

# Виртуални функции

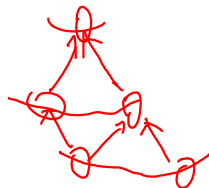
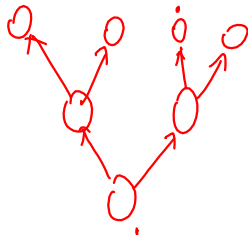
Трифон Трифонов

Обектно-ориентирано програмиране,  
спец. Компютърни науки, 1 поток,  
спец. Софтуерно инженерство,  
2016/17 г.

18–25 май 2017 г.

# Статично свързване

- Как компилаторът избира кой метод или коя функция да бъде извикана?



# Статично свързване

- Как компилаторът избира кой метод или коя функция да бъде извикана?
- Прави се сравнение между формални и фактически параметри и се избира най-точното съвпадение

# Статично свързване

- Как компилаторът избира кой метод или коя функция да бъде извикана?
- Прави се сравнение между формални и фактически параметри и се избира най-точното съвпадение
  - в случай, че има няколко най-точни, грешка за нееднозначност

# Статично свързване

- Как компилаторът избира кой метод или коя функция да бъде извикана?
- Прави се сравнение между формални и фактически параметри и се избира най-точното съвпадение
  - в случай, че има няколко най-точни, грешка за нееднозначност
- Методът, който ще се извика се предопределя **по време на компилация** и при всяко изпълнение е един и същ

# Статично свързване — примери

- **Пример:**

```
Player* pp = new Hero("Гандалф", 45, 15);
```

# Статично свързване — примери

- **Пример:**

```
Player* pp = new Hero("Гандалф", 45, 15);
```

- Кой метод ще се извика при `pp->print()`

# Статично свързване — примери

- **Пример:**

```
Player* pp = new Hero("Гандалф", 45, 15);
```

- Кой метод ще се извика при `pp->print()`
  - `Player::print` или `Hero::print`?



# Статично свързване — примери

- **Пример:**

```
Player* pp = new Hero("Гандалф", 45, 15);
```

- Кой метод ще се извика при `pp->print()`
  - `Player::print` или `Hero::print`?
  - **подсказка:** кой отговор може да се определи със сигурност по време на компилация?

# Статично свързване — примери

- **Пример:**

```
Player* pp = new Hero("Гандалф", 45, 15);
```

- Кой метод ще се извика при `pp->print()`
  - `Player::print` или `Hero::print`?
  - **подсказка:** кой отговор може да се определи със сигурност по време на компилация?
- Свързването по време на компилация нариаме **СТАТИЧНО** или **ранно** (early binding)

# Статично свързване — примери

- **Пример:**

```
Player* pp = new Hero("Гандалф", 45, 15);
```

- Кой метод ще се извика при `pp->print()`
  - `Player::print` или `Hero::print`?
  - **подсказка:** кой отговор може да се определи със сигурност по време на компилация?
- Свързването по време на компилация нариаме **СТАТИЧНО** или **ранно** (early binding)
- В C++ по подразбиране свързването е статично

# Статично свързване — примери

- **Пример:**

```
Player* pp = new Hero("Гандалф", 45, 15);
```

- Кой метод ще се извика при `pp->print()`
  - `Player::print` или `Hero::print`?
  - **подсказка:** кой отговор може да се определи със сигурност по време на компилация?
- Свързването по време на компилация нариаме **СТАТИЧНО** или **ранно** (early binding)
- В C++ по подразбиране свързването е статично
  - има езици, в които по подразбиране свързването не е статично!

# Статично свързване — примери

- **Пример:**

```
Player* pp = new Hero("Гандалф", 45, 15);
```

- Кой метод ще се извика при `pp->print()`
  - `Player::print` или `Hero::print`?
  - **подсказка:** кой отговор може да се определи със сигурност по време на компилация?
- Свързването по време на компилация нариаме **СТАТИЧНО** или **ранно** (early binding)
- В C++ по подразбиране свързването е статично
  - има езици, в които по подразбиране свързването не е статично!
  - Java, Python, Ruby

## Защо само статично свързване не стига

- **Пример:** създаване на обект от клас, избран от потребителя

```
Player* pp = NULL; char c;  
cin >> c;  
if (c == 'b') pp = new Bot;  
if (c == 'h') pp = new Hero;  
...  
if (pp != NULL) pp->print();
```

## Защо само статично свързване не стига

- **Пример:** създаване на обект от клас, избран от потребителя

```
Player* pp = NULL; char c;  
cin >> c;  
if (c == 'b') pp = new Bot;  
if (c == 'h') pp = new Hero;  
...  
if (pp != NULL) pp->print();
```

- Винаги се извиква `Player::print()`!

## Защо само статично свързване не стига

- **Пример:** създаване на обект от клас, избран от потребителя

```
Player* pp = NULL; char c;  
cin >> c;  
if (c == 'b') pp = new Bot;  
if (c == 'h') pp = new Hero;  
...  
if (pp != NULL) pp->print();
```

- Винаги се извиква `Player::print()`!
- Няма как компилаторът да знае какво ще въведе потребителят, затова “залага на сигурното”



## Защо само статично свързване не стига

- **Пример:** създаване на обект от клас, избран от потребителя

```
Player* pp = NULL; char c;  
cin >> c;  
if (c == 'b') pp = new Bot;  
if (c == 'h') pp = new Hero;  
...  
if (pp != NULL) pp->print();
```

- Винаги се извиква `Player::print()`!
- Няма как компилаторът да знае какво ще въведе потребителят, затова “залага на сигурното”
- Как да извикаме “правилния” метод?

## Решение №1

Проверяваме входа от потребителя при всяка операция

```
Player* pp = NULL; char c;  
cin >> c;  
if (c == 'b') pp = new Bot;  
if (c == 'h') pp = new Hero;  
...  
if (pp != NULL) {  
    if (c == 'b') ((Bot*)pp)->print();  
    if (c == 'h') ((Hero*)pp)->print();  
    ...  
}
```

## Решение №1

Проверяваме входа от потребителя при всяка операция

```
Player* pp = NULL; char c;  
cin >> c;  
if (c == 'b') pp = new Bot;  
if (c == 'h') pp = new Hero;  
...  
if (pp != NULL) {  
    if (c == 'b') ((Bot*)pp)->print();  
    if (c == 'h') ((Hero*)pp)->print();  
    ...  
}
```

### Проблеми:

- Пазим “маркера” на всеки обект, за да го ползваме при всяка операция

## Решение №1

Проверяваме входа от потребителя при всяка операция

```
Player* pp = NULL; char c;  
cin >> c;  
if (c == 'b') pp = new Bot;  
if (c == 'h') pp = new Hero;  
...  
if (pp != NULL) {  
    if (c == 'b') ((Bot*)pp)->print();  
    if (c == 'h') ((Hero*)pp)->print();  
    ...  
}
```

### Проблеми:

- Пазим “маркера” на всеки обект, за да го ползваме при всяка операция
- При добавяне на нов клас, трябва да се правят много промени по кода

## Решение №1

Проверяваме входа от потребителя при всяка операция

```

Player* pp = NULL; char c;
cin >> c;
if (c == 'b') pp = new Bot;
if (c == 'h') pp = new Hero;
...
if (pp != NULL) {
    if (c == 'b') ((Bot*)pp)->print();
    if (c == 'h') ((Hero*)pp)->print();
    ...
}

```

### Проблеми:

- Пазим “маркера” на всеки обект, за да го ползваме при всяка операция
- При добавяне на нов клас, трябва да се правят много промени по кода
- Много повторение на код

## Решение №1

Проверяваме входа от потребителя при всяка операция

```
Player* pp = NULL; char c;
cin >> c;
if (c == 'b') pp = new Bot;
if (c == 'h') pp = new Hero;
...
if (pp != NULL) {
    if (c == 'b') ((Bot*)pp)->print();
    if (c == 'h') ((Hero*)pp)->print();
    ...
}
```

## Проблеми:

- Пазим “маркера” на всеки обект, за да го ползваме при всяка операция
- При добавяне на нов клас, трябва да се правят много промени по кода
- Много повторение на код
- Грозно е

## Решение №2

```
const int PLAYER = 0, HERO = 1, BOT = 2;
struct SmartPlayer {
    Player* player;
    int type;
    void print() const {
        if (type == PLAYER) player->print();
        if (type == HERO) ((Hero const*)player)->print();
        if (type == BOT) ((Bot const*)player)->print();
    }
};
SmartPlayer pp = { NULL, PLAYER }; char c;
cin >> c;
if (c == 'h') { pp.player = new Hero; pp.type = HERO; }
if (c == 'b') { pp.player = new Bot; pp.type = BOT; }
pp.print();
```

## Решение №2

```

const int PLAYER = 0, HERO = 1, BOT = 2;
struct SmartPlayer {
    Player* player;
    int type;
    void print() const {
        if (type == PLAYER) player->print();
        if (type == HERO) ((Hero const*)player)->print();
        if (type == BOT) ((Bot const*)player)->print();
    }
};
SmartPlayer pp = { NULL, PLAYER }; char c;
cin >> c;
if (c == 'h') { pp.player = new Hero; pp.type = HERO; }
if (c == 'b') { pp.player = new Bot; pp.type = BOT; }
pp.print();

```

**Проблеми:**

- Копиране на код



## Решение №2

```

const int PLAYER = 0, HERO = 1, BOT = 2;
struct SmartPlayer {
    Player* player;
    int type;
    void print() const {
        if (type == PLAYER) player->print();
        if (type == HERO) ((Hero const*)player)->print();
        if (type == BOT) ((Bot const*)player)->print();
    }
};
SmartPlayer pp = { NULL, PLAYER }; char c;
cin >> c;
if (c == 'h') { pp.player = new Hero; pp.type = HERO; }
if (c == 'b') { pp.player = new Bot; pp.type = BOT; }
pp.print();

```

**Проблеми:**

- Копиране на код
- SmartPlayer трябва да знае за всички наследници на Player

# Какво всъщност ни трябва?

- И при двете решения:

# Какво всъщност ни трябва?

- И при двете решения:
  - Програмата трябва да “помни” допълнителни неща

# Какво всъщност ни трябва?

- И при двете решения:
  - Програмата трябва да “помни” допълнителни неща
  - Не можем да разширяваме лесно: промяна при всеки нов наследник на `Player`

# Какво всъщност ни трябва?

- И при двете решения:
  - Програмата трябва да “помни” допълнителни неща
  - Не можем да разширяваме лесно: промяна при всеки нов наследник на Player
- Какво всъщност ни трябва?

# Какво всъщност ни трябва?

- И при двете решения:
  - Програмата трябва да “помни” допълнителни неща
  - Не можем да разширяваме лесно: промяна при всеки нов наследник на Player
- Какво всъщност ни трябва?
- Всеки обект да има “етикет” от кой клас е

# Какво всъщност ни трябва?

- И при двете решения:
  - Програмата трябва да “помни” допълнителни неща
  - Не можем да разширяваме лесно: промяна при всеки нов наследник на Player
- Какво всъщност ни трябва?
- Всеки обект да има “етикет” от кой клас е
- При извикване на операция, етикетът се използва, за да се определи коя предефинирана версия на метода да се използва

# Динамично свързване

- Когато методът, който ще се извика, се определя **по време на изпълнение**, свързването се нарича **динамично** или **късно** (late binding)



# Динамично свързване

- Когато методът, който ще се извика, се определя **по време на изпълнение**, свързването се нарича **динамично** или **късно** (late binding)
- Извиква се методът на този клас, от който е обектът, към който сочи указателя

# Динамично свързване

- Когато методът, който ще се извика, се определя **по време на изпълнение**, свързването се нарича **динамично** или **късно** (late binding)
- Извиква се методът на този клас, от който е обектът, към който сочи указателя
- Обектът може да е от базовия клас или от клас-наследник

# Виртуални член-функции

- C++ поддържа едновременно статично и динамично свързване

# Виртуални член-функции

- C++ поддържа едновременно статично и динамично свързване
- Можем да указваме какво е свързването за всяка отделна член-функция

# Виртуални член-функции

- C++ поддържа едновременно статично и динамично свързване
- Можем да указваме какво е свързването за всяка отделна член-функция
- Функция, за която свързването е динамично, се нарича **виртуална**

# Виртуални член-функции

- C++ поддържа едновременно статично и динамично свързване
- Можем да указваме какво е свързването са всяка отделна член-функция
- Функция, за които свързването е динамично, се нарича **виртуална**
- **virtual** <сигнатура>;

# Виртуални член-функции

- C++ поддържа едновременно статично и динамично свързване
- Можем да указваме какво е свързването са всяка отделна член-функция
- Функция, за които свързването е динамично, се нарича **виртуална**
- **virtual** <сигнатура>;
- Класове с виртуални функции се наричат **полиморфни**

# Виртуални член-функции

- C++ поддържа едновременно статично и динамично свързване
- Можем да указваме какво е свързването са всяка отделна член-функция
- Функция, за която свързването е динамично, се нарича **виртуална**
- **virtual** <сигнатура>;
- Класове с виртуални функции се наричат **полиморфни**
- **Примери:**

```
class Player { ... virtual void print() const; ... };
Player p, *pp = &p;
pp->print(); // Player::print()
Hero h; pp = &h;
pp->print(); // Hero::print()
Bot b; pp = &b;
pp->print(); // Bot::print()
```



# Особености на виртуалните функции

- Само член-функции могат да бъдат виртуални

# Особености на виртуалните функции

- Само член-функции могат да бъдат виртуални
- Конструкторите не могат да са виртуални

# Особености на виртуалните функции

- Само член-функции могат да бъдат виртуални
- Конструкторите не могат да са виртуални
  - те се извикват “преди” обектът да е създаден

# Особености на виртуалните функции

- Само член-функции могат да бъдат виртуални
- Конструкторите не могат да са виртуални
  - те се извикват “преди” обектът да е създаден
- Наследяващата член-функция в производния клас **трябва** да е със същата сигнатура

# Особености на виртуалните функции

- Само член-функции могат да бъдат виртуални
- Конструкторите не могат да са виртуални
  - те се извикват “преди” обектът да е създаден
- Наследяващата член-функция в производния клас **трябва** да е със същата сигнатура
  - ако сигнатурата е различна, за C++ това е друга функция

# Особености на виртуалните функции

- Само член-функции могат да бъдат виртуални
- Конструкторите не могат да са виртуални
  - те се извикват “преди” обектът да е създаден
- Наследяващата член-функция в производния клас **трябва** да е със същата сигнатура
  - ако сигнатурата е различна, за C++ това е друга функция
- наследяващите функции са автоматично виртуални и **virtual** може да се пропусне

# Особености на виртуалните функции

- Само член-функции могат да бъдат виртуални
- Конструкторите не могат да са виртуални
  - те се извикват “преди” обектът да е създаден
- Наследяващата член-функция в производния клас **трябва** да е със същата сигнатура
  - ако сигнатурата е различна, за C++ това е друга функция
- наследяващите функции са автоматично виртуални и **virtual** може да се пропусне
- **virtual** се пише само пред декларацията, не пред дефиницията

# Видимост на виртуални функции

**Общо правило:** Видимостта на една виртуална функция се определя от видимостта ѝ в класа на указателя или псевдонима, през който се извиква.



# Видимост на виртуални функции

**Общо правило:** Видимостта на една виртуална функция се определя от видимостта ѝ в класа на указателя или псевдонима, през който се извиква.

**Въпроси:**

- може ли `private` виртуална функция да се извика извън класа?

# Видимост на виртуални функции

**Общо правило:** Видимостта на една виртуална функция се определя от видимостта ѝ в класа на указателя или псевдонима, през който се извиква.

## Въпроси:

- може ли `private` виртуална функция да се извика извън класа?
- какъв трябва да е спецификаторът за достъп на виртуална функция в основния клас?

# Видимост на виртуални функции

**Общо правило:** Видимостта на една виртуална функция се определя от видимостта ѝ в класа на указателя или псевдонима, през който се извиква.

## Въпроси:

- може ли `private` виртуална функция да се извика извън класа?
- какъв трябва да е спецификаторът за достъп на виртуална функция в основния клас?
- какъв трябва да е спецификаторът за достъп при наследяването на основен клас с виртуална функция?

# Извикване на виртуални функции

Какво се случва, ако виртуална функция се извика:

- чрез обект

# Извикване на виртуални функции

Какво се случва, ако виртуална функция се извика:

- чрез обект
  - `Player p; p.print();`

# Извикване на виртуални функции

Какво се случва, ако виртуална функция се извика:

- чрез обект
  - `Player p; p.print();`
  - статично свързване, понеже типът се знае предварително

# Извикване на виртуални функции

Какво се случва, ако виртуална функция се извика:

- чрез обект
  - `Player p; p.print();`
  - статично свързване, понеже типът се знае предварително
- чрез указател

# Извикване на виртуални функции

Какво се случва, ако виртуална функция се извика:

- чрез обект
  - `Player p; p.print();`
  - статично свързване, понеже типът се знае предварително
- чрез указател
  - `Player* pp = &h; pp->print();`



# Извикване на виртуални функции

Какво се случва, ако виртуална функция се извика:

- чрез обект
  - `Player p; p.print();`
  - статично свързване, понеже типът се знае предварително
- чрез указател
  - `Player* pp = &h; pp->print();`
  - динамично свързване

# Извикване на виртуални функции

Какво се случва, ако виртуална функция се извика:

- чрез обект
  - `Player p; p.print();`
  - статично свързване, понеже типът се знае предварително
- чрез указател
  - `Player* pp = &h; pp->print();`
  - динамично свързване
- чрез псевдоним

# Извикване на виртуални функции

Какво се случва, ако виртуална функция се извика:

- чрез обект
  - `Player p; p.print();`
  - статично свързване, понеже типът се знае предварително
- чрез указател
  - `Player* pp = &h; pp->print();`
  - динамично свързване
- чрез псевдоним
  - `Player& ap = b; ap.print();`

# Извикване на виртуални функции

Какво се случва, ако виртуална функция се извика:

- чрез обект
  - `Player p; p.print();`
  - статично свързване, понеже типът се знае предварително
- чрез указател
  - `Player* pp = &h; pp->print();`
  - динамично свързване
- чрез псевдоним
  - `Player& ap = b; ap.print();`
  - еквивалентно на указател, динамично свързване

# Извикване на виртуални функции

Какво се случва, ако виртуална функция се извика:

- чрез обект
  - `Player p; p.print();`
  - статично свързване, понеже типът се знае предварително
- чрез указател
  - `Player* pp = &h; pp->print();`
  - динамично свързване
- чрез псевдоним
  - `Player& ap = b; ap.print();`
  - еквивалентно на указател, динамично свързване
- чрез указване на област

# Извикване на виртуални функции

Какво се случва, ако виртуална функция се извика:

- чрез обект
  - `Player p; p.print();`
  - статично свързване, понеже типът се знае предварително
- чрез указател
  - `Player* pp = &h; pp->print();`
  - динамично свързване
- чрез псевдоним
  - `Player& ap = b; ap.print();`
  - еквивалентно на указател, динамично свързване
- чрез указване на област
  - `Player::print();`

# Извикване на виртуални функции

Какво се случва, ако виртуална функция се извика:

- чрез обект
  - `Player p; p.print();`
  - статично свързване, понеже типът се знае предварително
- чрез указател
  - `Player* pp = &h; pp->print();`
  - динамично свързване
- чрез псевдоним
  - `Player& ap = b; ap.print();`
  - еквивалентно на указател, динамично свързване
- чрез указване на област
  - `Player::print();`
  - статично свързване, указали сме кой метод да се извика

# Косвено извикване на виртуални функции

Какво се случва, ако виртуална функция се извика:

- от член-функция



# Косвено извикване на виртуални функции

Какво се случва, ако виртуална функция се извика:

- от член-функция
  - `void Player::f() { ... print(); ... }`

# Косвено извикване на виртуални функции

Какво се случва, ако виртуална функция се извика:

- от член-функция
  - `void Player::f() { ... print(); ... }`
  - еквивалентно на извикване през `this`, динамично свързване!

# Косвено извикване на виртуални функции

Какво се случва, ако виртуална функция се извика:

- от член-функция
  - `void Player::f() { ... print(); ... }`
  - еквивалентно на извикване през `this`, динамично свързване!
- от конструктор на основен клас

# Косвено извикване на виртуални функции

Какво се случва, ако виртуална функция се извика:

- от член-функция
  - `void Player::f() { ... print(); ... }`
  - еквивалентно на извикване през `this`, динамично свързване!
- от конструктор на основен клас
  - `Player::Player() { ... print(); ... }`

# Косвено извикване на виртуални функции

Какво се случва, ако виртуална функция се извика:

- от член-функция
  - `void Player::f() { ... print(); ... }`
  - еквивалентно на извикване през `this`, динамично свързване!
- от конструктор на основен клас
  - `Player::Player() { ... print(); ... }`
  - статично свързване, обектът от производен клас още не е построен!

# Косвено извикване на виртуални функции

Какво се случва, ако виртуална функция се извика:

- от член-функция
  - `void Player::f() { ... print(); ... }`
  - еквивалентно на извикване през `this`, динамично свързване!
- от конструктор на основен клас
  - `Player::Player() { ... print(); ... }`
  - статично свързване, обектът от производен клас още не е построен!
- от деструктор на основен клас

# Косвено извикване на виртуални функции

Какво се случва, ако виртуална функция се извика:

- от член-функция
  - `void Player::f() { ... print(); ... }`
  - еквивалентно на извикване през `this`, динамично свързване!
- от конструктор на основен клас
  - `Player::Player() { ... print(); ... }`
  - статично свързване, обектът от производен клас още не е построен!
- от деструктор на основен клас
  - `Player::~~Player() { ... print(); ... }`

# Косвено извикване на виртуални функции

Какво се случва, ако виртуална функция се извика:

- от член-функция
  - `void Player::f() { ... print(); ... }`
  - еквивалентно на извикване през `this`, динамично свързване!
- от конструктор на основен клас
  - `Player::Player() { ... print(); ... }`
  - статично свързване, обектът от производен клас още не е построен!
- от деструктор на основен клас
  - `Player::~~Player() { ... print(); ... }`
  - статично свързване, обектът от производния клас вече е разрушен!



## Косвено динамично свързване

```
void Player::prettyPrint() const {  
    cout << „----- [ Player Info ] “-----;  
    print();  
    cout << „-----“-----;  
}
```

## Косвено динамично свързване

```
void Player::prettyPrint() const {
    cout << „----- [ Player Info ] “-----;
    print();
    cout << „-----“;
}
```

Дадено е Hero h; какво ще изведат:

- `Player* pp = &h; pp->prettyPrint();`

## Косвено динамично свързване

```
void Player::prettyPrint() const {
    cout << „----- [ Player Info ] “-----;
    print();
    cout << „-----“-----;
}
```

Дадено е Hero h; какво ще изведат:

- `Player* pp = &h; pp->prettyPrint();`
- `Player& ap = h; ap.prettyPrint();`

## Косвено динамично свързване

```
void Player::prettyPrint() const {
    cout << „----- [ Player Info ] “-----;
    print();
    cout << „-----“;
}
```

Дадено е Hero h; какво ще изведат:

- `Player* pp = &h; pp->prettyPrint();`
- `Player& ap = h; ap.prettyPrint();`
- `Player p = h; p.prettyPrint();`

## Косвено динамично свързване

```
void Player::prettyPrint() const {
    cout << „----- [ Player Info ] “-----;
    print();
    cout << „-----“;
}
```

Дадено е Hero h; какво ще изведат:

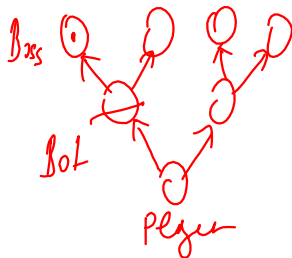
- `Player* pp = &h; pp->prettyPrint();`
- `Player& ap = h; ap.prettyPrint();`
- `Player p = h; p.prettyPrint();`
- **Извод:** “Виртуалността” **се разпростира автоматично** и сред член-функциите, които извикват член-функции

## Определяне на “правилната” функция

- Не е задължително виртуална функция да има нова реализация във всеки производен клас

# Определяне на “правилната” функция

- Не е задължително виртуална функция да има нова реализация във всеки производен клас
- Избира се виртуалната функция, която е “най-близо” до класа, от който е обекта



## Определяне на “правилната” функция

- Не е задължително виртуална функция да има нова реализация във всеки производен клас
- Избира се виртуалната функция, която е “най-близко” до класа, от който е обекта
  - избраната функция се нарича **final override**



## Определяне на “правилната” функция

- Не е задължително виртуална функция да има нова реализация във всеки производен клас
- Избира се виртуалната функция, която е “най-близко” до класа, от който е обекта
  - избраната функция се нарича **final override**
  - търсенето е отдолу-нагоре по йерархията

## Определяне на “правилната” функция

- Не е задължително виртуална функция да има нова реализация във всеки производен клас
- Избира се виртуалната функция, която е “най-близко” до класа, от който е обекта
  - избраната функция се нарича **final override**
  - търсенето е отдолу-нагоре по йерархията
- При множествено наследяване могат да се получат нееднозначности:

# Определяне на “правилната” функция

- Не е задължително виртуална функция да има нова реализация във всеки производен клас
- Избира се виртуалната функция, която е “най-близо” до класа, от който е обекта
  - избраната функция се нарича **final overrider**
  - търсенето е отдолу-нагоре по йерархията
- При множествено наследяване могат да се получат нееднозначности:
  - ако Boss не дефинираше `print()`, какво щеше да изведе следният код?  
`Boss b; Player* pp = &b; pp->print();`



# Определяне на “правилната” функция

- Не е задължително виртуална функция да има нова реализация във всеки производен клас
- Избира се виртуалната функция, която е “най-близо” до класа, от който е обекта
  - избраната функция се нарича **final overrider**
  - търсенето е отдолу-нагоре по йерархията
- При множествено наследяване могат да се получат нееднозначности:
  - ако Boss не дефинираше `print()`, какво щеше да изведе следният код?  
`Boss b; Player* pp = &b; pp->print();`
  - нееднозначността се вижда още по време на компилация!

# Механизъм на виртуалните таблици

- Как по време на изпълнение програмата избира кой метод да се изпълни?

# Механизъм на виртуалните таблици

- Как по време на изпълнение програмата избира кой метод да се изпълни?
- За всеки клас с виртуални функции се създава таблица с указатели към тях (**виртуална таблица**)

# Механизъм на виртуалните таблици

- Как по време на изпълнение програмата избира кой метод да се изпълни?
- За всеки клас с виртуални функции се създава таблица с указатели към тях (**виртуална таблица**)
- В началото на всеки обект с виртуални функции се поставя указател **към виртуална таблица**

# Механизъм на виртуалните таблици

- Как по време на изпълнение програмата избира кой метод да се изпълни?
- За всеки клас с виртуални функции се създава таблица с указатели към тях (**виртуална таблица**)
- В началото на всеки обект с виртуални функции се поставя указател **към виртуална таблица**
- При динамично свързване, се случва следното:



# Механизъм на виртуалните таблици

- Как по време на изпълнение програмата избира кой метод да се изпълни?
- За всеки клас с виртуални функции се създава таблица с указатели към тях (**виртуална таблица**)
- В началото на всеки обект с виртуални функции се поставя указател **към виртуална таблица**
- При динамично свързване, се случва следното:
  - компилаторът изчислява номера  $i$  на извикваната виртуалната функция

# Механизъм на виртуалните таблици

- Как по време на изпълнение програмата избира кой метод да се изпълни?
- За всеки клас с виртуални функции се създава таблица с указатели към тях (**виртуална таблица**)
- В началото на всеки обект с виртуални функции се поставя указател **към виртуална таблица**
- При динамично свързване, се случва следното:
  - компилаторът изчислява номера  $i$  на извикваната виртуалната функция
  - компилаторът генерира код, който

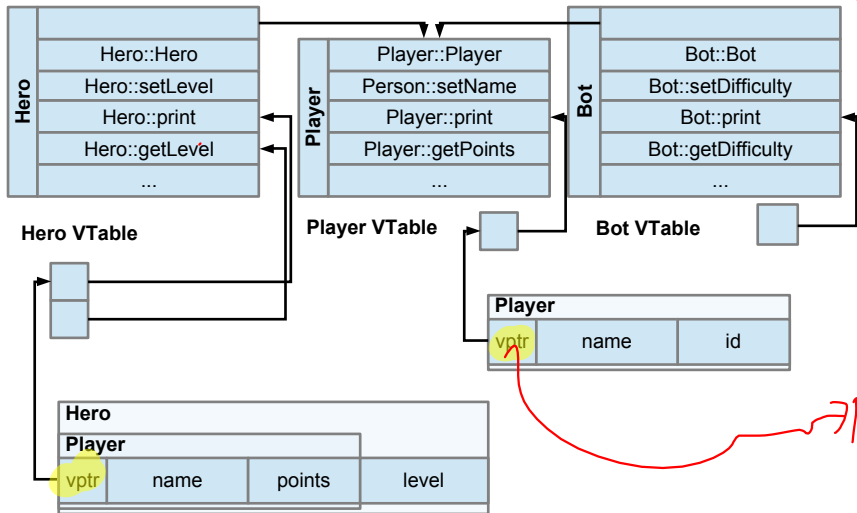
# Механизъм на виртуалните таблици

- Как по време на изпълнение програмата избира кой метод да се изпълни?
- За всеки клас с виртуални функции се създава таблица с указатели към тях (**виртуална таблица**)
- В началото на всеки обект с виртуални функции се поставя указател **към виртуална таблица**
- При динамично свързване, се случва следното:
  - компилаторът изчислява номера  $i$  на извикваната виртуалната функция
  - компилаторът генерира код, който
    - намира  $i$ -тия указател във виртуалната таблица на обекта

# Механизъм на виртуалните таблици

- Как по време на изпълнение програмата избира кой метод да се изпълни?
- За всеки клас с виртуални функции се създава таблица с указатели към тях (**виртуална таблица**)
- В началото на всеки обект с виртуални функции се поставя указател **към виртуална таблица**
- При динамично свързване, се случва следното:
  - компилаторът изчислява номера  $i$  на извикваната виртуалната функция
  - компилаторът генерира код, който
    - намира  $i$ -тия указател във виртуалната таблица на обекта
    - извиква функцията, която се сочи от този указател

## Представяне в паметта



# Виртуални таблици и множествено наследяване

- При множествено наследяване се създава по една виртуална таблица за всеки основен клас

# Виртуални таблици и множествено наследяване

- При множествено наследяване се създава по една виртуална таблица за всеки основен клас
- Във всеки обект има по един указател към виртуална таблица за всеки основен клас

# Виртуални таблици и множествено наследяване

- При множествено наследяване се създава по една виртуална таблица за всеки основен клас
- Във всеки обект има по един указател към виртуална таблица за всеки основен клас
- Ако имаме и виртуално наследяване, представянето става още по-сложно



# Виртуални таблици и множествено наследяване

- При множествено наследяване се създава по една виртуална таблица за всеки основен клас
- Във всеки обект има по един указател към виртуална таблица за всеки основен клас
- Ако имаме и виртуално наследяване, представянето става още по-сложно
- За щастие, в рамките на този курс няма да пишем компилатор за C++

# Типова информация по време на изпълнение (RTTI)

- В C++ има механизъм за намиране на типа на даден обект по време на изпълнение

# Типова информация по време на изпълнение (RTTI)

- В C++ има механизъм за намиране на типа на даден обект по време на изпълнение
- `typeid(<израз>)`

# Типова информация по време на изпълнение (RTTI)

- В C++ има механизъм за намиране на типа на даден обект по време на изпълнение
- `typeid(<израз>)`
- връща обект от тип `type_info`

# Типова информация по време на изпълнение (RTTI)

- В C++ има механизъм за намиране на типа на даден обект по време на изпълнение
- `typeid(<израз>)`
- връща обект от тип `type_info`
  - ако <израз> е lvalue от полиморфен клас, връща динамичния тип на <израз>

# Типова информация по време на изпълнение (RTTI)

- В C++ има механизъм за намиране на типа на даден обект по време на изпълнение
- `typeid(<израз>)`
- връща обект от тип `type_info`
  - ако <израз> е lvalue от полиморфен клас, връща динамичния тип на <израз>
  - иначе, връща статичния тип на <израз>

# Типова информация по време на изпълнение (RTTI)

- В C++ има механизъм за намиране на типа на даден обект по време на изпълнение
- `typeid(<израз>)`
- връща обект от тип `type_info`
  - ако <израз> е lvalue от полиморфен клас, връща динамичния тип на <израз>
  - иначе, връща статичния тип на <израз>
- можем да получим името на даден тип:  

```
cout << typeid(pp).name() << ' ' << typeid(*pp).name();
```

# Типова информация по време на изпълнение (RTTI)

- В C++ има механизъм за намиране на типа на даден обект по време на изпълнение
- `typeid(<израз>)`
- връща обект от тип `type_info`
  - ако <израз> е lvalue от полиморфен клас, връща динамичния тип на <израз>
  - иначе, връща статичния тип на <израз>
- можем да получим името на даден тип:  

```
cout << typeid(pp).name() << ' ' << typeid(*pp).name();
```
- два типа могат да се сравняват с `==` или `!=`  

```
typeid(p) != typeid(s), typeid(*pp) == typeid(HeroHero)
```



# Виртуални деструктори

- `Player* pp = new Bot; ... delete pp;`

# Виртуални деструктори

- `Player* pp = new Bot; ... delete pp;`
- Кой деструктор ще се извика?

# Виртуални деструктори

- `Player* pp = new Bot; ... delete pp;`
- **Кой деструктор ще се извика?**
  - статично свързване, `~Player()`

# Виртуални деструктори

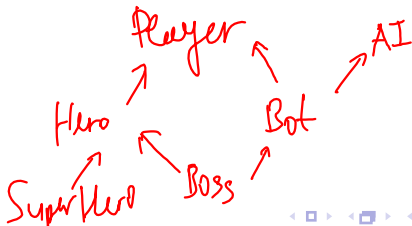
- `Player* pp = new Bot; ... delete pp;`
- **Кой деструктор ще се извика?**
  - статично свързване, `~Player()`
  - динамичната памет на `Bot` остава неосвободена, **изтича памет!**

# Виртуални деструктори

- `Player* pp = new Bot; ... delete pp;`
- **Кой деструктор ще се извика?**
  - статично свързване, `~Player()`
  - динамичната памет на `Bot` остава неосвободена, **изтича памет!**
- Искаме да се вика правилният деструктор!

# Виртуални деструктори

- `Player* pp = new Bot; ... delete pp;`
- **Кой деструктор ще се извика?**
  - статично свързване, `~Player()`
  - динамичната памет на `Bot` остава неосвободена, **изтича памет!**
- Искаме да се вика правилният деструктор!
- Можем да декларираме **деструктора като виртуален**



# Виртуални деструктори

- `Player* pp = new Bot; ... delete pp;`
- **Кой деструктор ще се извика?**
  - статично свързване, `~Player()`
  - динамичната памет на `Bot` остава неосвободена, **изтича памет!**
- Искаме да се вика правилният деструктор!
- Можем да декларираме **деструктора като виртуален**
- Тогава свързването е динамично и ще се извика деструкторът на `Bot`