Основы ООП

Перегруженные операции. Объекты-исключения

Перегруженные операции (формализм)

C++ позволяет переопределить большинство операций языка так, чтобы при использовании с объектами конкретных классов эти операции выполнялись специфическим образом, через специально заданные функции.

Конкретно, можно перегружать следующие операции языка:

```
+   -   *   /   %   ^   &   |   ~   !  =   <   >   +=  -=  *=  /=  %=  ^=  &=  |=
<<  >>  >>= <<= ==  !=  <=  >=  && ||  ++  --  ->* ,   ->  []  ()  new  delete
```

Последние четыре операции означают: индексацию, вызов функции, размещение в динамической памяти и освобождение памяти.

Для дальнейшего изложения, обозначим @ - некоторую перегружаемую операцию языка *C++*. Пусть x и y – объекты некоторых классов *C++*.

Перегрузка унарных операций.

@x – унарная операция над объектом x в префиксной форме интерпретируется как вызов функции-члена

x.operator@( ) без аргументов

или вызов friend-функции с именем

operator@(x) с одним аргументом.

x@ – унарная операция над объектом x в постфиксной форме интерпретируется как вызов функции-члена

x.operator@(int) , где аргумент типа int - фиктивный

или вызов friend-функции с именем

operator@(x, int) , где аргумент типа int – фиктивный.

.

Перегруженные операции

Перегрузка бинарных операций.

x @ y – бинарная операция с объектами x и y интерпретируется как вызов функции-члена

x.operator@(y) с одним аргументом

или вызов friend-функции с именем

operator@(x , y) с двумя аргументами.

Замечания к перегруженным операциям.

1. При перегрузке операций, полностью сохраняется синтаксис языка *C++*, в том числе приоритет и порядок выполнения операций.
2. Можно перегрузить операции new и delete и начать самостоятельно управлять процессами выделения и освобождения динамической памяти.
3. Можно перегрузить операцию индексации []. Индексы могут иметь любой тип (например, цвет, текст, др.). Это есть ассоциативные массивы.
4. Итераторы в контейнерах можно заменить «умными» перегруженными операциями «++» и «--», которые при обходе не позволяют выйти за пределы контейнера.
5. Можно переопределить операцию косвенной адресации объекта «->» для работы с «умными» указателями, которые при каждом обращении к объекту будут выполнять любую заданную вами работу.
6. Можно перегрузить операцию вызова функции, тогда запись вида <объект> (<список-аргументов>) будет рассматриваться как бинарная операция operator()(…).
7. Можно перегрузить операцию преобразования типа (собственный кастинг).

Перегруженные операции (пример)

Пример. Перегруженные операции для строк.

```
#pragma warning( disable : 4996 4267 ) // выключим некоторые предупреждения
#include <string.h>                    // понадобится для сцепления строк
#include <iostream>                    // задействуем потоковый ввод/вывод
using namespace std;                   // пространство имен для потокового в/в
class str {                            // опишем класс «строки»
        char* s;                       // указатель на символы
        int   n;                       // емкость буфера для хранения строки
public:
        str() { n=1; s=NULL; }         // 0-конструктор (по умолчанию)
        str(const str&);               // конструктор копирования
        str(const char*);              // конструктор общего вида
        ~str() { if(s) delete s; }     // деструктор
        str& operator~();              // операция инверсии строки
        str& operator=(char*);         // операция присваивания
        str& operator=(str&);          // операция присваивания
        friend str& operator+(str&, str&);     // операция сложения
        friend str& operator+(str&, char*);    // операция сложения
        friend str& operator+(char*, str&);    // операция сложения
        friend istream& operator>>(istream&, str&);    // операция ввода
        friend ostream& operator<<(ostream&, str&);  // операция вывода
};
```

Перегруженные операции

Конструкторы и инверсия строк

```
// конструктор копирования
str::str(const str &x) {
        s = new char[n=x.n];        // получим память
        strcpy(s, x.s);          // небезопасная функция!
}
```

```
// конструктор общего вида
str::str(const char *text) {
        s = new char[n=strlen(text)+1];        // получим память
        strcpy(s, text);                  // небезопасная функция!
}
```

```
// инверсия строки
str& str::operator~() {
        if(n>2) {    // имеет смысл для строк длиной больше 1 символа
                char temp, *beg=s, *end=s+n-2;  // рабочие указатели
                while(beg<end) {                // пока они не встретятся
                        temp=*beg; *beg++=*end; *end--=temp; }
        }
        return *this;      // результат работы – сам объект
}
```

Перегруженные операции

Операторы присваивания

```
// оператор присваивания char*
str& str::operator=(char *text) {
        if(s) delete[] s;  // освободим ранее использованную память
        s = new char[n=strlen(text)+1];        // получим память
        strcpy(s, text);  // небезопасная функция!
        return *this;      // результат работы – сам объект
}
```

```
// оператор присваивания str
str& str::operator=(str &x) {
        if(this != &x) {   // не присваиваем ли объект самому себе?
                if(s) delete[] s;  // освободим ранее использованную память
                s = new char[n=x.n];        // получим память
                strcpy(s, x.s);          // небезопасная функция!
        }
        return *this;      // результат работы – сам объект
}
```

Перегруженные операции

Операторы сложения

```
// дружественная функция: сложение двух объектов str
str& operator+(str &x1, str &x2) {
        str *sum = new str;   // новый объект – в нем будет сумма строк
        sum->s = new char[ sum->n = x1.n + x2.n - 1];       // получим память
        strcpy(sum->s, x1.s);       // строка из первого объекта
        strcat(sum->s, x2.s);       // цепляем строку из второго объекта
        return *sum;            // результат – сцепленные строки
}
```

```
// дружественная функция: сложение объекта str + char*
str& operator+(str &x, char *text) {
        str *sum = new str;   // новый объект – в нем будет сумма строк
        sum->s = new char[ sum->n = x.n + strlen(text) ];   // получим память
        strcpy(sum->s, x.s);        // строка из первого объекта
        strcat(sum->s, text);       // цепляем строку из второго объекта
        return *sum;            // результат – сцепленные строки
}
```

Перегруженные операции

Сложение, ввод и вывод строк

```
// дружественная функция: сложение char* и объекта str
str& operator+(char *text, str &x) {
        str *sum = new str;   // новый объект – в нем будет сумма строк
        sum->s = new char[ sum->n = strlen(text) + x.n ];    // получим память
        strcpy(sum->s, text);        // строка из первого объекта
        strcat(sum->s, x.s);         // цепляем строку из второго объекта
        return *sum;             // результат – сцепленные строки
}
```

```
// дружественная функция: потоковый ввод объекта str
istream& operator>>(istream &z, str &x) {
        char buf[BUFSIZ];            // рабочий буфер для ввода
        z.get(buf,BUFSIZ,'\n');      // используем метод get() объекта z
        x=buf;      // операция присваивания для класса str уже определена
        return z;           // возвращаем объект класса istream
}
```

```
// дружественная функция: потоковый вывод объекта str
ostream& operator<<(ostream &z, str &x) {
        return z<<x.s;  // потоковый вывод для char* уже определен в z
}
```

Перегруженные операции

Тестирование класса str

```
// посмотрим, как работает класс str
void main(void) {
        str a, b="xxx", c;      // обзаведемся тремя объектами
        cin>>a;                 // потоковый ввод
        c=a+b;                  // сложение объектов
        cout<<(a+" "+b+" "+c)<<endl;  // сложение и потоковый вывод
        cout<<~a<<b<<c<<endl; // инверсия и потоковый вывод
}
```

Объекты-исключения

В качестве введения в тему рассмотрим два фрагмента кода.

Фрагмент 1.

```
File* f1() {        // f1 – функция, которая пытается открыть некий файл
File *fp;           // возможна нештатная ситуация – ошибка открытия файла
if ( ( fp = fopen ( …, … ) ) == NULL )      // возникла ошибка открытия файла
    { cerr << “file open err”; return NULL; }       // сообщение и аварийный выход
return fp; }                    // успех, нормальный выход
```

При ошибке открытия файла следует проанализировать причины неудачи и предпринять некоторые разумные действия для исправления ситуации.

Фрагмент 2.

```
int f2() {          // f2 – функция, которая пытается создать объект-потомок
class X : public Y { … };   // возможна нештатная ситуация –
X *p = new X;           // объект не будет создан из-за нехватки памяти
if ( p == NULL )        // объект не создан, но конструктор предка уже отработал
    { cerr << “object create err”; return -1; }       // сообщение и аварийный выход
return 0; }             // успех, нормальный выход
```

Объект-потомок класса X не был создан, но конструктор объекта-предка класса Y уже отработал и мог получить (заблокировать) какие-то системные ресурсы.

Вопрос: какова дальнейшая судьба этих ресурсов, будут ли они освобождены?