ОСНОВЫ ТЕОРИИ И ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Пышкин Евгений Валерьевич
к.т.н., доцент

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ

Блок 7. Иерархии классов, наследование

Отношения между объектами

- Взаимосвязь любого рода между объектами различных типов
- Объекты, встречающиеся в определенной задаче, вместе с отношениями между ними образуют **объектную модель**

Пример объектной модели - игра в шахматы

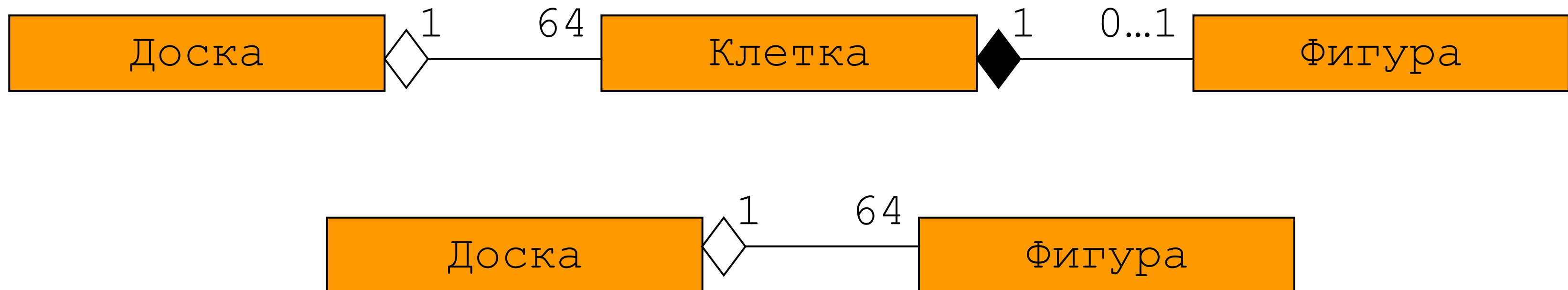
- **Доска** состоит из **клеток**, на каждой из которых может находиться **фигура**
- **Фигура** может быть **королем**, **ферзем**, **ладьей**, **слоном**, **конем** или **пешкой**
- **Ход** заключается в передвижении **фигуры** с одной **клетки** на **другую**
- **Игроки** могут делать **ходы**
- **Игрок** может быть **человеком** или **компьютером**
- …

Унифицированный язык моделирования

- **UML, Unified Modeling Language**
- Язык, использующий графические обозначения для создания модели системы (задачи)
- Одно из направлений использования - **формальное** описание объектных моделей при разработке программ

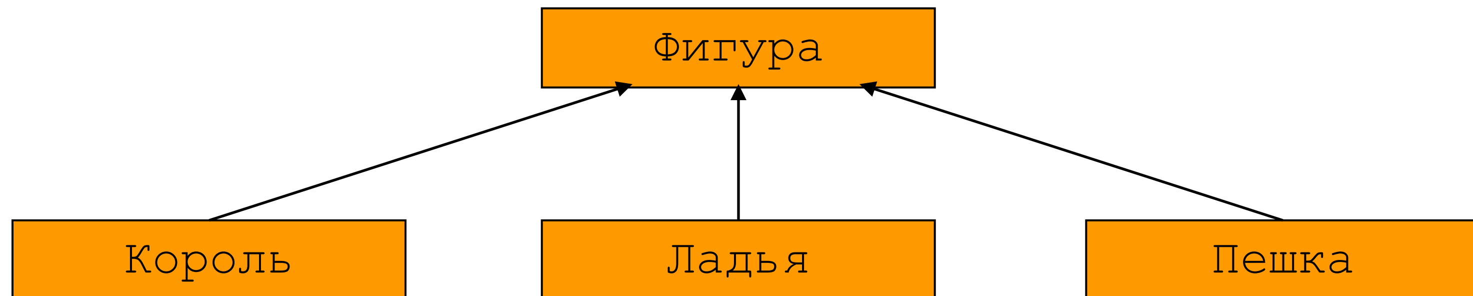
Основные отношения (ассоциации)

- Включение (агрегация, композиция, has a ...) - объект типа A включает в себя один или несколько объектов типа B
  - агрегация - ноль ... много
  - композиция - ноль или один



Основные отношения (ассоциации)

- Разновидность (наследование, генерализация, is a…) - тип A (надтип, суперкласс, базовый класс) является обобщением типа B (подтип, производный класс)
- При этом любой объект типа B принадлежит также и к типу A (is a)
- Говорят также, что тип B наследует тип A



Представление отношений в языке C++

- Включение - тип A включает в себя члены-данные типа B

```
struct Cell
{
  int x, y;
  Cell(int xx, int yy);
};
class Board
{
  Figure* pFigures[8][8]; // или
  map<Cell, Figure*> pFigures;
  // ...
};
```

Представление отношений в языке C++

- Разновидность - используется механизм **наследования (открытого)**
  - **class** King: **public** Figure
  - {
  -   //…
  - };
  - **class** Rook: **public** Figure
  - {
  -   //…
  - };

Свойства наследования - разновидность

- Указатель или ссылка на производный класс может использоваться везде, где требуется указатель или ссылка на базовый класс (**is a…**)
  - **class** Board
  - {
  - Figure* pFigures[8][8];
  - *// pFigures[i][j] может (и должен!)*
  - *// по факту указывать*
  - *// на объект типа «Король» или «Ладья»*
  - *// Не существует «Просто фигуры»*
  - };

Свойства наследования - разновидность

- Объект производного класса содержит в себе все данные и функции базового класса (и, возможно, свои данные и свои функции)
- Права доступа к данным и функциям базового класса зависят от спецификаторов, использованных в базовом классе (**public**, **private**, **protected**)

Свойства наследования - открытые члены

- Все открытые (**public**) члены-данные и члены-функции базового класса являются открытыми и в производном классе
  - **class** Figure
  - {
  - *// ...*
  - **public**:
  - **void** moveTo(**int** x, **int** y);
  - };
  - *// ...*
  - **int** main(**void**)
  - {
  - Rook rook;
  - rook.moveTo(3, 6);
  - }

Свойства наследования - конструкторы

- Однако, производный класс должен иметь собственные конструкторы (из которых должны вызываться конструкторы базового класса)

```
class Figure
{
  Cell cell;
public:
  Figure(int xx, int yy);
};
class Rook: public Figure
{
public:
  Rook(int x, int y);
};
```

Свойства наследования - конструкторы

- Реализация конструкторов:
  - Figure::Figure(**int** xx, **int** yy): cell(xx, yy) {}
  - Rook::Rook(**int** xx, **int** yy): Figure(xx, yy) {}

Свойства наследования - деструкторы

- Производный класс также может иметь собственный деструктор
- Деструктор базового класса вызывается автоматически после окончания работы деструктора производного класса

Свойства наследования - деструкторы


```
class Player {
  Move* moveArray;
public:
  ~Player();
};
class AIPlayer: public Player {
  BeginBase* beginBase;
public:
  ~AIPlayer();
};
Player::~Player() {
  delete[] moveArray;
}
AIPlayer::~AIPlayer() {
  delete beginBase;
}
```

Свойства наследования - закрытые члены

- К закрытым (**private**) членам-данным и членам-функциям базового класса нет доступа в производном классе
  - **void** AIPlayer::reset() {
  - **delete**[] moveArray; *// XXX*
  - }
- Отсутствие доступа **не означает**, что их нет в производном классе!

Свойства наследования - защищенные члены

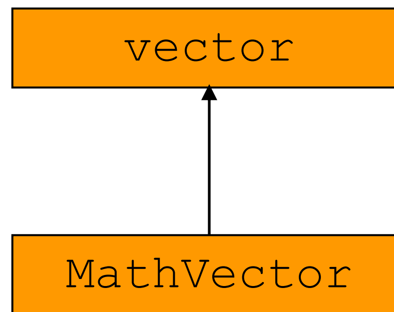
- Защищенные (**protected**) члены-данные и члены-функции базового класса могут использоваться в **методах** как **базового**, так и **производного** класса
  - **class** Figure {
  - **protected**:
  - Cell pos;
  - **bool** isWhite() **const**;
  - };
  - // …
  - **bool** Pawn::canTransform() {
  - **return** (isWhite() && pos.y==8) ||
  - (!isWhite() && pos.y==1);
  - }

Использование наследования - расширение возможностей

- Ранее был реализован класс А с определенным набором возможностей (он может использоваться самостоятельно)
- В нашем объекте требуются все возможности класса А плюс еще какие-то возможности
- Решение - реализовать класс В, унаследовав его от класса А (реализация на основе)

Пример

- Хотим расширить возможности вектора операциями сложения, вычитания и умножения



Для этого

- Создаем класс MathVector, унаследованный от vector<double>
- Определяем необходимые конструкторы
- Определяем необходимые операторы

Определение класса


```
class MathVector: public vector<double> {
public:
  MathVector();
  MathVector(int size);
  MathVector(const MathVector& v);
  MathVector& operator +=(const MathVector& v);
  MathVector& operator -=(const MathVector& v);
  MathVector operator -() const;
  friend MathVector operator +(const MathVector& a,
                               const MathVector& b);
  friend MathVector operator -(const MathVector& a,
                               const MathVector& b);
  friend double operator *(const MathVector& a,
                           const MathVector& b);
  friend ostream& operator <<(ostream& out, const MathVector& v);
};
```

Определения конструкторов

- MathVector::MathVector():
  vector<**double**>() {}

- MathVector::MathVector(**int** size):
  vector<**double**>(size) {}

- MathVector::MathVector
  (**const** MathVector& v):
    vector<**double**>(v) {}

Определение оператора +=


```
MathVector& MathVector::operator +=
    (const MathVector& v)
{
  if (size() != v.size())
    throw SizeException();
  for (int i=0; i<(int)size(); i++)
    (*this)[i] += v[i];
  return *this;
}
```

Определения операторов -, -=


```
MathVector& MathVector::operator -=
  (const MathVector& v)
{
  return (*this += (-v));
}

MathVector MathVector::operator -() const
{
  MathVector v(*this);
  for (int i=0; i<(int)size(); i++)
    v[i] = -v[i];
  return v;
}
```

Определение оператора *


```
double operator *
  (const MathVector& a,
   const MathVector& b)
{
  if (a.size() != b.size())
    throw SizeException();
  double res = 0.0;
  for (int i=0; i<(int)a.size(); i++)
    res += a[i] * b[i];
  return res;
}
```

Определение оператора <<

- ostream& **operator** <<
  (ostream& out, **const** MathVector& v)
- {
- out<<'['<<' ';
- **for** (**int** i=0; i<(**int**)v.size(); i++)
- out<<v[i]<<' ';
- out<<']';
- **return** out;
- }

Тестирование


```
int main(void)
{
  MathVector v(3), w(3);
  v[0] = 1;
  v[1] = 3;
  v[2] = 6;
  w[0] = 2;
  w[1] = -1;
  w[2] = 4;
  cout<<v<<" + "<<w<<" = "<<v+w<<endl;
  cout<<v<<" - "<<w<<" = "<<v-w<<endl;
  cout<<v<<" * "<<w<<" = "<<v*w<<endl;
  return 0;
}
```

Использование наследования - создание иерархий

- Необходимо создать несколько похожих друг на друга объектов, отличающихся в деталях
- Для этого создается базовый класс A, описывающий общие свойства всех объектов
- После чего создаются производные классы B_1, B_2, B_3, ..., описывающие специфические свойства объектов
- Примеры - шахматные фигуры, игрок-человек и игрок-компьютер

Создание иерархий - особенности

- Во многих случаях существуют действия, которые могут выполняться всеми объектами иерархии - но, **по-разному**!
- Шахматные фигуры?
- Игрок-человек и игрок-компьютер?

Полиморфизм

- Каждый объект по-своему выполняет определенные действия - в зависимости от своего типа

Поддержка полиморфизма в C++


```
struct Cell {
  int x, y;
};
struct Move {
  Cell src, dst;
};
class Figure {
protected:
  Cell pos;
public:
  virtual vector<Move>* getMovesArray(const Board& board)=0;
};
class Rook: public Figure {
public:
  virtual vector<Move>* getMovesArray(const Board& board);
};
```

Виртуальные функции

- Если функция объявлена в базовом классе как виртуальная, значит, производный класс может переопределить ее по-своему

Пример переопределения


```
vector<Move>* Rook::getMovesArray
  (const Board& board) {
  vector<Move>* pResult = new vector<Move>(14);
  Move move;
  move.src = pos;
  move.dst = pos;
  for (move.dst.x--; move.dst.x>=1; move.dst.x--)
    pResult->push_back(move);
  for (move.dst.x=pos.x+1; move.dst.x<=8; move.dst.x++)
    pResult->push_back(move);
  //...
  return pResult;
}
```

Механизм действия полиморфизма

- Если имеется указатель (или ссылка) на объект базового класса…
  - **class** Board {
  -   Figure* pFigures[8][8];
  - **public**:
  -   vector<Move>* calcWhiteMoves();
  - };
- …то при вызове виртуальной функции через указатель она будет выбрана в соответствии с реальным типом

Пример


```
vector<Move>* Board::calcWhiteMoves() {
  vector<Move>* pResult = new vector<Move>();
  for (int i=0; i<8; i++) {
    for (int j=0; j<8; j++) {
      if (pFigures[i][j] && pFigures[i][j]->isWhite()) {
        vector<Move>* pFigMove =
          pFigures[i][j]->calcMoves(*this);
        for (int mv=0; mv<pFigMove->size(); mv++)
          pResult->push_back((*pFigMove)[mv]);
        delete[] pFigMove;
      }
    }
  }
  return pResult;
}
```

Виртуальные деструкторы

- Если деструктор объявлен как виртуальный, то при вызове его через указатель на объект базового класса (через **delete**) будет вызван вначале деструктор производного класса, а затем деструктор базового класса

Пример


```
class Player {
  Move* moveArray;
public:
  virtual ~Player();
};
class AIPlayer: public Player {
  BeginBase* beginBase;
public:
  virtual ~AIPlayer();
};
Player::~Player() {
  delete[] moveArray;
}
AIPlayer::~AIPlayer() {
  delete beginBase;
}
```

Пример

- **class** Game {
-   Player* whitePlayer;
-   Player* blackPlayer;
- **public**:
-   **virtual** ~Game();
-   *// …*
- };
- Game::~Game() {
-   **delete** whitePlayer;
-   **delete** blackPlayer;
- }

Динамическое преобразование типов

- Часто требуется определить, на какой именно объект направлен данный указатель
- Для этого используется динамическое преобразование типов

Динамическое преобразование типов


```
// ...
Pawn* pPawn =
  dynamic_cast<Pawn*>(pFigures[i][j]);
if (pPawn) {
  if (pPawn->canTransform()) {
    // ...
  }
}
```

Другие виды наследования в C++

- Закрытое наследование
  - **class** Derived: **private** Base { … };
- Защищенное наследование
  - **class** Derived: **protected** Base { … };
- Множественное наследование
  - class Derived: public Base1, public Base2 { … };
- Применяются существенно реже открытого наследования

Итоги

- Рассмотрены понятия объектной модели и отношений между объектами
- Рассмотрен механизм работы наследования в языке C++
- Рассмотрен механизм переопределения функций (полиморфизма) в языке C++
- Рассмотрено динамическое преобразование типов