I have " << c2.getCents() << " cents.\n";
}
```


Перегрузка операторов

Операторы **присваивания** (=), **индекса** ([]), **вызова** функции (()) и выбора члена (->) должны быть перегружены как функции-члены, потому что так требует язык.

Перегрузка операторов

```
class Cents{
private:
    int m_cents;

public:
    Cents(int cents) { m_cents = cents; }

    // Перегрузка Cents + int
    Cents operator+(int value);

    int getCents() const { return m_cents; }
};

// обратите внимание: эта функция является функцией-членом!
// параметр cents в дружественной версии теперь неявный параметр *this
Cents Cents::operator+(int value)
{
    return Cents(m_cents + value);
}

int main()
{
    Cents cents1(6);
    Cents cents2 = cents1 + 2;
    std::cout << "I have " << cents2.getCents() << " cents.\n";
}
```

Перегрузка операторов

```
class Cents
{
private:
    int m_cents;
public:
    Cents(int cents=0): m_cents(cents){
    }

    // перегруженное приведение в int
    operator int() const { return m_cents; }

    int getCents() const { return m_cents; }
    void setCents(int cents) { m_cents = cents; }
};
```

Перегрузка операторов

```
class Matrix
{
private:
    double m_data[4][4]{};

public:
    double& operator()(int row, int col);
    double operator()(int row, int col) const; // для константных объектов
};

double& Matrix::operator()(int row, int col)
{
    assert(col >= 0 && col < 4);
    assert(row >= 0 && row < 4);

    return m_data[row][col];
}

double Matrix::operator()(int row, int col) const
{
    assert(col >= 0 && col < 4);
    assert(row >= 0 && row < 4);

    return m_data[row][col];
}

int main()
{
    Matrix matrix;
    matrix(1, 2) = 4.5;
    std::cout << matrix(1, 2) << '\n';
}
```

Итератор

Итератор

Итератор — предоставляет последовательный доступ к элементам множества (контейнера), независимо от его внутреннего устройства.

Итератор предоставляет набор методов позволяющих:

- Получить доступ к текущему элементу;
- Переключится на следующий элемент множества;
- Узнать достигнут ли конец множества.

`begin()` — позволяет получить итератор на начало контейнера;

`end()` — позволяет получить итератор на конец контейнера (элемент за последним);


`*` (оператор звёздочка) — позволяет получить доступ к текущему элементу;

`++` (инкремент) — позволяет перейти на следующий элемент;

`+ N` — позволяет перейти на N элементов вперёд (работает не для всех).

Если нужно брать элементы в обратном порядке: `rbegin()`, `rend()`

Итератор

```
vector<int> v {1, 2, 3, 4, 5, 6, 7, 8, 9, };  
auto i = begin(v);  
int x = *i; // x: 1  
++i; // advance by 1  
auto j = begin(v) + 3;  
int y = *j; // y: 4  
auto e = end(v);  
*j = 47; // change element value: 4 → 47
```

DO NOT ACCESS 'END' WITH '*'!
(does not refer to valid memory)
ONLY USE AS POSITION SPECIFIER!

sort



`sort(@begin, @end, compare(o,o) ↦ bool)`



a custom function(object) for comparing elements can be passed as 3rd argument

[cppreference](#)

```
std::vector<int> v {8,9,3,1,2,3,5,4,7,6};  
  
// sort subrange (as shown in image):  
sort(begin(v)+2, begin(v)+8);  
for (int x : v) { cout << x << ' '; } // 8 9 1 2 3 3 4 5 7 6  
  
// sort entire vector:  
sort(begin(v), end(v));  
for (int x : v) { cout << x << ' '; } // 1 2 3 3 4 5 6 7 8 9  
  
// sort vector in descending order:  
sort(begin(v), end(v), std::greater<>{});  
for (int x : v) { cout << x << ' '; } // 9 8 7 6 5 4 3 3 2 1
```



`ranges::sort(range, compare(o,o) ↦ bool) → @end`



a custom function(object) for comparing elements can be passed as 2nd argument

[cppreference](#)

```
std::vector<int> range {3,1,2,3,5,4};  
std::ranges::sort(range);  
for (int x : range) { cout << x << ' '; } // 1 2 3 3 4 5  
  
// sort in descending order:  
std::ranges::sort(range, std::greater<>{});  
for (int x : range) { cout << x << ' '; } // 5 4 3 3 2 1
```

