22. НЕКОТОРЫЕ ОСОБЕННОСТИ ПЕРЕОПРЕДЕЛЕННЫХ ОПЕРАЦИЙ

Ограничениями при переопределении операций =, [], (), -> является то, что реализующие их функции operator = () и т. д. обязательно должны являться членами класса и не могут быть статическими функциями. Если говорить о механизме наследования, то обычно производный класс наследует все свойства класса базового. Из этого правила есть два исключения:

1) производный класс не может наследовать конструкторы своего базового класса;

2) операция присваивания, переопределенная для базового класса, не считается переопределенной для его производных классов.

Все остальные операции наследуются обычным образом, то есть если для производного класса нужная операция не переопределена, но она переопределена в его базовом классе, то будет вызвана операция базового класса.

22.1. Операция =

Операция присваивания = является предопределенной для любого абстрактного типа данных.

При этом такая предопределенная операция присваивания интерпретируется не как получение побитовой копии объекта, а как последовательное выполнение операции присваивания над его членами (как стандартных, так и абстрактных типов). Побитовое копирование происходит только тогда, когда операция = не определена.

Предопределенную операцию = можно переопределить. Пример 1:

```
struct memberone{
int i;
memberone & operator = (memberone & a){ cout <<
"Операция копирования класса memberone\n\n"; return
a;
}
};

struct membertwo{
int j;
```

```
membertwo & operator = (membertwo & a){ cout <<
"Операция копирования  класса  membertwo\n\n"; return
a;
}
};

struct contain{ int k; memberone
mo;
membertwo  mt;
};            void
main(){
contain   from;
from.mo.i = 1;
from. mt.j = 2;
from.k   =   3;
contain       to;
to.mo.i   =   0;
to.mt.j    =  0;
to.k  =  0;  to  =
from;
cout << "to.mo.i = " << to.mo.i  << "\n\n"
<< "to.mt.j = "  << to.mt.j   << "\n\n"
<< "to.k = " << to.k << "\n\n"; }
```

Результат работы программы:

```
Операция копирования класса memberone
Операция копирования класса membertwo to.mo.i = 0
to.mt.j = 0 to.k = 3
```

Пример 2:

Рассмотрим снова класс **stroka.**

```
class stroka{
char  *c;  int
len; public:
. . .
stroka & operator = ( stroka & str );
. . .
};
```

```
stroka & stroka::operator = (stroka & str){
if( str.len > len ){
cout << "Длина строки мала! Копирование невозможно!\n";
}
else{
strcpy( c, str.c ); len = str.len;
}
return * this;
}  void  main(
){
stroka A("Строка A"), B("Строка"), C("Str");
A = B; A.display( );
B = C;  B.display( );
C = A;  C.display( ); }
```

В результате выполнения этой программы на мониторе появится:

```
Длина строки: 6
Содержание строки: Строка
Длина строки: 3
Содержание строки: Str
Длина строки мала! Копирование невозможно!
Длина строки: 3
Содержание строки: Str
```

22.2. Операция [ ]

Выражение **x[y]**, где x – объект абстрактного типа Class, интерпретируется как

x.operator [ ] (y)

Заметим, что массив объектов абстрактного типа, как и любого стандартного, имеет стандартный тип – указатель.

Даже если array – массив элементов абстрактного типа Class, выражение array[i] по-прежнему означает *(array + i), вне зависимости от того, переопределена операция [ ] для типа Class или нет. Пример:

```
class A{ int
a [10];
public:
A(){
```

```
for ( int i = 0; i < 10; i ++ )  a[i]
= i + 1;
}
int operator [ ] ( int j ){  return
a [j];
}    };    void
main(   ){   A
array [20];
cout << "array[3][5] = " << array [3][5] << ".\n"; }
```

Результатом работы программы будет: array[3][5] = 6.

Очевидно, что операция [ ], использованная в конструкторе класса A, является стандартной, так как она выполняется над именем массива.

Рассмотрим теперь выражение array [3][5]. Результат его вычисления является таким, как и ожидалось, по следующей причине: операция [ ] выполняется слева направо. Следовательно, выражение array [3][5] интерпретируется как (array[3]).operator[ ](5).

Первая из двух операций [ ] является стандартной, так как выполняется над именем массива. При этом неважно, какой тип имеют его элементы.

Вторая операция [ ] – переопределенная, так как результатом первой операции [ ] является объект типа A.

Встает вопрос: когда имеет смысл переопределять операцию [ ]?

22.3. Классы Array и Matrix

Попробуем создать АТД, который можно было бы использовать в программе подобно массиву.

Чтобы создание такого типа имело смысл, необходимо преодолеть основные недостатки, свойственные обычным массивам C++, а именно:

- необходимость задания размера массива на стадии компиляции;
- отсутствие контроля выхода за границы массива;
- невозможность задания произвольных границ изменения индекса;
- отсутствие предопределенных операций присваивания массивов, выполнения над ними арифметических операций и т. д.

Создадим класс **Array**, являющийся формализацией концепции «**одномерный массив целых**».

Для простоты предполагаем, что массив типа Array имеет тот же диапазон изменения индексов, что и обычный массив C++, а изменение его размера после создания невозможно.

Определим над типом Array операции присваивания, сложения и вывода. Чтобы обращаться к элементам такого массива, переопределим операцию [ ].

```
// Файл Array.cpp
# include < iostream > #
include < stdlib.h >
class Array {
int *pa;                                    // Массив целых;
int   sz;              public:              // размер массива.
Array( const Array &v );
Array( const int a[ ], int s );
Array( int s );
virtual ~Array( ){ delete []pa; }
int size ( ){ return  sz; }
int & operator [ ] ( int );
Array & operator = ( Array& );

// Результат возвращается по ссылке для возможности //
множественного присваивания типа a=b=c;
Array & operator + ( Array& );
ostream & print( ostream& ); };

Array::Array(const int a[], int s ){
// Инициализация массива типа  Array  обычным массивом.
if( s <= 0 ){ cout << "Неверный размер массива: " << s << "\n";
exit (1); }
if(!( pa = new (std::nothrow) int [ sz = s ] )){ cout <<
"Неудача при выделении памяти \n"; exit (1); }
for ( int i = 0; i<sz; i++)pa[i] = a[i];}
 Array::Array( const Array &v ) {         // Конструктор копирования.
if(!( pa = new (std::nothrow) int [ sz = v.sz ] )){
cout << "Неудача при выделении памяти \n";  exit(1);} for
( int i = 0; i < sz; i++) pa[i] = v.pa[i];
}
Array::Array( int s ){
```

```
// Создание неинициализированного массива размером s. if(
s <= 0){ cout << "Неверный размер массива \n"; exit(1); }
if(!(pa = new (std::nothrow) int[ sz = s ])){ cout
<<"Неудача при выделении памяти \n";
exit(1);
}
}

int & Array::operator [ ]( int index ){ if(
index < 0 || index >= sz){
cout << "Выход за границу массива!\n";  exit(1); }
return pa[index]; }
```

/* Так как результат возвращается по ссылке, то возвращается не значение элемента, а сам этот элемент, и поэтому выражение вида c[i], где c – типа Array, может находиться в левой части операции присваивания. */ ostream & Array::print(ostream& out){ out << '\n';

```
for ( int i = 0; i < sz; i++) out << pa[i]<< "  "; out <<"\n";
return out; } ostream & operator << ( ostream & out, const
Array & v ){ v.print ( out ); return out;  }

Array & Array::operator + ( const Array & op2 ){ if(
sz != op2.sz ){
cout << "Попытка сложить массивы разных размерностей!\n";
exit(1); }

Array & tmp = *( new Array (sz));
for ( int i = 0 ; i < sz ; i ++ )  tmp[
i ] = pa[i] + op2.pa[i]; return tmp;
}

Array & Array::operator = ( const Array &v ){ if(
sz != v.sz ){
cout <<"Разные размеры массивов при присваивании!\n";  exit
(1);}
for ( int i = 0; i < sz ; i ++ ) pa[ i ] = v[ i ]; return
( *this );}
// Конец файла Array.cpp.
```

Теперь можно написать следующую программу:

```
# include "Array.cpp" void
main( ){
int a[ ] = { 1, 7, 3, 15, 6, 20, 7 };
```

```
Array mas(a, sizeof  a / sizeof (int));
Array b(7);  // Неопределенный массив. Array c = mas;
// Конструктор копирования. b = mas + c ; mas = b + ( c
= mas ); for ( int i=0; i < 7; i++) cout << a[i] << "  ";  cout
<< "\n";  cout << mas << b << c;        // Сравните эти два
вывода!
}
```

В рассмотренном выше классе Array есть ряд недостатков. В частности, изменение размера объекта типа Array после создания невозможно; при использовании переопределенных операций типа сложения, вычитания и т. д. происходит утечка памяти.

Поэтому целесообразно пользоваться шаблонным классом std::vector, переопределив для него нужные операции.

Создадим теперь класс Matrix, являющийся формализацией концепции двумерного массива. Поскольку двумерный массив – это массив одномерных массивов, будем его строить с использованием шаблонного класса std::vector.

```
// Файл Matr.h
#pragma once
// препроцессорная директива: файл при компиляции
// подключается строго один раз
#include < iostream >
#include < stdlib.h >
#include < vector > using
namespace std;
vector<double> operator +
(const  vector<double>&  a,  const  vector<double>&  b){  //
Переопределение операции + для класса vector<double>
// Аналогично можно переопределить и другие нужные операции.
int n = a.size(); if(n != b.size()){
cout << "Разные размеры массивов при сложении!\n"; exit(1);}
vector<double>res(n); for (int i = 0; i < n; i++) res[i] = a[i] + b[i]; return
res;
}

vector<double>operator * (const vector<double> &a, double v){
// умножение вектора на число справа
int sz = a.size();
vector<double> t(sz);
```

```
for (int i = 0; i < sz; i++)
t[i] = a[i] * v; return t; }
vector<double> operator * ( double v, vector<double> &a){
// умножение вектора на число слева
int sz = a.size(); vector<double> t(sz); for (int i = 0; i < sz; i++) t[i] =
a[i] * v; return t; } double operator*(const vector<double>&a, const
vector<double>&b){
// скалярное произведение векторов
int n = a.size(); if(n
!= b.size()){
cout << "Разные размеры векторов при умножении!\n";
exit(1); }
double res = 0.0;
for (int i = 0; i < n; i++)res += a[i] * b[i];
return res; }
ostream& print(ostream & out, const vector<double>&v){ out
<< '\n';
for (int i = 0; i < v.size(); i++){ out
<< v[i] << "  ";
}   out   <<
'\n';  return
out; }
ostream& operator << (ostream& s, const vector<double> m){
print(s, m); return s;
}
//_____________________________________________________
class Matrix { vector <vector <double> >pm = {};
    // vector из векторов с компонентами double int r = 0; int c =0;
// Размерности матрицы. public:
Matrix() : r(0), c(0), pm{} {}      // Конструктор по умолчанию
Matrix( const Matrix & v);          // Конструктор копирования
Matrix(int, int);  ~Matrix(); int row(){
return r; } int col(){ return c; } vector
<double> & operator [ ] (int index){
// Результатом операции [ ], примененной к объекту типа Matrix,
// должен быть объект типа vector<double>
if(index < 0 || index >= r){
cout << "Выход за границу массива! \n"; exit(1);} return
pm[index];
```

```
}
Matrix operator = (const Matrix&); // Операция присваивания
Matrix operator * (double v){//умножение матрицы на число справа
Matrix t = Matrix(r, c); for
( int i = 0; i < r; i++) for(
int j=0; j<c; j++) t[i][j] =
pm[i][j] * v; return (t);
}
Matrix operator + (const Matrix& v){// сложение массивов int
m, n;
if(((m = r) != v.r) || ((n = c) != v.c)){
cout << "Разные размеры массивов при сложении!\n";
exit(1);} Matrix t(m, n);
for (int i = 0; i < m; i++) t[i] = pm[i] + v.pm[i]; return
t;
}

Matrix operator*(const Matrix& v2){// умножение массивов
int m = r, n = c, k = v2.c; if(c != v2.r){
cout << " Нельзя перемножить матрицы!\n";
exit(1);} Matrix t(m, k); for (int i = 0; i < m;
i++) for (int j = 0; j < k; j++){ t[i][j] = 0.0; for
(int l = 0; l < n; l++) t[i][j] += pm[i][l] *
v2.pm[l][j];}
return t; }
ostream& print(ostream& s);
};
//_____
// Конструктор копирования:
Matrix::Matrix( const Matrix& v): r(v.r), c(v.c ){ if(r){
pm.resize(r);                                    // изменение размера вектора
for (int i = 0; i < r; i++) pm[i] = v.pm[i];
}else pm = {};
}
Matrix::Matrix(int row, int col): r(row), c(col){         // Конструктор
```

```
if(r <= 0 || c<=0){
cout << "Неверные размеры матрицы: ";
cout << row <<"  "<< col << "\n"; exit(1);
}

pm.resize(r);
for (int i = 0; i < r; i++)
pm[i] = vector<double>(c); }
Matrix:: ~Matrix(){                                    //деструктор

pm.clear(); r = 0; c
= 0;  }

Matrix Matrix::operator = (const Matrix &v){

// операция присваивания
if(this == &v) return *this; // случай самоприсваивания if(r
!= v.r || c != v.c){
cout << "Разные размеры матриц при присваивании!\n";
exit(1);  } r = v.r; c = v.c; if(v.r){
for (int i = 0; i < r; i++)
pm[i] = v.pm[i]; }else
pm = {}; return (*this);
}

Matrix operator*(double a, Matrix& v){
                                    // умножение матрицы на число слева
int m = v.row(), n = v.col();
Matrix t(m, n); for ( int i =
0; i < m; i++)
for ( int j = 0; j < n; j++) t[i][j]= v[i][j] * a; return
t;
}

ostream& Matrix::print(ostream& out){ out
<< '\n';
for (int i = 0; i < r; i++){
for (int j=0; j < c; j++) out << pm[i][j] << "  "; out
<< "\n";
```

```
} out <<
'\n'; return
out;
}

ostream& operator << (ostream& s, Matrix &m){
m.print(s); return s;
}
// Конец файла Matrix.h
```

Теперь можно написать программу, в которой используется новый тип данных Matrix.

```
#include < iostream
> #include
"Matrix.h" void
main( ){ int m = 2;
int n = 3; int k = 5;

vector<double> arr(n);

for (int i = 0; i < n; i++){ arr[i] = i;
} cout << "arr:" << endl; cout << arr;
vector<double> arr2 = 2 * arr; cout
<< "arr2:" << endl; cout << arr2;
vector<double> arr3 = arr + 2 * arr2;
cout << "arr3:" << endl; cout <<
arr3; Matrix mas(m, n); for (int i = 0;
i < m; i++) mas[i] = arr * i; cout <<
"mas= " << mas << endl; Matrix
tbl(n, k); for (int i = 0; i < n; i++) for
(int j = 0; j < k; j++){ tbl[i][j] = i + j;
}

Matrix mas3(m, k);
mas3 = mas * tbl;
cout << "mas3:\n";
cout << mas3;
Matrix mas4 = mas3 * 3;
cout << "mas4:\n"; cout
<< mas4;
```

}

В результате выполнения этой программы в консоли появится:

```
arr: 0
1  2
arr2: 0
2  4
arr3: 0
5  10
mas=
0  0  0
0  1  2 mas3:
0  0  0  0  0 5
8  11  14  17
mas4:
0  0  0  0  0
15  24  33  42  51
```