Dědičnost

Karel Richta a kol.

katedra počítačů FEL ČVUT v Praze

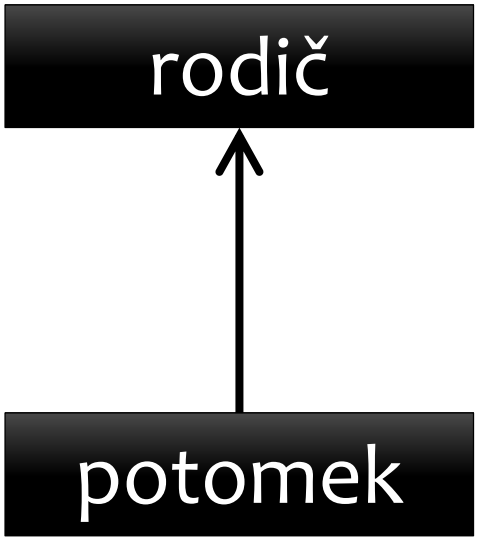
Programování v C++, B6B36PJC

11/2019, Lekce 8

https://cw.fel.cvut.cz/wiki/courses/b6b36pjc/start

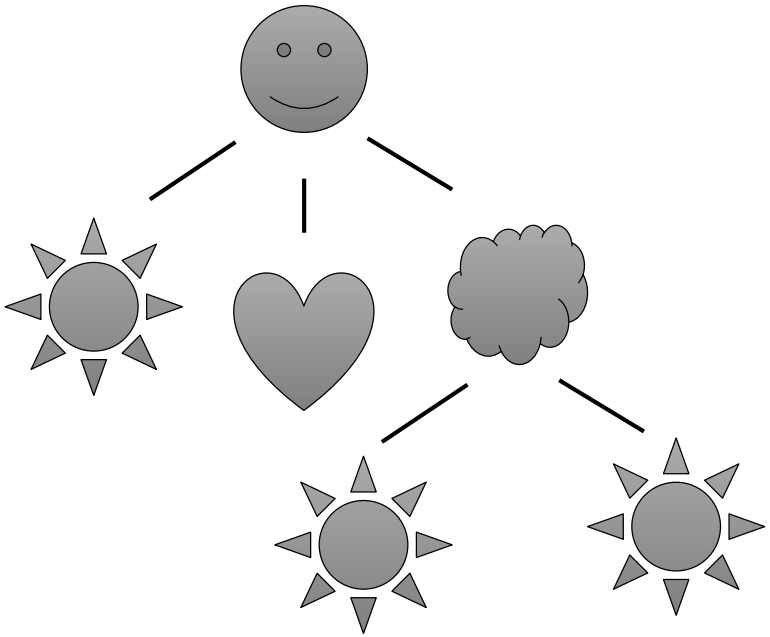
Dědičnost – co to je?

- **Dědění** je návrh nových tříd na základě tříd již existujících.
- Třídu, ze které dědíme, nazýváme **rodič** nebo **předek**. Nově vzniklou třídu nazýváme **potomek**.



Dědičnost – k čemu to slouží?

- Dědění používáme ke konstrukci **heterogenních hierarchií** objektů.



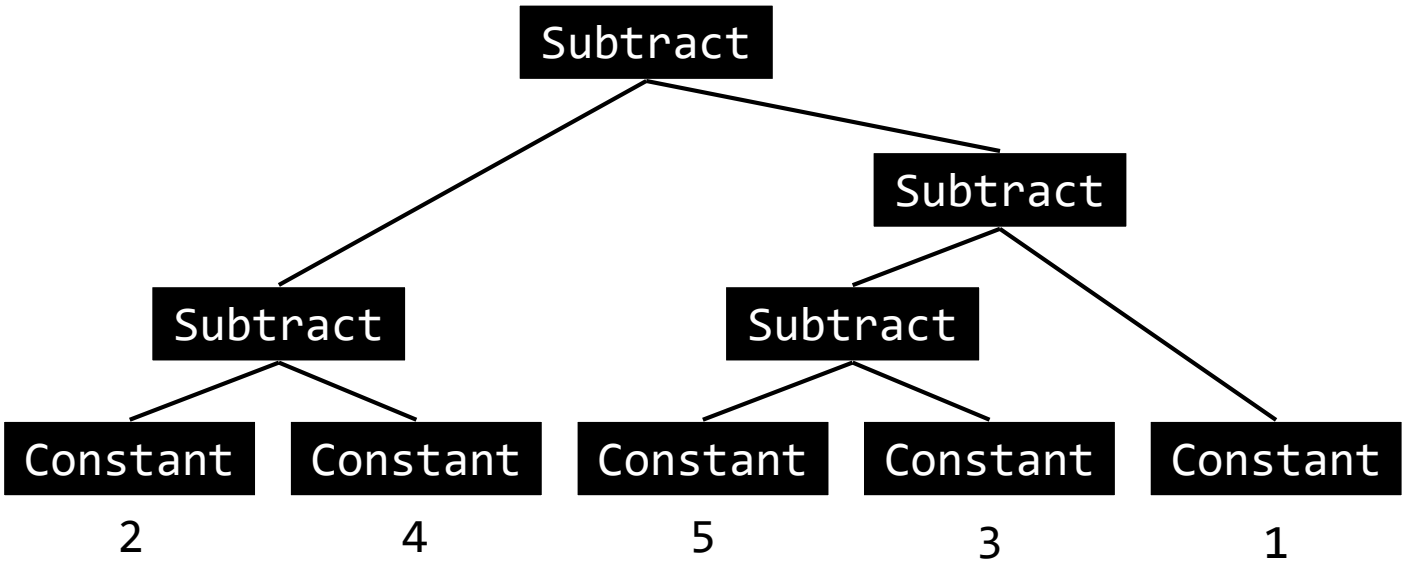
- Heterogenní = objekty v hierarchii mohou být různých typů.

Příklad heterogenní hierarchie

- Program má za úkol spravovat aritmetické výrazy.
- Mějme aritmetický výraz:

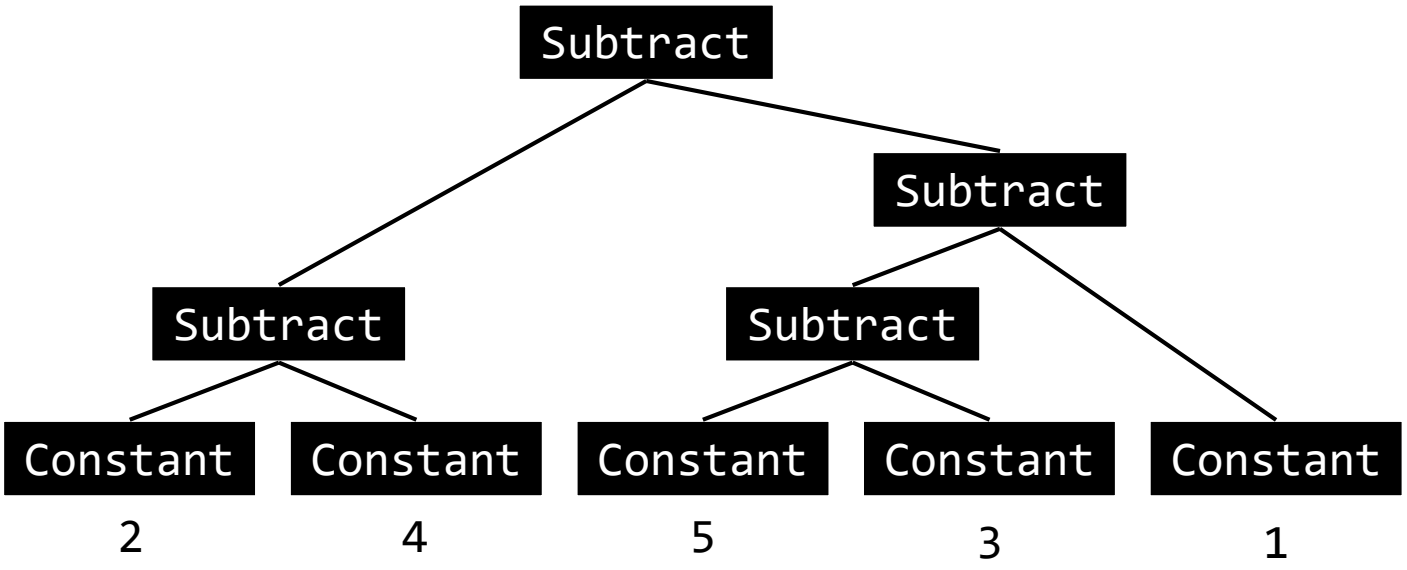
```
(2 - 4) - ((5 - 3) - 1)
```

- Program může tento výraz uložit v paměti jako hierarchii různorodých objektů:



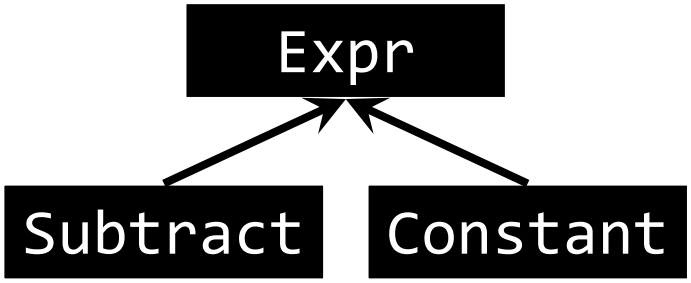
Příklad heterogenní hierarchie

- V této hierarchii je třída `Subtract` nadřazena jak dalším objektům typu `Subtract`, tak objektům typu `Constant`.
- Vzniká problém při implementaci: na jaký typ má `Subtract` odkazovat? Řešením je dědičnost.



Příklad heterogenní hierarchie

- V této hierarchii je třída `Subtract` nadřazena jak dalším objektům typu `Subtract`, tak objektům typu `Constant`.
- Vzniká problém při implementaci: na jaký typ má `Subtract` odkazovat? Řešením je dědičnost.
- Zavedeme třídu `Expr` (expression – výraz), která bude předkem oběma třídám `Subtract` a `Constant`.



Dědění – syntax

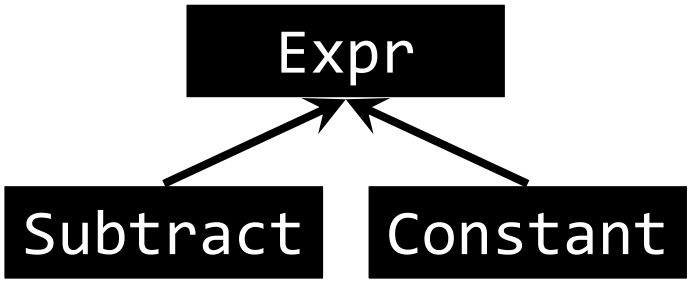
```
class Expr {
    ...
};

class Subtract : public Expr {
    ...
};

class Constant : public Expr {
    ...
};
```

Díky dědění se nyní na místo objektu třídy `Expr` může dosadit objekt třídy `Subtract` nebo `Constant`.

Význam klíčového slova `public` probereme později.



Příklad heterogenní hierarchie

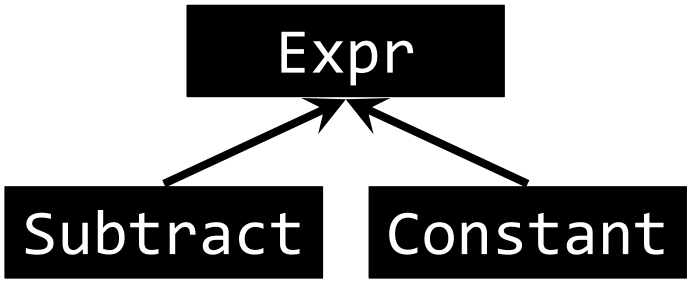
```
class Expr {
    ...
};

class Subtract : public Expr {
    Expr* lhs;
    Expr* rhs;
};

class Constant : public Expr {
    ...
};
```

Hierarchii implementujeme tak, že se ve třídě `Subtract` odkazujeme na třídu `Expr`.

Varianta 1/3: ukazatele



Příklad heterogenní hierarchie

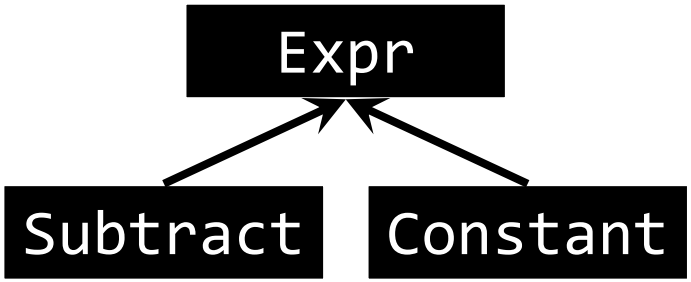
```
class Expr {
    ...
};

class Subtract : public Expr {
    std::unique_ptr<Expr> lhs;
    std::unique_ptr<Expr> rhs;
};

class Constant : public Expr {
    ...
};
```

Hierarchii implementujeme tak, že se ve třídě `Subtract` odkazujeme na třídu `Expr`.

Varianta 2/3: `unique_ptr`



Příklad heterogenní hierarchie

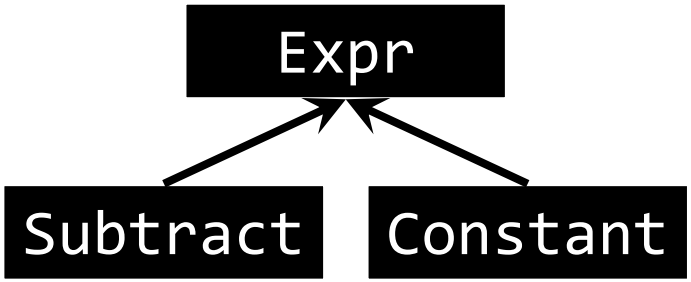
```
class Expr {
    ...
};

class Subtract : public Expr {
    Expr lhs;
    Expr rhs;
};

class Constant : public Expr {
    ...
};
```

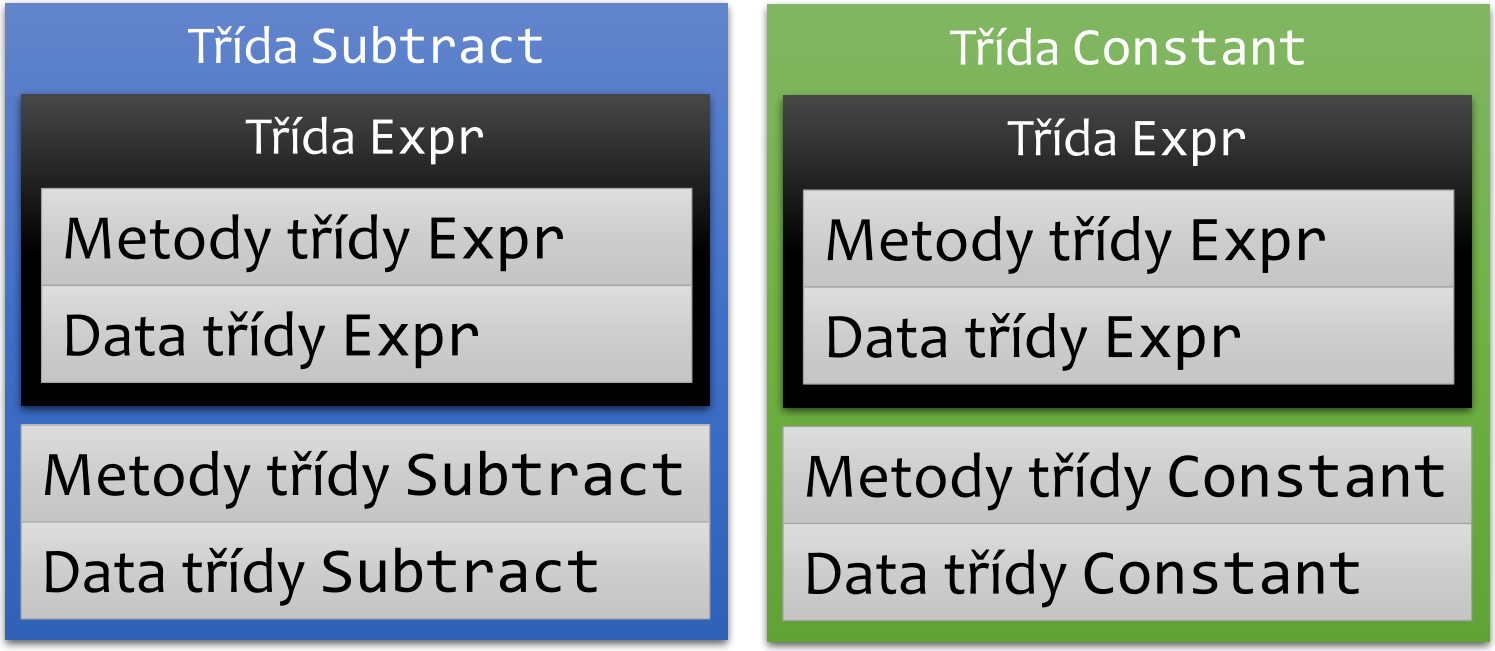
Hierarchii implementujeme tak, že se ve třídě `Subtract` odkazujeme na třídu `Expr`.

Varianta 3/3: hodnoty
Takto ne! (viz object slicing dále v přednášce)



Význam dědění

- Dědění zakládá vztah *potomek je druh rodiče* (odčítání je druh výrazu, konstanta je druh výrazu).
- Děděním potomek získá všechna data a metody předka, vyjma konstruktorů (o tom později).
- Můžeme si představit, že potomek předka obsahuje:



Příklad: přesné datum

```
struct Datum {
    int rok;
    int mesic;
    int den;
};
struct PresneDatum : Datum {
    int hodiny;
    int minuty;
    int sekundy;
};
```

```
PresneDatum vostok1() {
    PresneDatum d;
    d.rok = 1961;
    d.mesic = 4;
    d.den = 12;
    d.hodiny = 7;
    d.minuty = 55;
    d.sekundy = 0;
    return d;
}
```

- `PresneDatum` získá všechna data třídy `Datum`.
- Proč ne `struct PresneDatum : public Datum`? Klíčové slovo `public` zde není potřeba, protože `PresneDatum` je `struct` (viz veřejná a soukromá dědičnost).

Kopírování potomka do rodiče

```
PresneDatum vostok1() {
    ...
}

int main() {
    Datum d = vostok1();

    std::cout << d.den << '.' << d.mesic << '.' <<
        d.rok << '\n';
}
```

- Je povoleno kopírovat objekt potomka do objektu rodiče. Zkopírují se pouze datové položky rodiče.
- Později si ukážeme problémy způsobené tímto chováním (viz object slicing).

Dynamická vazba

- Dynamická vazba je způsob volání metod, který umožňuje **volat metody potomka prostřednictvím ukazatele na rodiče nebo reference na rodiče.**
- Příklad: Každé zvíře dělá vlastní zvuk, pes štěká a kočka mňouká. Zvířata reprezentujeme v programu jako různé potomky třídy `Zvire`. Když pak máme referenci (nebo ukazatel) na `Zvire`, očekáváme, že volání metody `udelejZvuk` vyloudí různé zvuky pro různá zvířata.

```
void foo(const Zvire& zvire) {
    zvire.udelejZvuk();  // haf pokud pes, mňau pokud kočka
}
```

```
void bar(Zvire* zvire) {
    zvire->udelejZvuk(); // haf pokud pes, mňau pokud kočka
}
```

Statická vazba v C++

- V jiných programovacích jazycích (C#, Java) považujeme dynamickou vazbu za samozřejmost. C++ ale používá **statickou vazbu**, která zavolá pouze tu metodu, která náleží typu ukazatele/reference.

```
struct Zvire {
    void udelejZvuk() const { std::cout << "!!!\n"; }
};
struct Pes : Zvire {
    void udelejZvuk() const { std::cout << "haf\n"; }
};
struct Kocka : Zvire {
    void udelejZvuk() const { std::cout << "mnau\n"; }
};
```

```
void foo(const Zvire& zvire) { zvire.udelejZvuk(); }
```

```
Kocka k; Pes p;
k.udelejZvuk(); p.udelejZvuk(); foo(k); foo(p);
```

mnau haf !!! !!!

Dynamická vazba v C++

- Abychom v C++ dosáhli dynamické vazby, musíme označit metodu jako **virtuální**.

```
struct Zvire {
    virtual void udelejZvuk() const { std::cout << "!!!\n"; }
};
struct Pes : Zvire {
    virtual void udelejZvuk() const { std::cout << "haf\n"; }
};
struct Kocka : Zvire {
    virtual void udelejZvuk() const { std::cout << "mnau\n"; }
};
```

```
void foo(const Zvire& zvire) { zvire.udelejZvuk(); }
```

```
Kocka k; Pes p;
k.udelejZvuk(); p.udelejZvuk(); foo(k); foo(p);
```

mnau haf mnau haf

16

Object slicing

- Pokud **zkopírujeme potomka do rodiče**, rodič nezíská data ani metody potomka. Tomuto chování říkáme **object slicing**.

```
Pes p;
Zvire* z = &p;
z->udelejZvuk(); // haf
```

```
Pes p;
Zvire& z = p;
z.udelejZvuk(); // haf
```

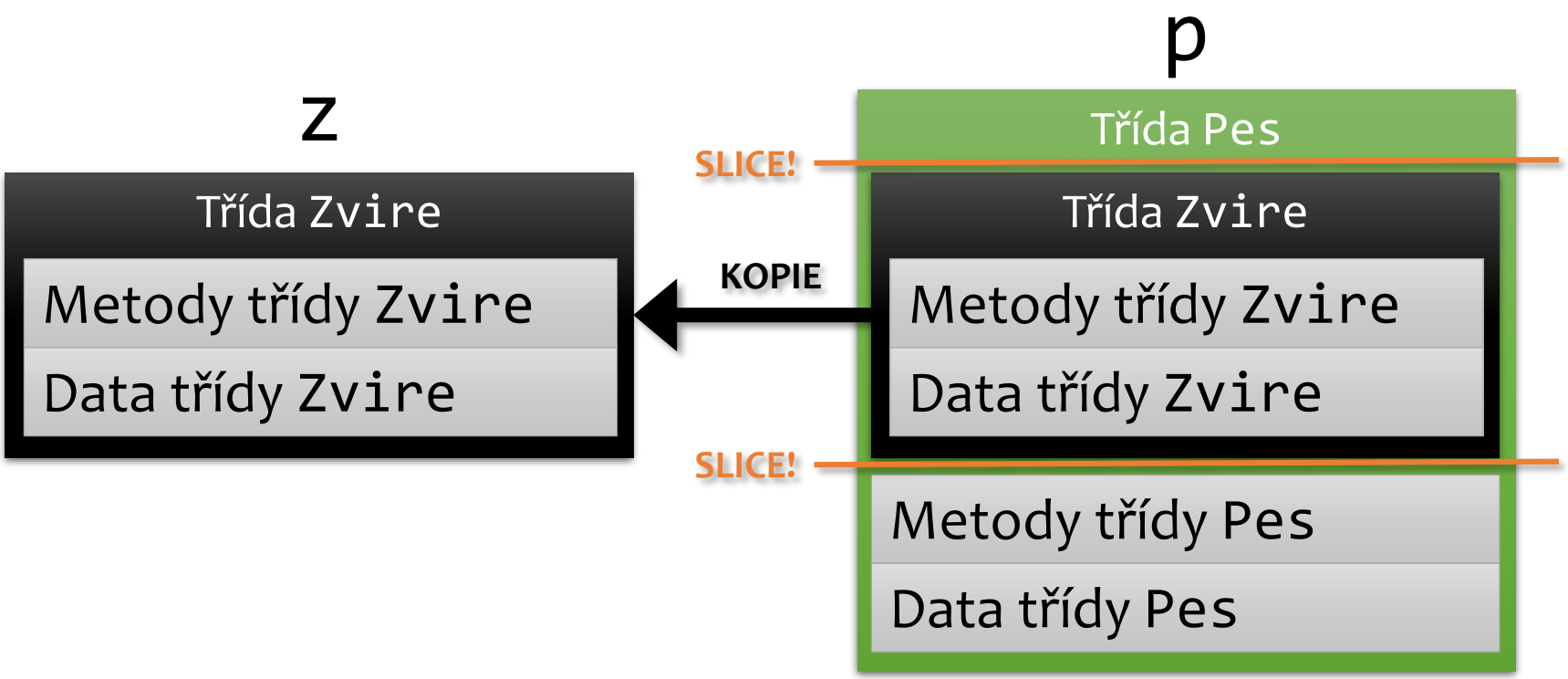
```
Pes p;
const Zvire& z = p;
z.udelejZvuk(); // haf
```

```
Pes p;
Zvire z = p;  SLICE!
z.udelejZvuk(); // !!!
```

- Pokud potřebujeme zabránit object slicingu, nesmíme kopírovat.

Object slicing - ilustrace

```
Pes p;
Zvire z = p;
```



Příklady object slicingu (1/4)

```
void foo(Zvire z) {
    z.udelejZvuk(); // !!!
}
int main() {
    Pes p;
    foo(p); SLICE!
}
```

Jak to spravit?

Příklady object slicingu (1/4)

```
void foo(Zvire z) {
    z.udelejZvuk(); // !!!
}
int main() {
    Pes p;
    foo(p); SLICE!
}
```

```
void foo(const Zvire& z) {
    z.udelejZvuk(); // haf
}
int main() {
    Pes p;
    foo(p);
}
```

Příklady object slicingu (2/4)

```
Zvire pes() {
    Pes p;
    return p;  SLICE!
}
int main() {
    Zvire z = pes();
    z.udelejZvuk(); // !!!
}
```

Jak to spravit?

Příklady object slicingu (2/4)

```
Zvire pes() {
    Pes p;
    return p;  SLICE!
}
int main() {
    Zvire z = pes();
    z.udelejZvuk(); // !!!
}
```

```
const Zvire& pes() {
    Pes p;
    return p;
}
int main() {
    const Zvire& z = pes();
    z.udelejZvuk(); // z ukazuje na smazaný objekt
}
```

Příklady object slicingu (2/4)

```
Zvire pes() {
    Pes p;
    return p;  SLICE!
}
int main() {
    Zvire z = pes();
    z.udelejZvuk(); // !!!
}
```

```
std::unique_ptr<Zvire> pes() {
    std::unique_ptr<Pes> p(new Pes);
    return std::move(p);
}
int main() {
    std::unique_ptr<Zvire> z = pes();
    z->udelejZvuk(); // haf
}
```

Příklady object slicingu (3/4)

```
int main() {
    std::vector<Zvire> zv;
    Pes p;
    zv.push_back(p); SLICE!
    zv.back().udelejZvuk(); // !!!
}
```

Jak to spravit?

Příklady object slicingu (3/4)

```
int main() {
    std::vector<Zvire> zv;
    Pes p;
    zv.push_back(p); SLICE!
    zv.back().udelejZvuk(); // !!!
}
```

```
int main() {
    std::vector<std::unique_ptr<Zvire>> zv;
    std::unique_ptr<Pes> p(new Pes);
    zv.push_back(std::move(p));
    zv.back()->udelejZvuk(); // haf
}
```

Příklady object slicingu (4/4)

```
struct Osoba {
    Zvire oblibeneZviratko;
};
int main() {
    Osoba pepa;
    Pes akim;
    pepa.oblibeneZviratko = akim; SLICE!
    pepa.oblibeneZviratko.udelejZvuk(); // !!!
}
```

Jak to spravit?

Příklady object slicingu (4/4)

```
struct Osoba {
    Zvire oblibeneZviratko;
};
int main() {
    Osoba pepa;
    Pes akim;
    pepa.oblibeneZviratko = akim; SLICE!
    pepa.oblibeneZviratko.udelejZvuk(); // !!!
}
```

```
struct Osoba {
    std::unique_ptr<Zvire> oblibeneZviratko;
};
int main() {
    Osoba pepa;
    std::unique_ptr<Pes> akim(new Pes);
    pepa.oblibeneZviratko = std::move(akim);
    pepa.oblibeneZviratko->udelejZvuk(); // haf
}
```

Object slicing – shrnutí

- Pokud potřebujeme zabránit object slicingu, musíme zabránit kopírování objektu potomka do objektu rodiče.
- Jak zabránit kopírování potomka typu P do rodiče typu R?
  - Pokud nechcete předat vlastnictví a P pouze zapůjčujete ke čtení, použijte `const R&`.
  - Pokud nechcete předat vlastnictví a P zapůjčujete ke čtení i zápisu, použijte `R&`.
  - Pokud chcete předat vlastnictví (tj. předat odpovědnost za smazání objektu P), vytvořte objekt P jako `std::unique_ptr` a předejte ho přesunem pomocí `std::move()`.
- Co když doopravdy potřebuji zkopírovat celý objekt potomka, ale znám jenom typ rodiče?
  - To není vždy možné, rodič i potomek musí mít zvláštní virtuální metodu typicky nazývanou `clone()` (viz. cvičení)

Otázka pro pozorné

- Proč dynamická vazba funguje jenom pro ukazatele a reference?

```
Zvire& ref = ...;
Zvire* ptr = ...;
Zvire val = ...;
ref.udelejZvuk(); // dynamická vazba, možná haf
ptr->udelejZvuk(); // dynamická vazba, možná haf
val.udelejZvuk(); // statická vazba, vždy !!!
```

Otázka pro pozorné

- Proč dynamická vazba funguje jenom pro ukazatele a reference?

```
Zvire& ref = ...;
Zvire* ptr = ...;
Zvire val = ...;
ref.udelejZvuk(); // dynamická vazba, možná haf
ptr->udelejZvuk(); // dynamická vazba, možná haf
val.udelejZvuk(); // statická vazba, vždy !!!
```

- Dynamická vazba pro proměnnou `val` by byla zbytečná. Pokud jsme do `val` zkopírovali potomka, **muselo dojít ke slicingu**. Proměnná typu `Zvire` vždy obsahuje jedině `Zvire`!

Konstruktory v hierarchiích tříd (1/3)

- Konstruktor rodiče je proveden vždy před konstruktorem potomka.

```
struct Zvire {
    Zvire() { std::cout << "Zvire(), "; }
};
struct Pes : Zvire {
    Pes() { std::cout << "Pes()\n"; }
};

int main() {
    Pes p; // Zvire(), Pes()
}
```

Konstruktory v hierarchiích tříd (2/3)

- Pokud vyžadujeme, aby byl zavolán jiný než výchozí konstruktor rodiče, určíme ho v inicializačním seznamu.

```
struct Zvire {
    Zvire() { std::cout << "Zvire(), "; }
    Zvire(int i) { std::cout << "Zvire(" << i << "), "; }
};
struct Pes : Zvire {
    Pes() : Zvire(42) { std::cout << "Pes()\n"; }
};

int main() {
    Pes p; // Zvire(42), Pes()
}
```

Konstruktory v hierarchiích tříd (3/3)

- V konstruktorech nefunguje pro vlastní metody dynamická vazba.

```
struct Zvire {
    Zvire() { std::cout << "Zvire dela "; udelejZvuk(); }
    virtual void udelejZvuk() const { std::cout << "!!!\n"; }
};
struct Pes : Zvire {
    Pes() { std::cout << "Pes dela "; udelejZvuk(); }
    virtual void udelejZvuk() const { std::cout << "haf\n"; }
};

int main() {
    Pes p; // Zvire dela !!!
}          // Pes dela haf
```

- Pozor na vlastní virtuální metody v konstruktoru!

Destruktory v hierarchiích tříd (1/3)

- Pokud smažeme potomka pomocí ukazatele nebo reference na rodiče, zavolá se pouze destruktor rodiče.

```
struct Zvire {
    ~Zvire() { std::cout << "~Zvire()\n"; }
};

struct Pes : Zvire {
    ~Pes() { std::cout << "~Pes(), "; }
};

int main() {
    std::unique_ptr<Zvire> z(new Pes);
} // ~Zvire()
```

- Stejně jako ostatní metody, i destruktor používá statickou vazbu, pokud neřekneme jinak.

Destruktory v hierarchiích tříd (2/3)

- Destruktory se zavolají správně, když jsou označeny jako virtuální.

```
struct Zvire {
    virtual ~Zvire() { std::cout << "~Zvire()\n"; }
};

struct Pes : Zvire {
    virtual ~Pes() { std::cout << "~Pes(), "; }
};

int main() {
    std::unique_ptr<Zvire> z(new Pes);
} // ~Pes(), ~Zvire()
```

- Pokud je třída součástí hierarchie tříd, vždy označte destruktor jako virtuální!

Destruktory v hierarchiích tříd (3/3)

- Destruktor potomka je proveden vždy před destruktorem rodiče.

```
struct Zvire {
    virtual ~Zvire() { std::cout << "~Zvire()\n"; }
};

struct Pes : Zvire {
    virtual ~Pes() { std::cout << "~Pes(), "; }
};

int main() {
    std::unique_ptr<Zvire> z(new Pes);
} // ~Pes(), ~Zvire()
```

(Za předpokladu, že jsme nezapomněli označit destruktory jako virtuální.)

Příklad na pořadí volání konstr. a destr.

```
struct A {
  A() { std::cout << "A"; }
  virtual ~A() { std::cout << "~A"; }
};

struct B : A {
  B() { std::cout << "B"; }
  virtual ~B() { std::cout << "~B"; }
};

struct C : B {
  C() { std::cout << "C"; }
  virtual ~C() { std::cout << "~C"; }
};
```

```
int main() {
  using ptr = std::unique_ptr<A>;
  std::vector<ptr> vec;
  std::cout << "[push1]";
  ptr bPtr(new B);
  vec.push_back(std::move(bPtr));
  std::cout << "[push2]";
  ptr cPtr(new C);
  vec.push_back(std::move(cPtr));
  std::cout << "[pop]";
  vec.pop_back();
  std::cout << "[end]";
}
```

- Program vypíše
  [push1]AB[push2]ABC[pop]~C~B~A[end]~B~A

Shrnutí – konstruktory a destruktory

- Konstruktor rodiče je proveden vždy před konstruktorem potomka.
- Destruktor potomka je proveden vždy před destruktorem rodiče.
- Pozor na vlastní virtuální metody v konstruktoru!
- Pokud je třída součástí hierarchie tříd, vždy označte destruktor jako virtuální!

Vícenásobná dědičnost

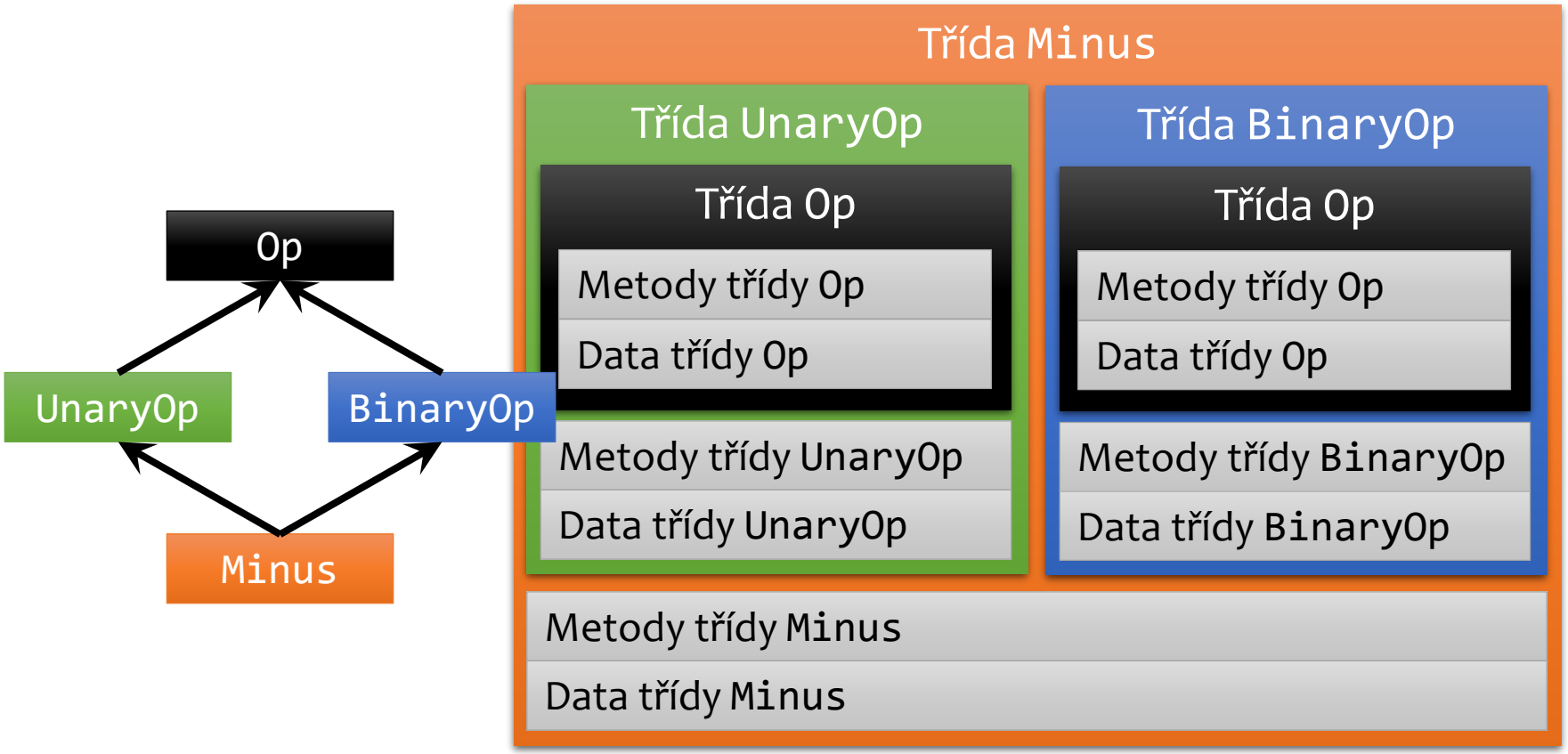
- Je povoleno dědit z více tříd najednou.

```
class Subtract : public Expr, public BinaryOp {
    ...
};
```

- Nedoporučujeme dědit z více tříd, protože může nastat tzv. **diamond problem**.

Diamond problem

- Mějme dvě třídy se společným předkem. Když je nějaká další třída potomkem obou, bude obsahovat jejich společného předka dvakrát.



Diamond problem – následky

- Pokud nastane diamond problem, nelze rozhodnout:
  - která data máme na mysli, když chceme přistoupit k datům společného předka.
  - kterou metodu máme na mysli, když chceme zavolat metodu společného předka.

```
struct Op { int data; void foo() {} };
struct UnaryOp : Op {};
struct BinaryOp : Op {};
struct Minus : UnaryOp, BinaryOp {};

int main() {
    Minus m;
    m.foo(); // chyba: které foo?
    m.data; // chyba: která data?
}
```

Diamond problem – řešení (1/2)

- Jak obejít diamond problem?
  - Nepoužívat vícenásobnou dědičnost.
    - Případy, kdy je dědičnost skutečně nejlepším řešením problému, jsou vzácné.
    - Preferujte metaprogramování se šablonami.

```
struct Op { int data; void foo() {} };
struct UnaryOp : Op {};
struct BinaryOp : Op {};
struct Minus : UnaryOp, BinaryOp {};

int main() {
    Minus m;
    m.foo(); // chyba: které foo?
    m.data; // chyba: která data?
}
```

Diamond problem – řešení (2/2)

- Jak obejít diamond problem?
  - Pokud je vícenásobná dědičnost nevyhnutelná a nastane diamond problem, lze použít klíčové slovo `virtual` při dědění společného předka.
  - Potomek poté obsahuje společného předka pouze jednou.

```
struct Op { int data; void foo() {} };
struct UnaryOp : virtual Op {};
struct BinaryOp : virtual Op {};
struct Minus : UnaryOp, BinaryOp {};

int main() {
    Minus m;
    m.foo(); // v pořádku
    m.data; // v pořádku
}
```

Veřejná a soukromá dědičnost

- Třídy uvozené klíčovým slovem `struct` používají veřejnou dědičnost, pokud není stanoveno jinak.
- Třídy uvozené klíčovým slovem `class` používají soukromou dědičnost, pokud není stanoveno jinak.

```
class Pes1 : Zvire {
    // Zvire zděděno soukromě
};
class Pes2 : public Zvire {
    // Zvire zděděno veřejně
};
struct Pes3 : Zvire {
    // Zvire zděděno veřejně
};
struct Pes4 : private Zvire {
    // Zvire zděděno soukromě
};
```

Význam veřejné dědičnosti (1/2)

- Pokud je rodič zděděn veřejně, kód vně obou tříd má přístup k metodám a funkcím rodiče prostřednictvím objektu potomka.

```
struct Zvire {
    void foo() {}
};
class Pes1 : Zvire {
    // Zvire zděděno soukromě
};
class Pes2 : public Zvire {
    // Zvire zděděno veřejně
};
int main() {
    Pes1 pes1; pes1.foo(); // chyba: foo je skryto
    Pes2 pes2; pes2.foo(); // v pořádku, foo není skryto
}
```

Význam veřejné dědičnosti (2/2)

- Pokud je rodič zděděn veřejně, kód vně obou tříd smí považovat ukazatel nebo referenci na potomka za ukazatel nebo referenci na rodiče.

```
struct Zvire {
    void foo() {}
};
class Pes1 : Zvire {
    // Zvire zděděno soukromě
};
class Pes2 : public Zvire {
    // Zvire zděděno veřejně
};
int main() {
    Pes1 pes1; Zvire& z1 = pes1; // chyba, Pes1 není Zvire
    Pes2 pes2; Zvire& z2 = pes2; // v pořádku, Pes2 je Zvire
}
```

Dědění konstruktorů

- Děděním získá potomek všechna data a metody rodiče, kromě konstruktorů.
- Konstruktory rodiče lze získat pomocí klíčového slova `using`.

```
struct Zvire {
    Zvire(std::string jmeno) : m_jmeno(std::move(jmeno)) {}
protected:
    std::string m_jmeno;
};
struct Pes : Zvire {
    using Zvire::Zvire;
};

int main() {
    Pes akim("Akim");
}
```

Změna viditelnosti

- Klíčové slovo `using` je užitečné také v případě, že chceme změnit viditelnost dat nebo metod rodiče.

```
struct Zvire {
    std::string m_jmeno;
};
struct Pes : Zvire {
private:
    using Zvire::m_jmeno;
};

int main() {
    Zvire felix;
    felix.m_jmeno = "Felix"; // OK, m_jmeno je přístupné
    Pes akim;
    akim.m_jmeno = "Akim"; // chyba, m_jmeno je nepřístupné
}
```

Abstraktní třída

- Pokud místo těla virtuální metody umístíme =0, jedná se o metodu bez implementace – tzv. **čistě virtuální metodu** (pure virtual member function).
- Třída, která obsahuje jednu nebo více čistě virtuálních metod, se nazývá **abstraktní třída**.
- Objekt abstraktní třídy nelze vytvořit. Abstraktní třídu lze pouze zdědit.

```
struct Zvire {
    virtual void udelejZvuk() const = 0;
};
struct Pes : Zvire {
    virtual void udelejZvuk() const { std::cout << "haf"; }
};
```

Override a final

- Pokud označíme metodu `override`, vyžadujeme, aby se jednalo o virtuální metodu nahrazující virtuální metodu v rodiči.
- Pokud označíme metodu `final`, vyžadujeme, aby se jednalo o virtuální metodu, která není nahrazena v žádném potomkovi.

```
struct Zvire {
    virtual ~Zvire() = default;
    virtual void udelejZvuk() const { std::cout << "!!!\n"; }
};
struct Pes : Zvire {
    virtual ~Pes() override = default;
    virtual void udelejZvuk() const override final {
        std::cout << "haf\n";
    }
};
```

Použití override (1/2)

- Pokud označíme metodu `override` a nejedná se o nahrazení virtuální metody v předkovi, kompilace selže.
- Takto `override` odhalí téměř neviditelné, ale velmi podstatné chyby v deklaracích metod.

```
struct Zvire {
    virtual ~Zvire() = default;
    virtual void udelejZvuk() const { std::cout << "!!!\n"; }
};
struct Pes : Zvire {
    virtual ~Pes() override = default;
    virtual void udelejZvuk() override { // chybí const
        std::cout << "haf\n";
    }
};
```

Použití override (2/2)

- Jiný příklad odhalené chyby:

```
struct Zvire {
    ~Zvire() = default;
    virtual void udelejZvuk() const { std::cout << "!!!\n"; }
};
struct Pes : Zvire {
    virtual ~Pes() override = default; // ~Zvire nevirtuální
    virtual void udelejZvuk() const override {
        std::cout << "haf\n";
    }
};
```

- Chyby tohoto charakteru je těžké nalézt čtením kódu.
- Používejte `override` pro všechny virtuální metody v potomcích!

Použití final (1/2)

- Pokud označíme metodu `final` a dědící třída se jí pokusí nahradit, kompilace selže.
- Užitečné, pokud má metoda důležitý invariant a bojíme se, že by ho potomci mohli nedodržet.

```
struct Zvire {...};
struct Pes : Zvire {
    virtual void udelejZvuk() const override final {
        // všichni psi budou dělat haf ~~~~~~^
        std::cout << "haf\n";
    }
};
struct Foxterier : Pes {
    virtual void udelejZvuk() const { // Kompilační chyba
        // Nelze nahradit final metodu
        std::cout << "vrrr\n";
    }
};
```

Použití final (2/2)

- Pokud označíme třídu `final` a jiná třída z ní dědí, kompilace selže.
- Užitečné pokud, má metoda důležitý invariant a bojíme se, že by ho potomci mohli nedodržet.
  - Navíc může kompilátor provést dodatečné optimalizace

```
struct Zvire {...};
struct Pes : Zvire {
    virtual void udelejZvuk() const override final {
        std::cout << "haf\n";
    }
};

struct Foxterier final : Pes {};
// Foxteriera nelze dědit
struct KratkosrstyFoxterier : Foxterier {};//Kompilační chyba
// Nelze dědit finální třídu
```

Přetypování na potomka

- Přetypovat ukazatel (referenci) na potomka na ukazatel na rodiče je povoleno vždy, když je rodič zděděn veřejně.
- V opačném směru, přetypovat ukazatel na rodiče na ukazatel na potomka, je možné pouze pomocí `dynamic_cast`.

```
void foo(Zvire& z) {
    Pes& p = dynamic_cast<Pes&>(z);
    ...
}
```

- To, zda je přetypování možné, je kontrolováno **za běhu programu**. Použití `dynamic_cast` není zdarma!
- Pokud se přetypování nezdaří, bude vyhozena vyjímka.
- Vyhýbejte se přetypování na potomka! Pokud je třeba použít `dynamic_cast`, pravděpodobně je chyba v návrhu hierarchie tříd.

Shrnutí – na co dávat pozor

- **Dynamická vazba** funguje pouze pro virtuální metody.
- Kopírování objektu potomka do objektu rodiče způsobuje **object slicing**.
- Pozor na volání vlastních virtuálních metod v konstruktoru!
- Pokud je třída součástí hierarchie tříd, vždy označte destruktor jako virtuální!
- Vyhýbejte se dědění z více tříd najednou, protože může nastat **diamond problem**.
- Používejte `override` pro všechny virtuální metody v potomcích!
- Vyhýbejte se přetypování na potomka!

Děkuji za pozornost.