Лекция 4. Перегрузка операций

Перегрузка операторов — один из способов реализации полиморфизма, заключающийся в возможности одновременного существования в одной области видимости нескольких различных вариантов применения оператора, имеющих одно и то же имя, но различающихся типами параметров, к которым они применяются.

Синтаксис перегрузки операторов очень похож на определение функции с именем operator@, где @ — это идентификатор оператора (например +, -,<<, >>). Рассмотримпростейшийпример:

```
class Integer
{
private:
   int value; public:
Integer(int i): value(i) {}
const Integer operator+(const Integer& rv) const { return (value + rv.value);
}
};
```

Вданном случае, оператор оформлен как член класса, аргумент определяет значение, находящееся в правой части оператора. Вообще, существует два основных способа перегрузки операторов: глобальные функции, дружественные для класса, или подставляемые функции самого класса.

Вбольшинстве случаев, операторы (кроме условных) возвращают объект, или ссылку на тип, к которому относятся его аргументы (если типы разные, то вы сами решаете как интерпретировать результат вычисления оператора).

Рассмотрим примеры перегрузки унарных операторов для определенного выше класса Integer. Заодно определим их в виде дружественных функций и рассмотрим операторы декремента и инкремента:

```
class Integer
{
private:
   int value; public:
Integer(int i): value(i) {}

//унарный +
friend const Integer& operator+(const Integer& i);

//унарный -
friend const Integer operator-(constInteger& i);


//префиксныйинкремент
friend const Integer& operator++(Integer& i);

//постфиксныйинкремент
friend const Integer operator++(Integer& i, int);

//префиксныйдекремент
friend const Integer& operator--(Integer&i);
```

```
//постфиксныйдекремент
friend const Integer operator--(Integer&i, int);
};

//унарный плюс ничего не делает.
const Integer& operator+(const Integer& i) { return i.value;
}

const Integer operator-(constInteger& i) { returnInteger(-i.value);
}

//префикснаяверсиявозвращаетзначениепослеинкрементаconstInteger&operator++(Integer&i) {
i.value++; return i;
}

//постфикснаяверсиявозвращаетзначениедоинкремента const Integer operator++(Integer& i, int) {
Integer oldValue(i.value); i.value++;
return oldValue;
}

//префикснаяверсиявозвращаетзначениепоследекремента const Integer& operator--(Integer&i) {
i.value--;return i;
}

//постфикснаяверсиявозвращаетзначениедодекремента const Integer operator--(Integer&i, int) {
Integer oldValue(i.value); i.value--;
return oldValue;
}
```

Теперь вы знаете, как компилятор различает префиксные и постфиксные версии декремента и инкремента. В случае, когда он видит выражение ++i, то вызывается функция operator++(a). Если же он видит i++, то вызывается operator++(a, int). То есть вызывается перегруженная функция operator++, и именно для этого используется фиктивный параметр

int в постфиксной версии.

Рассмотрим синтаксис перегрузки бинарных операторов. Перегрузим один оператор, который возвращает l-значение,один условный оператор и один оператор, создающий новое значение (определим их глобально):

```
class Integer
{
private:
   int value; public:
Integer(int i): value(i) {}
friend const Integer operator+(const Integer& left, const Integer& right);

friend Integer& operator+=(Integer& left, const Integer& right);

friend bool operator==(const Integer& left, const Integer& right);
};

const Integer operator+(const Integer& left, const Integer& right) { return Integer(left.value + right.value);
}
```

```
Integer& operator+=(Integer& left, const Integer& right) { left.value += right.value;
  return left;
  }

bool operator==(const Integer& left, const Integer& right) { return left.value == right.value;
  }
```

Во всех этих примерах операторы перегружаются для одного типа, однако, это необязательно. Можно, к примеру, перегрузить сложение нашего типа Integer и определенного по его подобию Float.

Как можно было заметить, в примерах используются различные способы передачи аргументов в функции и возвращения значений операторов.

•Если аргумент не изменяется оператором, в случае, например унарного плюса, его нужно передавать как ссылку на константу. Вообще, это справедливо для почти всех арифметических операторов (сложение, вычитание, умножение...)

•Тип возвращаемого значения зависит от сути оператора. Если оператор должен возвращать новое значение, то необходимо создавать новый объект (как в случае бинарного плюса). Если вы хотите запретить изменение объекта как l-value,то нужно возвращать его константным.

•Для операторов присваивания необходимо возвращать ссылку на измененный элемент. Также, если вы хотите использовать оператор присваивания в конструкциях вида (x=y).f(), где функция f() вызывается для для переменной x, после присваивания ей y, то не возвращайте ссылку на константу, возвращайте просто ссылку.

•Логические операторы должны возвращать в худшем случае int, а в лучшем bool. Некоторые операторы в C++ не перегружаются в принципе. По всей видимости, это сделано из соображений безопасности.

•Оператор выбора члена класса ".".

•Оператор разыменования указателя на член класса ".*"

•В C++ отсутствует оператор возведения в степень (как в Fortran) "**".

•Запрещено определять свои операторы (возможны проблемы с определением приоритетов).

•Нельзя изменять приоритеты операторов

Как мы уже выяснили, существует два способа операторов — в виде функции класса и в виде дружественной глобальной функции. Роб Мюррей, в своей книге C++ Strategies and Tactics определил следующие рекомендации по выбору формы оператора:

Оператор	**Рекомендуемая форма**
Все унарные операторы	Член класса
= () [] ->->*	Обязательно член класса
+= -= /=*= ^= &= |= %= >>= <<=	Член класса
Остальные бинарные операторы	Не член класса

Файловый ввод/вывод

Все операции, применимые в стандартному вводу и выводу, могут быть также применены к файлам. Чтобы использовать файл для ввода или вывода, мы должны включить еще один заголовочный файл:

```
#include <fstream>
```

Перед тем как открыть файл для вывода, необходимо объявить объект типа ofstream:

```
ofstream outfile("name-of-file");
```

Проверить, удалось ли нам открыть файл, можно следующим образом: if (! outfile) // false, если файл не открыт

```
cerr << "Ошибка открытия файла.\n"
```

Так же открывается файл и для ввода, только он имеет тип ifstream: ifstream infile("name-of-file");
if (! infile) // false, если файл не открыт cerr << "Ошибка открытия файла.\n"

Ниже приводится текст простой программы, которая читает файл с именем in_file и выводит все прочитанные из этого файла слова, разделяя их пробелом, в другой файл, названный out_file.

```
#include <iostream> #include <fstream> #include <string> int main()
{
ifstream infile("in_file"); ofstream outfile("out_file");

if ( ! infile ) {
cerr<< "Ошибкаоткрытиявходногофайла.\n" return -1;
}

if ( ! outfile ) {
cerr<< "Ошибкаоткрытиявыходногофайла.\n" return -2;
}
string word;
while ( infile >> word ) outfile << word << ' ';
return 0;
}
```

Шаблоны

Шаблоны (англ. template) — средство языка C++, предназначенное для кодирования обобщённых алгоритмов, без привязки к некоторым параметрам (например, типам данных, размерам буферов, значениям по умолчанию).

В C++ возможно создание шаблонов функций и классов.

Шаблоны позволяют создавать параметризованные классы и функции. Параметром может быть любой тип или значение одного из допустимых типов (целое число, enum, указатель на любой объект с глобально доступным именем).

Шаблон функции начинается с ключевого слова template, за которым в угловых скобках следует список параметров. Затем следует объявление функции:

```
template< typename T >
void sort( T array[], int size ); // прототип: шаблон sort объявлен, но не определён

template< typename T >
void sort( T array[], int size ) // объявлениеиопределение
{
T t;
for (int i = 0; i < size - 1; i++) for (int j = size - 1; j > i; j--)if (array[j] <array[j-1])
{
t = array[j];
array[j] = array[j-1];array[j-1]= t;
}

                    }

                    template< int BufferSize >          // целочисленный параметр
                    char* read()
                    {
char *Buffer = new char[ BufferSize ]; /* считываниеданных */
return Buffer;
}
```

Ключевое слово typename появилось сравнительно недавно, поэтому стандарт допускает использование class вместо typename:

```
template< class T >
```

Вместо T допустим любой другой идентификатор.

Простейшим примером служит определение минимума из двух величин. Если a меньше b то вернуть a, иначе - вернуть b

В отсутствие шаблонов программисту приходится писать отдельные функции для каждого используемого типа данных. Хотя многие языки программирования определяют встроенную функцию минимума для элементарных типов (таких как целые и вещественные числа), такая функция может понадобится и для сложных (например «время» или «строка») и очень сложных («игрок» в онлайн-игре)объектов.

Так выглядит шаблон функции определения минимума:

```
template< typename T >
T min( T a, T b )
{
return a < b ? a : b;
}
```

Для вызова этой функции можно просто использовать её имя:

```
min( 1, 2 ); min( 'a', 'b' );
min( string( "abc" ), string( "cde" ) );
```

Вообще говоря, для вызова шаблонной функции, необходимо указать значения для всех параметров шаблона. Для этого после имени шаблона указывается список значений в угловых скобках:

```
int i[5] = { 5, 4, 3, 2, 1 }; sort< int >( i, 5 );

char c[] = "бвгда";
sort< char >( c, strlen( c ) );

sort< int >( c, 5 ); // ошибка: у sort< int >параметр int[] ане char[]

char *ReadString = read< 20 >(); delete [] ReadString; ReadString = read< 30 >();
```

Для каждого набора параметров компилятор генерирует новый экземпляр функции. Процесс создания нового экземпляра называется инстанцированием шаблона.

В примере выше компилятор создал две специализации шаблона функции sort (для типов char и int) и две — шаблона read (для значений BufferSize 20 и 30). Последнее скорее всего расточительно, так как для каждого возможного значения параметра компилятор будет создавать новые и новые экземпляры функций, которые будут отличаться лишь одной константой.