Семестр 1. Лекция 10. Перегрузка операторов. Умные указатели.

Евгений Линский

17 Ноября 2017

Параметры по умолчанию

GaussNumber.h

```
class  GaussNumber {
private:
  int re; int im;
public:
  GaussNumber(int re = 0, int im = 0);
  // GaussNumber(int re, int im = 0); ok
  // GaussNumber(int re = 0, int im); error
}
```

main.cpp

```
#include "GaussNumber.h"
GaussNumber gn1(2, 3);
GaussNumber gn2(2);
GaussNumber gn2();
```

- Почему параметры по умолчанию должны быть указаны в header файле?

Параметры по умолчанию

GaussNumber.h

```
class  GaussNumber {
private:
  int re; int im;
public:
  GaussNumber(int re = 0, int im = 0);
  // GaussNumber(int re, int im = 0); ok
  // GaussNumber(int re = 0, int im); error
}
```

main.cpp

```
#include "GaussNumber.h"
GaussNumber gn1(2, 3);
GaussNumber gn2(2);
GaussNumber gn2();
```

- Почему параметры по умолчанию должны быть указаны в header файле?
- Их необходимо знать при компиляции *main.cpp*. Другие cpp файлы в этот момент не используются.

Перегрузка операторов

1. Некоторые операторы (скорей всего не все):

```
+ - * / % += -= *= /= %=
+a -a
++a a++ --a a--
&& || !
& | ~ ^ &= |= ^= << >> <<= >>=
=
== != < > >= <=
&a *a a-> () [] (type) . ,  (a ? b : c)
```

2. Перегрузка
   int average(int, int); double average(double, double).
3. Можно считать, что компилятор видит
   $c = a + b$ как $c = operator+(a,b)$
4. Можно стандартный оператор *int operator+(int, int)* перегрузить для своего класса BigInt:
   BigInt operator+(const BigInt&, const BigInt&).
5. Операторы "." и "a?b:c" перегружать нельзя

Перегрузка операторов

BigInt — класс для длинной арифметики (например, 2048 двоичных разрядов).

```
// 1.  Outside class
BigInt operator+(const BigInt& a, const BigInt & b){
   ...
}
// 2.  Inside class:  'this' instead 'a'
BigInt BigInt::operator+(const BigInt & b) {...}
```

const/non-const

```
class BigInt {
  char operator [](size_t i) const;
  char& operator [](size_t i);
};
```

- Почему non-const метод возвращает ссылку?

const/non-const

```
class BigInt {
  char operator [](size_t i) const;
  char& operator [](size_t i);
};
```

- Почему non-const метод возвращает ссылку?
- Чтобы можно было делать так *BigInt a(74574); a[3] = 5;*

const/non-const

```
class BigInt {
  char operator [](size_t i) const;
  char& operator [](size_t i);
};
```

- Почему non-const метод возвращает ссылку?
- Чтобы можно было делать так *BigInt a(74574); a[3] = 5;*
- Зачем нужен const метод?

const/non-const

```
class BigInt {
  char operator [](size_t i) const;
  char& operator [](size_t i);
};
```

- Почему non-const метод возвращает ссылку?
- Чтобы можно было делать так *BigInt a(74574); a[3] = 5;*
- Зачем нужен const метод?
- Чтобы можно было передать *BigInt a(453);* в *print(const BigInt&)*

operator=

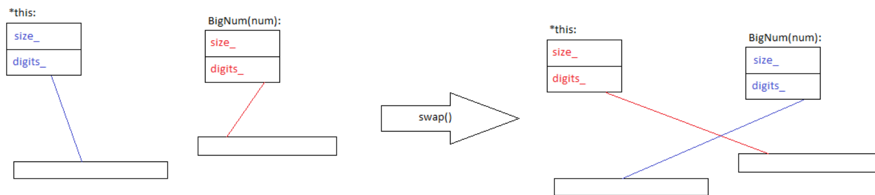
```
class BigInt {
  size_t size_; char* digits_;
  BigInt(const BigInt& num) { ... }

  void swap(BigInt & b) {
    std::swap(size_, b.size_);
    std::swap(digits_, b.digits_);
  }
  BigInt& operator=(const BigInt& num) {
    if(this != &num) {
      BigInt tmp(num);
      tmp.swap(*this);
    }
    return *this;
  }
};
```

operator=

- *std::swap(a,b)* — функция из стандартной библиотеки C++:
  - Меняет местами значения a и b.
  - Можно считать, что перегружена для примитивных типов и указателей (детали: шаблоны, след. семестр).
- Создав временный объект равный num, поменяем его значения с текущими значениями объекта *this. Выйдя из функции временный объект удалится, а в *this останутся новые значения.
- Короткая версия с использованием безымянной переменной.

```
BigInt(num).swap(*this);
```

operator=



- BigInt(num) (как и tmp) — локальная переменная, когда закончится срок ее жизни, то вызовется декструктор.
- Этот деструктор уничтожит то, что до swap было в this.

“Анонимные” переменные - I

```
int max(int a, int b) { ... }

main() {
    int a = 2;
    int b = 3;
    int c = max(a, b);
}
```

или с помощью анонимные переменных

```
main() {
    int c = max(2, 3);
}
```

“Анонимные” переменные - II

```
BigInt max(const BigInt& a, const BigInt& b) { ... }

main() {
    BigInt a("32323223232323232322");
    BigInt b("37382787387382738273");
    BigInt c = max(a, b);
}
```

или с помощью анонимные переменных

```
main() {
    BigInt c = max(BigInt("32323223232323232322"),
        BigInt("37382787387382738273"));
}
```

prefix/postfix

```
BigInt& operator++();    // prefix
BigInt operator++(int);// postfix

BigInt& BigInt::operator++() {
  ...
  return *this;
}

//int is unused
BigInt BigInt::operator++(int){
  BigInt t(*this);
  ++(*this);
  return t;
}
```

Постфикс через префикс.

Приведение типов

- int к BigInt: BigInt a = 3;

```
//using constructor
class BigInt {
  BigInt(int a) { .. }
};
```

- Это не всегда удобно Matrix m = 3. Что хотел автор? Заполнить матрицу тройками? Создать матрицу 3x3?
  Запрет использования конструктора для приведения типов

```
class Matrix {
  explicit Matrix(size_t a) { .. }
};
```

- BigInt к int: BigInt a(32424); int b = a;

```
class BigInt {
  operator int() const {
    return ...;
  }
};
```

Операторы сравнения

Достаточно реализовать *operator<* и *operator==*.

```
bool operator<(BigInt const & a, BigInt const & b)
{ ... }
bool operator==(BigInt const & a, BigInt const & b) { ... }

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

Если не важна производительность, то можно выразить все сравнения через *operator<* (Как?).

Внутри класса или снаружи

```
BigInt a(3); BigInt b(2);
```

- Операторы сравнения лучше определять вне класса.
  - a < b Выполнится в обоих случаях.
  - a < 2 Выполнится в обоих случаях. "2" приведется к BigInt.
  - 3 < b Будет работать только если оператор сравнения определен вне класса.
- Унарные операторы (+=, -=, *= и т.д.) лучше делать внутри класса. А бинарные операторы (+, -, * и т.д.) на их основе, но уже снаружи класса.

```
BigInt operator+(BigInt a, const BigInt& b) {
    a+=b;
    return a;
}
```

Вместо создания временной переменной можно работать с копией параметра.

Умный указатель

Object — некоторый класс.

```
void f() {
  Object *p = new Object(...);
  p->save_to_file("out.txt");
  delete p;
}
```

Хотим, сделать “обертку” (MyArray —“обертка” над динамическим массивом) над указателем.

```
void f() {
  scoped_ptr p(new Object(...));
  p->save_to_file("out.txt");
}
```

Не нужно думать про освобождение памяти! Об этом позаботится деструктор scoped_ptr.

scoped_ptr.h

```
class scoped_ptr {
private:
  Object* myPointer;
public:
  scoped_ptr(Object* ptr);
  ~scoped_ptr();
public:
  Object* ptr();
  Object& operator *() const;
  Object* operator->() const;
  bool isNull() const;

private:
  scoped_ptr(const scoped_ptr& p);
  const scoped_ptr& operator=(const scoped_ptr& p);
}
```

scoped_ptr.cpp

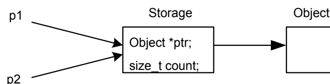
```
scoped_ptr::scoped_ptr(Object* ptr) {
   myPointer = ptr;
}
scoped_ptr::~scoped_ptr() {
  if (myPointer != 0) {
    delete myPointer;
  }
}
Object& scoped_ptr::operator*() const {
  return *myPointer;
}
Object* scoped_ptr::operator->() const {
  return myPointer;
}
bool scoped_ptr::isNull() const {
  return myPointer == 0;
}
```

unique_ptr

- scoped_ptr — нельзя копировать (copy) и передавать в функции
- unique_ptr — можно перемещать (move) и передать в функцию один раз
- shared_ptr — можно все! (почти)

```
unique_ptr::(unique_ptr& p) {
  myPointer = p.myPointer;
  p.myPointer = 0;
}
unique_ptr& unique_ptr::operator=(unique_ptr& p) {
  if (this != &p) {
    if (myPointer != 0) {
      delete myPointer;
    }
    myPointer = p.myPointer;
    p.myPointer = 0;
  }
  return *this;
}
```

shared_ptr



1. Создаём вспомогательный объект Storage, в котором есть счётчик ссылок (и кроме того указатель на Object), устанавливаем его на 1.
   shared_ptr p1 = new Object;
2. Пусть теперь создали другой указатель и присвоили ему значение 1ого, тогда увеличиваем счётчик на 1.
   shared_ptr p2 = p1;
3. Теперь область видимости заканчивается и вызывается деструктор p2. Уменьшаем счётчик на 1, т.к. значение счётчика не 0, то больше ничего не делаем.
4. Вызывается деструктор p1, уменьшаем счётчик на 1, т.к. он стал равен нулю, вызываем деструктор для Object, а затем для Storage.

Когда такой подход не сработает? Подсказка: циклические ссылки.