21. ПЕРЕОПРЕДЕЛЕНИЕ СТАНДАРТНЫХ ОПЕРАЦИЙ

21.1. Основные определения и свойства

В C++ есть возможность распространения действия стандартных операций на операнды абстрактных типов данных.

Для того, чтобы переопределить одну из стандартных операций для работы с операндами абстрактных типов, программист должен написать функцию с именем **operator** □,

где □ – обозначение этой операции (например, + - | += и т. д.).

При этом в языке существует несколько ограничений:

□ нельзя создавать новые символы операций; □

нельзя переопределять следующие операции:

:: (операция разрешения области видимости)

?: (тернарная операция) операция **sizeof**

. (операция уточнения)

.* (операция извлечения из члена класса);

• символ унарной операции не может использоваться для переопределения бинарной операции и наоборот. Например, символ << можно использовать только для бинарной операции, ! – только для унарной, а & – и для унарной, и для бинарной;

• переопределение операций не меняет ни их приоритетов, ни порядка их выполнения (слева направо или справа налево);

• при переопределении операции компьютер не делает никаких предположений о ее свойствах. Это означает, что если стандартная операция **+=** может быть выражена через операции + и =, т. е. a += b эквивалентно a = a + b, то для переопределения операций в общем случае таких соотношений не существует, хотя, конечно, программист может их обеспечить. Кроме того, не делается предположений, например, о коммутативности операции +: компилятор не имеет оснований считать, что a + b, где a и b – переменные абстрактных типов, – это то же самое, что и b + a;

• никакая опрация не может быть переопределена для операндов стандартных типов.

Функция operator () является обычной функцией, которая может содержать от 0 до 2 явных аргументов. Она может быть, а может и не быть функцией-членом класса.

```
class cl{ int i; public:
int get (){return i;} int operator + (int);        //
Бинарный плюс.
}; int operator + (cl&, float);   // Бинарный
плюс.
```

В первой форме бинарного плюса не один, а два аргумента. Первый – неявный. Его имеет любая нестатическая функция-член класса; этот аргумент является указателем на объект, для которого она вызвана. Реализация обеих функций может выглядеть так:

```
int cl::operator + (int op2){ return
i + op2;}
int operator + (cl &op, float op2){ return
op.get() + op2;}
```

Что будет, если в глобальной функции ::operator + () второй аргумент будет иметь тип не float, а int? В этом случае компилятор выдаст сообщение об ошибке, так как он не сможет сделать выбор между функциями cl::operator + () и ::operator + () – обе подходят в равной степени.

Для выполнения переопределенной унарной операции **x** (или **x**), где x – объект некоторого абстрактного типа Class, компилятор пробует найти либо функцию Class::operator (void), либо ::operator (Class). Если найдены одновременно оба варианта, то фиксируется ошибка.

Интерпретация выражения осуществляется либо как x.operator (void), либо как operator (x). Для выполнения переопределенной бинарной операции **x y**, где **x** обязательно является объектом абстрактного типа Class, компилятор ищет либо функцию Class::operator (type y), либо функцию ::operator (Class, type y), причем type может быть как стандартным, так и абстрактным типом.

Выражение **x y** интерпретируется либо как x.operator (y), либо как operator (x, y).

Как для унарной, так и для бинарной операции число аргументов функции operator () должно точно соответствовать числу операндов этой операции. Заметим, что часто удобно передавать значения параметров в функцию operator () не по значению, а по ссылке.

Рассмотрим для примера операцию сложения, определенную над классом «комплектное число»:

```
class complex { double re,
im; public:  double &
real(){return re;} double &
imag(){return im;}
//. . .
};
complex operator + (complex a, complex b){ complex
result;
result.real() = a.real() + b.real();
result.imag() = a.imag() + b.imag(); return
result;}
```

Здесь оба аргумента функции operator + () передаются по значению, то есть выполняется копирование четырех чисел типа double. Подобные затраты могут оказаться слишком накладными, особенно если операция переопределяется над таким, например, классом, как «матрица».

Можно было бы попытаться избежать накладных расходов, передавая по значению не сами объекты, а указатели на них:

```
complex operator + (complex* a, complex *b){. . .}
```

Но так поступать нельзя, так как оба аргумента теперь являются объектами стандартного типа – указателями, а переопределение операций для стандартных типов запрещено.

В этой ситуации необходимо использовать ссылки – они не изменяют тип операндов, а только влияют на механизм передачи параметров:

```
complex operator + (complex &a, complex &b){ complex
result;
result.real() = a.real() + b.real(); result.imag()
= a.imag() + b.imag();
return result; }
```

Тело функции operator + () при этом не изменилось. Пример:

определение операции + для класса stroka:

```
class stroka{ char *c; // Указатель на строку. int len;
// Длина строки. public:
stroka(int N = 80): len(0)          // Строка, не содержащая информацию;
{c = new char[N +1];       // выделение памяти для массива. c[0] = '\0';
}  // Конструктор выделяет память для строки и делает ее пустой.
```

```
stroka(const char * arg){
len = strlen (arg); c =
new char [len + 1];
strcpy(c, arg);
}
int & len_str()                    // Возвращает ссылку на длину строки.
{return len;
}
char * string( )                   // Возвращает указатель на строку.
{return c;}
void display()                     // Печать информации о строке.
{cout << "Длина строки: "<< len << ".\n"; cout
<< "Содержимое строки: " << c <<".\n"; }
~stroka(){delete []c;}
}; stroka & operator + (stroka &a, stroka &b){ int ii = a.len_str() +
b.len_str();   // Длина строки-результата.  stroka * ps = new
stroka (ii);
strcpy(ps -> string(), a.string());          // Копирует строку из a; strcat(
ps->string(), b.string());  // присоединяет строку из  b; ps ->
len_str() = ii;          // записывает значение длины строки; return
*ps;   // возвращает новый объект stroka.
}   void   main(){   stroka
X("Вася");   stroka   Y("
едет");   stroka   Z("   на
велосипеде");
stroka T; T = X
+ Y + Z;
T.display(); }
```

Результат выполнения программы:

```
Длина строки: 23.
Содержимое строки: Вася едет на велосипеде.
```

Заметим, что вместо T = X + Y + Z возможна и такая форма обращения к operator + ():

```
T = operator + (X, Y);
T = operator + (T, Z);
```

Отметим, что для работы со строками хорошо и желательно использовать класс std::string из стандартной библиотеки C++. Перед этим нужно использовать включающую директиву #include <string>.

21.2. Операции *new* и *delete* при работе с абстрактными типами

Операции new и delete реализуются через функции, и вне зависимости от того, описаны или нет operator new() и operator delete как static, они всегда являются статическими функциями. Операция new предопределена для любого типа, в том числе и для абстрактного, определенного через механизм классов. Можно переопределять как глобальную функцию operator new(), так и функцию class x::operator new(). Глобальные new и delete переопределяются обычным образом через механизм соответствия сигнатур.

Как и при переопределении глобальной функции operator new(), переопределенная функция classX::operator new() должна возвращать результат типа void*, а ее первый аргумент должен иметь тип size_t (то есть unsigned), в которой хранится размер выделяемой памяти. Заметим, что при использовании операции new этот аргумент не указывается, а размер необходимого участка памяти вычисляется автоматически, исходя из указанного типа.

21.3. Использование *new* при создании динамического объекта абстрактного типа

Рассмотрим фрагмент: class

C{ . . .

public:

C(int arg){ . . .} };

. . .

C *cp = new C(3);

Создание динамического объекта типа C можно разбить на две стадии:

1. Собственно создание объекта – это выполняет конструктор.
2. Размещение этого объекта в определенной области памяти – это делает операция new.

При этом вначале выполняется функция operator new(), а затем уже конструктор размещает создаваемый объект в выделенной памяти. Операцию new можно переопределить: class cl{ . . . public: cl(){

```
cout << "Конструктор класса cl.\n";
} void* operator new
(unsigned);
}; void* cl::operator new (unsigned size){ cout <<"Функция
operator new() класса cl;\n"; void* p = new (std::nothrow) char
[size];          // Глобальная new! if(p) return p;  else {
cout << "Нет памяти для объекта типа cl!\n"; exit(1);}
} void main(){
cl * cp = new cl;
}
```

Результат:

```
Функция operator new() класса cl; Конструктор
класса cl.
```

21.4. Операция *delete*

Выполнение операции delete применительно к указателю на объект абстрактного типа приводит к вызову деструктора для этого объекта.

```
cl * clp = new cl(5);          // Вызов конструктора cl(5);
. . .
delete clp;                    // вызов деструктора ~ cl()
                               // перед освобождением  //
                               динамической памяти.
```

Функцию x::operator delete() можно переопределить в классе x, причем она может иметь только две формы: void operator delete(void *); void operator delete(void *, size_ t);

Если присутствует вторая форма данной операции, то компилятор использует именно ее.

21.5. Преобразование типов

Преобразование типов можно разделить на 4 группы:

1. стандартный к стандартному;
2. стандартный к абстрактному;
3. абстрактный к стандартному; 4. абстрактный к абстрактному.

Первые преобразования уже были нами рассмотрены. Преобразования второй группы основаны на использовании конструкторов – как явном, так и неявном.

Снова рассмотрим касс complex:

```
class complex { double re, im; public: complex (double r = 0,
double i = 0){ re = r;  im = i ; }
. . .
};
```

Объявления вида

```
complex c1;
complex c2(1.8);
complex c3(1.2,
3.7);
```

обеспечивают создание комплексных чисел.

Но конструктор может вызываться и неявно, в том случае, когда в выражении должен находиться операнд типа complex, а на самом деле присутствует операнд типа double:

```
complex operator + (complex  & op, complex & op2 ); complex
operator -  (complex  & op, complex & op2 ); complex operator
*  (complex  & op, complex & op2 ); complex operator  /
(complex  & op, complex & op2 ); complex operator  -
(complex  & op ); // Унарный минус.
complex res;
res = - (c1 + 2) * c2 / 3 + .5 * c3;
```

Интерпретация, например, выражения -(c1 + 2) будет следующей:

operator -((operator + (c1, complex (double (2)))).

При вычислении этого выражения неявные вызовы конструкторов создадут временные константы типа complex: (2.0, 0.0), (3.0, 0.0), (4.5, 0.0), которые будут уничтожены сразу же после того, как в них отпадет надобность. Заметим, что здесь не только происходит неявный вызов конструктора complex, но и неявное стандартное преобразование значения типа int к типу double. Число уровней неявных преобразований ограничено. При этом правила таковы: компилятор может выполнить не более одного неявного стандартного преобразования и не более одного неявного преобразования, определенного программистом. Пример:

```
class A{public: A(double
d){. . .}
```

```
}; class B{
public:
B(A va){. . .}
}; class C{
public:
C(B vb){ . . .}
};

A var1 (1.2);          // A(double)
B var2 (3.4);          // B(A(double))
B var3 (var1);         // B(A)
C var4 (var3);         // C(B)
C var5 (var1);         // C(B(A))
C var6 (5.6);          // Ошибка! Неявно вызывается C(B(A(double)))
C var7 (A(5.6));       // C(B(A))
```

Ошибка при создании переменной var6 связана с тем, что здесь необходимы два уровня неявных нестандартных преобразований, выполняющихся с помощью вызова конструкторов: double к A, а затем A к B.

При создании переменной var7 одно из этих преобразований – double к A – сделано явным, и теперь все будет нормально.

Таким образом, конструктор с одним аргументом Class::Class(type) всегда определяет преобразование типа type к типу Class, а не только способ создания объекта при явном обращении к нему.

Для преобразования абстрактного типа к стандартному или абстрактного к абстрактному в C++ существует средство – функция, выполняющая преобразование типов, или **оператор-функция преобразования типов**.

Она имеет вид

```
Class::operator type (void);
```

Эта функция выполняет определенное пользователем преобразование типа Class к типу type. Эта функция должна быть членом класса Class и не иметь аргументов. Кроме того, в ее объявлении не указывается тип возвращаемого значения. Обращение к этой функции может быть как явным, так и неявным. Для выполнения явного преобразования можно использовать как традиционную, так и «функциональную» форму. Пример 1:

```
class X{int a, b; public:
```

```
X (X & vx){a = vx.a; b = vx.b;} X
(int i, int j){a = 2 * i, b = 3 * j;}
operator double(){ return (a + b)/2.0;}          // Преобразование  //
                                    абстрактного типа к стандартному.
}; int i =
5;
double d1 = double(i);    // Явное преобразование типа int к double;
double d2 = i ;       // неявное преобразование int к double;
X       xv(5, -8); double d3 = double(xv);  // явное
преобразование типа  X  к  double; double d4 = xv;      //
неявное преобразование X  к  double.
```

Пример 2:

```
// Преобразование абстрактного типа к абстрактному.
class Y{
char* str1;    // Строки  str1  и  str2  хранят символьное  char* str2;
// представление целых чисел. public:
Y       (char* s1, char* s2): str1(s1), str2 (s2){} operator X(){
return X(atoi (str1), atoi (str2));} };
. . .
Y  yvar ("12", "-25");
X  xvar = yvar;
```

При создании переменной xvar перед вызовом конструктора копирования X::X(X&) будет выполнено неявное преобразование значения переменной yvar к типу X. Это же преобразование в явном виде может выглядеть так:

```
X xvar = X(yvar);
X xvar = (X)yvar;
```

Для выражения

```
X xvar = X("12", "-25");
```

компилятор выдаст сообщение об ошибке «не найден конструктор с указанными аргументами». Дело в том, что в отличие от конструктора, оператор-функция преобразования типа не может создать объект абстрактного типа. Она способна только выполнить преобразование значения уже созданного объекта одного типа к значению другого типа.

В последнем же примере объект типа Y еще не существует.