Виртуални функции

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
- Важно: Методът, който ще се извика се предопределя по време на компилация и при всяко изпълнение е един и същ

Статично свързване

- Пример:
  Player* pp = new Hero(„Gandalf“, 45, 15);
- Кой метод ще извика pp->print()?
  - Hero::print или Player::print?
  - подсказка: кое от двете може да се определи със сигурност по време на компилация, а не по време на изпълнение?
- Свързването по време на компилация наричаме статично или ранно (early binding)
- В C++ по подразбиране свързването е статично
  - има езици, в които по подразбиране не е статично!

Защо само статично свързване не стига

- Пример:
```
Player* pp = NULL; char c;
cin >> c;
if (c == 'b') pp = new Bot;
if (c == 'h') pp = new Hero;
...
if (pp != NULL) pp->print(); // Player::print
```
- Няма как компилаторът да знае какво ще въведе потребителят, затова се „залага на сигурното“
- Как можем да направим така, че да се извика „този метод, който трябва“?

Защо само статично свързване не стига

Решение №1:

```
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();
}
```

Защо само статично свързване не стига

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();
```

Защо само статично свързване не стига

- И двете решения не са напълно добри, понеже изискват програмата да „помни“ допълнителни неща...
- Колко хубаво би било, ако можеше със създаването си обектът да има „етикет“ и по време на изпълнение етикетът се използва, за да се определи кой метод да се извика

Динамично свързване

- При динамичното (късно) свързване (late binding) методът, който ще се извика, се определя по време на изпълнение
- извиква се методът на този клас, от който всъщност е даденият обект
  - независимо че указателят може да е дефиниран към базов клас

Виртуални член-функции

- В C++ динамичното свързване може да се включи за всяка отделна член-функция, като тя се обяви като **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
- Конструкторите не могат да са виртуални
  - те се извикват „преди“ обектът да е създаден
- Статичните член-функции не могат да са виртуални
  - те могат да се извикват без обект
- Наследяващата член-функция в производния клас трябва да е със същата сигнатура
  - ако сигнатурата е различна, това е друга функция
  - наследяващите функции са автоматично виртуални и запазената дума **virtual** може да се пропусне
- **virtual** се пише само пред декларацията, не пред дефиницията

Видимост на виртуални функции

- Правило:
  Видимостта на една виртуална функция се определя от видимостта ѝ в класа на обекта, (указателя, псевдонима), през който се извиква
- Това означава ли, че:
  - може private виртуална функция да се извика извън класа?
  - няма смисъл виртуална функция в основния клас да е private или protected, понеже няма как да се извика?
  - основният клас, който съдържа виртуалната функция, трябва да е наследен с public?

Извикване на виртуални функции

Какво става 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

Извикване на виртуални функции

Какво става 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 << „----------------------------------------“;
}
```

Ако Hero h; какво ще изведат:

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

Косвено динамично свързване

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

Ако Hero h; какво ще изведат:

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

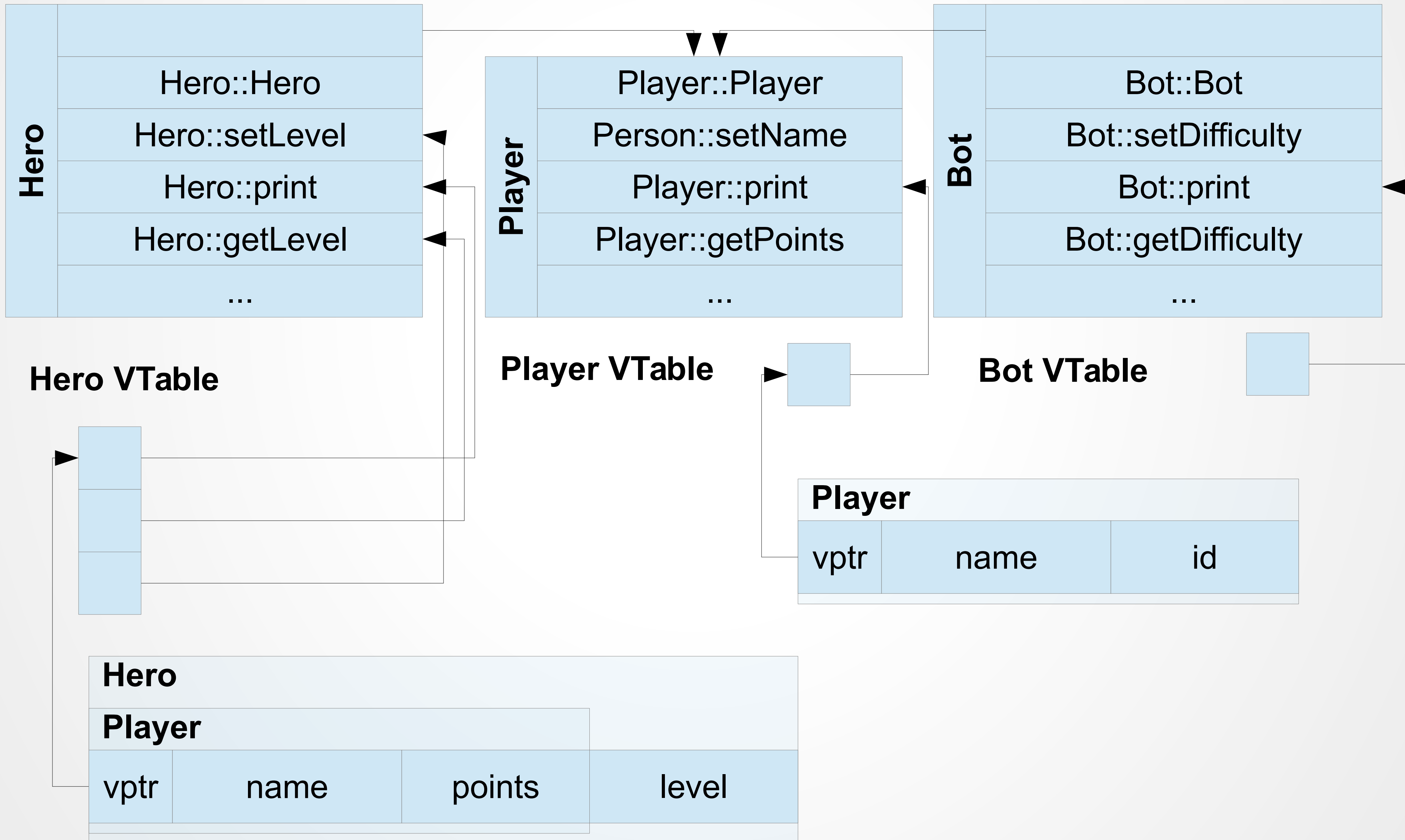
Коя реална функция ще се извика?

- 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
  - final overrider
  - търси се отдолу-нагоре
- 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 програмата решава по време на изпълнение кой метод да се изпълни?
- За всеки клас с виртуални функции се създава таблица с указатели към тях (**виртуална таблица**)
- За всеки обект с виртуални функции в началото 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

Механизъм на виртуалните таблици



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
- 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>
- можете да получите 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(Student)

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, деструкторът на Player
  - динамичната памет на Bot остава неосвободена (изтичане на памет)!
- Искаме да се вика правилният деструктор!
- Можем да декларираме деструктора като виртуален
- Тогава свързването е динамично и ще се извика деструкторът на Bot