

6

Polymorfizmus (virtualita)

Polymorfizmus

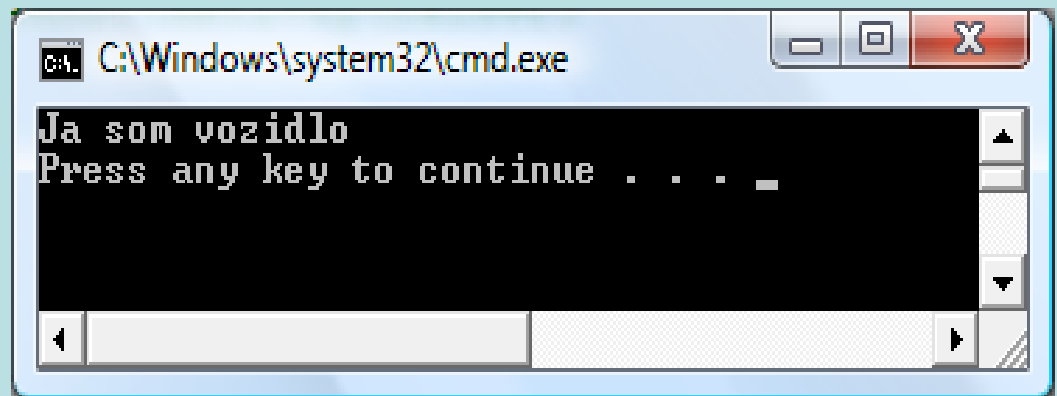
- Mechanizmus, ktorý umožňuje zmeniť spôsob fungovania metód základnej triedy
- Implementovaný prostredníctvom virtuálnych funkcií
- Virtuálna funkcia dovoľuje jednému typu vyjadriť svoje odlišnosti od iného, podobného typu, pokiaľ sú obidva typy odvodené z tej istej základnej triedy. Tento rozdiel je vyjadrený prostredníctvom rozdielov v správaní funkcií, ktoré voláme prostredníctvom základnej triedy.
- Dovoľuje vylepšovať organizáciu a čitateľnosť kódu
- Umožňuje tvorbu rozširovateľných programov, ktoré sa dajú rozširovať nielen počas počiatočného vývoja projektu, ale i v prípade požiadaviek na nové vlastnosti.

Pretypovanie smerom nahor (upcasting)

```
class Vozidlo {  
public:  
    void Identifikuj() const { printf("Ja som vozidlo\n"); }  
};  
// Objekt triedy Autobus je Vozidlo - majú rovnaké rozhranie:  
class Autobus : public Vozidlo {  
public:  
    // Predefinovanie funkcie rozhrania:  
    void Identifikuj() const { printf("Ja som autobus\n"); }  
};
```

```
void KtoSom(Vozidlo &i)  
{  
    i.Identifikuj();  
}  
int main()  
{
```

```
    Autobus zaa100;  
    KtoSom(zaa100); // Pretypovanie nahor
```



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window has a black background and white text. It displays the output of the program: "Ja som vozidlo" followed by "Press any key to continue . . . _". The cursor is positioned at the end of the second line. The window has standard Windows window controls (minimize, maximize, close) in the top right corner.

Väzba volania funkcie

- Prepojenie volania funkcie s telom funkcie sa nazýva **väzba**.
 - **včasná väzba** - vytvára sa pred odštartovaním programu (kompilátorom a spojovacím programom-linkerom)
 - **neskorá väzba** - vytvorí sa až za chodu programu, pričom bude založená na **type objektu** (*dynamická väzba, väzba za chodu programu*)

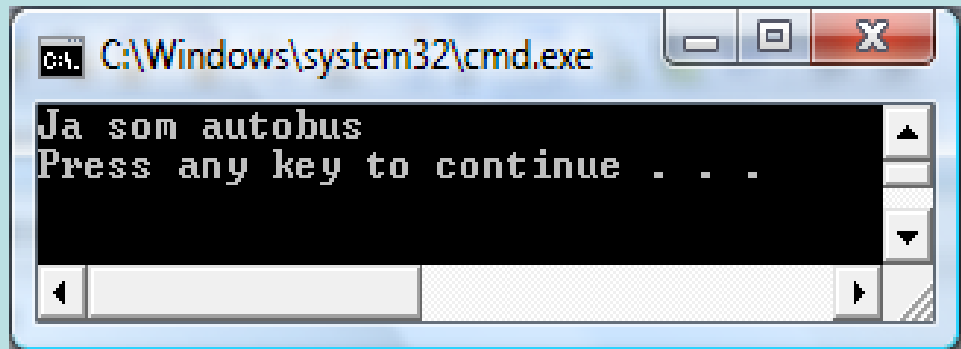
Virtuálne metódy

- Ak chceme C++ donútiť, aby konkrétne metódy používali neskorú väzbu, **musíme** pri deklarácií metódy v základnej triede použiť kľúčové slovo **virtual**
- Kľúčové slovo **virtual** sa zadáva **len v deklarácií metódy, nie v definícií**
- Ak je metóda deklarovaná ako **virtual** v základnej triede, potom bude **virtual** vo všetkých odvodených triedach. Všetky metódy v odvodených triedach, ktoré majú zhodnú signatúru s deklaráciou v základnej triede, sa budú volať použitím virtuálneho mechanizmu.
- Predefinovanie virtuálnej metódy v odvodenej triede sa zvyčajne nazýva **prevažovanie**

Pretypovanie smerom nahor (upcasting)

```
class Vozidlo {  
public:  
    virtual void Identifikuj() const { printf("Ja som vozidlo\n");}  
};  
// Objekt triedy Autobus je Vozidlo - majú rovnaké rozhranie:  
class Autobus : public Vozidlo {  
public:  
    // Predefinovanie funkcie rozhrania:  
    virtual void Identifikuj() const { printf("Ja som autobus\n"); }  
};
```

```
void KtoSom(Vozidlo &i)  
{  
    i.Identifikuj();  
}  
int main()  
{  
    Autobus zaa100;  
    KtoSom(zaa100); // Pretypovanie nahor  
}
```



Rozšiřovatelnost

```
class Vozidlo {  
public:  
    virtual void Identifikuj() const { printf("Ja som vozidlo"); }  
    virtual char* Druh() const { return "Vozidlo"; }  
    virtual void Nastav(int) {}  
};
```

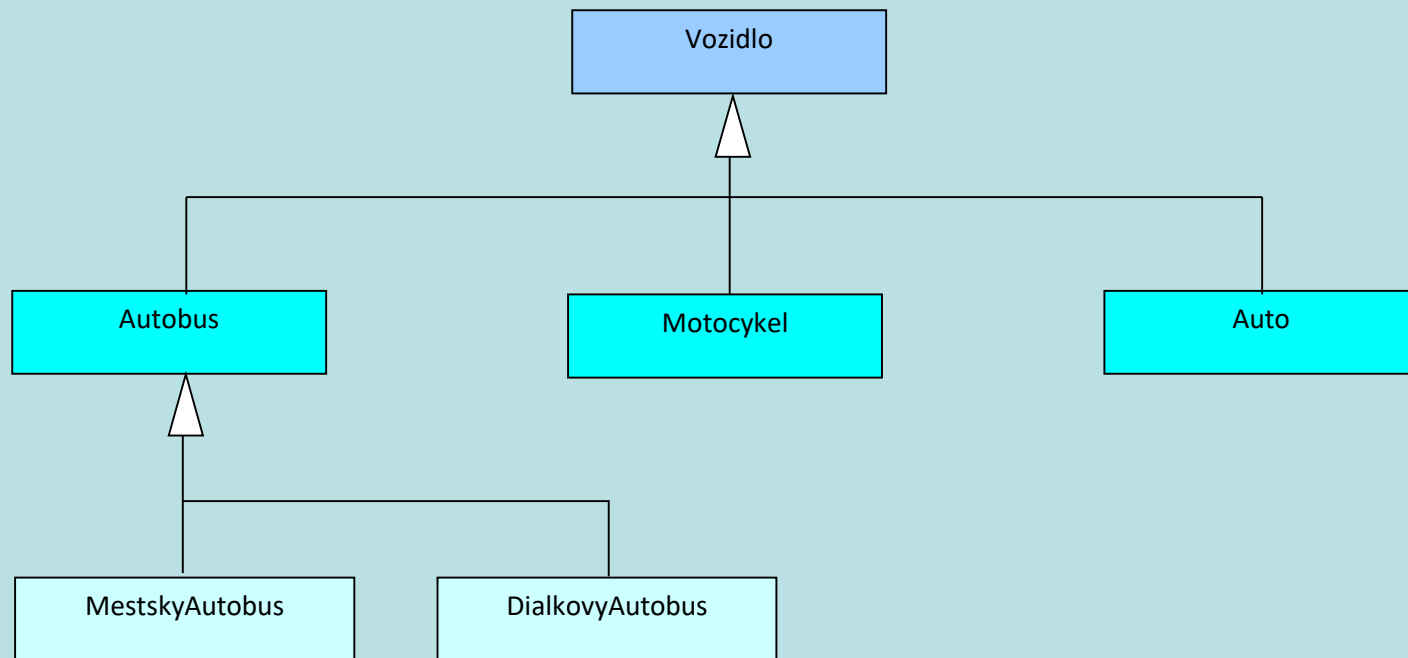
```
class Motocykel : public Vozidlo {  
public:  
    void Identifikuj() const {printf("Ja som motocykel"); }  
    char* Druh() const { return "Motocykel"; }  
    void Nastav(int) {}  
};
```

```
class Auto : public Vozidlo {  
public:  
    void Identifikuj() const { printf("Ja som auto"); }  
    char* Druh() const { return "Auto"; }  
    void Nastav(int) {}  
};
```

Rozšiřovatelnost

```
class MestskyAutobus : public Autobus {  
public:  
    void Identifikuj() const {  
        printf("Ja som mestsky autobus");  
    }  
    char* Druh() const { return "Mestsky autobus"; }  
};  
  
class DialkovyAutobus : public Autobus {  
public:  
    void Identifikuj() const {  
        printf("Ja som dialkovy autobus");  
    }  
    char* Druh() const { return "Dialkovy autobus"; }  
};
```


Hierarchia

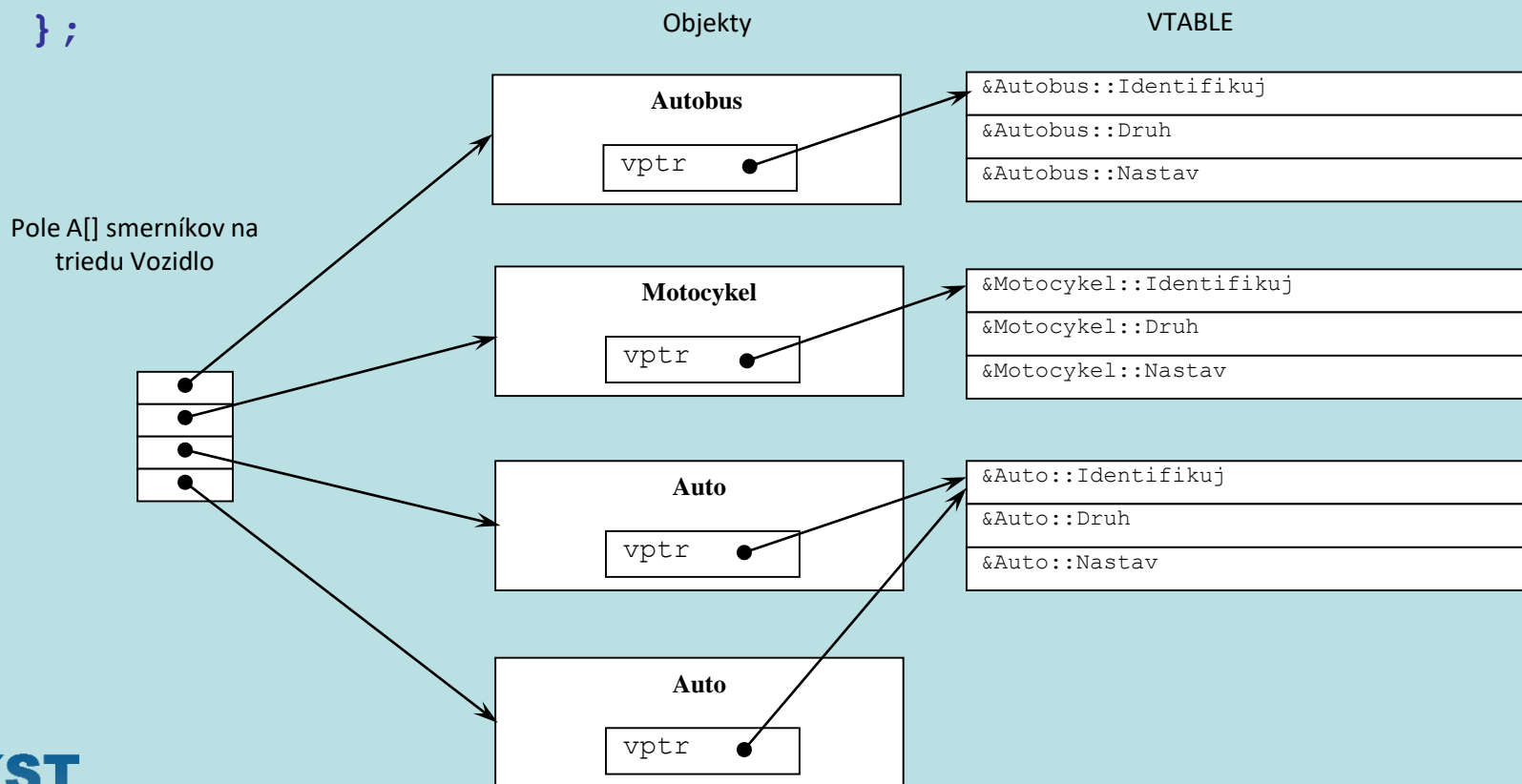


C++ a implementácia neskorej väzby

- Kompilátor vytvára pre každú triedu jednu tabuľku (nazývanú VTABLE), ktorá obsahuje virtuálne funkcie.
- V každej triede, ktorá obsahuje aspoň jednu virtuálnu funkciu, je skryté uložený smerník, nazývaný **vpoiner** (skrátene VPTR), ktorý ukazuje na VTABLE triedy.
- Keď zavoláme virtuálnu funkciu prostredníctvom smerníka na základnú triedu (t.j. keď urobíme tzv. **polymorfné volanie**), kompilátor potichu vygeneruje kód na výber VPTR a vyhľadanie adresy funkcie vo VTABLE, čím sa zavolá správna funkcia a realizuje sa neskorá väzba.

Zobrazenie virtuálnych funkcií

```
Vozidlo* A[] = {  
    new Autobus(),  
    new Motocykel(),  
    new Auto(),  
    new Auto()  
};
```

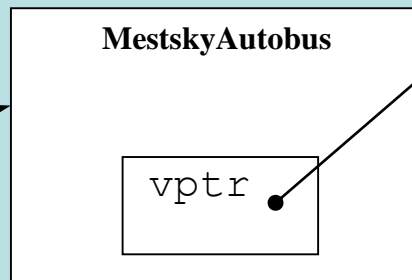
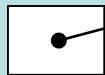


Volanie virtuálnej funkcie

- `p->Nastav(3);`

Smerník na
triedu Vozidlo

Vozidlo* p



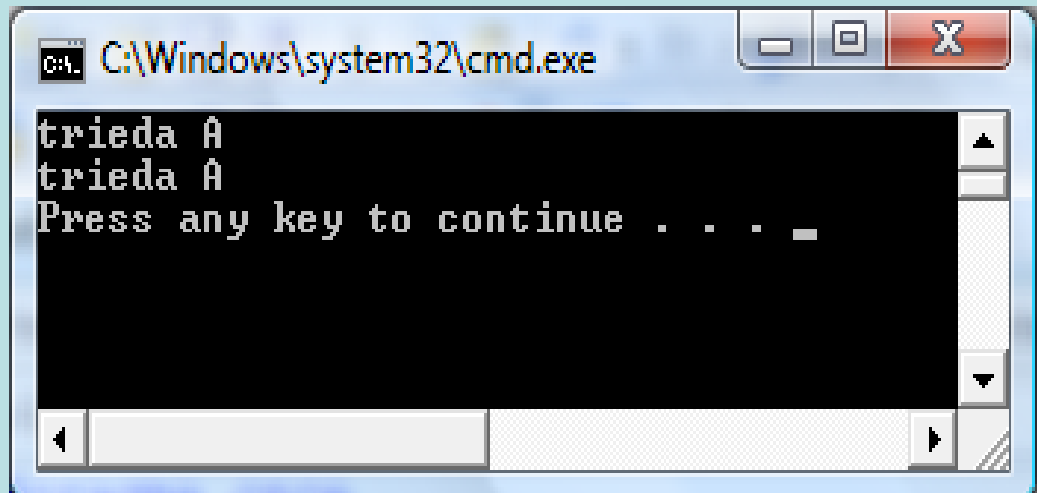
VTABLE triedy

MestskyAutobus

[0]	<code>&MestskyAutobus::Identifikuj</code>
[1]	<code>&MestskyAutobus::Druh</code>
[2]	<code>&MestskyAutobus::Nastav</code>

Príklad

```
class A {  
protected:  
    const char* ToString() { return "trieda A"; }  
public:  
    void Print() { cout << ToString() << '\n'; }  
};  
  
class B : public A {  
protected:  
    const char* ToString() { return "trieda B"; }  
};  
  
int main()  
{  
    A a;  
    B b;  
    a.Print();  
    b.Print();  
    return 1;  
}
```



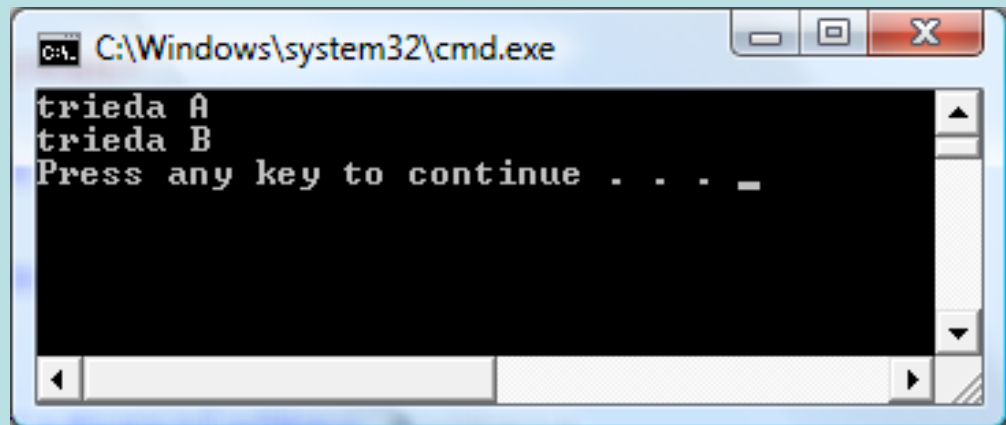
The screenshot shows a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window has a black background with white text. The output of the program is displayed as follows:

```
trieda A  
trieda A  
Press any key to continue . . .
```

The text "trieda A" is printed twice, corresponding to the calls to `a.Print()` and `b.Print()` in the `main` function. The prompt "Press any key to continue . . ." is shown at the bottom, indicating that the program has finished execution and is waiting for a key press to close the window.

Virtuálne

```
class A {  
protected:  
    virtual const char* ToString() { return "trieda A"; }  
public:  
    void Print() { cout << ToString() << '\n'; }  
};  
  
class B : public A {  
protected:  
    virtual const char* ToString() { return "trieda B"; }  
};  
  
int main()  
{  
    A a;  
    B b;  
    a.Print();  
    b.Print();  
    return 1;  
}
```



The screenshot shows a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window has a black background and white text. The output of the program is displayed as follows:

```
trieda A  
trieda B  
Press any key to continue . . . _
```

Abstraktná trieda

- Abstraktná trieda je trieda s aspoň jednou čisto virtuálnou metódou
- Čisto virtuálna metóda je virtuálna metóda, ktorej v prototype priradíme nulu

virtual int metoda(int param) = 0;

- Z abstraktnej triedy nie je možné vytvoriť objekt
- Prinútenie potomka definovať metódu
- Často sa používa pre definovanie **rozhrania** (interface)

Konštruktory a deštruktory

- Virtualita nefunguje v konštruktore a deštruktore – volá lokálnu funkciu
- Konštruktor nemôže byť virtuálny
- Virtuálny deštruktor

Dynamická identifikácia

```
#include <iostream>
#include <typeinfo>
using namespace std;
class A
{
    virtual void f(){}
};
class B : public A {};
void main()
{
    B b1;
    A* a1=&b1;
    cout<<typeid(*a1).name()<<endl;
}
```

- Typeid je typu type_info. Má definované operátory == a !=.
- Musí byť zapnuté Run-Time Type Information (RTTI)
- Vypíše „class B“
- Zapoznámkovaním virtual funkcie vypíše „class A“
- Ak sa nedá určiť typ (napr. je null) vyhodí sa výnimka „bad_typeid“

Pretypovanie

```
#include <iostream>

using namespace std;

class A
{
    virtual void f() {}
};

class A1 : public A {};

class A2 : public A {};

class B {};
```

```
void main()
{
    A* a=new A;
    A* x;
    A1* x1;
    B* b;
    b=reinterpret_cast<B*>(a);
    A1* a1=new A1;
    A2* a2=new A2;
    //b=static_cast<B*>(a);           // chyba pri preklade
    //a2=static_cast<A2*>(a1);        // chyba pri preklade
    x=static_cast<A*>(a1);           // nemá veľký význam, lebo to je ako x=a1;
    cout<<x<<endl;
    x1=static_cast<A1*>(a);          // neciste
    cout<<x1<<endl;
    x=dynamic_cast<A*>(a1);
    cout<<x<<endl;
    x1=dynamic_cast<A1*>(a);         // ciste
    cout<<x1<<endl;
    x1=(A1*)(a);
    cout<<x1<<endl;
    getchar();
}
```

- reinterpret_cast je obecné „nebezpečné“ pretypovanie na najnižšej úrovni – natvrdo
- static_cast je pretypovanie z potomka na predka (nemá moc význam) aj z predka na potomka ale staticky – nebezpečné. Nedá sa pretypovať A* na B*, ak nie sú predok - potomok.
- dynamic_cast je dynamické pretypovanie z predka na potomka pomocou RTTI. Ak sa nedá pretypovať, vracia „0“.