НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ»
Кафедра информатики и процессов управления (№17)

Лекции по информатике и программированию

Лекция 5

Абстрактные типы данных (часть 3)

Содержание

8. Способы перегрузки операций

8.1. Перегрузка операций простыми функциями

В рассмотренном ранее классе `Complex` символы арифметических операций перегружены функциями-членами (методами) вида `operator+`, а также имеется конструктор преобразования. Тогда возможно прибавить к комплексному числу действительное число типа `double`, например:

```
Complex z, t;
// ...
z = t + 0.5;
```

Здесь к объекту `t` класса `Complex` с помощью конструктора преобразования прибавляется константа `0.5` типа `double`.

Но перестановка операндов вызовет ошибку:

```
z = 0.5 + t;     // ошибка компиляции!
```

поскольку функция-член (метод) `operator+` может быть вызван только для объекта класса `Complex`, а константа `0.5` таковым не является.

Компилятор не преобразует объекты, для которых вызывается метод. В отличие от аргументов функции.

Операцию сложения двух комплексных чисел можно реализовать как стороннюю функцию (вне класса `Complex`), которая по двум заданным комплексным числам возвращает третье комплексное число:

```
Complex operator+(const Complex& a, const Complex& b) {
    return Complex(a.get_re() + b.get_re(), a.get_im() + b.get_im());
}
```

Теперь если убрать метод `operator+` из класса `Complex`, то без ошибок выполняются:

```
Complex z, t;
// ...
z = t + 0.5;
z = 0.5 + t;
```

Можно считать, что выражение для бинарной (двухместной) операцией `a+b` компилятор превращает в одно из следующих выражений:

`a.operator+(b)` или `operator+(a, b)`

Аналогично для унарной (одноместной) операции, (например, `~a`):

`a.operator~()` или `operator~(a)`

Оба варианта равнозначны и не имеют приоритета при компиляции: если в одной программе имеется и метод, и внешняя функция для одной и той же операции (для одинаковых типов операндов), то компилятор выдаст ошибку.

8.2. Дружественные функции и классы

В языке Си++ имеется возможность делать исключения из запретов, налагаемых механизмом защиты. Все детали реализации класса (или структуры с защищенной частью) станут доступны некоторой функции, если объявить эту функцию ***дружественной*** с помощью ключевого слова `friend`. Например:

```
class MyClass {
    friend void func(int, const char *);
    ...
};
```

```
void func(int, const char *) {
    // здесь доступны защищенные поля MyClass
    ...
}
```

Можно объявить дружественным другой класс (или структуру), при этом все его методы станут дружественными:

```
class MyClassA {
    friend class MyClassB;
    ...
};
class MyClassB {
    // во всех методах MyClassB доступны все защищенные поля MyClassA
    ...
}
```

Однако, применять механизм дружественных функций следует **с осторожностью** – необдуманный обход защиты может нанести существенный вред.

Применение дружественных функций можно считать практически безопасным при выносе символов стандартных операций за пределы класса. Пример для рассмотренного ранее класса `Complex`:

```
class Complex {
    ...
    friend Complex operator+(const Complex&, const Complex&);
    ...
};
Complex operator+(const Complex& a, const Complex& b) {
    return Complex(a.re + b.re, a.im + b.im);
}
```

8.3. Перегрузка операций присваивания

В языке Си++ для переопределения операции присваивания (=) и операций с присваиванием (+=, -=, *=, /= и др.) действует ограничение: ***операции присваивания можно перегружать только как методы класса (или структуры)***. Перегружать их в виде обычных функций запрещено.

В языках Си и Си++ операция присваивания возвращает значение, которое было присвоено. По аналогии с этим перегруженная операция присваивания обычно возвращает либо **копию объекта**, либо **константную ссылку на объект** (для которого эта операция вызвана). Последнее предпочтительнее.

Например для класса `Complex`:

```
class Complex {
    ...
    const Complex& operator=(const Complex& c) {
        re = c.re; im = c.im;
        return *this;
    }
    const Complex& operator+=(const Complex& c) {
        re += c.re; im += c.im;
        return *this;
    }
    ...
};
```

Параметр операции присваивания не обязан иметь тот же тип, что и описываемый класс. Например, комплексной переменной можно присвоить действительное число:

```
class Complex {
    ...
    const Complex& operator=(double x) {
        re = x; im = 0.0;
        return *this;
    }
    ...
};
```

8.4. Методы, возникающие неявно

При необходимости компилятор Си++ может сам автоматически (***неявно***) генерировать следующие методы объектов (или структур):

- конструктор умолчания;
- конструктор копирования;
- деструктор;
- метод, реализующий операцию присваивания объекту того же типа.

Если описание класса (или структуры) не содержит никаких конструкторов, то переменную такого типа всё же можно создать. Поскольку в Си++ каждый экземпляр класса (или структуры) является объектом, то конструктор генерируется компилятором автоматически и называется ***неявным конструктором***.

Компилятор Си++ неявно генерирует только два типа конструкторов:
- конструктор умолчания (без параметров),
- конструктор копирования (один параметр типа «ссылка на объект описываемого класса»).

Конструктор копирования неявно генерируется для любого класса (или структуры), для которого конструктор копирования не описан в коде явно. Конструктор копирования присутствует вообще в любом классе (или структуре), явно или неявно.

Неявный конструктор копирования работает наиболее очевидным способом:
- поля, имеющие конструкторы копирования, копируются с их помощью;
- остальные поля копируются побитово.

Конструктор умолчания генерируется неявно, если в классе (или структуры) не описано вообще ни одного конструктора. Если в коде явно описать хотя бы один конструктор (любой), то компилятор неявно не будет генерировать конструктор умолчания.

Неявный конструктор умолчания использует для инициализации полей класса другие конструкторы умолчания (определенные для соответствующих типов).

В Си++ считается, что **деструктор** имеется в любом классе (или структуре). Если деструктор не описан в коде явно, то компилятор автоматически создает ***неявный деструктор***. Такой деструктор может ничего не делать. Но если в классе (или структуре) есть поля, имеющие свои деструкторы, то неявный деструктор содержит их вызовы.

Операция присваивания объекту того же типа может быть **неявно** сгенерирована компилятором Си++. Например, если в классе `MyClass` не описать явно операцию присваивания с параметром, имеющим тот же тип `MyClass` или ссылку на него `MyClass&`, то компилятор неявно создаст метод с этой операцией:

```
class MyClass {
    ...
    const MyClass& MyClass::operator=(const MyClass& other);
    ...
};
```

Такая неявная операция присваивания копирует содержимое каждого поля с помощью другой операции присваивания, определенной для этого поля (если она у этого поля имеется). Если для некоторых полей операция присваивания не найдется (например, поле типа «ссылка» или константное поле), то компилятор не сможет сгенерировать неявную операцию присваивания.

Существует способ **запретить копирование объектов класса**: для этого конструктор копирования достаточно описать явно в приватной части класса.

```
class MyClass {
    ...
private:
    MyClass(const MyClass& ref); // копирование запрещено
    ...
};
```

Такой объект не может быть передан в функцию по значению и не может быть возвращен из функции.

Аналогично можно запретить присваивание объектов класса:

```
class MyClass {
    ...
private:
    void operator=(const MyClass& ref); // присваивание запрещено
    ...
};
```

8.5. Перегрузка операции индексирования

В языке Си операция индексирования (получения значения элемента массива по индексу, `[]`) является операцией адресной арифметики над указателем и целым числом:

`a[i]` эквивалентно `*(a+i)`

В языке Си++ для переопределения операции индексирования (`[]`) действует ограничение: ***операцию индексирования можно перегружать только как метод класса (или структуры)***, она не может быть перегружена отдельной функцией.

Рассмотрим пример: в приватной части класса содержится динамический массив целых чисел с заранее неизвестным размером, который при обращении к его элементам по индексам автоматически увеличивает размер. Пусть исходно создается массив из 16 элементов, а затем его размер будет удваиваться до тех пор, пока нужный индекс не станет допустимым.

```
class IntArray {
    int *p;                 // указатель на динамический массив
    unsigned int size;      // текущий размер динамического массива
public:
    IntArray() {            // конструктор умолчания
        size = 16;
        p = new int[size];
    }
    ~IntArray() {           // деструктор
        delete[] p;
    }
    int& operator[](unsigned int i); // перегрузка операции индексирования
private:
    void Resize(unsigned int new_i); // изменение размера массива
```

```
    void operator=(const IntArray& ref) { }  // присваивание запрещено
    IntArray(const IntArray& ref) { }        // копирование запрещено
};
```

В приведенной версии класса для упрощения запрещено присваивание и копирование объектов класса `IntArray`.

Метод `operator[]` возвращает ссылку (`int&`) на соответствующий элемент массива, чтобы сделать возможным как получение значения этого элемента, так и присваивание ему нового значения:

```
int& IntArray::operator[](unsigned int i) { // операция индексирования
    if (i >= size)
        Resize(i);     // измененяем размер массива
    return p[i];       // возвращаем ссылку(int&) на i-й элемент массива
}
```

Вспомогательный метод `Resize` объявлен приватным, поскольку он является деталью реализации, а вне класса динамический массив представляется безразмерным (бесконечным).

```
void IntArray::Resize(unsigned int new_i) { // изменение размера массива
    unsigned int new_size = size;
    while (new_size <= new_i)    // определяем новый размер массива
        new_size *= 2;
    int *new_array = new int[new_size]; // выделяем память для нового массива
    for (unsigned int i = 0; i < size; ++i)
        new_array[i] = p[i];    // поэлементно копируем массив в новый
    delete[] p;    // освобождаем память старого массива
    p = new_array;      // переопределяем поле указателя на массив
    size = new_size;    // переопределяем поле текущего размера массива
}
```

Теперь в программе можно использовать:

```
IntArray arr;
arr[500] = 123;
arr[1000] = 456;
arr[10] = arr[500] + 78;
arr[20]++;
```

Развивая такой подход, следует заметить, что операция индексирования должна иметь ровно один параметр, но этот параметр может быть любого типа. Например, использование строк в качестве индексов позволяет создавать ***ассоциативный массив (словарь)*** вида:

```
array["key"] = "value";
```

8.6. Перегрузка инкремента и декремента

Операции инкремента (`++`) и декремента (`--`) имеют две формы: **префиксную** (`++i`) и **постфиксную** (`i++`). По сути префиксная и постфиксная форма – это две разные операции, поэтому в языке Си++ принято соглашение:

- ***префиксную*** форму переопределяют методы без параметров:
  `operator++` и `operator--` ;
- ***постфиксную*** форму методы с одним фиктивным параметром типа `int`:
  `operator++(int)` и `operator--(int)` .

Фиктивный параметр никогда не используется, он введен исключительно ради различения функций при перегрузке имен функций.

Операции инкремента и декремента можно перегружать и как методы класса (или структуры), и как отдельные функции вне классов. В последнем случае фиктивный параметр должен стоять вторым параметром функции.

Проиллюстрируем разницу перегрузки префиксной и постфиксной форм на примере:

```
class MyClass {
public:
    void operator++()    { printf(" ++a \n"); }
    void operator--()    { printf(" --a \n"); }
    void operator++(int) { printf(" a++ \n"); }
    void operator--(int) { printf(" a-- \n"); }
};
```

Теперь можно использовать:

```
MyClass a;
++a;    // выводится: ++a
a++;    // выводится: a++
--a;    // выводится: --a
a--;    // выводится: a--
```

В рассмотренном примере перегруженные операции ничего не возвращают (`void`).

Исходно префиксная и постфиксная формы различаются именно возвращаемым значением: операция в префиксной форме возвращает новое значение переменной, а операция в постфиксной форме – старое.

Например, реализуем операции инкремента для класса, инкапсулирующего целочисленную переменную:

```
class MyInt {
    int i;
public:
    MyInt(int x) : i(x)  { }
    const MyInt& operator++() {  // перегрузка префиксной операции
        i++;
        return *this;  // возврат константной ссылки на объект
    }
    MyInt operator++(int) {  // перегрузка постфиксной операции
        MyInt temp(*this);  // создание временной (локальной) копии объекта
        i++;
        return temp;  // возврат локальной копии объекта по значению
    }
    ...
};
```

В постфиксной форме копирование объекта происходит дважды: при создании переменной `temp` и при возврате ее по значению с помощью оператора `return`. Для сложных классов это может привести к неоправданным затратам, поэтому реализацию постфиксной формы часто опускают.

8.7. Перегрузка операции «стрелка» (выборка поля)

В языке Си++ операция «стрелка» (`->`) выполняет **выборку поля или метода из структуры (или класса), на которую ссылается указатель.**

Поскольку в объектах поля приватные, то операция `->` чаще используется для структур:

```
struct MySt {   // описание структуры
    int a, b;
};
...
MySt *p = new MySt;   // p - указатель на структуру
...
p->a = 123;   // обращение к полю a структуры по указателю p
```

Для переопределения операции «стрелка» (`->`) действует ограничение: ***операцию «стрелка» (выборка поля) можно перегружать только как метод класса (или структуры)***, она не может быть перегружена отдельной функцией.

В Си++ принято соглашение: операция `->` переопределяется методом `operator->` без параметров, который обязан возвращать значение одного из следующих типов:

- указатель на некоторую другую структуру (или класс);
- объект (или ссылку на объект), для которого операция `->` переопределена как метод.

При использовании перегруженной операции `->` компилятор последовательно вставляет в код вызовы методов `operator->`, пока один из них не вернет указатель на структуру (или класс). Затем по этому указателю выполнится выборка заданного поля.

Продолжая предыдущий пример, составим класс, объекты которого будут вести себя как указатели на `MySt`. Но при исчезновении такого объекта-указателя объект, на который он указывает, уничтожается.

```
class PtrMySt {  // Описание класса, имитирующего указатель на структуру.
    MySt *p;  // Поле p - указатель на объект (структуру MySt):
public:       // нулевой указатель расценивается как отсутствие объекта.
    PtrMySt(MySt *ptr = 0) : p(ptr) { }  // Конструктор умолчания.
    ~PtrMySt() {            // Деструктор:
        if (p)  delete p;  // удаляем имеющийся объект по указателю p.
    }
    MySt* operator=(MySt *ptr) {  // Операция присваивания обычного адреса:
        if (p)  delete p;  // 1) удаляем имеющийся объект;
        p = ptr;   // 2) сохраняем в поле p адрес присваиваемого объекта;
        return p;  // 3) возвращаем указатель на присвоенный объект.
    }
    MySt& operator*() const {  // Операция разыменования.
        return *p;
    }
    MySt* operator->() const {  // Операция «стрелка» (выборка поля).
        return p;
    }
private:
    PtrMySt(const PtrMySt&) { }        // Запрет побитового копирования.
    void operator=(const PtrMySt&) { } // Запрет побитового присваивания.
};
```

Объекты-указатели на структуру удобно использовать внутри функций: им можно присваивать адреса новых экземпляров структур `MySt`, при этом старый экземпляр каждый раз будет удаляться. А при завершении функции будет удален последний экземпляр структуры `MySt`. Например:

```
int func() {
    PtrMySt p1, p2;  // Создаем p1 и p2 - объекты-указатели на структуру.
    p1 = new MySt; // Создаем структуру (анонимная, доступна только по p1).
    p1->a = 12;
    p1->b = 34;
    p2 = new MySt; // Создаем структуру (анонимная, доступна только по p2).
    p2->a = 56;
    p2->b = p1->a + 78;  // Результат 12+78=90 сохранится в поле b.
    p1 = p2;  // Ошибка компиляции – побитовое присваивание запрещено.
    ...
}  // При завершении функции func вся выделенная память будет освобождена.
```

Побитовые операции копирования и присваивания придется запретить, чтобы один и тот же объект-указатель типа `PtrMySt` не удалялся из памяти дважды (операцией `delete` при вызове деструктора).

8.8. Перегрузка операции вызова функции

Операция вызова функции обозначается в языках Си и Си++ круглыми скобками, внутри которых может находиться список параметров (через запятую).

Для переопределения операции вызова функции действует ограничение: ***операцию вызова функции можно перегружать только как метод класса***, она не может быть перегружена отдельной функцией.

В Си++ операция вызова функции переопределяется методом `operator()`. Объекты классов, для которых перегружена операция вызова функции, называют ***функциональными объектами*** (или ***функторами***, или ***функционалами***).

Пример:

```
class MyFunc {
public:
    void operator()() {  // без параметров
        printf("func0\n");
    }
    void operator()(int a) {  // с одним параметром
        printf("func1: %d\n", a);
    }
    void operator()(int a, int b) {  // с двумя параметрами
        printf("func2: %d %d\n", a);
    }
};
```

Вызов функтора

```
MyFunc f;
f();
f(123);
f(45, 67);
```

В результате на экран выводится:

```
func0
func1: 123
func2: 45 67
```

8.9. Перегрузка операции преобразования типов

В языках Си и Си++ перация преобразования типов выражения может выполняться выполняется **неявно**. Например:

```
int i;
double x;
...
i = x;  // Неявное преобразование типа double к типу int.
```

В языке Си++ для переопределения операции неявного преобразования типов действует ограничение: ***операцию неявного преобразования типов можно перегружать только как метод класса (или структуры)***, она не может быть перегружена отдельной функцией.

Операция преобразования типов переопределяется методом, имя которого состоит из слова `operator` и имени типа, к которому необходимо преобразовать тип выражения. Пример:

```
class MyClass {
    ...
public:
    ...
    operator int() const {  // Перегрузка операции неявного
        ...                 // преобразования типа MyClass к типу int.
    }
};
```

Теперь возможен такой код:

```
MyClass a;
int i;
...
i = a;  // Неявное преобразование типа MyClass к типу int.
```

Однако, если компилятор находит больше одного способа преобразования одного типа к другому, то не применяет ни один из них и выдает ошибку. Поэтому перегрузка операции преобразования типов на практике используется крайне редко.