Лекция 10. Перегрузка операторов, преобразование типов и приведения новых стилей

Казахский национальный университет имени аль-Фараби

Кафедра компьютерных наук

Объектно-ориентированное программирование

Лектор: старший преподаватель, PhD, Кенжебек Ержан Ғалымжанұлы

Цель лекции

Цель лекции

Изучить перегрузку операторов в C++, дружественные функции, а также преобразования и приведения типов нового стиля.

Ожидаемые результаты

1. Перегружать операторы как методы класса и как дружественные функции
2. Знать правила и ограничения перегрузки
3. Выбирать между static_cast, dynamic_cast, reinterpret_cast и const_cast
4. Использовать dynamic_cast с указателями

План лекции

- Перегрузка операторов — чрезвычайно интересная особенность C++. Это не только интересно и увлекательно, но и незаменимый инструмент для дизайнера классов. В этой лекции объясняется следующее:
- - концепция перегрузки операторов,
- - поддержка C++ для перегрузки операторов,
- - необходимость перегружать операторов,
- - правила перегрузки операторов,
- - использование и неправильное использование перегрузки операторов, и
- - подводные камни при перегрузке оператора

Перегрузка оператора

- Давайте сначала разберемся в значении перегрузки операторов и в том, как реализована эта полезная функция языка C++. Перегрузка оператора означает программирование оператора для работы с операндами тех типов, для которых он еще не предназначен. Например, оператор сложения может работать с операндами типа char, int, float и double. Однако если «s1», «s2» и «s3» являются объектами класса String, который мы определили ранее, то следующие операторы:
- $s3 = s1 + s2$
- не будет компилироваться, если только создатель класса String явно не перегрузит оператор сложения для работы на объектах своего класса. Ниже описан способ реализации такой перегрузки.

Перегрузка оператора (синтаксис)

- Операторы перегружаются написанием функций перегрузки операторов. Эти функции являются либо функциями-членами, либо дружественными функциями того класса, чьи объекты предназначены в качестве операндов перегруженного оператора. Функции перегрузки операторов очень похожи на функции-члены и дружественные функции, о которых мы все время читали. Единственное, что в них примечательно, так это их название. Имена функций перегрузки операторов состоят из ***operator*** ключевого слова, за которым следует **символ** перегружаемого оператора. Синтаксис для функций-членов, которые перегружают данный оператор, выглядит следующим образом:

```
class <class_name>
{
   <return_type> operator <op> (<arg_list>); //prototype
};
<return_type> <class_name> :: operator <op> (<arg_list>)
//definition
{
   //function body
}
```

Правила перегрузки операторов

Что нельзя

- Создавать новые операторы
- Перегружать ::, ., .*, ?: и sizeof
- Менять число операндов, приоритет и ассоциативность оператора

Рекомендации

- Хотя бы один операнд — пользовательский тип
- Операторы =, [], () и -> перегружаются только как методы класса
- Сохраняйте привычный смысл оператора: + не должен вычитать
- Не перегружайте оператор без необходимости

Перегрузка оператора + для класса String

```
/*Beginning of String.h*/
class String
{
public:
 String operator + (const String &) const; //prototype
 /*
 rest of the class String
 */
};
/*End of String.h*/

/*Beginning of String.cpp*/
#include"String.h"
String String :: operator + (const String & ss) const //definition
{
 //function body
}
/*
 definitions of the rest of the functions of class String
*/
/*End of String.cpp*/

/*Beginning of SomeProgram.cpp*/
#include"String.h"
void f() //some function
{
 String s1,s2,s3;
 /*
 rest of the function f()
 */
 s3 = s1 + s2;
 /*
 rest of the function f()
 */
}
/*End of SomeProgram.cpp*/
```

- Мы можем заметить, что функция была объявлена как открытый член класса. Это связано с тем, что оператор обычно используется в перегруженной форме внутри функций, не являющихся членами.

Перегрузка оператора + для класса String с помощью friend function

```
/*Beginning of String.h*/
class String
{
public:
 friend String operator + (const String &, const String &) const; //prototype
 /*
 rest of the class String
 */
};
/*End of String.h*/

/*Beginning of String.cpp*/
#include"String.h"
String operator + (const String & ss1, const String &ss2) const //definition
{
 //function body
}
/*
 definitions of the rest of the functions of class String
*/
/*End of String.cpp*/

/*Beginning of SomeProgram.cpp*/
#include"String.h"
void f() //some function
{
 String s1,s2,s3;
 /*
 rest of the function f()
 */
 s3 = s1 + s2;
 /*
 rest of the function f()
 */
}
/*End of SomeProgram.cpp*/
```

Интерпретация

- s3 = s1 + s2;
- s3 = s1.**operator** + (s2); //member function
- s3 = **operator** + (s1, s2); // в случае friend function

Концепт перегрузки используя friend function

- Мы могли бы задаться вопросом, почему дружественные функции используются для перегрузки операторов. В конце концов, функции-члены, кажется, служат этой цели. Чтобы понять это, давайте рассмотрим два класса A (которые мы определили) и B (существующий класс или встроенный тип данных). Мы понимаем, что по какой-то причине к объекту класса B будет добавлен только объект класса A, чтобы получить другой объект класса A. Это будет сделано следующим образом:
- `a2 = b1 + a1;  //a1, a2 – объекты класса A,  b1 –  объект класса B`

Сложение объектов двух классов

```
class A
{
public:
 A operator + (const B &);
};

a2 = b1 + a1;
//interpret
a2 = b1.operator + (a1);
a2 = operator + (b1,a1);
```

- Прототип функции-члена не удовлетворяет ни одной из этих двух интерпретаций. Компилятор, естественно, выдаст ошибку. **Объявление функции перегрузки операторов как дружественной функции с объектом класса B в качестве первого формального аргумента решает проблему.**

Перегрузка оператора с использованием friend function

```
class A
{
public:
  friend A operator + (const B&, const A&);
    //prototype
};

A operator + (const B& bb, const A& aa) //definition
 {
  //function body
 }
```

Преобразование и приведение новых стилей

- C++ предоставляет новый набор операторов для приведения типов. Эти операторы можно использовать вместо весьма подверженного ошибкам метода приведения типов, предоставляемого языком C.
- Приведения новых стилей безопасны в использовании и могут быть легко обнаружены в исходных кодах с помощью средства поиска редактора, в котором был открыт исходный код. Последнее преимущество особенно полезно в больших исходных кодах.В идеале программа вообще не должна нуждаться в приведениях. Однако существуют различные шаблоны программирования, где они необходимы. Чтобы удовлетворить эту потребность, вместо старого традиционного стиля следует использовать слепки нового стиля.Есть четыре оператора приведения нового стиля.
  - dynamic_cast
  - static_cast
  - reinterpret_cast
  - const_cas

Syntax

- Каждый из этих операторов преобразует объект, переданный ему в качестве операнда, предварительно определенным образом и возвращает преобразованный объект. Общий синтаксис этих операторов:

  operator <тип>(значение, тип которого необходимо преобразовать)

оператор dynamic_cast

- Информация о типе времени выполнения (RTTI) позволяет нам найти тип значения и сравнить типы двух значений. C++ предоставляет оператор dynamic_cast и оператор typeid длявнедрение RTTI.Оператор dynamic_cast используется для определения того, указывает ли конкретный указатель базового класса на объект базового класса или объект одного из производных классов во время выполнения. Он также используется для определения того, ссылается ли ссылка на базовый класс на объект базового класса или на объект одного из производных классов во время выполнения. Из главы 5 мы знаем, что указатель базового класса может указывать на объект производного класса, в то время как указатель производного класса не может указывать на объект базового класса.

```
A A1, * APtr;
B B1, * BPtr;
APtr=&B1;              //line 1: OK: Can convert from B* to A*
BPtr=&A1;              //line 2: ERROR: Cannot convert from A* to B*
BPtr=APtr;             //line 3: ERROR: Cannot convert from A* to B*
```

dynamic cast с указателями

```cpp
/*Beginning of dynamicCast01.cpp*/
#include<iostream.h>
class A
{
public:
virtual void f1()
 {
 cout<<"A::f1() called\n";
 }
};
class B : public A
{
public:
void f2()
 {
 cout<<"B::f2() called\n";
 }
};
class C : public A
{
public:
void f3()
 {
 cout<<"C::f3() called\n";
 }
};
void main()
{
 A * APtr;
 B B1, * BPtr;
 C C1;
 APtr=&B1; //APtr points at an object of class B.
 BPtr=dynamic_cast<B*>(APtr); //APtr is actually of
 //type B* and type is
 //also B*. Hence, cast
 //returns address of B1.
if(BPtr!=NULL) //BPtr is not NULL. It contains the
 //address of B1.
 BPtr->f2();
else
 cout<<"Invalid cast\n";
 APtr=&C1; //APtr points at an object of class C.
 BPtr=dynamic_cast<B*>(APtr); //APtr is actually of
 //type C* and type is
 //B*. Hence, cast
 //returns NULL.
if(BPtr!=NULL) //BPtr is NULL.
 BPtr->f2();
else
 cout<<"Invalid cast\n";
}
/*End of dynamicCast01.cpp*/
```


```
/*Beginning of dynamicCast01.cpp*/
#include<iostream.h>
class A
{
public:
virtual void f1()
 {
 cout<<"A::f1() called\n";
 }
};
class B : public A
{
public:
void f2()
 {
 cout<<"B::f2() called\n";
 }
};
class C : public A
{
public:
void f3()
 {
 cout<<"C::f3() called\n";
 }
};
void main()
{
 A * APtr;
 B B1, * BPtr;
 C C1;
 APtr=&B1; //APtr points at an object of class B.
 BPtr=dynamic_cast<B*>(APtr); //APtr is actually of
 //type B* and type is
 //also B*. Hence, cast
 //returns address of B1.
if(BPtr!=NULL) //BPtr is not NULL. It contains the
 //address of B1.
 BPtr->f2();
else
 cout<<"Invalid cast\n";
 APtr=&C1; //APtr points at an object of class C.
 BPtr=dynamic_cast<B*>(APtr); //APtr is actually of
 //type C* and type is
 //B*. Hence, cast
 //returns NULL.
if(BPtr!=NULL) //BPtr is NULL.
 BPtr->f2();
else
 cout<<"Invalid cast\n";
}
/*End of dynamicCast01.cpp*/
```


Output

B::f(2) called

Invalid cast

ДЗ

- Рассмотреть другие операторы приведения и преобразования типов

Выводы

1. Перегрузка операторов позволяет использовать привычные операции с объектами пользовательских классов.
2. Оператор перегружается функцией operator@ — как метод класса или как дружественная функция.
3. Дружественная функция нужна, когда левый операнд не является объектом данного класса.
4. Нельзя создавать новые операторы и перегружать ::, ., .*, ?: и sizeof; нужно сохранять привычный смысл операторов.
5. Приведения нового стиля (static_cast, dynamic_cast, reinterpret_cast, const_cast) безопаснее и заметнее в коде, чем приведение в стиле C.

Контрольные вопросы

1. Что такое перегрузка оператора? Зачем она нужна?
2. Как перегрузить оператор + как метод класса и как дружественную функцию?
3. Когда необходимо использовать дружественную функцию?
4. Какие операторы нельзя перегружать? Какие правила нужно соблюдать?
5. Назовите четыре оператора приведения типов в C++ и их назначение.
6. Для чего нужен dynamic_cast и когда он возвращает нулевой указатель?

Список литературы

Основная литература

1. Лафоре Р. Объектно-ориентированное программирование в C++. 4-е изд. — СПб.: Питер, 2004.
2. Страуструп Б. Язык программирования C++. Специальное издание. — М.: Бином, 2011.
3. Липпман С., Лажойе Ж., Му Б. Язык программирования C++. Базовый курс. 5-е изд. — М.: Вильямс, 2014.

Дополнительная литература

4. Шилдт Г. Полный справочник по C++. 4-е изд. — М.: Вильямс, 2006.
5. Майерс С. Эффективный и современный C++. — М.: Вильямс, 2016.
6. Страуструп Б. Программирование: принципы и практика с использованием C++. 2-е изд. — М.: Вильямс, 2016.

Электронные ресурсы

7. C++ Core Guidelines: isocpp.github.io/CppCoreGuidelines
8. Справочник по C++ и стандартной библиотеке: en.cppreference.com

- Спасибо за внимание!