Zpracoval: houzvjir@fel.cvut.cz

9. Datový typ ukazatel, přetěžování funkcí, typ reference, vstup a výstup, třídy, staticky vázané metody, dědění, dynamicky vázané metody, abstraktní třídy, polymorfní datové struktury. (A7B36PJC)

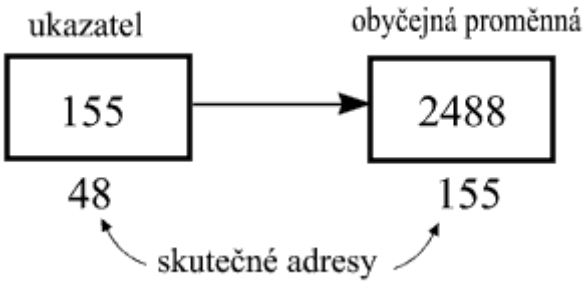
Obsah

Datový typ ukazatel 2
Typ reference 3
Přetěžování funkcí 4
Vstup a výstup 4
Soubory 7
Třídy 8
Staticky vázané metody 13
Dědění 13
Dynamicky vázané metody 15
Abstraktní třídy 15
Polymorfní datové struktury 16

Datový typ ukazatel

Vedle obvyklého přístupu k datům v paměti pomocí proměnných, nabízí jazyk C i možnost pracovat s těmito daty nepřímo pomocí ukazatelů.

Ukazatel je v podstatě proměnná nebo konstanta jako každá jiná. Liší se jen způsob, jak se s ní pracuje. Klasická proměnná představuje určitý kus paměti, ve které je uložena hodnota této proměnné. To platí i pro pointery, jenže v jejich případě je touto hodnotou adresa nějakého místa v paměti. Říkáme pak, že pointer na místo určené touto adresou ukazuje. Samotná adresa, kterou pointer uchovává je pro nás většinou nezajímavá, důležitější je to, co se na této adrese nachází.



Bázový typ ukazatele

Bázový typ ukazatele určuje, jak se bude interpretovat místo v paměti, kam tento pointer ukazuje. Například, je-li bázovým typem ukazatele typ int, znamená to, že obsah paměti, kam pointer ukazuje, se bude interpretovat jako datový objekt typu int. Bázový typ ukazatele určujeme při jeho definici, je ale možné určit ho až za běhu programu.

Definice proměnné typu ukazatel:

bázový_typ *identifikátor;

Ukazatele definujeme podobně jako jiné proměnné. Rozdílem je, že před identifikátorem musíme uvést ještě znak ‘*‘, kterým dáme překladači vědět, že definujeme pointer (tj. ukazatel).

Potřebuji-li pracovat s hodnotou, na kterou ukazuje pointer, použiji dereferenci. To provedu tak, že před název proměnné typu pointer předřadím znak „*“.

int a, *b=3;

a = *b; `// *b je dereference – do a vloží hodnotu 3 z adresy, na kterou ukazuje b`

Příklady deklarací:

```
int * a; // ukazatel na typ int
const int * b; // ukazatel na konstantu typu int
int * const c; // konstantní ukazatel na typ int
const int * const d; // konstantní ukazatel na konstantu typu int
```

int * e (void); // funkce e vrací ukazatel na typ int
int (*f)(void); // ukazatel na funkci
int *g [20]; // pole 20 ukazatelů na int
int (*h)[40]; // ukazatel na pole 40 int
int (*(i(void)))(int) // funkce vrací ukazatel na funkci
int (*j[30])(int) // pole 30 ukazatelu na funkci
int (*(*k)[50])(int) // ukazatel na 50 prvkove pole ukazatelu na funkci

Příklady práce s pointery:

```
int a[5] = {1,2,3,4,5};
int *b, c, *d;

b = &a[2]; // priradi do b ukazatel na polozku s indexem 2 v poli a
c = b[2];  // do c vlozi hodnotu 5
// = mohu indexovat promennou deklarovanou jako pointer

d = a+4; // d nyni ukazuje na hodnotu 5 v poli a
// = s pointery mohu provadet matematicke operace
d = &a[4]; // stejny vysledek jako predchozi radka
```

Typ reference

Typ reference je podporován jen v C++, používá se hlavně pro předávávání parametru odkazem. Rozdíl oproti pointeru je, že nelze změnit adresu, na kterou ukazuje – reference je tedy bezpečnější než pointer. Hned při deklaraci musí být inicializován na již existující proměnnou. Není třeba dereferencovat pomocí *, dereference probíhá automaticky.

Příklad:

```
void plus_jedna(int & r_i) { // v parametru je
      r_i++;
}

int _tmain(int argc, _TCHAR* argv[])
{
  int a=1;
  plus_jedna(a);
  printf("a=%d",a); // vystupem je a=2
  return 0;
}
```

Příklad:

```
int _tmain(int argc, _TCHAR* argv[])
{
  int a=2;
  int *b = &a; // do b vlozi pointer na a
  int &c = *b; // reference na int, na ktery ukazuje b, cili vlastne na a
  c++;
  printf("a=%d",a); // vypise a=3
```

```
  return 0;
}
```

Přetěžování funkcí

Přetížení funkce (anglicky function overloading) je v informatice taková metoda zápisu zdrojového kódu programu, kdy je deklarováno více funkcí pod stejným názvem, přičemž se navzájem liší ve struktuře seznamu parametrů (počet parametrů nebo datovým typem). Při volání funkce překladač analyzuje parametry a podle toho určí odpovídající funkci. Přetížení se týká i návratové hodnoty, překladač analyzuje typ požadované návratové hodnoty na levé straně přiřazovacího operátoru = a podle toho vybere příslušnou funkci.

Nelze přetěžovat na základě návratové hodnoty.

Přetěžování funkcí je v praxi často používané jak ve strukturovaném, tak i v objektově orientovaném programování. Velice výhodné může být přetížení konstruktorů.

```
double prumer(double n1, double n2){
        return (n1+n2)/2;
}

double prumer(int n1, int n2){
        return (n1+n2)/2;
}

double prumer(int n1, int n2, int n3){
        return (n1+n2+n3)/3;
}

int main(){
        double d1, d2, pr;
        int i1, i2, i3;
        pr = prumer(i1, i2, i3); // Tady se vola treti funkce
        pr = prumer(d1, d2); // Tady se vola prvni funkce
        pr = prumer(i1, i2); // Tady se vola druha funkce
        return 0;
}
```

Vstup a výstup

Vstup a výstup - neformátovaný

- Reprezentace dat v paměti a na vstupu/výstupu je stejná.
- Čtecí/zápisové operace nezasahují do zpracovávané posloupnosti bajtů.
- Příklad – celá čísla:

Hodnota	**v paměti**	**v souboru**
1	00 00 00 01 (4B)	00 00 00 01 (4B)
1000	00 00 03 E8 (4B)	00 00 03 E8 (4B)
1000000	00 0F 42 40 (4B)	00 0F 42 40 (4B)

Vstup a výstup - formátovaný

- Paměťová (binární) reprezentace odlišná od textové reprezentace na vstupu/výstupu.
- Zápisové operace provádí konverzi z paměťové (binární) reprezentace na text.
- Čtecí operace převádějí textovou reprezentaci zpět na paměťovou binární.
- Ošetření chyb při konverzi.
- Příklad – celá čísla:

Hodnota	**v paměti**	**v souboru (např.)**
1	00 00 00 01 (4B)	31 (1B)
1000	00 00 03 E8 (4B)	31 30 30 30 (4B)
1000000	00 0F 42 40 (4B)	46 34 32 34 30 (5B)

http://statnice.stm-wiki.cz/doku.php?id=archiv:si:si10

- vstupní a výstupní operace jsou v C a C++ dány knihovnami
- v C++ je možné používat C funkce na vstup a výstup (následuje seznam některých C funkcí)
    - formátované funkce
        - **scanf**: int scanf (const char * format, &promenna1, &promenna2, …); *pro standardní vstup * fscanf, sscanf: obdoby pro čtení ze souboru a stringu * **printf**: int printf (const char * format, promenna1, promenna2, …);* pro standardní vstup
            - fprintf, sprintf: obdoby pro čtení ze souboru a stringu
    - neformátované
        - **getc**: přečte znak ze vstupu
        - **ungetc**: odpřečte znak ze vstupu (příští volání getc vrátí tento znak, pozor na opakované volání)
        - **putc**: zapíše znak na výstup
        - **fread**: přečte blok ze souboru
        - **fwrite**: zapíše blok dat do souboru
        - **feof**: test na konec souboru
        - **ferror**: test na chybu
        - **clearerr**: vynuluje chybové indikátory

Následující část kapitoly se zabývá hlavně C++.

- prostředkem pro vstup nebo výstup je **datový proud** (stream)
    - vstupní proud standardního vstupu je k dispozici pod názvem **cin**
    - výstupní proudy jsou:
        - **cout** na standardní výstup
        - **cerr** na chybový výstup
        - **clog** na výstup do logu
- datový proud (dále jen proud) realizuje tok dat od zdroje ke spotřebiči
    - je-li spotřebičem program, jde o vstupní proud (zdrojem může být soubor nebo řetězec)
    - je-li zdrojem program, jde o výstupní proud (spotřebičem může být soubor nebo řetězec)

- vstup a výstup může být formátovaný nebo neformátovaný
    - **formátovaný vstup**: zdrojová data jsou posloupnosti znaků, které se konvertují do vnitřní (binární) reprezentace
    - **formátovaný výstup**: zdrojová data ve vnitřní reprezentaci se konvertují na posloupnosti znaků
    - **neformátovaný vstup a výstup**: zdrojová data proudí do spotřebiče jako posloupnosti bytů

Třída istream

- Společné vlastnosti všech vstupních proudů definuje třída istream.
- Metody pro neformátovaný vstup:

```
int get()
istream& get(char *, int len, char = '\n')
istream& get(char&)
istream& getline(char *, int len, char = '\n')
istream& ignore(int n = 1, int delim = EOF)
int peek()
istream& putback(char)
istream& read(char *, int)
long tellg()
istream& seekg(long)
```

Příklady

Operátor » pro formátovaný vstup

```
istream& operator>>(istream&, T&) // pro všechny standarní typy T
```

Formátovaný vstup lze řídit manipulátory

```
int n;
...
cin >> oct >> n; // vstup v osmičkové soustavě
```

Chyby

Nastane-li při čtení ze vstupního proudu chyba, další vstupní operace s daným proudem se až do vymazání příznaku chyby ignorují.

```
bool fail()  // kontroluje zda-li se operace zdařila
void clear() // odstraní příznak chyby
```

Příklad čtení celého čísla s kontrolou chyby:

```
int ctiInt(istream& is)
{
  int x;
  bool chyba = false;

  cout << "zadejte cele cislo: ";
  do
  {
    is >> x;
    chyba = is.fail();
```

```
    if (chyba)
    {
      cout << "spatne zadane cislo, zadejte znovu\n";
      is.clear();
      is.ignore(100, "\n");
    }
  }
  while(chyba);

  return(x);
}
```

Soubory

- Konstruktory tříd **ifstream** a **ofstream** s jedním parametrem otevírají soubory jako textové.
- Textové soubory se člení na řádky.
    - Oddělovačem řádků v paměti (v řetězcích) je jediný znak '\n' s kódem 10 (LF)
    - Oddělovačem řádků v souboru je:
        - v OS Windows dvojice znaků CR LF s kódy 13 a 10
        - v OS Unix (Linux a pod.) je jediný znak LF
    - Rozdílné reprezentace oddělovače řádků v souboru řeší knihovny C++ tak, že v OS Windows:
        - Při čtení textového souboru se dvojice bytů s hodnotami 13 a 10 přečte jako jediný znak s kódem 10 (LF).
        - Při zápisu do textového souboru se znak s kódem 10 zapíše jako dvojice bytů s hodnotami 13 a 10.
- Při otevírání binárních souborů, jejichž obsahem nejsou znaky a nečlení se na řádky, je třeba výše uvedenou transformaci potlačit uvedením příznaku binary.

```
ofstream out("data.bin", ios::out|ios::binary);
```

Příklad

Zápis a přečtení binárního souboru po blocích.

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
  int pole[5] = {1,2,3,4,5};                        // vytvotri pole
  ofstream out("pole.bin",ios::out|ios::binary);    // otevte soubor
  out.write((const char*)pole,5*sizeof(int));       // zapise pole
  out.close();                                      // zavreni souboru

  for (int i=0; i<5; pole[i++]=0);                  // vynulovani pole

  ifstream inp("pole.bin",ios::in|ios::binary);     // otevreni pro cteni
  inp.read((char*)pole,5*sizeof(int));              // cteni
  inp.close();                                      // zavreni

  for (i=0; i<5; cout << pole[i++] << ' ');         // vypis pole

  return 0;
}
```

Třídy

- Třídy:
  - Reprezentace modelované skutečnosti v programu.
  - Soustředí související údaje do jednoho celku.
  - Uzavření před vnějším světem (omezení přístupu).
  - Rozhraní pro práci (metody).
  - Využitelné pro polymorfismus a dědění.
  - Pouze v C++.
  - V C třídy nejsou, pouze struktury (bez metod a řízení přístupu).
- Objekty:
  - instance nějaké (právě jedné) třídy
  - třída má typicky mnoho instancí

Příklad třídy v C++

```
class CCplx
 {
   public:
           CCplx(double r, double i=0) {re=r; im=i;}
           CCplx(const char * str) { … }
          ~CCplx(void) { }
    double Re    (void) {return re;}
    double Im    (void) {return im;}
    double Abs   (void) {return sqrt (re*re+im*im);}
    double Phi   (void) {return atan2 (im, re);}
   private:
    double re, im;
 };
```

Deklarace třídy v C++

- Konstruktor:
  - jméno stejné jako třída,
  - inicializace členských proměnných,
  - volán vždy při vzniku instance,
  - konstruktorů může být ve třídě více,
  - rozlišení – viz pravidla pro přetěžování funkcí.
- Destruktor:
  - jméno stejné jako třída s ~,
  - maximálně jeden,
  - volán vždy, když je odstraňována instance,
  - úklid členských proměnných.
- Řízení přístupu:
  - public: – neomezený přístup,
  - protected:– pouze pro metody třídy a metody potomků,
  - private: – pouze pro metody třídy.
- Viditelnost je daná blokově, podle poslední použité direktivy public/protected/private.
- implicitní viditelnost je private,
- Klíčové slovo class:
  - zahajuje deklaraci třídy,

Inline způsob deklarace třídy:

```
// inline zpusob
class CCplx
 {
   public:
            CCplx(double r, double i=0) {re=r; im=i;}
            CCplx(const char * str) { … }
           ~CCplx(void) { }
    double Re   (void) {return re;}
    double Im   (void) {return im;}
    double Abs  (void) {return sqrt (re*re+im*im);}
    double Phi  (void) {return atan2 (im, re);}
   private:
    double re, im;
 };
```

Deklarace třídy pomocí .h a .cpp souborů:

```
// .h soubor - deklarace, ne tela metod
class CCplx
 {
   public:
            CCplx(double r, double i=0);
            CCplx(const char * str)
           ~CCplx(void);
    double Re   (void);
    double Im   (void);
    double Abs  (void);
    double Phi  (void);
   private:
    double re, im;
 };
```

```
// .cpp soubor - definice tel metod
#include "CCplx.h"

        CCplx::CCplx(double r, double i)
 { // uz bez implicitnich hodnot parametru
   re = r;
   im = i;
 }

double CCplx::Re    (void)
 {
   return re;
 }

…
```

Konstruktory v C++

- Inicializace členských proměnných při vzniku instance.
- Výběr konstruktoru parametry v závorce při vytváření instance:

CCplx a (10, 20);

CCplx b ("10 + 2i");

Destruktory v C++

- Volán při zániku instance.
- Uvolnění prostředků, které objekt vlastnil:
  - dynamicky alokovaná paměť,
  - prostředky OS (soubory, sokety, thready, semafory, mutexy, ...).
- Není povinný, systém si "domyslí" prázdný destruktor, pokud neexistuje jiný.

Staticky vytvořená instance:

```
void foo1 ( CCplx & x )
 {
   cout << x . Abs ();
 }
void foo2 ( CCplx * x )
 {
   cout << x -> Abs ();
 }
void foo3 ( void )
 {
   CCplx a ( 10, 20 );    // konstruktor
   CCplx b ( "1 + 2i" ); // konstruktor

   foo1 ( a );
   foo2 ( &a );
   cout << a . Re ();     // pristup k metode pomoci .
 }  // destruktor a, destruktor b
```

Dynamicky vytvořená instance:

```
void foo1 ( CCplx * x )
 {
   ...
   cout << x -> Abs ();
 }

void foo2 ( void )
 {
   CCplx * a;
   CCplx b = new CCplx ( "1 + 2i" ); // konstruktor

   a = new CCplx ( 10, 20 ); // konstruktor
   foo1 ( a );
   cout << a -> Re ();  // pristup k metode pomoci ->
   delete a; // destruktor a
   delete b; // destruktor b
 }
```

- Oba operátory slouží pro:
  - přístup ke členským proměnným,
  - volání metod.
- Operátor . se použije pro:
  - práci přímo s instancí,
  - práci s referencí.
- Operátor -> se použije pro práci s ukazatelem na instanci.
- Platí:

```
X .  foo () <=> (&X) -> foo ();
Y -> foo () <=> (*Y) .  foo ();
```

Metody

- Konstantní metody
  - Konstantní metoda je metoda, která nemění vlastnost třídy (nemění datovou položky třídy) a nevolá nekonstantní metodu.
- Instanční metody
  - Metoda spuštěná nad konkrétní instancí
- Třídní metody
  - Spuštěná bez konkrétní instance
  - Má přístup pouze ke svým parametrům a ke globálním proměnným
  - Deklarace pomocí klíčového slova static
  - Volání třídní metody pomocí čtyřtečkové notace ::

Příklad na třídní metodu:

```
class CCplx
 {
   public:
    ...
    static CCplx Add ( const CCplx & a,
                       const CCplx & b )
     {
       return CCplx ( a.re + b.re, a.im + b.im );
     }
    ...
 };

CCplx u ( 2,3 ), v (4, 5), w;

w = CCplx::Add ( u, v );
```

Třídní proměné

- Instanční proměnné:
  - každý objekt má vlastní sadu instančních proměnných,
  - instanční proměnné různých objektů se vzájemně "nevidí".
- Třídní proměnné:
  - vázané na třídu,
  - v systému existuje právě 1x, bez ohledu na počet aktivních instancí.
- Deklarace třídní proměnné – klíčové slovo static.
- Zpřístupnění třídní proměnné – čtyřtečková notace ::
- Využití – hlavně pro tabulky předpočítaných hodnot.

Staticky vázané metody

jsou *časně* vázané metody (*early-bound*). Jejich volání je vyřešeno již při překladu. Technicky se volání statické metody neliší od volání procedury nebo funkce.

Dědění

Deklarace podtřídy

- podtřída má implicitně všechny vlastnosti jako nadřazená třída
- je možné přidat nové položky
- je možné přidat či předefinovat metody
- je možné omezit přístup k metodám nadřazené třídy
  - private změní vše na private
  - public zachová původní přístupnost
- Potomek může být přiřazen předkovi.
- Předek nemůže být přiřazen potomkovi.

```
class T1 : public T
{
  deklarace nových položek
  prototypy nových (předefinovaných) metod
};
```

V uvedeném příkladu je T1 podtřídou třídy T

Řízení přístupu a dědění

- přístupová práva mohou být děděním zachována nebo změněna (omezena)

```
class Podtrida: public Nadtrida { ... };
```

přístupová práva jsou zachována (nejčastější případ)

```
class Podtrida: protected Nadtrida { ... };
```

nebo *přístupová práva jsou omezena*

```
class Podtrida: private Nadtrida { ... };
```

	: public	: protected	: private
public:	public:	protected:	private:
protected:	protected:	protected:	private:
private:	*nepřístupné*	*nepřístupné*	*nepřístupné*

- typová kompatibilita mezi nadtřídou a podtřídou:

```
Nadtrida Nad, *uNad = &Nad;
Podtrida Pod, *uPod = &Pod;
Nad = Pod;          // O.K., přiřazeny jen zděděné položky
Pod = Nad;          // nelze: přidané položky by nebyly inicializované
uNad = uPod;        // O.K.
uNad = &Pod;        // také O.K.
uPod = uNad;        // nelze: byly by zpřístupněny neexistující položky
```

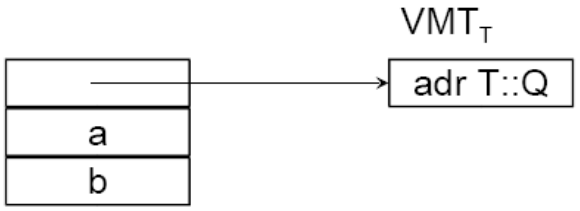
Dynamicky vázané metody

- Dynamicky vázané metody začínají klíčovým slovem **virtual**.
- Obvykle jsou vnitřně řešeny pomocí **tabulky virtuálních metod** *(VMT: Virtual Method Table)*. Ta pro každou hlavičku virtuální metody vytvoří odkaz na metodu. Příslušná metoda je na místo odkazu vyplněna až za běhu při spuštění konstruktoru příslušného objektu.
- Volání virtuální metody je časově mírně náročnější oproti staticky vázané.

Příklad:

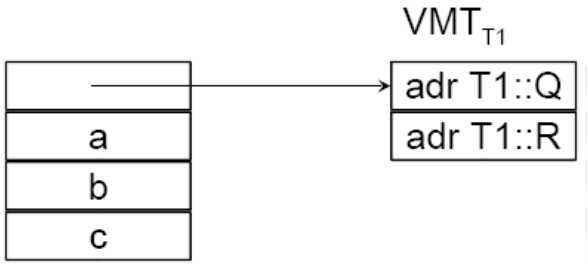
```
class T
{
  int a,b;

public:
  void P();
  virtual void Q();
};
```




```
class T1 : public T
{
  int c;

public:
  void P();
  virtual void Q();
  virtual void R();
  void S();
};
```




```
T *t = new T();
T *t1 = new T1();

t->P();  // zavola T::P()
t1->P(); // zavola T:P()
t->Q();  // zavola T:Q()
t1->Q(); // zavola T1:Q() (pomoci tabulky virtualnich metod)
t1->R(); // prekladova chyba, T nema metodu R
```

Abstraktní třídy

- jejich základní vlastností je, že **není možné z nich vytvářet instance** (volat konstruktor) a to ani staticky ani dynamicky
- lze ale pracovat s ukazateli a referencemi typu abstraktní třída
- mají alespoň jednu **abstraktní metodu**
    - abstraktní metody mouhou být pouze v abstraktních třídách - jakmile třída obsahuje alespoň jednu abstraktní metodu, stává se automaticky abstraktní
    - je dáno **pouze rozhraní metody** (jméno, parametry, …)
    - **tělo** metody je definováno **v potomcích**

    - metoda však **existuje i v předkovi** (aby se vyhradil prostor v VMT)
    - musí být vždy **virtual** (aby byla při volání byla zavolána správná metoda pomocí Dynamické vazby - viz výše)
    - lze vytvořit pouze abstraktní instanční metodu
    - nelze vytvořit abstraktní konstruktor, destruktor a třídní metodu
    - potomek musí implementovat všechny abstraktní metody, v opačném případě se jedná opět o abstraktní třídu

- může obsahovat implementaci metod, které jsou společné všem potomkům
- *použití*
    - výchozí bod (jakousi šablonu) pro složitější datové struktury (základ pro dědění)
    - poskytují jednotný pohled na více heterogenních objektů
    - využití rozhraní vyšší úrovně, není třeba rozlišovat detaily implementace v jednotlivých podtřídách - poskytují rozhraní společné všem potomkům
    - pro implementaci Polymorfních datových struktur - viz dále

```
class Shape
{
public:
  virtual int Area() = 0; // virtualni metoda
}
```

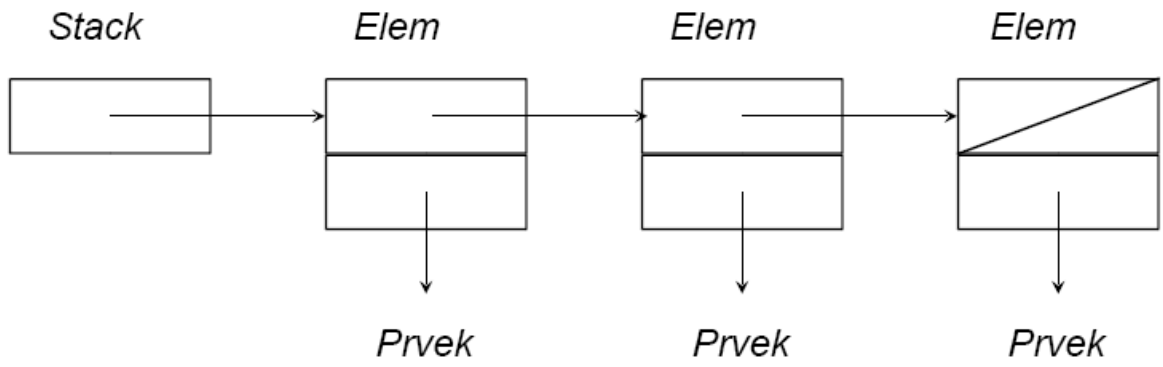
```
class Rectangle: Shape
{
  int a;
public:
  Rectanlge(int size) { a = size; }
  virtual int Area() { return a * a; }
}
```

Polymorfní datové struktury

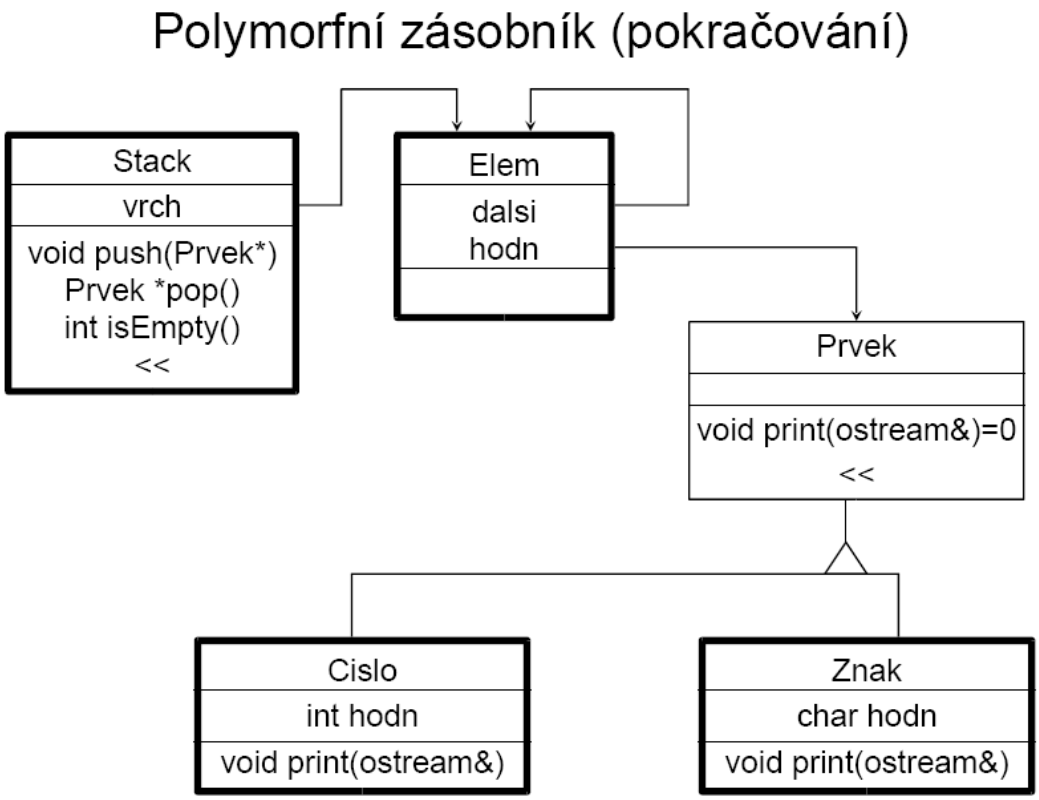
- jedná se o datové struktury, které v sobě mohou uchovávat instance odvozené od různých tříd
- typickým příkladem je třeba abstraktní datový typ ***zásobník, fronta, tabulka, množina, strom, halda*** apod
- datová struktura obsahuje **prvky**, které **podporují určité operace**, jinak řečeno implementují definové rozhraní
- polymorfní datové struktury jsou z pravidla **implementovány pomocí abstraktních tříd** (viz výše), které právě definují příslušné rozhraní. **Typy prvků**, které je taková struktura schopna ukládat jsou **podtřídami** (ve smyslu dědičnosti) dané abstraktní třídy
- *dělí se na*
    - **homogenní** - obsahují **prvky stejného** typu ⇒ operace se provádí pro všechny prvky stejně
    - **heterogenní** - obsahuje **prvky různých** typů ⇒ operace se mohou provádět různě v závislosti na typu prvku

Příklad - polymorfní zásobník

- implemtován spojových seznamem dynamických proměnných
- tyto proměnné obsahují ukazatel na prvek zásobníku, situace je znázorněna na následujícm obrázku



- typy prvků, které bude zásobník schopen uchovávat budou podtřídami abstraktní třídy CPrvek - v této třídě bude pro všechny prvky přetížen operátor výstupu «, který bude volat abstraktní metodu Print.
- vše výše popsané zachycuje následující obrázek



Zdrojový kód

```
class Stack
{
  struct Elem
  {
      Prvek * hodn;
      Elem  * dalsi;
      Elem (Prvek *h, Elem *d){hodn = h; dalsi = d;
  };
  Elem * vrch;
  public:
      Stack ()
      {
          vrch = 0;
      }
      void push ( Prvek * p)
      {
           vrch = new Elem (p, vrch);
      }
      Prvek * pop ()
      {
           Elem *pom = vrch;
           Prvek *prvek = vrch->hodn;
           vrch = vrch->dalsi;
           delete pom;
           return prvek;
      }
      int isEmpty()
      {
          return vrch == 0;
      }
      friend ostream& operator<<(ostream&, Stack&);
};


/* stack.cpp */
#include "stack.h"
ostream& operator<<(ostream& s, Stack& stc)
{
  Stack::Elem *pom = stc.vrch;
  while (pom)
  {
      s << *pom->hodn << endl;
      pom = pom->dalsi;
  }
  return s;
}


/* main.cpp */
/* Stack s astraktnim prvkem */
#include "stack.h"
#include <iostream>
using namespace std;
class Cislo : public Prvek
{
    int hodn;
    public:
        Cislo(int h) {hodn = h;}
        virtual void print(ostream& s) const {s << hodn;}
```

```
};
class Znak : public Prvek
{
    char hodn;
    public:
        Znak(char h) {hodn = h;}
        virtual void print(ostream& s) const {s << hodn;}
};

void main()
{
    Stack s;
    Znak z('b');
    Cislo c(222);
    cout << "\nPolymorfní zásobník\n";
    s.push(new Cislo(10));
    s.push(new Znak('a'));
    s.push(&z);
    s.push(&c);
    cout << "Vypis zásobníku\n" << s;
    cout << "Odebirani ze zásobníku\n";
    while (!s.isEmpty())
    {
        cout << *s.pop() << endl;
    }
}
```

Poznámky k parametrům

- Je-li parametr funkce (metody, konstruktoru) typu reference na třídu T, přípustným skutečným parametrem je objekt typu T', kde T' je T nebo podtřídou T. Ve druhém případě se předá odkaz na objekt, aniž by se prováděla konverze na T.
- Funkce operator« definovaná pro třídu Prvek se vyvolá i pro objekt typu Cislo nebo Znak; vlastní výpis provede metoda print, která je definovaná pro každou podtřídu jinak.
- Parametr typu reference na nadtřídu může být též výstupním parametrem

Výběr ukazatele z polymorfní datové struktury

- Při výběru ukazatele na prvek polymorfní datové struktury je někdy třeba jej uložit do proměnné typu ukazatel
- Obsahuje-li datová struktura ukazatele na instance podtříd abstraktní třídy T, lze vybraný ukazatel uložit přímo (bez přetypování) pouze do proměnné typu T*

Příklad 1:

```
Stack s;
s.push(new Cislo(10));
s.push(new Znak('a'));
Znak *pc = s.pop(); // chyba při překladu
Cislo *pc = s.pop(); // chyba při překladu
```

Příklad 2:

```
Stack s;
s.push(new Cislo(10));
s.push(new Znak('a'));
Prvek *p1 = s.pop(); // O.K.
Prvek *p2 = s.pop(); // O.K.
```

Dynamické přetypování na ukazatel na podtřídu

- Ukazatel na nadtřídu lze dynamicky přetypovat na ukazatel na podtřídu

Příklad: Součet čísel v polymorfním zásobníku

```
int main()
{
    Stack s;
    s.push(new Cislo(10));
    s.push(new Cislo(29));
    s.push(new Znak('a'));
    cout << s << endl;
    Cislo *pc;
    int soucet = 0;
    while (!s.isEmpty())
    {
        pc = dynamic_cast<Cislo*>(s.pop());
        if (pc!=NULL)
        {
            soucet += pc->hodn;
        }
    }
    cout << "soucet cisel v zasobniku = " << soucet << endl;
    return 0;
}
```

Dynamický test typu objektu

- Někdy stačí pouze zjistit typ objektu, na který ukazuje proměnná typu nadtřída objektu

Příklad: počet čísel a znaků v polymorfním zásobníku

```
#include <typeinfo>
int main ()
{
    Stack s;
    s.push(new Cislo(10));
    s.push(new Cislo(29));
    s.push(new Znak('a'));
    cout << s << endl;
    Prvek *p;
    int pocetCisel = 0, pocetZnaku = 0;
    while (!s.isEmpty())
    {
        p = s.pop();
        if (typeid(*p) == typeid(Cislo))
        {
```

```
            pocetCisel++;
        }
        else if (typeid(*p) == typeid(Znak))
        {
            pocetZnaku++;
        }
    }
    cout << "pocet cisel = " << pocetCisel << endl;
    cout << "pocet znaku = " << pocetZnaku << endl;
    return 0;
}
```