Урок 3

Виртуальные функции и полиморфизм

Основы виртуальных функций. Чистые виртуальные функции и абстрактные классы. Интерфейсные классы. Раннее и позднее связывание. Перегрузка функций и методов.

Указатели и ссылки при наследовании

Вспомним, как работают указатели и ссылки. Допустим, есть базовый класс **Parent** и производный **Child**:

```
class Parent
{
protected:
    int m_age;
public:
    Parent(int age) : m_age(age)
    { }
    int getAge() { return m_age; }
};

class Child : public Parent
{
public:
    Child(int age) : Parent(age)
    { }
    int getAge() { return m_age / 2; }
```

Создадим объект класса **Child** и обратимся к его членам-функциям:

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

int main()
{
    Child child(40);
    cout << "child has age " << child.getAge() << '\n';

    Child &rChild = child; // ссылка на объект child
    cout << "rChild has age " << rChild.getAge() << '\n';

    Child *pChild = &child; // указатель на объект child
    cout << "pChild has age " << pChild->getAge() << '\n';

    return 0;
}
```

Результат выполнения программы:

child has age 20

rChild has age 20

pChild has age 20

Обратите внимание, что при вызове члена-функции через указатель **pChild** был использован оператор **->** (стрелка). Теперь, если мы создадим указатель и ссылку типа **Parent** на объект **child**, также сможем вызывать члены-функции объекта **child**. Но при одном условии: если эти функции будут членами класса **Parent**.

```
int main()
{
    Child child(40);
    Parent &rParent = child;
    Parent *pParent = &child;

    cout << "child has age " << child.getAge() << '\n';
    cout << "rParent age " << rParent.getAge() << '\n';
    cout << "pParent age " << pParent->getAge() << '\n';

    return 0;
}
```

Результат выполнения программы:

child has age 20

rParent age 40

pParent age 40

Через **rParent** и **pParent** можно вызвать только члены базового класса, при этом они не видят члены производного класса. Это проблема, решение которой рассмотрим дальше. Можно было бы отказаться от создания ссылок или указателей типа базового класса и сразу создавать их с типом производного класса. Но у такого подхода есть ряд недостатков. Например, есть несколько дочерних классов (**Child1**, **Child2**), и нам необходимо создать функцию **writeAge()**, которая будет выводить на экран значение члена-функции **getAge** для любого объекта из этих классов (причем эти члены-функции для каждого класса разные). Параметром функции **writeAge()** может быть объект любого из дочерних классов, поэтому логичнее сделать этот параметр типа базового класса:

```
void writeAge(Parent &someChild)
{}
```

Но мы не получим нужного решения, так как доступ к уникальным членам дочерних классов через ссылку закрыт.

Виртуальные функции

Виртуальная функция — это метод класса, который может быть переопределен в дочерних классах так, что конкретная реализация метода для вызова будет определяться во время исполнения. При использовании виртуальной функции вызывается «самый дочерний» метод — при условии, что имя, тип параметров, тип возврата дочернего метода совпадает с методом родительского класса. Если тип параметров, их количество, тип возврата метода не будет совпадать, то реализуется механизм перегрузки. Для создания виртуальной функции перед ее объявлением нужно указать ключевое слово **virtual**. С помощью виртуальных функций может быть решена проблема, поставленная в предыдущем разделе.

```
class Parent
{
protected:
    int m_age;
public:
    Parent(int age) : m_age(age)
    { }
    virtual int getAge() { return m_age; }
};
class Child: public Parent
{
public:
    Child(int age) : Parent(age)
    { }
    int getAge() override
      { return m_age / 2; }
};

void writeAge( Parent &someChild)
{
    cout << "rParent age " << someChild.getAge() << '\n';
}

int main()
{
    Child child(40);
    Parent &rParent = child;
    writeAge(rParent);
    return 0;
}
```

Результат выполнения программы:

rParent age 20

Так как **getAge()** является виртуальной функцией, компилятор проверяет, есть ли переопределение этой функции в дочерних классах. Если есть, выполняется именно метод из дочернего класса. Обратите внимание, что переопределенная функция имеет идентификатор **override**, который используется, чтобы явно указать, что функция должна переопределять виртуальную функцию, объявленную в базовом классе. Идентификатор **override** помогает избегать ошибок в коде, а также лучше понимать текст программы. Применять данный идентификатор не обязательно, поэтому мы его не будем использовать в дальнейшем.

Если функция объявлена как виртуальная, то и все переопределения функции тоже являются виртуальными, даже если это не указано явно в коде. Обратите внимание, что тип возврата виртуальной функции должен совпадать во всех ее переопределениях. Нельзя вызывать виртуальные функции в теле конструктора или деструктора, так как всегда будет вызываться родительская версия метода.

Иногда бывает необходимо проигнорировать вызов переопределений, созданных виртуальными функциями. Для этого перед вызовом функции следует указать имя класса, которому принадлежит нужная версия метода, с использованием оператора разрешения области видимости (**::**).

Несмотря на явные преимущества виртуальных функций, повсеместное их использование будет неэффективным, так как обработка и вызов виртуального метода занимает больше времени.

Теперь можем поговорить о еще одном принципе ООП — полиморфизме.

Полиморфизм — это возможность применять одноименные методы с одинаковыми или различными наборами параметров в одном классе или в группе классов, связанных отношениями наследования. Другими словами, полиморфизм позволяет переопределять методы для классов-наследников. Он реализуется с помощью виртуальных функций.

Виртуальные деструкторы

Виртуальные деструкторы работают по тому же принципу, что и виртуальные функции. При вызове деструктора базового класса вначале должны вызываться деструкторы производных классов. Это возможно, если деструктор объявлен как виртуальный. Рассмотрим на примере:

```
#include <iostream>
using namespace std;
class Parent
{
public:
    virtual ~Parent() // Деструктор виртуальный
    {
        cout << "Calling ~Parent()" << endl;
    }
};

class Child: public Parent
{
private:
    int* m_grades;

public:
    Child(int length)
    {
        m_grades = new int[length];
    }

    ~Child() // Деструктор виртуальный
    {
        delete[] m_grades;
    }
};

int main()
{
    Child *child = new Child(7);
    Parent *parent = child;
    delete parent;

    return 0;
}
```

Если бы деструкторы были не виртуальными, то при удалении указателя **parent** вызывался бы только деструктор базового класса. При этом член-переменная **m_grades** не будет удалена. Объявление деструктора виртуальным исправляет эту проблему.

При наследовании создавайте деструкторы виртуальными.

Абстрактные классы

Встречаются задачи, где не нужно определять виртуальную функции внутри базового класса. В таком случае объявляют **чисто виртуальную функцию** (или абстрактную функцию). Чтобы ее объявить необходимо написать прототип функции и присвоить ей значение 0. Класс, в котором присутствует

чисто виртуальная функция, называется **абстрактным**. При объявлении абстрактной функции должно выполняться два условия:

- нельзя создавать объекты абстрактного класса;
- все производные классы должны переопределять все чисто виртуальные функции.

Абстрактные классы — крайне полезный инструмент. Их можно применять для создания указателей и использования преимуществ всех полиморфных способностей. Например:

```
using namespace std;
class Animal
{
public:
    virtual void say()=0;
};
class Cat: public Animal {
public:
    void say()
    {
        cout << "Meow" << endl;
    }
};
class Dog: public Animal
{
public:
    void say() {
        cout << "Woof" << endl;
    }
};

int main() {
    Cat cat;
    Dog dog;
    Animal *animal1 = &cat;
    Animal *animal2 = &dog;

    animal1 -> say();
    animal2 -> say();
}
```

В данном примере невозможно создать объект класса **Animal**. Но используя указатель типа **Animal**, можно вызывать методы нужных классов.

Интерфейсные классы

Интерфейсные классы не имеют переменных-членов, и все их методы — это чисто виртуальные функции. Любой класс-наследник должен предоставлять свою реализацию виртуальных методов. Интерфейсные классы создаются для конструирования «рабочих» классов.

Использование интерфейсных классов снимает большинство проблем, которые могли бы возникнуть при множественном наследовании. Это связано с отсутствием переменных-членов и определений методов.

Один интерфейсный класс может наследовать от нескольких интерфейсных классов. Таким образом создается многоуровневая направленная иерархия.

Обычные классы могут наследовать не от одного интерфейсного класса, но он должен дать определение всем методам, перечисленным в заголовках родительских классов. Если хотя бы один метод будет не переопределен, то новый класс автоматически становится абстрактным.

Интерфейсным классам принято давать названия, начинающиеся с буквы **I**. Пример такого класса:

```
class IErrorLog
{
public:
    virtual bool openLog(const char *filename) = 0;
    virtual bool closeLog() = 0;

    virtual bool writeError(const char *errorMessage) = 0;

    virtual ~IErrorLog() {}; // создаём виртуальный деструктор в случае, если удалим указатель на IErrorLog, то чтобы вызывался соответствующий деструктор дочернего класса
};
```

Любой класс, который наследует **IErrorLog**, должен предоставить свою реализацию всех трех методов класса **IErrorLog**. Вы можете создать дочерний класс с именем **FileErrorLog**, где **openLog()** открывает файл на диске, **closeLog()** закрывает файл, а **writeError()** записывает сообщение в файл. Можно создать еще один дочерний класс с именем **ScreenErrorLog**, где **openLog()** и **closeLog()** ничего не делают, а **writeError()** выводит сообщение во всплывающем окне на экран.

Раннее и позднее связывание

Связывание — это процесс, во время которого компилятор сопоставляет с идентификаторами адреса в памяти. Связывание производится как для переменных, так и для функций. Существует два типа связывания: **раннее** (статическое) или **позднее** (динамическое). Рассмотрим их на примере функций.

Ранее связывание означает, что компилятор до выполнения кода определяет адрес функции и затем уже просто обращается по этому адресу. До этого мы использовали только раннее связывание.

Механизм позднего связывания представляет выбор реализации нужной функции на этапе выполнения программы. Заранее должен быть известен тип ссылки на функцию, и уже потом выбирается ее нужная реализация. В C++ данный процесс реализуют указатели на функции. Например:

```
#include <iostream>
using namespace std;
int add(int a, int b)
{
    return a + b;
}

int main()
{
    // Создаем указатель на функцию add
    int (*pF)(int, int) = add;
    cout << pF(4, 5) << endl; // вызов add(4 + 5)

    return 0;
}
```

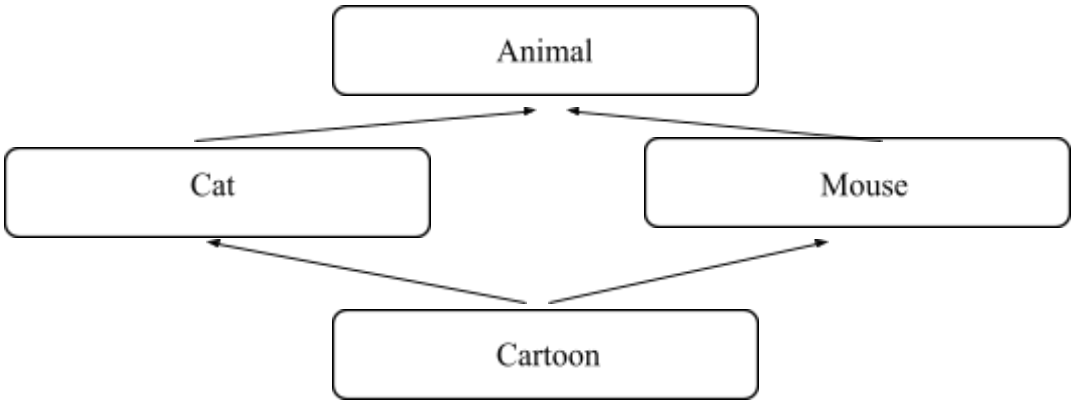
Здесь был создан указатель на функцию **add()**. Позднее связывание обладает большей гибкостью, так как не нужно заранее знать, какую функцию вызывать. Но позднее связывание менее эффективно, так как появляется дополнительный указатель, который хранит адрес вызываемой функции.

Виртуальные функции — это пример использования позднего связывания. Если у дочернего класса существует переопределение виртуальной функции, то выполняется именно это переопределение. В ином случае выполняется функция, определенная в базовом классе. При создании указателя базового класса и вызове виртуальной функции через этот указатель компилятор не знает, виртуальная функция объекта какого класса (базового или производного) будет вызвана. Именно поэтому используется позднее связывание.

Виртуальный базовый класс

Вспомним проблему «алмаз смерти», которая возникает при множественном наследовании. Вкратце: от базового класса A наследуют два класса: B и C, — а от B и C наследует один класс — D. Проблема в том, что при создании объекта класса D непонятно, какой класс создает объект класса A — класс B или C?

Рассмотрим множественное наследование на конкретном примере:



Для решения данной проблемы создаются виртуальные базовые классы. Таким образом компилятор понимает, что за создание базового класса отвечает класс **Cartoon** (самый дочерний).

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

class Animal
{
public:
    Animal(char a[]) {
        cout << a << endl;
    }
};

class Cat: virtual public Animal
{
public:
    Cat(char a[], char c[]) : Animal(a)
    {
        cout << c << endl;
    }
};

class Mouse: virtual public Animal
{
public:
    Mouse(char a[], char m[]) : Animal(a)
    {
        cout << m << endl;
    }
};

class Cartoon: public Cat, public Mouse
{
public:
    Cartoon(char a[], char c[], char m[]) :
    Cat(a,c), Mouse(a,m), Animal(a)
    { }
};

int main() {
    Cartoon animal("animal", "cat", "mouse");
}


int main() {
    Cartoon animal("animal", "cat", "mouse");
}
```

Результат выполнения программы:

animal
cat
mouse

Теперь понятно, что в момент создания объекта класса **Cartoon** именно этот класс и создает объекты своих родительских классов. Это необходимо учитывать при написании конструктора класса **Cartoon**.

Перегрузка операторов

Перегрузка функций — это определение нескольких функций с одинаковым именем, но различными параметрами. Наборы параметров перегруженных функций могут отличаться порядком следования, количеством, типом. Таким образом, перегрузка функций нужна для того, чтобы избежать дублирования имен функций, выполняющих сходные действия, но обладающих различной программной логикой.

В C++ оператором называется действие или функция, обозначенная специальным символом. Чтобы распространять эти действия на новые типы данных, сохраняя естественный синтаксис, в C++ была введена возможность перегрузки операторов.

Список операторов:

```
+ - * / %               //Арифметические операторы
+= -= *= /= %=
+a -a                   //Операторы знака
++a a++ --a a--         //Префиксный и постфиксный инкременты
&& || !                 //Логические операторы
& | ~ ^
&= |= ^=
<< >> <<= >>=           //Битовый сдвиг
=                       //Оператор присваивания
== !=                   //Операторы сравнения
< > >= <=
```

Специальные операторы:

```
&a  *a  a->  a->*
()  []
(type)
.   ,   (a ? b : c)
```

Перегрузить можно почти все операторы, кроме тернарного оператора (?:), оператора (sizeof), оператора разрешения области видимости (::), оператора выбора члена (.) и указателя в качестве оператора выбора члена (.*).

Есть ряд правил и ограничений для перегрузки операторов:

1. **Перегружать можно только существующие операторы** — нельзя создавать новые или переименовывать существующие.
2. **Хотя бы один операнд перегруженного оператора должен быть пользовательского типа данных** — нельзя перегружать операторы только со встроенными типами данных.
3. **Нельзя изменять количество операндов, поддерживаемых операторами** — унарный оператор имеет только один операнд, бинарный — два, тернарный — три.
4. **Все операторы сохраняют свой приоритет и ассоциативность по умолчанию**.

Перегрузить операторы можно тремя способами:

- через дружественные функции;
- через обычные функции;
- через методы класса.

Перегрузка операторов через дружественные функции

Покажем пример перегрузки оператора +:

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

class Dollars
{
private:
    int m_dollars;

public:
    Dollars(int dollars) { m_dollars = dollars; }

    // выполняем Dollars + Dollars через дружественную функцию
    friend Dollars operator+ (const Dollars &d1, const Dollars &d2);

    int getDollars() const { return m_dollars; }
};

Dollars operator+(const Dollars &d1, const Dollars &d2)
{
    // используем конструктор Dollars и operator+(int, int)
    // мы имеем доступ к закрытому члену m_dollars, поскольку эта функция
является дружественной классу Dollars
    return Dollars(d1.m_dollars + d2.m_dollars);
}

int main()
{
    Dollars dollars1(7);
    Dollars dollars2(9);
    Dollars dollarsSum = dollars1 + dollars2;
    cout << "I have " << dollarsSum.getDollars() << " dollars.";
    return 0;
}
```

Результат исполнения программы:

I have 16 dollars.

Поскольку перегруженная функция является дружественной для класса **Dollars**, то она имеет доступ к закрытым членам этого класса. Определять дружественную функцию можно и внутри класса, но лучше записывать определения функций в отдельном .cpp-файле, чтобы разделять интерфейс и реализацию.

Необходимо предусмотреть случай операторов с разными типами операндов. Например, в нашем случае необходим еще добавить **operator+(Dollars, int)**, а также симметричный ему оператор **operator+(int, Dollars)**.

Аналогичным образом можно перегружать операторы -, *, /. Обратите внимание, что операторы - и / не являются ассоциативными, то есть порядок следования операндов имеет значение.

Перегрузка операторов через обычные функции

Если у вас в классе присутствуют get-функции, можно перегружать операторы через обычные функции, которые не являются членами класса.

```
#include <iostream>
using namespace std;
class Dollars
{
private:
    int m_dollars;

public:
    Dollars(int dollars) { m_dollars = dollars; }

    int getDollars() const { return m_dollars; }
};

// Эта функция не является ни методом класса, ни дружественной классу Dollars!
Dollars operator+(const Dollars &d1, const Dollars &d2)
{
    // используем конструктор Dollars и operator+(int, int)
    // здесь нам не нужен прямой доступ к закрытым членам класса Dollars
    return Dollars(d1.getDollars() + d2.getDollars());
}

int main()
{
    Dollars dollars1(7);
    Dollars dollars2(9);
    Dollars dollarsSum = dollars1 + dollars2;
    cout << "I have " << dollarsSum.getDollars() << " dollars.";

    return 0;
}
```

В данном примере перегрузка осуществляется через обычную функцию. Данный способ рекомендуется использовать, когда в классе есть get-функции. Если их нет, специально создавать не надо, чтобы не загружать код дополнительными функциями. Поэтому необходимо использовать перегрузку через дружественные функции.

Перегрузка операторов через методы класса

Пример перегрузки оператора через метод класса:

```
#include <iostream>
using namespace std;
class Dollars
{
private:
    int m_dollars;

public:
    Dollars(int dollars) { m_dollars = dollars; }
    // Эта функция является методом класса!
    // Вместо параметра dollars в перегрузке через дружественную функцию здесь неявный параметр, на который указывает указатель *this
    Dollars operator+(int value)
    {
        return Dollars(m_dollars + value);
    }
    int getDollars() { return m_dollars; }
};

int main()
{
    Dollars dollars1(7);
    Dollars dollars2 = dollars1 + 3;
    cout << "I have " << dollars2.getDollars() << " dollars.\n";
    return 0;
}
```

В данном примере у оператора только один аргумент. На самом деле, он явный, а есть еще один — неявный, который ссылается с помощью указателя ***this** на текущий объект. Поэтому в строке **return Dollars(m_dollars + value)** переменная-член **m_dollars** является членом того объекта, который вызвал этот метод.

Через методы класса нужно переопределять операторы присваивания (=), индекса ([]), вызова функции (()) и выбора члена (->). Это требование языка C++.

Через методы класса нельзя переопределять операторы ввода (>>) и вывода (<<). Также не получится переопределить бинарный оператор, левый операнд которого не является классом (например, **int**) или это класс, который мы не можем изменить (например, **ostream**). Это следует из того, что левый операнд должен обязательно быть членом данного класса.

Каким из трех способов переопределять операторы, решать вам. Но обычно бинарные операторы, которые не изменяют левый операнд, переопределяют через обычные или дружественные функции. Если же левый операнд изменяется, то используют перегрузку через методы класса (например, **operator+=**). Унарные операторы тоже перегружаются через методы класса, так как в таком случае параметры не используются вовсе.

Операторы	Перегрузка через обычную или дружественную функцию	Перегрузка через метод класса
Унарный оператор	-	+
Бинарный оператор, не изменяющий левый операнд	+	-
Бинарный оператор, изменяющий левый операнд	-	+

Перегрузка унарных операторов

Перегрузим унарный оператор минус (-) и логический оператор НЕ (!).

```
#include <iostream>
using namespace std;
class Vector
{
private:
    double m_x, m_y, m_z;

public:
    Vector(double x = 0.0, double y = 0.0, double z = 0.0) :
    m_x(x), m_y(y), m_z(z)
    {
    }

    Vector operator- () const
    {
        return Vector(-m_x, -m_y, -m_z);
    }

    bool operator! () const
    {
        return (m_x == 0.0 && m_y == 0.0 && m_z == 0.0);
    }

    double getX() {return m_x;}
    double getY() {return m_y;}
    double getZ() {return m_z;}
};

int main()
{
    Vector v1; // используем конструктор по умолчанию со значениями 0.0, 0.0,
0.0
    if (!v1)
        cout << "Vector_1 is null.\n";
    else
        cout << "Vector_1 is not null.\n";

    Vector v2(1,2,3);
    cout << "Opposite vector_2 is (" << -v2.getX() << ", " << -v2.getY() << ", "
<< -v2.getZ() << ").";

    return 0;
}
```

Результат выполнения программы:

Vector_1 is null.

Opposite vector_2 is (-1, -2, -3).

Обратите внимание, что методы класса объявлены константными. Это сделано потому, что оператор не изменяет объект, вызвавший этот метод. При этом константный метод также необходим, если мы используем его для константного объекта этого класса.

Создание классов для игры Blackjack

Начиная с этого урока, будем с использованием полученных знаний по ООП создавать игру Blackjack. Суть проста: раздаются карты, за каждую из которых начисляются очки. Игрок пытается заработать 21 очко, не больше. За каждую карту с числом дается столько очков, сколько на ней указано. За туз — либо 1 очко, либо 11 (что больше подходит игроку), а за валета, даму и короля — по 10 очков.

Компьютер выступает дилером и играет против группы от одного до семи игроков. В начале кона все участники, включая дилера, получают две карты. Игроки могут видеть все свои карты, а также сумму очков. Однако одна из карт дилера скрыта на протяжении всего кона.

Далее каждый игрок может брать дополнительные карты. Если сумма очков у него превысит 21, он проигрывает. Когда все игроки получили возможность набрать дополнительные карты, дилер открывает скрытую карту. Далее он обязан брать новые карты до тех пор, пока сумма его очков не превышает 16. Если у дилера перебор, все игроки, не имеющие перебора, побеждают. В противном случае сумма очков каждого из оставшихся игроков сравнивается с суммой очков дилера. Игрок побеждает, если сумма его очков больше, чем у дилера. В противном случае — проигрывает. Если суммы очков игрока и дилера одинаковы, засчитывается ничья.

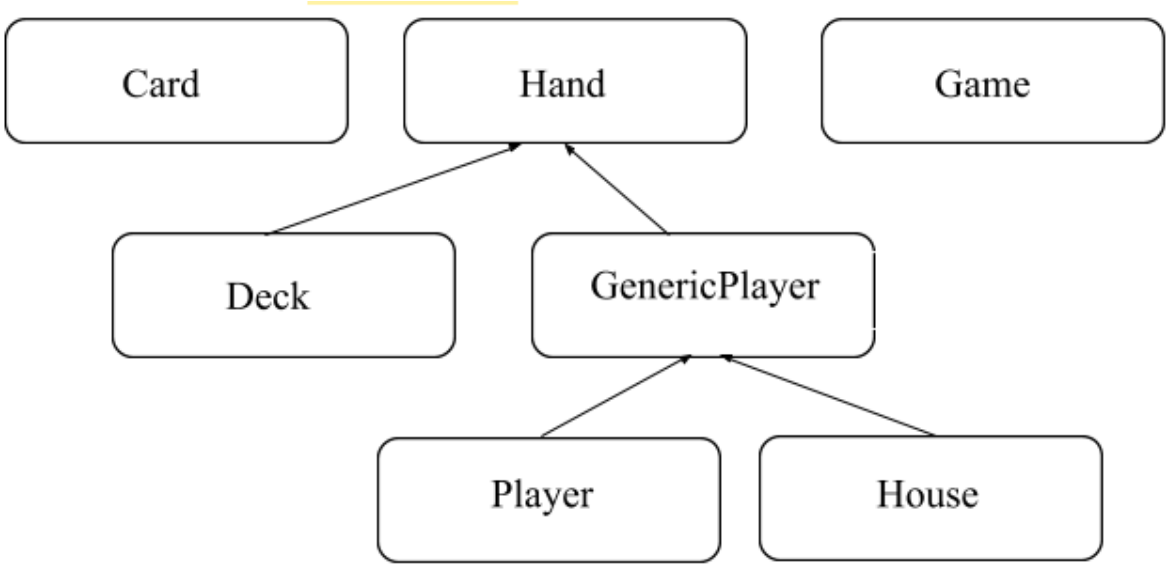
Разработаем классы, которые будем использовать в нашей программе:

Класс	**Родительский класс**	**Описание**
Card	нет	Карта
Hand	нет	Набор карт, коллекция объектов класса **Card**
Deck	Hand	Имеет дополнительную функциональность, которая отсутствует в классе **Hand**, в частности — тасование и раздачу
GenericPlayer	Hand	Обобщенно описывает игрока. Не является полноценным игроком, а лишь содержит элементы, характерные как для игрока-человека, так и для игрока-компьютера
Player	GenericPlayer	Человек-игрок
House	GenericPlayer	Компьютер-игрок
Game	нет	Игра

Обратите внимание: класс **GenericPlayer** создается для того, чтобы общая функциональность классов **Player** и **House** не дублировалась в обоих классах.

Колода карт отделена от дилера, поэтому карты колоды будут раздаваться игрокам-людям и игроку-компьютеру на равных. Это значит, что функция-член, предназначенная для раздачи карт, будет полиморфной.

Приведем иерархию данных классов:



Практические задания

1. Создать абстрактный класс Figure (фигура). Его наследниками являются классы Parallelogram (параллелограмм) и Circle (круг). Класс Parallelogram — базовый для классов Rectangle (прямоугольник), Square (квадрат), Rhombus (ромб). Для всех классов создать конструкторы. Для класса Figure добавить чисто виртуальную функцию area() (площадь). Во всех остальных классах переопределить эту функцию, исходя из геометрических формул нахождения площади.
2. Создать класс Car (автомобиль) с полями company (компания) и model (модель). Классы-наследники: PassengerCar (легковой автомобиль) и Bus (автобус). От этих классов наследует класс Minivan (минивэн). Создать конструкторы для каждого из классов, чтобы они выводили данные о классах. Создать объекты для каждого из классов и посмотреть, в какой последовательности выполняются конструкторы. Обратить внимание на проблему «алмаз смерти».

   Примечание: если использовать виртуальный базовый класс, то объект самого "верхнего" базового класса создает самый "дочерний" класс.

3. Создать класс: Fraction (дробь). Дробь имеет числитель и знаменатель (например, 3/7 или 9/2). Предусмотреть, чтобы знаменатель не был равен 0. Перегрузить:
   - математические бинарные операторы (+, -, *, /) для выполнения действий с дробями
   - унарный оператор (-)
   - логические операторы сравнения двух дробей (==, !=, <, >, <=, >=).

   Примечание: Поскольку операторы < и >=, > и <= — это логические противоположности, попробуйте перегрузить один через другой.

   Продемонстрировать использование перегруженных операторов.

4. Создать класс Card, описывающий карту в игре БлэкДжек. У этого класса должно быть три поля: масть, значение карты и положение карты (вверх лицом или рубашкой). Сделать поля масть и значение карты типом перечисления (enum). Положение карты - тип bool. Также в этом классе должно быть два метода:
   - метод Flip(), который переворачивает карту, т.е. если она была рубашкой вверх, то он ее поворачивает лицом вверх, и наоборот.
   - метод GetValue(), который возвращает значение карты, пока можно считать, что туз = 1.

Дополнительные материалы

1. Бьерн Страуструп. Программирование. Принципы и практика использования C++.
2. Стивен Прата. Язык программирования C++. Лекции и упражнения.
3. Роберт Лафоре. Объектно-ориентированное программирование в C++.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Онлайн справочник программиста на C и C++. Полиморфизм.
2. Бьерн Страуструп. Программирование. Принципы и практика использования C++.
3. Ральф Джонсон, Ричард Хелм, Эрих Гамма. Приемы объектно-ориентированного программирования. Паттерны проектирования.