Семинар 11
Специальные методы классов и перегрузка операций при наследовании

Специальные методы классов при наследовании

К специальным методам класса относятся:

- конструктор умолчания
- конструктор копирования
- деструктор
- операция-функция присваивания

Если программист не определяет эти функции, то компилятор создаст их автоматически. Конструктор и операция присваивания не наследуются.

Присваивание при наследовании

При выполнении присваивания из метода operator=() производного класса неявно вызывается одноимённый метод базового класса.
Пример.

```
struct BAS{int b;};
struct DIR:BAS{double d;};
int main()
{
  DIR one, two;
  one.b=12;
  one.d=3.4;
  two=one; // Об. произв. класса присваивается объект произв. класса
  cout<<"two.b="<<two.b<<" two.d="<<two.d<<endl;
  return 0;
}
// Вывод программы: two.b=12 two.d=3.4
```

Присваивание при наследовании

Перегрузка операции-функции присваивания может потребоваться, например, при использовании классов ресурсоёмких объектов.
Пример перегрузки операции-функции присваивания, в которой происходит вызов операции-функции присваивания базового класса:

```
DIR& operator= (const DIR& x){
        if (this==&x) return *this;
        BAS::operator=(dynamic_cast<const BAS &>(x));
        this -> d = x.d;
        return *this;
}
```

Присваивание при наследовании

Также возможно присваивание объекту базового класса значение объекта производного класса. Но это потенциально опасная операция, т.к. базовый класс ничего не знает о других полях производных классов. В этом случае программист в базовый класс должен добавить соответствующий метод.

При присваивании объекту производного класса значения объекта базового класса необходимо определять оператор-функцию присваивания для производного класса таким образом, чтобы программист явно определял, какие значения нужно присваивать полям производного класса, которых не было в базовом:

```
DIR& operator=(const BAS &x){
        BAS::operator=(x);
        this -> d = 3.14159;
        return *this;
}
```

Конструкторы при наследовании

Особенности:

• при создании объекта производного класса из его конструктора всегда явно или неявно вызывается конструктор базового класса

• конструктор базового класса должен быть вызван и выполнен до исполнения операторов тела конструктора производного класса:

```
circle(double x0, double y0, double r) :point(x0, y0), radius(r) {}
```

• Конструктор производного класса не имеет права непосредственно инициализировать поля данных базового класса без вызова конструктора базового класса, даже если эти поля данных унаследованы как открытые

• Если в инициализаторе конструктора производного класса не предусмотреть вызов конструктора базового класса, то неявно будет вызван конструктор умолчания базового класса.

Конструкторы при наследовании

Иногда конструкторы могут применяться для организации присваивания объектов базового класса объектам производного класса.

```
#include <iostream>
using namespace std;

struct BAS{int b;};
struct DIR:BAS{
        double d;
        DIR(const BAS& bi): BAS(bi), d(2.71828) {}
};

int main() {
        BAS bas;
        DIR one(bas); // Здесь происходит вызов конструктора
        bas.b = 10;
        one.d = 0.0;
        one = bas; // И здесь происходит вызов конструктора, а затем присваивание
        cout<<"one.b="<<one.b<<"\tone.d="<<one.d<<endl;
        return 0;
}
```

Деструкторы при наследовании

Деструктор производного класса выполняется раньше деструктора базового класса и автоматически вызывает деструктор базового класса. Уничтожение объектов происходит в обратном порядке по сравнению с их созданием. Сначала уничтожаются компоненты, добавленные при наследовании, а затем объект базового класса.

При множественном наследовании деструкторы базовых классов вызываются в обратном порядке по отношению к перечислению базовых классов в списке спецификаторов баз производного класса.

Перегрузка операций при наследовании

При перегрузке операций при наследовании следует применять операции приведения типов static_cast и dynamic_cast. Приведение типов
dynamic_cast<имя_базы &>(объект класса-наследника) или
static_cast<имя_базы &>(объект класса-наследника)
Выделяет из объекта производного класса базовую часть. Не во всех компиляторах получится выделить базовую часть с помощью dynamic_cast из неполиморфного класса (класса не имеющего виртуальных функций). В этом случае используйте static_cast.

Попытка приводить объекты к базовому классу, а не к ссылке на базовый класс приведёт к появлению объектов-копий, что не всегда желательно (например, при вводе данных данные будут введены в поля объекта-копии, но не в поля того объекта, в который вы желаете ввести данные).

Перегрузка операций при наследовании

И снова о приведении типов:
операции
(имя_типа) выражение и
имя_типа(выражение)
Выполнят обращение к конструктору приведения типов: будет создан новый объект

Когда нужно осуществить приведение указателей и ссылок (т.е. приведение типа указателя/ссылки на один объект к указателю/ссылке на другой объект), то используют специальные операции static_cast и dynamic_cast.

При использовании static_cast не выполняется никаких проверок во время выполнения программы, и возможны некорректные преобразования типов.

При использовании dynamic_cast осуществляется проверка допустимости преобразования. Если преобразование недопустимо, возвращается значение 0 или формируется исключение bad_cast.

Перегрузка операций при наследовании

Пример.

```
#include "stdafx.h"
#include<iostream>
using namespace std;

class BAS{
protected: int k;
  friend istream & operator >> (istream & in, BAS & b);
  friend ostream & operator << (ostream & out, const BAS & b);
};
istream & operator >> (istream & in, BAS & b){
        cout << "k = ";
        in >> b.k;
return in;
}
ostream & operator << (ostream & out, const BAS & b){
        out << "k = " << b.k << endl;
        return out;
}
```

Перегрузка операций при наследовании


```
class DIR : public BAS{
double z;
        friend istream & operator >> (istream & in, DIR & d);
        friend ostream & operator << (ostream & out, const DIR & d);
};
istream & operator >> (istream & in, DIR & d){
        cout << "BAS: ";
        operator>>(in, static_cast<BAS&>(d)); // Для ввода значений базовой части
        cout << "DIR::z = ";
        in >> d.z;
        return in;
}
ostream & operator << (ostream & out, const DIR & d){
        out << "BAS: ";
        out << static_cast<const BAS &>(d);
        out << "DIR::z = " << d.z << endl;
        return out;
}
```

```
int _tmain()
{
        DIR one, two;
        cin >> one;
        two = one;
        cout << two;
        return 0;
}
```

Перегрузка операций при наследовании

Путём использования const_cast<целевой тип>(выражение) можно отменить атрибут const и volatile (последний атрибут говорит компилятору о невозможности оптимизации кода, связанного с этой переменной).
Например, изменив код с двух предыдущих страниц, можно каждый раз при выводе поля b базового класса BAS инкрементировать его значение:

```
ostream & operator << (ostream & out, const BAS & b){
        (const_cast<BAS &>(b)).k++;
        out << "k = " << b.k << endl;

        // две строки выше для простоты восприятия можно заменить:
        // BAS & c = const_cast<BAS &>(b);
        // c.k++;
        // out << "k = " << c.k << endl;

        return out;
}
```

Изменим код в main():

```
int _tmain()
{
DIR one, two;
cin >> one;
two = one;
cout << two;
cout << two;
cout << two;
return 0;
}
```