**Перегрузка операторов**

Для перегрузки оператора создается оператор-функция. Чаще всего, оператор-функция является членом класса или дружественной классу. В C++ под операторами, в частности, понимаются оператор индексирования [], оператор вызова функции (), операторы new и delete, операторы . (точка) и -> (стрелка).

Синтаксис оператор-функций – членов класса:

возвр\_тип имя\_класса ::operator#(список аргументов)

{

// выполняемая операция

}

**Замечание:** Наиболее часто типом возвращаемого значения оператор-функции является класс, для которого она определена.

**Замечание:** Оператор-функция, за исключением оператора =, наследуется производным классом.

Замечание: Оператор-функции не могут иметь параметров по умолчанию.

# Перегрузка

Когда оператор-функция – член класса перегружает бинарный оператор, у функции будет только один параметр. Этот параметр получит тот объект, который расположен справа от оператора. Объект слева генерирует вызов оператор-функции и передается неявно, с помощью указателя *this*.

**Пример**: // Перегрузка оператора +, = относительно класса complex

#include <iostream>

using namespace std;

class complex {

int re\_val, im\_val; // значения координат

public:

complex() { re\_val = 0; im\_val = 0; }

complex(int re, int im) { re\_val = re; im\_val = im; }

void get\_re\_im(int &i, int &j) { i = re\_val; j = im\_val; }

complex operator +(complex z2);

complex operator =(complex z2);

};

complex complex::operator+(complex z2)

{

complex temp;

temp.re\_val = re\_val + z2.re\_val; // temp.re\_val=this->re\_val +z2.re\_val;

temp.im\_val = im\_val + z2.im\_val; // temp.im\_val=this->im\_val+z2.im\_val;

return temp;

}

complex complex::operator = (complex z2)

{

re\_val = z2.re\_val;

im\_val = z2.im\_val;

return \*this;

}

int main()

{

complex z1(10, 15), z2(5, 3), z3, z4;

int x, y;

z3 = z1 + z2; // сложение двух объектов (вызов функции operator+())

z3.get\_re\_im(x, y);

cout << "re(z1+z2) = " << x << ", im(z1+z2) = " << y << "\n";

z4 = z1; // присваивание объекта (вызов функции operator=())

z4.get\_re\_im(x, y);

cout << "re z4 = " << x << ", im z4 = " << y << "\n";

system("pause");

return 0;

}

Результаты выполнения программы:

re(z1+z2) = 15, im(z1+z2) = 18

re z4 = 10, im z4 = 15

Следует обратить внимание на то, что для вывода результатов сложения двух комплексных чисел (объект **z3**) и присваивания (**z4**) используются две переменные **x** и **y** и ссылки на эти переменные – **i** и **j**.

**Перегрузка унарных операторов**

Перегрузка унарных операторов подобна перегрузке бинарных операторов. Но, при перегрузке унарного оператора с использованием функции-члена у функции нет параметров, так как есть только один операнд, который и генерирует вызов оператор-функции.

**Пример**: // Перегрузка унарного оператора – инкремента (++)

#include <iostream>

using namespace std;

class complex {

int re\_val, im\_val;

public:

complex() { re\_val = 0; im\_val = 0; }

complex(int re, int im) { re\_val = re; im\_val = im; }

void get\_re\_im(int &i, int &j) { i = re\_val; j = im\_val; }

complex operator++();

};

complex complex::operator++()

{

++re\_val;

++im\_val;

return \*this;

}

int main()

{

complex z(10, 15);

int x, y;

++z;

z.get\_re\_im(x, y);

cout << "re(++z) = " << x << ", im(++z) = " << y << "\n";

system("pause");

return 0;

}

Результаты выполнения программы:

re(++z) = 11, im(++z) = 16

**Дружественные оператор-функции**

Для перегрузки оператора можно использовать дружественную функцию. Дружественной функции указатель *this* не передается. В случае бинарного оператора это означает, что дружественной оператор-функции передаются оба операнда, а в случае унарного – один.

**Пример**: // *Арифметика комплексных чисел (с использованием дружественных функций)*

*Примерный интерфейс:*

Введите первое комплексное число:

1 2

Введите второе комплексное число:

3 4

МЕНЮ

1. Сложение двух комплексных чисел

2. Вычитание двух комплексных чисел

3. Умножение двух комплексных чисел

4. Деление двух комплексных чисел

5. Выход

Выбор пункта меню: 2

z1 = (1,2)

z2 = (3,4)

z1 - z2 = (-2,-2)

МЕНЮ

1. Сложение двух комплексных чисел

2. Вычитание двух комплексных чисел

3. Умножение двух комплексных чисел

4. Деление двух комплексных чисел

5. Выход

Выбор пункта меню: 5

# §8. Динамический полиморфизм. Виртуальные функции (virtual functions)

Виртуальные функции используются для динамического полиморфизма.

В C++ полиморфизм поддерживается двумя способами:

* при компиляции, путем перегрузки операторов и функций;
* при выполнении программы посредством виртуальных функций.

Основой для использования виртуальных функций и динамического полиморфизма являются указатели на производные классы.

**Замечание**: Указатель, объявленный в качестве указателя на базовый класс, может использоваться как указатель на любой класс, производный от этого класса.

Таким образом, для указания на объект производного класса можно воспользоваться указателем базового класса.

**Замечание**: Указатель производного класса нельзя использовать для доступа к объектам базового класса.

Арифметика указателей связана с типом данным. Если указатель базового класса указывает на объект производного класса и затем инкрементируется, то он уже, в общем случае, не будет указывать на следующий объект производного класса.

Виртуальная функция является членом класса. Она объявляется внутри базового класса и переопределяется в производном классе. Для того, чтобы функция стала виртуальной, перед объявлением функции ставится ключевое слово virtual.

Если класс, содержащий виртуальную функцию, наследуется, то в производном классе виртуальная функция переопределяется.

Каждое переопределение виртуальной функции в производном классе определяет ее реализацию. При переопределении виртуальной функции в производном классе, ключевое слово virtual, не ставится.

Виртуальная функция может вызываться так же, как и любая другая функция-член. Однако, только *вызов через указатель*, позволяет осуществить динамический полиморфизм. При этом определение конкретной версии виртуальной функции имеет место не в процессе компиляции, а в процессе выполнения программы.

**Определение**: Класс, содержащий виртуальную функцию, называют виртуальным классом.

Если в виртуальной функции базового класса отсутствует значимое действие, тогда в каждом классе, производном от этого базового, виртуальная функция должна быть переопределена. Это положение в C++ поддерживается использованием чистых виртуальных функций (pure virtual function).

Чистые виртуальные функции не определяются в базовом классе, туда включаются только прототипы этих функций.

**Основная форма:**

virtual тип имя\_функции(список\_аргументов)=0;

Эта форма определения предполагает, что чистая виртуальная функция обязательно должна подменяться в каждом производном классе. Если этого нет, то при компиляции возникнет ошибка.

**Определение**: Если класс содержит хотя бы одну чистую виртуальную функцию, то о нем говорят как об абстрактном классе (abstract class).

Переопределение виртуальной функции внутри производного класса отличается от перегрузки функций. Эти два процесса совершенно различны:

Перегруженная функция отличается типом и (или) числом параметров, а переопределяемая виртуальная функция должна иметь точно такой же тип параметров, то же их число, и такой же тип возвращаемого значения. Если меняется тип или число параметров виртуальной функции, то она теряет свою виртуальность и становится перегружаемой функцией.

Виртуальная функция должна быть членом класса.

Чтобы подчеркнуть разницу между перегружаемыми функциями и переопределяемыми виртуальными функциями, для описания переопределения виртуальной функции используется термин *подмена* (overriding).

Рассмотрим еще два понятия объектно-ориентированного программирования – раннее связывание и позднее связывание.

Раннее связывание относится к событиям, которые происходят в процессе компиляции. Функции раннего связывания – нормальные функции, перегруженные, дружественные. Преимущество – обеспечивается быстродействие выполнения программы.

Позднее связывание относится к событиям, которые происходят в процессе выполнения программы. Виртуальная функция является объектом позднего связывания. Преимущество – повышается гибкость во время выполнения программы.

Рассмотрим пример, в котором доступ к функции производного класса осуществляется через указатель на объект базового класса. Виртуальные функции не используются.

**Пример**: // *Доступ без virtual*

#include <iostream>

using namespace std;

class B {

protected:

int x;

public:

B(int i) { x = i; }

void show() { cout << " x=" << x << "\n"; }

void modify() { x \*= x; }

};

class D : public B {

public:

D(int x) :B(x) {};

void modify() { x += x; }

};

int main()

{

B \*p;

B b\_ob(2);

D d\_ob(5);

b\_ob.show();

d\_ob.show();

p = &b\_ob;

p->modify();

p->show();

p = &d\_ob;

p->modify();

p->show();

system("pause");

return 0;

}

Результаты выполнения программы:

x=2

x=5

x=4

x=25

**Комментарий**: Переопределения функции **modify()** в производном классе **D** не происходит. Вместо ожидаемого значения **10** результатом будет число **25**.

Рассмотрим модификацию предыдущей программы с использованием виртуальной функции.

**Пример:** // Доступ c virtual

#include <iostream>

using namespace std;

class B {

protected:

int x;

public:

B(int i) { x = i; }

void show() { cout << " x=" << x << "\n"; }

virtual void modify() { x \*= x; }

};

class D : public B {

public:

D(int x) :B(x) {};

void modify() { x += x; }

};

int main()

{

B \*p;

B b\_ob(2);

D d\_ob(5);

b\_ob.show();

d\_ob.show();

p = &b\_ob;

p->modify();

p->show();

p = &d\_ob;

p->modify();

p->show();

system("pause");

return 0;

}

Результаты выполнения программы:

x=2

x=5

x=4

x=10

**Комментарий**: Произошло переопределения функции **modify()** в производном классе **D.**

/\* В этом примере показана работа виртуальной функции при наличии случайных событий во время выполнения программы.\*/

#include <iostream>

//#include <cstdlib>

#include<ctime>

using namespace std;

class area {

double a, b;

public:

void set\_a\_b(double x, double y) { a = x; b = y; }

void get\_a\_b(double &x, double &y) { x = a; y = b; }

virtual double get\_area() = 0;

};

class el : public area {

public:

double get\_area() { double x, y; get\_a\_b(x, y); return 3.14\*x\*y; }

};

class re : public area {

public:

double get\_area() { double x, y; get\_a\_b(x, y); return x \* y; }

};

int main()

{

area \*p;

el e;

re r;

e.set\_a\_b(4, 5);

r.set\_a\_b(2, 3);

float j;

srand((unsigned)time(NULL));

j = (float)rand() / (float)RAND\_MAX;

cout << "j=" << j << endl;

if ((j > 0.5))

{

p = &e; cout << "S (ellipse)=" << p->get\_area() << endl;

}

else

{

p = &r; cout << "S (rectangle)=" << p->get\_area() << endl;

}

system("pause");

return 0;

}