Перегрузка операций, наследование

Артамонов Ю.Н.

Филиал «Котельники» университета «Дубна»

17 декабря 2018 г.

Содержание

Основные принципы перегрузки операций

C++ дает программисту возможность перегружать большинство операций, так чтобы они были чувствительны к контексту, в котором они используются. Например, операции $>>$, $<<$ в обычном режиме означают побитовый сдвиг вправо, влево соответственно. Однако в C++ они перегружены для работы с потоками ввода, вывода. Несмотря на то, что C++ не позволяет создавать новые операции, он позволяет перегружать существующие операции и, когда они применяются к объектам класса, операции приобретают смысл, соответствующий новым типам. Это очень сильная сторона C++ и важная причина его популярности.

Основные принципы перегрузки операций

Операции перегружаются посредством написания обычного определения функции (с заголовком и телом) за исключением того, что именем функции становится ключевое слово **operator** с последующим символом перегружаемой операции. Например, имя функции **operator**+ можно использовать для перегрузки операции сложения. При этом действует ряд ограничений:

- перегрузить можно только имеющиеся операции, нельзя ввести новую операцию;
- для перегруженной операции можно использовать только тот же самый графический символ;
- перегрузить операцию можно только на новый тип, нельзя, скажем перегрузить сложение целых чисел.
- нельзя изменить правило применения или приоритет операции;
- нельзя перезагружать условную операцию, операцию разрешения видимости и операцию взятия члена структуры - точка.

Пример перегрузки операции сложения

Рассмотрим пример перегрузки операции сложения целых чисел для объектов класса

```
#include <iostream>
using namespace std;
class number{
public:
  int x;
  int operator+ (number);
};
int number::operator+ (number b){  return this->x-b.x;}
int main(){
  number a,b;
  a.x=9;
  b.x=2;
  cout<<a+b<<endl;
  return 0;
}
```

Видно, что здесь вместо сложения будет выполняться вычитание.

Запреты, разрешения на перезагрузку операций

Таблица 1: Операции, которые нельзя перегружать

.	.*	::	?:	sizeof

При перезагрузке ( ), [ ], ->, = перегружающая функция должна быть методом класса. Для других операций перегружающие функции могут быть друзьями.
Перезагрузка операции присваивания и операции суммирования с целью разрешить такие операции, как
object2 = object1+object2;
не означает, что автоматически будет перегружена операция +=. Однако такого поведения можно добиться, если явно перегрузить операцию +=.

Дружественные функции или методы класса

Функции-операции могут как быть методами класса, так и не быть ими. Во втором случае они обычно являются друзьями класса. Методы класса используют неявный указатель **this**, чтобы получить один из аргументов-объектов своего класса. В противном случае такой аргумент должен быть получен явным образом.
Когда функция-операция объявляется в качестве метода класса, левый операнд должен быть объектом, или ссылкой на объект, принадлежащий классу. Если необходимо, чтобы левый операнд был объектом другого класса, эта функция операция должна объявляться как функция, не являющаяся элементом класса. В этом случае такой функции-операции нужно быть другом класса, если она должна получать прямой доступ к закрытым или защищенным элементам класса.

Дружественные функции или методы класса

Перегруженная операция $<<$ должна иметь в качестве левого операнда тип **ostream &** (такой, как **cout** в выражении **cout** $<<$ **classObject**), так что она должна быть функцией, не являющейся методом класса. Аналогично операция $>>$ должна иметь в качестве левого операнда тип **istream &**(такой, как **cin** в выражении **cin** $>>$ **classObject**) и не являться методом класса. Кроме того, каждая из таких перегружающих функций должна иметь доступ к закрытым элемента класса, который должен выводиться или вводиться. Поэтому такие перегружающие функции удобно сделать дружественными функциями класса.

Пример перегрузки операций передачи в поток и извлечения из потока

В C++ имеется возможность ввода и вывода стандартных типов данных с использованием операций извлечения из потока $>>$ и передачи в поток $<<$. Эти операции являются перегруженными и могут обрабатывать любой стандартный тип данных, включая строки и адреса памяти. Кроме этого, операции передачи и извлечения данных из потока могут быть перегружены, чтобы выполнять ввод и вывод типов, определяемых пользователем. Рассмотрим пример такого рода. Определим класс телефонного номера, который будем вводить и выводить с использованием стандартных потоков. Например, два объекта могут вводиться следующим образом:

```
cin >> phone1 >> phone2;
```

Пример перегрузки операций передачи в поток и извлечения из потока (продолжение)

```
#include <iostream>
using namespace std;
class phonenumber
{
  friend ostream &operator<<(ostream &, phonenumber &);
  friend istream &operator>>(istream &, phonenumber &);

private:
  char first[2]; //цифра 8 и null
  char prefix[4];
  char two[4];
  char three[3];
  char four[3];
};
```

Пример перегрузки операций передачи в поток и извлечения из потока (продолжение)

```
ostream &operator<<(ostream &output, phonenumber &num)
{
  output << num.first << "-"<<"(" << num.prefix <<")"<<"-"<<
    num.two<<"-"<<num.three<<"-"<<num.four;
  return output;
}
istream & operator>>(istream &input, phonenumber &num)
{
  char s[18];
  input.getline(s, 18);  num.first[0]=s[0];  num.first[1]='
    \0';
  num.prefix[0]=s[3];  num.prefix[1]=s[4];  num.prefix[2]=s
    [5];  num.prefix[3]='\0';
  num.two[0]=s[8];  num.two[1]=s[9];  num.two[2]=s[10];  num
    .two[3]='\0';
  num.three[0]=s[12];  num.three[1]=s[13];  num.three[2]='\0
    ';
  num.four[0]=s[15];  num.four[1]=s[16];  num.four[2]='\0';
  return input;
}
```

Пример перегрузки операций передачи в поток и извлечения из потока (продолжение)

```
int main()
{
  phonenumber phone;
  cout<<"введите телефонный номер в формате 8-(926)-345-12-23"<<
    endl;
  cin >> phone;
  cout<<"Итак, Вы ввели телефонный номер:"<<phone<<endl;
  return 0;
}
```

Пример работы с массивом с перегруженными операциями ++, +=, =, ==, < <

Рассмотрим здесь еще один пример работы с классом массива целых чисел. В конструкторе будем задавать массив с определенной размерностью.
Перегрузим для класса массива операции:

- Операции инкремента (постинкремент, прединкремент);
- Операцию += увеличивает все элементы массива на одно и тоже число;
- Операцию = присваивает один массив другому массиву;
- Операцию == сравнивает один массив с другим;
- Операцию < < выводит элементы массива в стандартный поток вывода.

Работа с классом массива

```
#include<iostream>
#include<time.h>
#include<stdlib.h>
using namespace std;
class array{
  friend ostream & operator<< (ostream &, array &);
  friend istream & operator>> (istream &, array &);
  friend array & operator+(array &, array &);
public:
  int & operator [] (int);
  array & operator++(); // Это прединкремент
  array & operator++(int); //Постинкремент требует фиктивного
    переменного
  array & operator +=(int);
  array & operator=(array  &);
  bool operator ==(array &);
  void filling_random(void);
  const  int get_size();
```

Работа с классом массива

```
array(int);
  array(array &); //Конструктор копирования
  ~array();
private:
  int size;
  int *mass;
};

array current(1);

array::array(int n)
{
  size = n;
  mass = new int [size];
  for (int i = 0; i<size; i++) mass[i]=0;
}
```

Работа с классом массива

```
array & operator+(array &ar1, array &ar2)
{
  current=((ar1.get_size()<ar2.get_size())? ar2:ar1);
  for (int i=0; i<current.get_size(); i++)
    {
      if ((i<ar1.get_size()) && (i<ar2.get_size()))
  current[i] = ar1[i]+ar2[2];
      if ((i<ar1.get_size()) && (i>=ar2.get_size()))
  current[i] = ar1[i];
      if ((i>=ar1.get_size()) && (i<ar2.get_size()))
  current[i] = ar2[2];
    }
  return current;
}
```

Работа с классом массива

```
array::array(array & copy){
  size = copy.get_size();
  mass = new int [size];
  for (int i = 0; i<size; i++) mass[i]=copy[i];
}
bool array::operator ==(array & ar2){
  if (size == ar2.get_size()){
      bool pr = true;
      for (int i=0; i<ar2.get_size();i++)
  if ((mass[i]) != (ar2[i])){
      pr = false;
      break;
    }
      return pr;
    }
  else
    return false;
}
```

Работа с классом массива

```
array & array::operator= (array  & ar)
{
  size = ar.get_size();
  delete [] mass;
  mass = new int [size];
  for (int i = 0; i<this->size; i++) this->mass[i]=ar[i];
  return *this;
}

array::~array(void)
{
  delete [] this->mass;
}

const int array::get_size()
{
  return this->size;
}
```

Работа с классом массива

```
int & array::operator[](int n){
  if (n < size)
    return mass[n];
  else{
      if (n<0)
  return mass[0];
      else
  return mass[size-1];
    }
}

ostream & operator<<(ostream & output, array & arr){
  for (int i = 0; i<arr.size; i++)
    output << (arr[i])<<"; ";
  output<<endl;
}
```

Работа с классом массива

```
istream & operator>>(istream & input, array &arr){
  cout<<"array size: ";
  cin>>arr.size;
  delete [] arr.mass;
  arr.mass = new int [arr.size];
  for (int i=0; i<arr.size;i++)
    {cout<<"element "<<i<<": ";
    cin>>arr[i];
    }
}

array& array::operator++()
{
  for (int i = 0; i<size;i++)
    mass[i]++;
  return *this;
}
```

Работа с классом массива

```
array & array::operator++(int)
{
  current = *this;
  for (int i = 0; i<size;i++)
    mass[i]++;
  return current;
}
array & array::operator+=(int value)
{
  for (int i = 0;i<size; i++)
    mass[i]+=value;
  return *this;
}
void array::filling_random()
{
  for (int i = 0; i<size; i++) mass[i] = rand()&100+1;
}
```

Общее представление о наследовании

Наследование представляет собой механизм повторного использования программного обеспечения, в соответствии с которым новые классы создаются на основе существующих. Эти классы наследуют свойства и поведение *базовых классов* и приобретают дополнительно новые качества. Это существенно экономит время, способствует повторному использованию отлаженного программного обеспечения. Новый класс, который наследует свойства и методы базового класса, называется *производным классом*. Каждый производный класс может, в свою очередь, быть базовым классом для каких-либо других новых классов. При *простом наследовании* производный класс наследует свойства и методы от одного базового класса. При *сложном наследовании* производный класс наследует свойства от многих (возможно, логически не связанных) классов. Новый класс имеет больше специфических свойств в сравнении с исходным базовым классом и представляет меньшую группу объектов. Каждый объект производного класса является также объектом базового класса (обратное неверно). Настоящая сила наследования заключается в возможности добавлять, замещать и уточнять наследуемые от базовых классов свойства и методы.

Понятие о защищенных элементах

Производный класс не может иметь доступа к закрытым элементам своего базового класса, такой доступ нарушал бы инкапсуляцию последнего. Однако производный класс может иметь доступ к открытым и так называемым *защищенным элементам базового класса*. Такие защищенные элементы базового класса помещаются в специальный раздел, отмеченный словом **protected**. Защищенный доступ представляет собой промежуточный уровень защиты между закрытым и открытым доступом. Защищенные элементы базового класса могут быть доступны только для элементов и друзей самого класса и для элементов и друзей производных классов. Элементы производного класса могут вызывать открытые и защищенные элементы базового класса просто по их имени (без использования специальных гетеров и сеторов). При этом необязательно использовать операцию разрешения области действия (::) – по умолчанию имеется в виду текущий объект.

Примеры базовых и проиводных классов

Система базовых и производных классов легко строится для объектов рельного мира, вступающих в иерархические отношения. Рассмотрим ряд примеров:

- Военнослужащий -> офицеры, сержанты, старшины, солдаты, матросы. Офицеры в свою очередь делятся на младших офицеров, старших офицеров, высших офицеров. Младшие офицеры: лейтенант, старший лейтенант, капитан. Старшие офицеры -> майор, подполковник, полковник. Высшие офицеры -> генерал, маршал, адмирал. Генералы, в свою очередь, делятся на -> генерал-майор, генерал-лейтенант, генерал-полковник, генерал армии.
- Фигура -> двухмерная, трехмерная. Двухмерная -> круг, треугольник, прямоугольник ... Трехмерная -> шар, пирамида, призма, цилиндр, куб, икосаэдр, додекаэдр, октаэдр ...
- Человек из вуза -> студент, аспирант, докторант, сотрудник. Студент -> магистр, бакалавр. Аспирант, докторант -> целевой, штатный. Сотрудник -> профессорско-преподавательский состав, сотрудники факультетов. Профессорско-преподавательский состав - ассистенты, преподаватели, старшие преподаватели, доценты, профессора. Сотрудники факультетов -> администраторы, технический персонал.

Синтаксис описания наследования по типам

Из наших примеров, для описания того, что круг является производным классом от фигуры, можно использовать следующий синтаксис:

```
class Circle : public Figure
{
  ...
};
```

Это пример *открытого (public) наследования*. В C++ существует три типа наследования: **public, protected, private**. Другие типы наследования будут иметь следующий синтаксис:

```
class C : protected A {...};
class Z : private A {...};
```

Классификация наследования по типам

Если класс объявлен как базовый для другого класса со спецификатором доступа:

- **public**: «public»-члены базового класса — доступны как «public»-члены производного класса; «protected»-члены базового класса — доступны как «protected»-члены производного класса;
- **protected**: «public»- и «protected»- члены базового класса — доступны как «protected»-члены производного класса;
- **private**: «public»- и «protected»- члены базового класса — доступны как «private»-члены производного класса.

Одним из основных преимуществ «public»-наследования является то, что указатель на классы-наследники может быть неявно преобразован в указатель на базовый класс, то есть при объявлении **class B : public A ...** можно написать: **A* a = new B();**.

Пример создания классов с наследованием

```
#include<iostream>

using namespace std;

class Point{
  friend ostream & operator<<(ostream &, const Point &);
public:
  Point(float = 0, float = 0);
  void setPoint(float,float);
  float getX() const{return x;}
  float getY() const{return y;}
protected:
  float x,y;
};
```

Пример создания классов с наследованием

```
class Circle : public Point{
  friend ostream & operator<<(ostream &, const Circle &);
public:
  //Конструктор по умолчанию
  Circle(float radius=0.0, float x=0, float y=0);
  void setRadius(float);
  float getRadius();
  float getArea();
protected:
  float radius;
};
```

Пример создания классов с наследованием

```
//Методы Point
Point :: Point(float a, float b){x=a; y=b;}
void Point:: setPoint(float a, float b){x=a; y=b;}
ostream & operator<<(ostream &output, const Point & p)
{
  output << "[" <<p.x<<","<<p.y<<"]";
  return output;
}
```

Пример создания классов с наследованием

```
//Методы Circle
Circle::Circle(float r, float a, float b)
{
  x=a;
  y=b;
  radius = r;}

void Circle::setRadius(float r) {radius = r;}
float Circle::getRadius(){return radius;}
float Circle::getArea(){return 3.14159*radius*radius;}
ostream & operator<< (ostream & output, const Circle &c)
{
  output << "Center = [" << c.x << ", " << c.y << "]; Radius
     = " << c.radius;
  return output;
}
```

Пример создания классов с наследованием

```
int main()
{
  Point *pPoint, p(3.6, 4.5);
  Circle *pCircle, c(3, 1.1, 2.3);
  cout<< "Point :" << p << endl;
  cout<< "Circle :" << c << endl;
  return 0;
}
```

Сложное наследование

Мы рассматривали механизм простого наследования, в котором каждый класс выводится только из одного базового класса. Однако в C++ разрешается конструировать производный класс путем наследования из нескольких базовых классов. Такой механинзм наследования называется *сложным наследованием*. Это мощный механизм дает интересные формы повторного использования программного кода, однако может порождать ряд проблем, связанных с неоднозначностью. Сложное наследование следует применять, когда между новым типом и двумя или более типами существует отношение является: тип A является как типом B, так и типом C. Рассмотрим соответствующий пример.

Пример сложного наследования

Создаем базовые классы

```
#include<iostream>

using namespace std;

class Base1{
public:
  Base1 (int x){value = x;}
  int getData() const{return value;}
protected:
  int value;
};

class Base2{
public:
  Base2(char c){letter = c;}
  char getData() const {return letter;}
protected:
  char letter;
};
```

Пример сложного наследования: создаем класс со сложным наследованием

```
class MyClass : public Base1, public Base2{
  friend ostream &operator<<(ostream &, const MyClass &);
public:
  MyClass(int, char, float);
  float getReal() const;
private:
  float real;
};
MyClass::MyClass(int i, char c, float f)
  : Base1(i), Base2(c)
{real = f;}
float MyClass::getReal() const{return real;}

ostream &operator<<(ostream &output, const MyClass &m){
  output << "Целое: " << m.value
         << "\n Символ: " << m.letter
         << "\n Вещественное: " << m.real;
  return output;
}
```

Пример сложного наследования: используем классы

```
int main()
{
  Base1 b1(10), *pBase1;
  Base2 b2('z'), *pBase2;
  MyClass M(7, 'p',3.5);
  cout << "B1.value = "<< b1.getData() << endl
       << "B2.letter = "<<b2.getData() << endl;
  cout << M << endl;
  cout << "Целое: " << M.Base1::getData() << endl;
  cout << "Символ: " << M.Base2::getData() << endl;
  pBase1 = &M;
  pBase2 = &M;
  cout << pBase1-> getData() << endl;
  cout << pBase2->getData() << endl;
  return 0;
}
```

Пример сложного наследования: пояснение

Класс **Base1** содержит один защищенный элемент **value**, а также один конструктор, который устанавливает **value**, и открытую функцию **getData**, которая считывает значение **value**.
Класс **Base2** аналогичен классу **Base1**, единственное отличие - защищенная переменная **letter** символьного типа.
Класс **MyClass** является производным классом от обоих классов **Base1**, **Base2**. Следует заметить, что конструктор производного класса вызывает каждый из своих базовых конструкторов посредством синтаксиса инициализатора элементов.
Перегруженная операция передачи в поток является дружественной для класса **MyClass**, поэтому она имеет доступ к закрытому полю **real** этого производного класса и к защищенным элементам **value**, **letter** базовых классов.

Пример сложного наследования: пояснение

Следует обратить внимание на неоднозначность: в базовых классах **Base1, Base2** имеется функция с одинаковым именем: **getData()**. Поэтому класс **MyClass** наследует эти две функции с одинаковым именем. Поэтому могут возникать трудности при вызове конкретной функции **getData()** базового класса из производного. Эта неоднозначность легко снимается с помощью операции разрешения области действия.