Программирование

Лекция 6. Перегрузка операторов. Перегрузка операторов внутри классов . Глобальные переменные. Статические переменные и функции. Ключевое слово friend

Перегрузка операторов

- Перегрузка операторов позволяет определять поведение встроенных операторов для объектов пользовательских классов.
- Какие операторы есть в C++?

Основные операторы

Арифметические

Принимают 1 аргумент

- Унарные: префиксные `+ - ++ --`, постфиксные `++ --`
- Бинарные: `+ - * / % += -= *= /= %=`

`a+=10; // a=a+10;`

```
int a = -10;
int b = ++a;
// a = -9; b = -9;
b = a++;
// a = -8; b = -9;
```

Битовые

Операторы для целых чисел, к-рые работают с ними, как с битыми строками.

- Унарные: `~`. ~010101 = 101010
- Бинарные: `& | ^ &= |= ^= >> <<`.

Побитовые сдвиги:
010101 >> 2 = 000101;
010101 << 2 = 010100

Логические

- Унарные: `!`.
- Бинарные: `&& ||`.
- Сравнения: `== != > < >= <=`

Поразрядные логические операции C++

- **И**, обозначение: &
- исключающее **ИЛИ**, обозначение: ^
- исключающее **ИЛИ**, обозначение: |

X	Y	F
0	0	0
0	1	0
1	0	0
1	1	1

&

0, если хотя бы один из битов 0. Если оба бита равны 1, то результат 1.

X	Y	F
0	0	0
0	1	1
1	0	1
1	1	0

^

0, если оба бита будут равны, во всех остальных случаях результат равен

X	Y	F
0	0	0
0	1	1
1	0	1
1	1	1

|

0, если оба бита будут равны 0, во всех остальных случаях результат равен

Другие операторы

1. Оператор присваивания: `=`
2. Специальные:
   - префиксные `* &`, (разыменование указателя и взятие адреса)
   - постфиксные `-> ->*`,
   - особые `, . ::` p.x; A::f, “,” – оператор последовательного выполнения b = (a+=d), (a+d); // b = a+d
3. Скобки: `[] ()` A[i], a(), a(1,2,”Hello”)
4. Оператор приведения `(type)`
5. Тернарный оператор: `x ? y : z`
6. Работа с памятью: `new new[] delete delete[]`

Нельзя перегружать операторы `. ::` и тернарный оператор.

Перегрузка операторов

```
Vector operator-(Vector const& v) {
    return Vector(-v.x, -v.y);
}

Vector operator+(Vector const& v,
                 Vector const& w) {
    return Vector(v.x + w.x, v.y + w.y);
}

Vector operator*(Vector const& v, double d) {
    return Vector(v.x * d, v.y * d);
}

Vector operator*(double d, Vector const& v) {
    return v * d;
}
```

Унарный оператор

Если оператор перегружен и внутри, и снаружи, то произойдёт ошибка!

Умн-е вектора на число

Умн-е числа на вектор

Исп-ем реализацию вектора на число

Перегрузка операторов внутри классов

- Для перегрузки операторов мы использовали внешние функции, но можно перегружать операторы и при помощи определения методов.

```
struct Vector {
    Vector operator-() const { return Vector(-x, -y); }
    Vector operator-(Vector const& p) const {
        return Vector(x - p.x, y - p.y);
    }
    Vector & operator*=(double d) {
        x *= d;
        y *= d;
        return *this;
    }
    double operator[](size_t i) const {
        return (i == 0) ? x : y;
    }
    bool   operator()(double d)   const   { ... }
    void   operator()(double a, double b) { ... }
    double x, y;
};
```

Нет аргументов

1 аргумент

Оператор умн-я числа на вектор внутри класса определить не получится

Для () может быть произвольное число аргументов, для [] – 1 аргумент

Перегрузка инкремента и декремента

```
struct BigNum {
    BigNum & operator++() { //prefix
        //increment
        ...
        return *this;
    }

    BigNum operator++(int) { //postfix
        BigNum tmp(*this);
        ++(*this);
        return tmp;
    }
    ...
};
```

Унарный оператор ++a

«заглушка» (чтобы различать постфикс и префикс)

Сохраняем тек. зн-е

Вызываем префикс

Вернем временное зн-е

```
a = 10;
b = a++;
// b = 10;
a=11
```

Посфикс «сложнее» - вызов префикса и создание копии

Переопределение операторов ввода-вывода

```
#include <iostream>

struct Vector { ... };

std::istream& operator>>(std::istream & is,
                         Vector & p) {
    is >> p.x >> p.y;
    return is;
}

std::ostream& operator<<(std::ostream &os,
                         Vector const& p) {
    os << p.x << ' ' <<  p.y;
    return os;
}
```

Поток ввода

Возвращаем ссылку на поток ввода

Поток вывода

Эти операторы всегда переопределяют как внешние функции – т.к. 1-ый аргумент – поток ввода и вывода

Операторы с особым порядком вычисления («и», «или», «,»)

```
int main() {
    int a = 0;
    int b = 5;
    (a != 0) && (b = b / a);
    (a == 0) || (b = b / a);

    foo() && bar();
    foo() || bar();
    foo(), bar();
}

// no lazy semantics
Tribool operator&&(Tribool const& b1,
                   Tribool const& b2) {
    ...
}
```

False && () = False

True || () = True

Вычисление 2-го операнда не будет происходить

Можно проверить с помощью ф-ций

Сначала выполняется 1-ый операнд, но возвращается второй

При перегрузке операторов данный порядок вычислений не гарантируется – все операнды будут вычислены

Переопределение арифметических и битовых операторов

```
struct String {
    String( char const * cstr ) { ... }

    String & operator+=(String const& s) {
        ...
        return *this;
    }
    //String operator+(String const& s2) const {...}
};

String operator+(String s1, String const& s2) {
    return s1 += s2;
}
```

Конструктор приведения от строки в стиле Си

Можно было определить как метод

Внешняя функция

```
    String s1("world");
    String s2 = "Hello " + s1;
```

Эта строчка не скомпилируется, если определить как метод

“Правильное” переопределение операторов сравнения

```
bool operator==(String const& a, String const& b) {
    return ...
}
bool operator!=(String const& a, String const& b) {
    return !(a == b);
}
bool operator<(String const& a, String const& b) {
    return ...
}
bool operator>(String const& a, String const& b) {
    return b < a;
}
bool operator<=(String const& a, String const& b) {
    return !(b < a);
}
bool operator>=(String const& a, String const& b) {
    return !(a < b);
}
```

Определяем только 2 оператора == и <

О чём стоит помнить

- Стандартная семантика операторов.

Не нужно определять + как *, а / как %.
Чтобы не запутаться

```
void operator+(A const & a, A const& b) {}
```

Оператор + ничего не возвращает?

- Приоритет операторов.

```
Vector a, b, c;
c = a + a ^ b * a; //?????
```

Допустим, ^ - это векторное умн-е.
Но здесь не ясен порядок

- Хотя бы один из параметров должен быть пользовательским.

```
void operator*(double d, int i) {}
```

Так нельзя, т.к. оба операнда – встроенные типы

Глобальные переменные

Объявление глобальной переменной:

```
extern int global;  // лучше в заголовочном файле .hpp

void f () {
    ++global;
}
```

Определение глобальной переменной:

```
int global = 10;  // разумно всегда инициализировать (.cpp)
```

Проблемы глобальных переменных:

- Масштабируемость. Н-р, многопоточные приложения
- Побочные эффекты. Сложно контролировать зн-е глоб. переменной
- Порядок инициализации. Если несколько глоб. переменных

Статические глобальные переменные

Статическая глобальная переменная — это глобальная переменная, доступная только в пределах модуля.

Определение:

```
static int global = 10;

void f () {
    ++global;
}
```

Проблемы статических глобальных переменных:

- Масштабируемость.
- Побочные эффекты.

Статические локальные переменные

Статическая локальная переменная — это глобальная переменная, доступная только в пределах функции.

Время жизни такой переменной — от первого вызова функции next до конца программы.

```
int next(int start = 0) {
    static int k = start;
    return k++;
}
```

next(10) -> 10
next(20) -> 11
Инициализация только 1 раз

Проблемы статических локальных переменных:

- Масштабируемость.
- Побочные эффекты. Пример на слайде

Статические функции

Статическая функция, доступная только в пределах модуля.

Файл 1.cpp:

```
static void test() {
    cout << "A\n";
}
```

Можно иметь ф-ции с одинаковыми названиями
Без слов static - ошибка

Файл 2.cpp:

```
static void test() {
    cout << "B\n";
}
```

Статические глобальные переменные и статические функции проходят *внутреннюю линковку*. (внутри модуля)

Статические поля класса

Статические поля класса — это глобальные переменные, определённые внутри класса.

Объявление:

```
struct User {
    ...
private:
    static size_t instances_;
};
```

Так мы можем посчитать кол-во экземпляров типа User

Инкремент в конструкторе, декремент в деструкторе

Определение:

```
size_t User::instances_ = 0;
```

Определять нужно снаружи класса

Для доступа к статическим полям не нужен объект.

Статические методы

Статические методы — это функции, определённые внутри класса и имеющие доступ к закрытым полям и методам.

Объявление:

```
struct User {
    ...
    static size_t count() { return instances_; }
private:
    static size_t instances_;
};
```

Есть доступ к закрытым полям класса

Для вызова статических методов не нужен объект.

```
    cout << User::count();
```

Ключевое слово inline

Советует компилятору встроить данную функцию.

```
inline double square(double x) { return x * x; }
```

Вместо вызова ф-ции будет замена на умножение

- В месте вызова `inline`-функции должно быть известно её определение.

Не получится разделить на определение в заголовочном файле и реализацию в .cpp

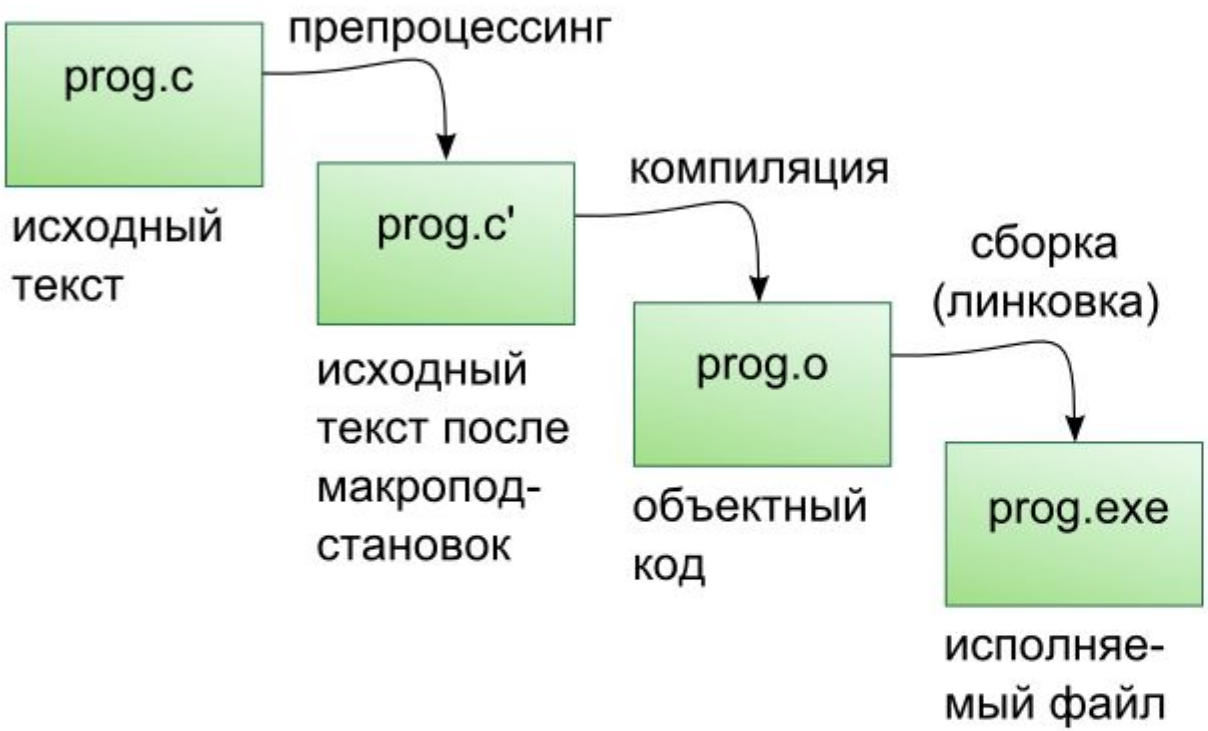
- `inline` функции можно определять в заголовочных файлах.
- Все методы, определённые внутри класса, являются `inline`.
- При линковке из всех версий `inline`-функции (т.е. её код из разных единиц трансляции) выбирается только одна.
- Все определения одной и той же `inline`-функции должны быть идентичными.
- `inline` — это совет компилятору, а не указ.

Если сложная ф-ция

Правило одного определения

Правило одного определения
(One Definition Rule, ODR) (т.е.одного файла)

- В пределах любой единицы трансляции сущности не могут иметь более одного определения. Ошибка на этапе компиляции
- В пределах программы глобальные переменные и не-`inline` функции не могут иметь больше одного определения. Ошибка на этапе линковки
- Классы и `inline` функции могут определяться в более чем одной единице трансляции, но определения обязаны совпадать. Не будет ошибки, но будет некорректная программа


prog.c
препроцессинг
исходный текст
prog.c'
компиляция
исходный текст после макропод-становок
prog.o
сборка (линковка)
объектный код
prog.exe
исполняе-мый файл

Вопрос 1

В заголовочном файле count.hpp определена следующая функция:

```
static int count() {
    static int counter = 0;
    return ++counter;
}
```

Этот файл подключается тремя файлами foo.cpp, bar.cpp и zoo.cpp.

Сколько различных экземпляров переменной counter будут существовать в программе?

Ответ: 3

Вопрос 2

В заголовочном файле foo.hpp есть определение функции:

```
inline void foo(int i) { std::cout << "i = " << i << std::endl; }
```

В программе есть три корректных файла с кодом first.cpp, second.cpp и third.cpp, которые подключают foo.hpp.

Отметьте все верные утверждения из списка.

- [ ] Файлы first.cpp, second.cpp и third.cpp компилируются без проблем, но на этапе компоновки возникает ошибка из-за множественного определения функции foo.
- [ ] При компоновке лишние определения функции foo будут отброшены.
- [ ] Файлы first.cpp, second.cpp и third.cpp компилируются без проблем.
- [ ] Программа компилируется и компонуется без проблем.

Вопрос 3

В заголовочном файле count.hpp определена следующая функция:

```
inline int count() {
    static int counter = 0;
    return ++counter;
}
```

Этот файл подключается тремя файлами foo.cpp, bar.cpp и zoo.cpp.

Сколько различных экземпляров переменной counter будут существовать в программе?

Ответ: 1

Вопрос 4

В заголовочном файле count.hpp определена следующая функция:

```
inline static int count() {
    static int counter = 0;
    return ++counter;
}
```

Этот файл подключается тремя файлами foo.cpp, bar.cpp и zoo.cpp.

Сколько различных экземпляров переменной counter будут существовать в программе?

Ответ: 3

Дружественные классы

```
struct String {
    ...
    friend struct StringBuffer;
private:
    char * data_;
    size_t len_;
};

struct StringBuffer {
    void append(String const& s) {
        append(s.data_);
    }
    void append(char const* s) {...}
    ...
};
```

Данные классы не связаны наследованием

Обращение к private-полям класса String

Дружественные функции

Дружественные функции можно определять прямо внутри описания класса (они становятся inline).

```
struct String {
    ...
    friend std::ostream&
        operator<<(std::ostream & os,
                   String  const& s)
    {
        return os << s.data_;
    }

private:
    char * data_;
    size_t len_;
};
```

Оператор вывода

Функцию можно не только объявить, но и определить

Получаем доступ к private-полям класса String

Дружественные методы

Для одного класса можно определить дружественным метод другого класса

```
struct String;
struct StringBuffer {
    void append(String const& s);
    void append(char const* s) {...}
    ...
};

struct String {
    ...
    friend
      void StringBuffer::append(String const& s);
};

void StringBuffer::append(String const& s) {
    append(s.data_);
}
```

Определение метода будет после