

Dědičnost

Karel Richta a kol.

katedra počítačů FEL ČVUT v Praze

Přednášky byly připraveny s pomocí materiálů, které vyrobili Ladislav Vágner, Pavel Strnad, Martin Hořeňovský, Aleš Hrabalík

© Karel Richta, 2015

Programování v C++, A7B36PJC

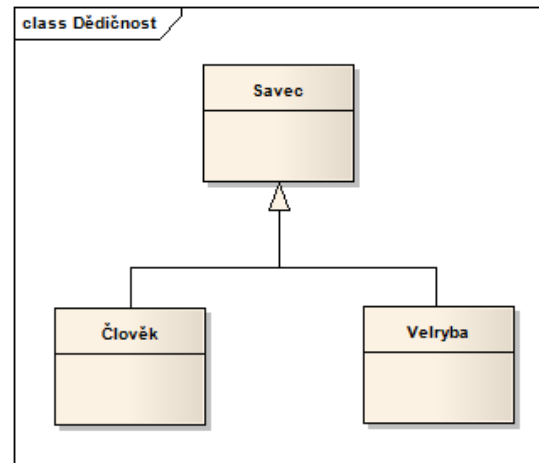
09/2015, Lekce 7

<https://cw.fel.cvut.cz/wiki/courses/a7b36pjc/start>



Dědičnost

- Dědičnost je technika disciplinovaného vývoje nových tříd ze tříd již existujících.
- Existující třídu, od které dědíme, nazýváme **rodič** (předek), třídu odvozenou děděním **potomek**.
- Rodič může mít více potomků.
- Potomek může mít více rodičů, ale zpravidla se tato vícenásobná dědičnost používá jen výjimečně, neboť může působit jisté problémy.
- Graficky dědění kreslíme:



Dědičnost (pokr.)

Základní vlastnosti:

- Potomek dědí vlastnosti a chování rodiče – členská data i metody.
- Potomek může přidávat nová data, nemůže nic ubrat.
- Potomek může metody přidávat, upravovat, nebo překrývat (přepsat).

Standardní postup:

- Identifikace podobných tříd v modelované oblasti.
- Určení společných vlastností a chování (členská data a metody).
- Vytvoření nové třídy obsahující právě společné atributy a metody.
- Nová třída se stane rodičem předchozích tříd – ty se stanou potomky.
- V původních třídách již nejsou společné atributy explicitně zmíněny.

Příklad:

Třída: pes, kočka, člověk.

Společné vlastnosti – jméno, stáří, pohlaví.

Nová třída – rodič Savec se společnými vlastnostmi.

Potomci mají navíc speciální vlastnosti – např. Pes (známka, barva_srsti).

Dědičnost vytváří hierarchii tříd (objektů) - generalizace a specializace.

Dědičnost (pokr.)

Podporuje abstrakci:

- Potomek dodržuje rozhraní zděděné od předka.
- Potomek mění chování předka – implementuje nové funkce a vlastnosti.

Omezuje duplicitu v kódu:

- Duplicita omezuje přehlednost kódu.
- Duplicita zvyšuje riziko chyb.
- Duplicita zvyšuje náročnost úprav (více míst).
- Duplicita zvyšuje nároky na paměť.

Podporuje znovupoužitelnost kódu:

V C se to zajišťuje pomocí funkcí:

- Funkce v sobě skrývá kód a její volání v různých místech jej opakovaně používá.
- Vytváří se hlavičkové soubory.

V C++ lze navíc využít dědičnost:

- Odstranění duplicit i pro proměnné.
- Dědičnost - mechanismus tvorby - odvozování tříd z již existujících.

Veřejná dědičnost

- Modeluje vztah „je“, „je druhu“, „is A“ (ISA).
- Pro úpravu vlastností tříd není nutné mít zdrojový kód, pouze prototypy.
- Odvození třídy potomek od rodiče (základní) je explicitně dáno v deklaraci potomka (v rodiči není nic uvedeno).
- Předpis: **class Potomek : public Rodic** – operátor :
- Příklad: **class Clovek : public Savec{}**
- Objekt odvozené třídy obsahuje objekt základní třídy.
- Veřejné položky rodiče se stanou veřejnými položkami potomka.
- Soukromé položky se stanou součástí potomka – přístupné pouze pomocí veřejných metod rodiče.
- Pokud vynecháte klíčové slovo **public** – je dědičnost automaticky **private** (soukromá dědičnost) – vše je soukromé.
- Položky potomka nejsou vůbec přístupné základní rodičovské třídě – ta o nich neví.

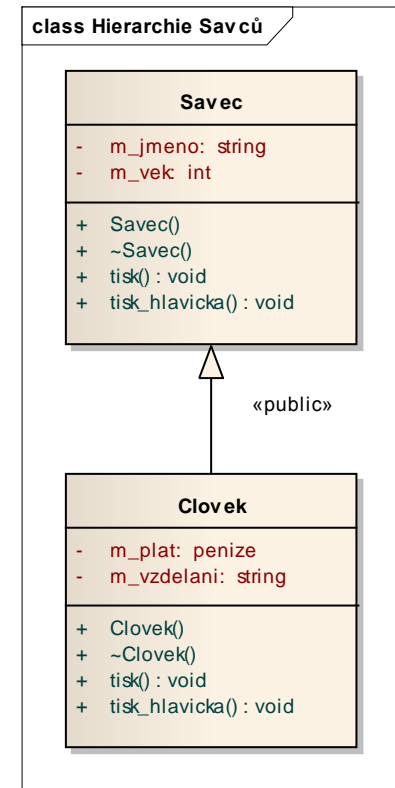
Veřejná dědičnost – příklad

- Základní třída (rodič):

```
class Savec {  
    string m_jmeno;  
    int m_vek;  
public:  
    Savec(string jm, int ve);  
    ~Savec();  
    void tisk();  
    void tisk_hlavicka();  
};
```

- Odvozená třída:

```
class Clovek : public Savec {  
    int m_plat;  
    string m_vzdelani;  
public:  
    Clovek(string jm, int ve, int pl, string vz);  
    ~Clovek();  
    void tisk();  
    void tisk_hlavicka();  
};
```



Veřejná dědičnost (pokr.)

- Odvozená třída - nové položky, nové metody - nedostupné v rodiči (ten o nich neví).
- Odvozená třída musí mít svůj konstruktor.
- Při vytváření objektu potomka se vyvolá:
 - 1. konstruktor rodiče,
 - 2. konstruktor potomka.
- Konstruktor potomka předá informace konstruktoru rodiče.
- Destruktor se potomkovi nemusí přidávat, pokud není nutné něco dynamicky uvolňovat – pak se musí definovat.
- Pokud platnost objektu končí, volá se:
 - 1. destruktory potomka,
 - 2. destruktory rodiče.
- Potomek dědí všechny metody, konstruktor a destruktory je volán automaticky, nebo v seznamu inicializátorů.
- Nedědí se operátor přiřazení.

Veřejná dědičnost – konstruktory

- Konstruktory základní třídy:

```
Savec::Savec(string jm, int ve) {  
    this->m_jmeno = jm;  
    this->m_vek = ve;  
};
```

- Konstruktory odvozené třídy:

```
Clovek::Clovek(string jm, int ve, int pl, string vz) {  
    m_plat = pl;  
    m_vzdelani = vz;  
    setVek(ve); // nelze: m_vek = vek  
    setJmeno(jm); /* lze, ale první se volá implicitní konstruktor  
                  základní třídy */  
};
```

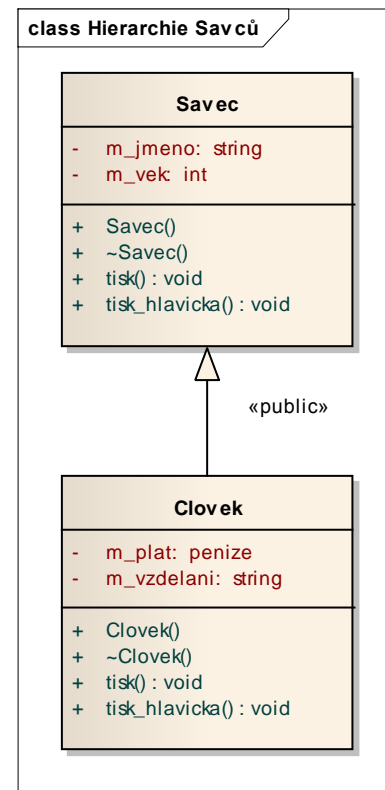
```
Clovek::Clovek(string jm, int ve, int pl, string vz) :  
    Savec(jm,ve) //seznam inicializátorů {  
    m_plat = pl;  
    m_vzdelani = vz;  
};
```

Konstruktor odvozené třídy volá implicitní konstruktor základní třídy.

Lze uvést explicitně jaký konstruktor se bude volat – seznam inicializátorů (podle prototypu).

Pokud je explicitní konstruktor uveden, implicitní se již nebude volat.

Z konstruktoru odvozené třídy je vždy volán konstruktor základní třídy.



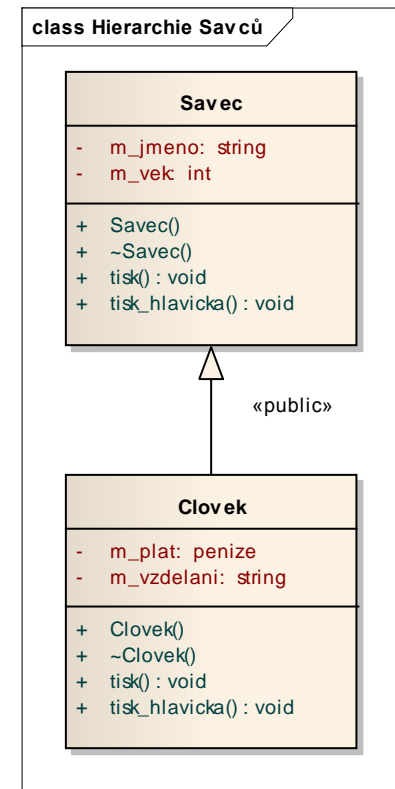
Veřejná dědičnost – destruktory

- Destruktor základní třídy:

```
Savec::~~Savec() {  
    pocet --;  
    cout << "\npocet objektu Savec: " << pocet;  
};
```

- Destruktor odvozené třídy:

```
Clovek::~~Clovek() {  
    pocet --;  
    cout << "\npocet objektu Clovek: " << pocet;  
};
```



Destruktor odvozené třídy volá destruktory základní třídy.

Inicializace objektu odvozené třídy objektem základní třídy

- Nový objekt je vytvořen pomocí existujícího objektu základní třídy.
- Použit lze kopírovací konstruktor, pro vytvoření kopie objektu základní třídy – ten naplní odpovídající část odvozeného objektu.
- Existuje implicitní kopírovací konstruktor, lze jej použít, pokud není nutné kopírovat dynamické objekty, nebo upravovat statická členská data.
- Konstruktor odvozené třídy:

```
Clovek::Clovek(Savec &sav, int pl, string vz) :  
    Savec(sav), m_plat(pl), m_vzdelani(vz) {};
```

```
Savec::Savec(const Savec &sav) : // kopírující konstruktor  
    m_jmeno(sav.m_jmeno), m_vek(sav.m_vek); {  
    cout<<"\nObjekt tridy Savec vytvoren (kop konstruktor): "<< sav.m_jmeno;  
};
```

Použití rozhraní základních tříd ve třídách odvozených

- Nelze přímo přistupovat k členským datům základní třídy ve třídě (metodách) třídy odvozené.
- Musí se použít veřejných metod základní třídy – rozhraní (pokud jsou členská data **private**).
- Pokud odvozená třída překryje metodu základní třídy, pak se použije.
- Jinak se bude používat metoda základní třídy.
- Pro exaktní určení metody je nutné použít operátor :: (úplné kval.jméno).

```
int Savec::get_vek() const {
    return m_vek;
};
void Savec::tisk() const {
    cout << "\nSavec jmeno: " << m_jmeno;
    cout << " vek: " << m_vek;
};
```

```
void Clovek::tisk_jmeno() const {
    cout << get_jmeno();
};
void Clovek::tisk() const {
    Savec::tisk();
    cout<<" plat: " << m_plat;
    cout<<" vzdelani: " << m_vzdelani;
};
```

Použití

```
int main() {  
    Savec s1("Pavel", 12);  
    Clovek c1("Josef", 24, 30000, "VS");  
    s1.tisk();  
    cout << endl << s1.get_vek() << endl;  
    c1.tisk();  
    cout << endl << c1.get_vek() << endl;  
}
```

```
Objekt tridy Savec c. 1 vytvoren: Pavel  
Objekt tridy Savec c. 2 vytvoren: Josef  
Objekt tridy Clovek c. 1 vytvoren : Josef  
Savec jmeno: Pavel vek: 12  
12
```

```
Savec jmeno: Josef vek: 24 plat: 30000 vzdelani: VS  
24
```

```
pocet objektu Clovek: 0  
pocet objektu Savec: 1  
pocet objektu Savec: 0
```

Statické proměnné a metody

- Jsou to data a metody třídy, nikoliv jednotlivých objektů.
- Jsou označena klíčovým slovem **static**.
- Statická data nejsou součástí žádné instance – musí se deklarovat odděleně.
- Statické metody mohou pracovat jen se statickými daty.
- Typický příklad: aktuální počet instancí třídy.

```
class Savec {  
private:  
    ...  
    static int pocet;  
public:  
    Savec (string, int);  
    Savec (const Savec &);  
    ~Savec();  
    static int kolik();  
};
```

```
int Savec::pocet = 0;
```

```
Savec::Savec (string jm, int ve) {  
    this->m_jmeno = jm;  
    this->m_vek = ve;  
    pocet++;  
};
```

```
Savec::~~Savec() { pocet --; };
```

```
int Savec::kolik() { return pocet; }
```

Základní příklad dědičnosti – rodič Savec

```
class Savec {
private:
    string m_jmeno;
    int m_vek;
    static int pocet;
public:
    Savec (string, int);
    Savec (const Savec &);
    ~Savec();
    int get_vek();
    string get_jmeno();
    void tisk();
    static int kolik();
};

int Savec::get_vek() {
    return m_vek; };

string Savec::get_jmeno() {
    return m_jmeno; };

int Savec::kolik() {
    return pocet; };
```

```
int Savec::pocet = 0;

Savec::Savec (string jm, int ve) :
    m_jmeno(jm), m_vek(ve) {
    pocet++; cout <<"\nObjekt tridy Savec c. „
        << pocet << “ vytvoren : ” << m_jmeno;
};

Savec::~Savec() { pocet --;
    cout <<
        "\nZrusen Savec, zbyva pocet objektu Savec: "<<
        pocet; };

void Savec::tisk() {
    cout<<"\nSavec jmeno: "<< m_jmeno;
    cout <<" vek: "<< m_vek; };

Savec::Savec(const Savec & sav) :
    m_jmeno(sav.m_jmeno), m_vek(sav.m_vek) {
    pocet++; cout<<"\nObjekt tridy Savec c. "
        << pocet <<" vytvoren (kop konstruktor): "
        << sav.m_jmeno;
};
```

Základní příklad dědičnosti – potomek Clovek

```
class Clovek:public Savec {
private:
    int m_plat;
    string m_vzdelani;
    static int pocet;
public:
    void tisk();
    Clovek(string, int, int,
            string);
    Clovek(Savec&, int,
            string);
    ~Clovek();
};

void Clovek::tisk() {
    Savec::tisk();
    cout <<" plat: "<< m_plat
         << " vzdelani: "
         << m_vzdelani;
};
```

```
int Clovek::pocet = 0;

Clovek::~~Clovek() { pocet --;
    cout <<"\nZrusen Clovek, pocet objektu Clovek: "<<
         pocet;
};

Clovek::Clovek(string jm, int ve, int pl, string vz)
: Savec(jm,ve), m_plat(pl), m_vzdelani(vz) {
    pocet++; cout<<"\nObjekt tridy Clovek c. "<<
         pocet<<" vytvoren : "<<jm;
};

Clovek::Clovek(Savec & sav, int pl, char* vz)
: Savec(sav), m_plat(pl), m_vzdelani(vz) {
    cout << "\nObjekt tridy Clovek c. "<<
         pocet <<" vytvoren: "<< sav.get_jmeno();
};
```

Základní příklad dědičnosti - použití

```
int main() {
    Savec sav1("Petr", 10);
    Clovek oso1("Jan",46, 10000,"VS");
    Clovek oso2(sav1,40000,"SS");
    sav1.tisk();
    oso1.tisk();
    oso2.tisk();
    system("pause");
}
```

```
Objekt tridy Savec c. 1 vytvoren : Petr
Objekt tridy Savec c. 2 vytvoren : Jan
Objekt tridy Clovek c. 1 vytvoren : Jan
Objekt tridy Savec c. 3 vytvoren (kop konstruktor): Petr
Objekt tridy Clovek c. 2 vytvoren: Petr
Savec jmeno: Petr vek: 10
Savec jmeno: Jan vek: 46 plat: 10000 vzdelani: VS
Savec jmeno: Petr vek: 10 plat: 40000 vzdelani: SS
Zrusen Clovek, zbyva pocet objektu Clovek: 1
Zrusen Savec, zbyva pocet objektu Savec: 2
Zrusen Clovek, zbyva pocet objektu Clovek: 0
Zrusen Savec, zbyva pocet objektu Savec: 1
Zrusen Savec, zbyva pocet objektu Savec: 0
Exit code: 0 (normal program termination)
```


Chráněná členská data (protected)

- Občas potřebujeme přístup k členským datům rodičů v potomcích – použijeme mód **protected** - podobné **private**, ale pro členské metody potomků jako **public**.
- Přístupné jsou jen pomocí členských metod.
- Potomci ovšem mají přímý přístup ke chráněným položkám základní třídy.

```
//savec.h
class Savec {
private:
    string m_jmeno;
    int m_vek;
    static int pocet;
protected:
    char m_pohlavi;
};
```

```
class Clovek : public Savec {
    string m_vzdelani;
    int m_plat;
public:
    tisk();
}
void Clovek::tisk() {
    Savec::tisk();
    cout <<" plat: "<< m_plat;
    cout <<" vzdelani: "<< m_vzdelani;
    cout << "pohlavi :"<< m_pohlavi;
};
```

Přetypování v dědičnosti

Motivace: chceme jeden typ ukazatele a více objektů, na které ukazuje.

Standardně není povoleno přiřadit jednomu typu ukazatele ukazatel na jiný typ:

```
double x = 2.2;
int * uk = & x; // nelze
```

Dědičnost – ukazatel na základní třídu může odkazovat i na objekt třídy odvozené.

Přetypování na předka:

- Nemusí se provést explicitní přetypování.
- Vše co může dělat objekt základní třídy může dělat i odvozená třída:

```
Clovek clo("Petr", 40, 1, "VS");
Clovek * uk_clo = &clo;
Savec * uk_sav = &clo;
uk_sav->getJmeno(); // všechny metody budou funkční...
```

Přetypování na potomka:

- Musí být provedeno explicitně.
- Odvozená třída má nové metody a data, které u základní třídy nelze použít:

```
Savec sav("Ivan", 30);
Clovek *uk = (Clovek*)&sav; /* měl by se použít dynamic_cast
                           Clovek *uk = dynamic_cast<Clovek*>(&sav); */
cout<<uk->getPohlavi(); // pouze metody ze základní třídy budou funkční
```

Příklad (přetypování na potomka)

```
class base {
    int i,j;
public:
    base() {}
    base(int _i,int _j) : i(_i),j(_j) {}
    virtual void show() { std::cout<<"Base "<<i<<" "<<j; }
};

class derived : public base {
    int i,j;
public:
    derived(int _i,int _j) : i(_i),j(_j) {}
    void show() { std::cout<<"Derived"<<i<<" "<<j; }
};

int main() {
    derived *d_ptr = dynamic_cast<derived *>(new base(1,2));
    d_ptr->show(); // Zde se vyhodí chyba "access violation error"
    base *b_ptr = dynamic_cast<base *>(new derived(1,2));
    b_ptr->show(); // Tady je to správně
    return 0;
}
```

Statická vazba

- Spuštění přetížené metody závisí na objektu, který ji vyvolal
<= platí jen pro spuštění metody pomocí tečkové notace:

```
Savec sav; // Savec::tisk() {cout<<"Savec";}
Clovek clo; // Clovek::tisk(cout<<"Clovek");
sav.tisk();
clo.tisk();
```

- V případě spouštění pomocí ukazatelů je implicitně spuštěna metoda určená typem ukazatele – ne konkrétního objektu:

```
Savec * uk_sav;
uk_sav = &sav;
uk_sav->tisk();
uk_sav = &clo;
uk_sav->tisk(); /* volá se Savec::tisk(), protože uk_sav
je ukazatel na savce - napraví to dynamická vazba */
```

Dynamická vazba

- Statická vazba – již v době překladu je jasné, která funkce bude spuštěna.
- Dynamická vazba – program zodpovídá za spuštění správné metody za běhu podle objektu který reprezentuje.
- Nástroj pro aktivaci dynamické vazby je virtuální členská metoda.
- V prototypu virtuální členské metody v základní třídě použijeme klíčové slovo **virtual**.
- Přetížené metody v základních třídách mohou překrýt danou funkci ze základní třídy – dle rozhodnutí programátora.
- Dynamická vazba musí udržovat informace o zpracovávaných objektech, statická nemusí – méně náročná na provedení.
- Pokud nepoužívám dědění – není nutná dynamická vazba.
- Pokud nechci metodu předefinovat v odvozené třídě – neuvedu **virtual**.
- Pokud není virtuální metoda v potomkovi vytvořena, bude použita metoda základní třídy.

Vzorový příklad statické vazby

```
class Rodic {
    int r;
public:
    Rodic(int a):r(a) {};
    Rodic() {r = 0;}
    ~Rodic() {
        cout<<"\nmazani rodice"; }
    void tisk () {
        cout<<"\nrodic - hodnota:"<<r; }
};

class Potomek: public Rodic {
    int p;
public:
    Potomek(int a):p(a) {};
    ~Potomek() {
        cout<<"\nmazani potomka";};
    void tisk() {
        cout<<"\npotomek "
            "hodnota: "<< p;}
};
```

```
int main() {
    Rodic * pole[2];
    pole[0] = new Rodic(10);
    pole[1] = new Potomek(100);
    pole[0]->tisk();
    pole[1]->tisk();
    delete pole[0];
    delete pole[1];
    system("PAUSE");
    return 0;
}
```

```
rodic - hodnota:10
rodic - hodnota:0
mazani rodice
mazani rodicePokračujte
```

Vzorový příklad dynamické vazby

```
class Rodic {
    int r;
public:
    Rodic(int a):r(a) {};
    Rodic() {r = 0;}
    virtual ~Rodic() {
        cout<<"\nmazani rodice";}
    virtual void tisk () {
        cout<<"\nrodic - hodnota:"<<r;}
};
```

```
class Potomek: public Rodic {
    int p;
public:
    Potomek(int a):p(a) {};
    ~Potomek() {
        cout<<"\nmazani potomka";}
    void tisk() override {
        cout<<"\npotomek hodnota:"<<p;}
};
```

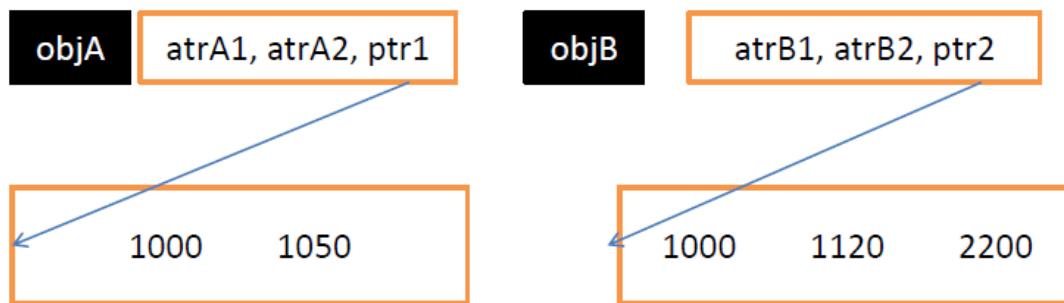
```
int main() {
    Rodic * pole[2];
    pole[0] = new Rodic(10);
    pole[1] = new Potomek(100);
    pole[0]->tisk();
    pole[1]->tisk();
    delete pole[0];
    delete pole[1];
    system("PAUSE");
    return 0;
}
```

```
rodic - hodnota:10
potomek hodnota:100
mazani rodice
mazani potomka
mazani rodicePokračujte stí
```

Virtuální funkce – tabulka funkcí VMT

- Statická vazba zná adresu volání členské metody již v době překladu.
- Dynamická vazba využívá tabulku virtuálních metod – zde za běhu programu najde adresu členské metody.
- Každý objekt má takovou tabulku – přístupná je přes skrytý ukazatel.
- Tabulka obsahuje adresy virtuálních metod definovaných ve třídě a adresy metod, které překrývají virtuální metody.

```
class A {  
public:  
    virtual void a1() {};  
    virtual void a2() {};  
};  
class B : public A {  
public:  
    void a2()  
    virtual void b1{};  
};  
A objA;  
B objB;
```



Virtuální metody

- Konstruktor nemůže být virtuální – není děděn, lze ho použít jen v konstruktorech.
- Destruktor by měl být vždy virtuální – uvolnění případné dynamické paměti správného objektu.
- Spřátelená funkce nemůže být virtuální – není členskou metodou.
- Není-li metoda překryta v potomkovi, je použita virtuální metoda základní třídy.
- Pozn.: při volání virtuálních metod v konstruktoru se tyto volají nevirtuálně.
- Příklad:

```
class Savec {  
private:  
    string jmeno;  
    int vek;  
...};  
class Clovek : public Savec {  
private:  
    string prijmeni;  
...}
```

Použití

```
Savec s("Zak",10);  
Clovek c("Ivan",30,"Hrozny");
```

```
Savec temps = s;          /* lze - pokud dynamická data -> kopírující  
                           konstruktor */
```

```
Clovek tempc = c;        /* lze - využít kopírující konstruktor  
                           rodiče a dodělat kopírování  
                           pro vlastní dynamická data */
```

```
Savec tempsavec = c;     /* lze - pokud dynamická data -> kopírující  
                           konstruktor */
```

```
Clovek tempclovek = s;  /* nelze -> konverzní funkce = konstruktor s  
                           1 argumentem */
```

```
Savec sa1; Clovek cl1;
```

```
sa1 = s;                 // lze - pokud dynamická data -> operátor přiřazení
```

```
cl1 = c;                 /* lze - pokud dynamická data - využít operátor  
                           přiřazení základní třídy a přiřadit vlastní data  
                           v operátoru přiřazení */
```

```
sa1 = c;                 // lze - pokud dynamická data -> operátor přiřazení
```

```
cl1 = s;                 // nelze -> konverzní funkce
```

Řízení možností dědění (final, override)

```
class A {
    virtual void foo() final; // A::foo je finální
    void bar() final; // Chyba: nevirtuální metoda nemůže být finální
};

class B final : A // Třída B je finální
{
    void foo(); // Chyba: metoda foo nemůže být přepsána, neboť je finální v A
};

class C : B // Chyba: B je finální, nelze z ní dědit
{};

class D : A {
    void foo() const override; // Chyba: B::foo nemůže přepsat A::foo
                                // (má jinou signaturu)
    void foo() override; // OK: B::foo přepíše (overrides) A::foo
    void bar() override; // Chyba: A::bar není virtuální
};
```

Kopírování a přiřazení v dědičnosti

```
class Savec {
private:
    char *jmeno;
    int vek;
public:
    ~Savec() {delete [] jmeno; cout<<"deIS";};
    Savec() : jmeno(0), vek(0) {};
    Savec(char * jm, int ve) : vek(ve) {
        jmeno = new char [strlen(jm)+1];
        strcpy(jmeno,jm);
    }
    void tisk() {cout<<"\n"<<jmeno<<" " <<vek;}
    Savec(const Savec & s) {
        jmeno = new char [strlen(s.jmeno)+1];
        strcpy(jmeno,s.jmeno);
        vek = s.vek;
    }
    Savec & operator=(const Savec & s) {
        if (this == &s) return *this;
        delete [] jmeno;
        jmeno = new char [strlen(s.jmeno)+1];
        strcpy(jmeno,s.jmeno);
        vek = s.vek;
        return *this;
    }
};
```

```
class Clovek : public Savec {
private:
    char * prijmeni;
public:
    ~Clovek() {delete [] prijmeni; cout<<"deIC";};
    Clovek() {prijmeni = 0;};
    Clovek(char *jm, int ve, char *pr):Savec(jm,ve) {
        prijmeni = new char [strlen(pr)+1];
        strcpy(prijmeni,pr);
    }
    Clovek (const Savec &s) : Savec(s) { // podivné
        prijmeni = new char [6];
        strcpy(prijmeni,"NEDEF");
    }
    Clovek (const Clovek & c) : Savec(c) {
        prijmeni = new char [strlen(c.prijmeni)+1];
        strcpy(prijmeni,c.prijmeni);
    }
    Clovek & operator=(const Clovek & c) {
        if (this == &c) return *this;
        Savec::operator =(c);
        delete [] prijmeni;
        prijmeni = new char [strlen(c.prijmeni)+1];
        strcpy(prijmeni,c.prijmeni);
    }
    void tisk() {
        Savec::tisk();
        cout << " pr: " << prijmeni;
    }
};
```

Zapouzdření programu

```
//osoba.h
class Osoba {
private:
    string jmeno;
    int vek;
public:
    Osoba() {}
    int getVek() {return vek;}
    string getJmeno() {return jmeno;}
    Osoba(string jm, int ve):vek(ve) {
        jmeno=jm;}
    void tisk() {
        cout<<"\n"<<jmeno<<" "<<vek;}
};
// případně osoba.cpp
```

```
// seznamosob.h
class SeznamOsob {
    Osoba *pole_osob[10];
    int pocet_osob;
public:
    SeznamOsob() { pocet_osob = 0; }
    int Pridej_osobu();
    void tisk();
    ~SeznamOsob();
};
```

Zapouzdření programu (seznam osob)

```
// seznamosob.cpp
int SeznamOsob::Pridej_osobu() {
    string jm;
    int ve;
    cout << "\nZadej jmeno: "; cin >> jm;
    if (jm == "konec") return 1;
    cout << "\nZadej vek: "; cin >> ve;
    pole_osob[pocet_osob++] = new Osoba(jm,ve);
    return 0;
}

void SeznamOsob::tisk() {
    for (int i = 0; i < pocet_osob; i++) {
        cout << "\n os jmeno: " << pole_osob[i]->
            getJmeno() << "\n vek: „
            << pole_osob[i]->getVek();
    }
}

SeznamOsob::~SeznamOsob() {
    for (int i = 0; i < pocet_osob; i++)
        delete pole_osob[i];
}
```

```
// main.cpp

#include "seznamosob.h"

int main() {
    SeznamOsob osoby;
    while (1) {
        if (osoby.Pridej_osobu())
            break;
    };
    osoby.tisk();
    return 0;
}
```

Abstraktní třídy

- Generalizace – dalším nástrojem pro její dosažení jsou abstraktní třídy (AT).
- AT – vytváří rozhraní, pomocí kterého je řízeno chování podobných objektů.
- Hledají se společné vlastnosti různých objektů.
- Hledají se metody – rozhraní pro práci s nimi:
 - např. zobrazení objektů na cílovou plochu (zobrazit),
 - není podstatné, jaká přesně bude (obrazovka, tiskárna) → AT ZobrazovacíZarizeni,
 - např. tiskárny → AT Tiskarna,
 - není podstatná technologie tisku (tisk).
- Rozšiřuje princip dědičnosti – AT je základní třídou.

Abstraktní třídy (pokr.)

- AT – hlavní motivace je tvorba rozhraní, jehož implementace se upřesňuje dle typu objektu.
- Příklad : všechny grafické objekty je možné vykreslit.
- Implementace společného předka AT → GeoUtvar.
- Do rozhraní AT zahrneme metody, které mají příslušníci třídy obecně společné, ale rozdílně implementované.
- Veřejné metody vytvoří rozhraní AT, např. tisk – všechny musí jít zobrazit, ale každý rozdílně.
- Potomci AT implementují toto chování.
- AT nemá důvod pro existenci vlastní instance.
- Protože je ale použito principu dědičnosti je vždy při vytváření potomka volán konstruktor rodiče.

Abstraktní třídy (pokr.)

- Pro vytvoření AT je nutné implementovat minimálně jednu čistě virtuální metodu.
- Ta má v AT pouze svoji deklaraci, implementace je v potomcích.
- Předpis:
`virtual navrat_typ metoda (argumenty) = 0;`
- Potomek překrývá tyto virtuální metody = dynamická vazba.
- AT nevytváří svoje instance – nelze z ní vytvořit objekt.
- AT – může obsahovat implementace některých metod.
- AT – může obsahovat i společná data.
- Pokud jsou všechny metody čistě virtuální = čistě abstraktní třída.
- Čistě virtuální metoda umožní definovat u předka metodu bez implementace a tím vynutí její implementaci v potomcích.

Abstraktní třída GeoUtvár

- Pro konstrukci některých geometrických útvarů lze použít vždy souřadnice středu a vzdálenost k hranicím. U všech lze vypočítat obsah.

```
class GeoUtvár {
private:
    int m_x;
    int m_y;
    static int poc;
public:
    GeoUtvár(int,int);
    virtual double obsah() = 0;
    virtual ~GeoUtvár();
    friend float vzdalenost(GeoUtvár * g1,
                           GeoUtvár * g2) {
        return sqrt(
            (pow((float)g2->m_x-g1->m_x,2)+
             pow((float)g2->m_y-g1->m_y,2)
            ));
    }
};

int GeoUtvár::poc = 0; // statická proměnná
```

```
class Ctverec : public GeoUtvár {
private:
    int m_vzd;
public:
    double obsah() override;
    Ctverec(int, int, int);
};

class Obdelnik : public GeoUtvár {
private:
    int m_vzd1;
    int m_vzd2;
public:
    double obsah() override;
    Obdelnik(int, int, int, int);
};
```

Abstraktní třída GeoUtvár (pokr.)

- Pro konstrukci některých geometrických útvarů lze použít vždy souřadnice středu a vzdálenost k hranicím. U všech lze vypočítat obsah.

```
GeoUtvár::GeoUtvár(int a, int b) :  
    m_x(a), m_y(b) {  
    poc++;  
};  
  
Ctverec::Ctverec(int a, int b, int c) :  
    GeoUtvár(a,b), m_vzd(c) {  
};  
  
GeoUtvár::~~GeoUtvár() {  
    poc --;  
    cout<<"\n pocet obj : "<<poc;  
};
```

```
Obdelnik::Obdelnik(int a,int b,int c,int d) :  
    GeoUtvár(a,b), m_vzd1(c), m_vzd2(d) {  
};  
  
double Ctverec::obsah() {  
    return (4*m_vzd*m_vzd);  
};  
  
double Obdelnik::obsah() {  
    return (2*m_vzd1*2*m_vzd2);  
};
```

Abstraktní třída GeoUtvar (pokr.)

```
int main(int argc, char* argv[]) {
    GeoUtvar* uk[2];
    uk[0] = new Ctverec(2,2,2);
    uk[1] = new Obdelnik(3,3,2,3);
    for(int i = 0; i<2; i++) {
        cout << "\n Obsah geoutvaru je: " << uk[i]->obsah();
    }
    cout << "\n vzdalenost: " << vzdalenost(uk[0],uk[1]);
    for(int i = 0; i<2; i++) {
        delete uk[i];
    }
    return 0;
}
```

Čistě abstraktní třída

- Čistě abstraktní třída nemusí mít členská data, jen metody.
- Nemusí mít konstruktor.
- Musí ale mít minimálně jednu čistě virtuální metodu.

```
class GeometrickyUtvar {
public:
    virtual float obvod() = 0;
};

class Kruh : public GeometrickyUtvar {
    float r;
public:
    Kruh(float polomer):r(polomer){};
    float obvod() override; // lze i virtual
};

class Obdelnik : public GeometrickyUtvar {
    float a,b;
public:
    Obdelnik(float s1, float s2):a(s1),b(s2){};
    float obvod() override;
};
```

Čistě abstraktní třída (pokr.)

```
float Kruh::obvod() { return 2*3.14*r; }
float Obdelnik::obvod() { return 2*(a+b); }

int main(void) {
    GeometrickyUtvar *k = new Kruh(1);
    GeometrickyUtvar *o = new Obdelnik(2,10);

    cout<< " obvod je " << k->obvod();
    cout<< " obvod je " << o->obvod();
    delete k;
    delete o;

    return 0;
}
```

Vícenásobná dědičnost

```
class A {
public:
    int atribut;
    void nastav(int x) { atribut = x; }
};

class B {
public:
    int atribut;
    void nastav(int x) { atribut = x; }
};

class C : public A, public B {
public:
    void nuluj();
};

void C::nuluj()
{
    atribut = 0; // Chyba - nejednoznačnost
};
```

The End