Programování v C++

Ostrava, 2008

Rostislav Fojtík

Obsah:

Úvodní lekce 5
1. Základy OOP v C++ 7
2. Nové prvky jazyka C++ 17
3. Třídy a instance 29
4. Statické datové členy a funkce. Přátelé. 43
5. Dědičnost 51
6. Polymorfismus 63
7. Vícenásobná dědičnost 71
8. Přetěžování operátorů 77
9. Šablony 83
10. Standardní knihovna šablon 89
11. Datové proudy 101
12. Výjimky 107
13. Vývoj programů 113
14. Programovací jazyk C# 117
15. Jednoduché řešené příklady v jazyku C# 125
16. Cvičení č.1 137
17. Cvičení č.2 141
18. Cvičení č.3 145
19. Cvičení č.4 152
20. Cvičení č.5 157
21. Cvičení č.6 162
22. Cvičení č.7 168
23. Cvičení č.8 170

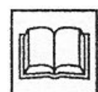
Vysvětlivky k používaným symbolům

Průvodce studiem – vstup autora do textu, specifický způsob, kterým se studentem komunikuje, povzbuzuje jej, doplňuje text o další informace

Příklad – objasnění nebo konkretizování problematiky na příkladu ze života, z praxe, ze společenské reality, apod.

Pojmy k zapamatování.

Shrnutí – shrnutí předcházející látky, shrnutí kapitoly.



Literatura – použitá ve studijním materiálu, pro doplnění a rozšíření poznatků.

Kontrolní otázky a úkoly – prověřují, do jaké míry studující text a problematiku pochopil, zapamatoval si podstatné a důležité informace a zda je dokáže aplikovat při řešení problémů.

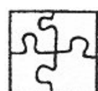
Úkoly k textu – je potřeba je splnit neprodleně, neboť pomáhají dobrému zvládnutí následující látky.

Korespondenční úkoly – při jejich plnění postupuje studující podle pokynů s notnou dávkou vlastní iniciativy. Úkoly se průběžně evidují a hodnotí v průběhu celého kurzu.

Úkoly k zamyšlení.

Část pro zájemce – přináší látku a úkoly rozšiřující úroveň základního kurzu. Pasáže a úkoly jsou dobrovolné.

Testy a otázky – ke kterým řešení, odpovědi a výsledky studující najdou v rámci studijní opory.



Řešení a odpovědi – vážou se na konkrétní úkoly, zadání a testy.

Úvodní lekce

Úvodní lekce slouží studentům k orientaci ve výukovém kurzu „Programování v C++". Kurz má zkratku PROC2 a je určen studentům oborů zaměřených na informatiku a výpočetní techniku, kteří studují distanční a kombinovanou formou studia. Učební materiály mohou sloužit rovněž studentům v prezenční výuce.

Pro zdárné absolvování kurzu „Programování v C ++" jsou základním předpokladem znalosti z předmětů ALDS1, ALDS2 a PROC1. Dále jsou vhodné znalosti z předmětu ARPOC, hlavně z oblasti dělení a organizace paměti, práce procesorů.

Poznámka: v databázi "Student" je kurz pro distanční studium pojmenován XPRO2. Stejně jako ostatní distanční kurzy je na první místo v názvu vloženo písmeno X. Důvodem je potřeba odlišit distanční kurzy od běžné prezenční formy.

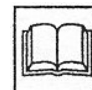
Cíle kurzu

Cílem kurzu "Programování v C++" je seznámit se základními rysy objektově orientovaného programování a jeho praktickým využitím v jazyce C++. Součásti kurzu je rovněž tématika analýzy a tvorby programů.

Po absolvování kurzu by student měl být schopen:

- tvořit programy v jazyku C++
- tvořit programy založené na objektově orientovaném principu
- vytvářet aplikace pomocí vizuálních programovacích nástrojů
- správně analyzovat a navrhovat řešení programů

Doporučená literatura



Stroustrup, B.: *C++ Programovací jazyk*, BEN, 1997
Pecinovský, R., Virius, M.: *Objektové programováni I*, Grada, 1996
Pecinovský, R., Virius, M.: *Objektové programováni II*, Grada, 1996
Virius, M.: *Pasti a propasti jazyka C++*, Grada, 1997
Virius, M.: *C++ Builder 4.0 podrobný průvodce*, Grada, 1999
Racek, S.: *Objektově orientované programování v C++*, KOPP, 1994
Večerka, A.: *Jazyk C++*, UP Olomouc, 1996
Vondrák, I., Šaloun, P.: *Objektově orientované programování*, VŠB-TU Ostrava, 1995
Kačmář, D.: *Programování v jazyce C++*, VŠB-TU Ostrava, 1995
Renner, G.: *Borland C++ kompendium*, UNIS, 1992
Nenadál, K., Václavíková, D.: *Borland C++*, Grada, 1992

Informační zdroje

www.builder.cz/cpp/
www.borland.cz
www.bloodshed.net
fora pro programátory:
www.programator.cz/fora.asp#cpp
www.seif.cz/web/index/html/index.php

1. ZÁKLADY OOP V C++

V této kapitole se dozvíte:

Lekce slouží k nejzákladnějšímu seznámení s objektově orientovaným programováním (dále jen OOP) v rámci programovacího jazyka C++. V učebních materiálech jsou vysvětleny základní pojmy jako třída, objekt, zapouzdření, dědičnost, polymorfismus, členské metody a podobně. Na jednoduchém zdrojovém textu je ukázán postup tvorby objektově orientovaných programů v programovacím jazyku C++.

Po absolvování lekce by student měl být schopen:

- pochopit základní vlastnosti objektově orientovaného programování zapouzdření, dědičnost a polymorfismus
- správně používat pojmy třída a objekt
- orientovat se v jednoduchém zdrojovém textu z programovacího jazyka C++
- vědět, jaké jsou rozdíly mezi jazyky C++, Java a C#

Klíčová slova této kapitoly:
Objektově orientované programování, Zapouzdření, Dědičnost, Polymorfismus, Třída, Objekt, Metody, Konstruktor, Destruktor

Čas potřebný k prostudování učiva kapitoly:
3 hodiny

Programovací jazyk C++ patří mezi nejčastěji využívané typy jazyků pro tvorbu aplikací. Jeho moderní vlastnosti jej předurčují pro práci v profesionálních vývojových týmech. K autorům jazyka C++ patří Bjarne Stroustrup, který při jeho návrhu vyšel z jazyka C a přidal do něj možnosti *objetkově orientovaného programování.* Přestože jazyk C je určitou podmnožinou C++, jsou oba jazyky samostatné. Mají svou samostatnou ANSI a ISO normu (C standard ISO 9899-1990, C++ standard ISO 14882-1998).

Způsob zápisů učebního textu:
instance - důležitý pojem (výraz), který bude dále vysvětlen
while - klíčové slovo programovacího jazyka v běžném textu
`void main(void)` - ukázky zdrojových zápisů

Obsah lekce:

Základní vlastnosti objektově orientovaného programování. Zapouzdření, dědičnost, polymorfismus. Přístupová práva k prvkům třídy (private, public, protected). Třída a objekt. V zhledem k tomu, že lekce se zabývá hlavně teoretickými základy objektově orientovaného programování, není tolik zaměřena na tvorbu konkrétních programů. Student by se měl řádně seznámit s principiálními vlastnostmi OOP a pochopit je. Tomuto tématu je potřeba věnovat dostatek času a energie, aby tvorba budoucích programů byla kvalitní.

Jednotlivé zdrojové soubory jsou pouze jako ilustrační a student je nemusí pochopit do všech detailů.

Testy a úkoly v lekci slouží pouze studentům jako zpětná vazba, zda učivo zvládli.

Nutné předpoklady pro úspěšné zvládnutí lekce

Nutným předpokladem pro práci s lekcí je zdárné absolvování kurzu "Programování v jazyku C" (PROC1). Ovládání programovacího jazyka C je nezbytné, neboť C++ z něj přebírá velmi mnoho konstrukcí a postupů. Přestože se jedná o dva samostatné jazyky, provázanost některých vlastností je veliká. V této i všech následujících lekcích budeme předpokládat důkladnou znalost jazyka C a vlastnosti, které jsou obsaženy v jazyku C, nebudeme znovu opakovat a vysvětlovat. Dalším předpokladem je schopnost pracovat s překladačem jazyka C++. Proto je nutné, aby si student některý z překladačů nainstaloval na svůj počítač. Kromě komerčních nástrojů pro vývoj programů je možné využívat freewarové aplikace. Vzhledem k neustále se vyvíjející standardizaci jazyka C++, je potřeba pracovat s novějšími verzemi překladačů. Například dříve oblíbený překladač Borland C/C++ 3.1 není pro výuku vhodný, neboť nepodporuje některé nové konstrukce.

Jako vhodný překladač pro kompilaci pod MS Windows nebo Linux doporučují freewarový vývojový nástroj Bloodshed Dev-C++, který si můžete stáhnout na adrese: http://www.bloodshed.net/devcpp.html. Volně si můžete rovněž stáhnout MS Visual C++ 2008 Express Edition na adrese: http://msdn.microsoft.com/vstudio/express/visualcsharp/

Vstupní test

1) Prohlédněte si krátkou ukázku zdrojového souboru v programovacím jazyce C a nalezněte chyby a nesprávné konstrukce.

```
#include <stdio.h>

int main()
{
  float pom;
  int i;

  while (i<10)
  {
    scanf("%f",pom);
    printf("Hodnota je %d\n",pom);
    i++;
  }
  return 0;
}
```

Úvod do OOP v C++

Základní vlastnosti objektově orientovaného programování. Zapouzdření, dědičnost, polymorfismus. Přístupová práva k prvkům třídy (private, public, protected). Třída a objekt.

Programovací jazyk C vychází z koncepce, která rozděluje program na data a algoritmické struktury (reprezentované např. funkcemi). Obě skupiny mohou být zpracovávány relativně nezávisle na sobě. Algoritmy musí být pouze vhodné pro vstup, zpracování a výstup určitých datových typů. Jinak však není vytvořeno žádné omezení. Sám programátor se musí postarat, aby se v jeho kódu neobjevily operace, které odporují logice programu. Například výpočtu faktoriálu proměnné, která obsahuje hodnotu dne v datu narození, jazyk C nijak nebrání. Jedná se přece o celé číslo a pro ně lze faktoriál příslušnou funkcí vypočítat. Ovšem vypočtená hodnota pomocí této operace postrádá jakýkoliv význam, tedy je nepoužitelná a z logického hlediska zcela nesmyslná. Ne vždy je však uplatnění proměnných natolik čitelné jako v uvedeném případě a trochu méně pozorný programátor se může dostat do značných obtíží. Jazyk C++ může využívat většiny postupů jazyka C, ale navíc principů objektově orientovaného programování. OOP vnímá data i příslušné algoritmy jako jeden celek, které jsou spojeny v objektech. Ve správně navrženém programu se pak s daty v objektech manipuluje pomocí metod (v jazyku C++ se metody mnohdy označují členskými funkcemi), které jsou součásti daného objektu. Často se hovoří o tom, že *objekty si posílají zprávy*. Toto posílání zpráv mezi objekty je realizováno jako vyvolání některé z jejich metod.

Základní vlastnosti OOP

• ***Zapouzdření (Encapsulation)*** - je vlastnost, která vytváří možnost spojení dat s metodami obsaženými v objektu. Zapouzdření rovněž určuje u jednotlivých dat a metod specifikaci přístupů.
• ***Dědičnost (Inheritance)*** - jedná se o možnost odvozovat nové třídy, které dědí data a metody z jedné nebo více tříd. Dědičnost určuje konkretizaci tříd potomků. V odvozených třídách je možno přidávat nebo předefinovávat nová data a metody.
• ***Polymorfismus*** - česky možno vyjádřit přibližně pojmem "vícetvarost". Umožňuje jediným příkazem zpracovávat "podobné" objekty. Používání těchto vlastností umožňuje vytvářet lépe strukturovaný a udržovatelný program.

Třídy v jazyce C++

Základem objektově orientovaného programování je *třída* (**class**). Typ třída se podobá struktuře v jazyce C. Může však navíc obsahovat i funkce nazývané metodami – princip zapouzdření. Zapouzdření kromě spojení členských dat a členských funkcí (metod) umožňuje jasně odlišit vlastnosti dané třídy, které mohou

být používány i mimo definici třídy od rysů, které lze využívat jen uvnitř třídy - rozlišení přístupových práv.

```
class Datum //příklad definice třídy
{
  private: //následují soukromé prvky třídy
   int den, mesic, rok; //privátní data
  public: //následují veřejné prvky třídy
  Datum(); //konstruktor
  Datum(int d, int m, int r); //konstruktor
  void VypisDatum() const; //veřejná metoda
  int DejDen() const;
  int DejMesic() const;
  int DejRok() const;
  void ZmenDatum(int d, int m, int r);
  void ZmenDen(int d);
  void ZmenMesic(int m);
 void ZmenRok(int r);
};
```

Třída a objekt

Pojem *objekt* budeme chápat jako konkrétní výskyt, instanci dané třídy.Viz následující deklarace:

```
Datum dnes(10,2,2001); //Datum je třída, dnes je
objekt
```

Srovnání struktur v C, struktur a tříd v C++ :

- struktura v jazyce C

```
typedef struct{
int a;
float f;
}hodnota;
```

- struktura v jazyce C++

```
struct hodnota{
 int a;
float f;
};
```

- třída v jazyce C++

```
class hodnota{
public:
int a;
float f;
};
```

Přístupová práva

Rozdíl mezi strukturami v C a třídami v C++ je v úrovni přístupu ke členům. Ten se určuje pomocí klíčových slov **public:**, **private:** a **protected:** (veřejné, soukromé a chráněné členy). Veřejné členy - na ně se můžeme přímo obracet všude, kde je objekt znám prostřednictvím libovolného jiného člena třídy, ale i prostřednictvím libovolné jiné funkce nebo výrazu. Soukromé členy -

obracet se na ně můžeme pouze prostřednictvím členů téže třídy nebo pomocí zvláštních funkcí, kterým se říká *spřátelené metody*. Chráněné členy - obracet se na ně můžeme pouze prostřednictvím členů té třídy, ve které byly chráněné členy definované, nebo pomocí členů jakékoli třídy z dané třídy *odvozené*.

Konstruktor má za úkol vytvoření objektu v paměti a inicializaci členských dat. Konstruktor musí mít stejné jméno jako třída. Tato speciální funkce nemá žádný návratový typ ani **void**(nemůže tudíž obsahovat příkaz **return**). Pokud programátor nevytvoří ani jeden svůj konstruktor, pak je vytvořen *implicitní konstruktor*, který však neinicializuje žádná členská data.

Destruktor je opakem konstruktoru a ruší objekt (uvolňuje paměť). Nelze však přetížit a nemá žádné parametry.

Konstruktor a destruktor musí být zařazeny mezi veřejné metody.

Dědičnost – inheritance

Inheritance umožňuje přidat ke třídě T1 další vlastnosti nebo stávající vlastnosti modifikovat a vytvořit novou odvozenou (podtřídu neboli potomka) třídu T2. Programovací jazyk C++ umožňuje vytvářet inheritanci následujících typů:

Jednoduchá dědičnost - třída má jen jednoho předka (rodiče). Vytváříme stromovou hierarchii tříd. Třídu v nejvyšší úrovní označujeme jako kořenovou třídu.

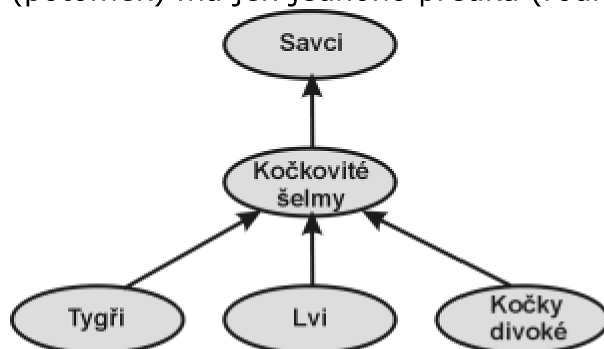
Vícenásobná dědičnost - třída má více předků.

Opakovaná dědičnost - třída může zdědit vlastnosti některého (vzdálenějšího) předka více cestami. Vztahy tříd v hierarchii jsou znázorňovány orientovaným acyklickým grafem (direct acyclic graph - DAG), označovaným také jako graf příbuznosti tříd.

```
class T1
{
  private: //soukromé datové prvky
  public: //veřejně přístupné metody
};

class T2: public T1 //třída T2 je potomkem třídy T1
{
  private: //soukromé datové prvky
  public: //veřejně přístupné metody
};
```

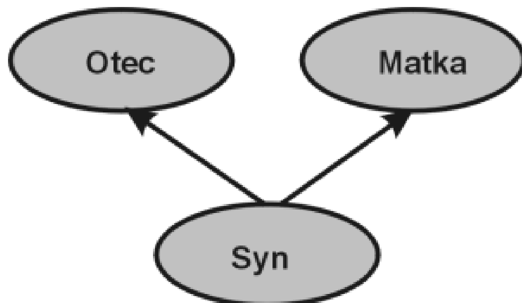
Jednoduchá dědičnost

Jednoduchá dědičnost určuje, že každá odvozená třída (potomek) má jen jednoho předka (rodiče).



Vícenásobná dědičnost

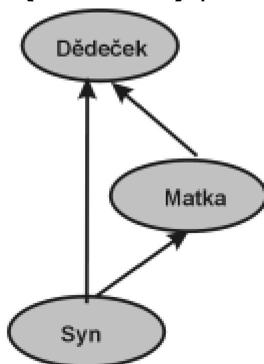
U vícenásobné dědičnosti může mít odvozená třída (potomek) více než jednoho předka (rodiče). Pozor na směr šipek v rámci



grafu. Šipka míří od potomka k rodiči. Vyjadřuje se tím závislot potomka na rodiči.

Opakovaná dědičnost

U opakované dědičnosti může odvozená třída zdědit vlastnosti potomků různými cestami. Například třída **C (syn)** dědí vlastnosti třídy **A (dědeček)** přímo nebo prostřednictvím třídy **B (matka)**.



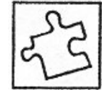
Řešené příklady

Prohlédněte si následující zdrojové soubory. V projektu je vytvořena třída „Napis", která zajistí výpis textu, který umístíme do objektu. Uvedené soubory slouží pouze jako příklad tvorby tříd, proto je řešení velmi zjednodušeno. V souboru NAPIS.H je třída deklarována. Definice jednotlivých metod třídy jsou v souboru NAPIS.CPP. HLAVNI.CPP je soubor, ve kterém je umístěna hlavní funkce **main**.

```
//*** Příklad - NAPIS.H ***//
/* soubor - NAPIS.H */
/* hlavičkový soubor obsahující deklarace třídy */
class Napis
{
  private:
   char text[100];
```

```
  public:
   Napis(char p[]); // první konstruktor - deklarace
   Napis(); // druhý konstruktor - deklarace
   void vypis(); // deklarace další metody
};
/*** NAPIS.H ***/

/* soubor - NAPIS.CPP */
#include <iostream.h>
#include <string.h>
#include "napis.h"
//definice jednotlivých metod třídy "Napis"
Napis::Napis(char p[])
{
  strcpy(text,p);
}
Napis::Napis()
{
  strcpy(text,"Konstruktor bez parametru");
}
void Napis::vypis()
{
  cout << text << endl;
}
/*** NAPIS.CPP ***/
/* soubor - HLAVNI.CPP */
#include "napis.h"
// definice globalni instance
Napis prvni("Prvni vypis");
Napis druhy("Druhy vypis");
Napis treti("Treti vypis");
Napis ctvrty;
int main()
{
  prvni.vypis();
  druhy.vypis();
  treti.vypis();
  ctvrty.vypis();
  return 0;
}
/*** HLAVNI.CPP ***/
```

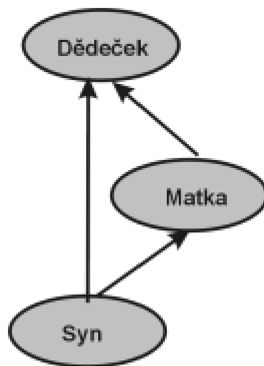
Opakovací test

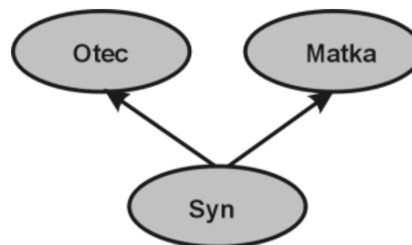


1) Zapouzdření je jedná ze tří základních vlastností objektově orientovaného programování. Co tato vlastnost vyjadřuje?
a) Spojení členských dat a členských funkcí (metod) v jedné třídě.
b) Možnost definování přístupových práv k jednotlivým členům třídy.
c) Možnost vytvářet nové třídy pomocí již dříve definovaných tříd.
d) Schopnost jednou metodou zpracovávat podobné objekty

2) Nakreslete si graf, který vyjadřuje vztahy dědičnosti mezi níže uvedenými třídami. O jaký typ dědičnosti se jedná? Třídy: tygři, savci, kočky divoké, lvi, savci

3) Prohlédněte si níže uvedené obrázky a určete jaký typ dědičnosti znázorňují:





Shrnutí kapitoly

Základním stavebním kamenem *objektově orientovaného programování* je *třída*. Ta odpovídá datovému typu, který určuje vlastnosti a schopnosti jednotlivých proměnných - *instancí (objektů)*.
Hlavními vlastnostmi objektově orientovaného programování jsou:
- zapouzdření
- dědičnost
- polymorfismus

Definice tříd se podobá definici struktur v jazyce C. Navíc se ve třídě definují *metody (členské funkce)*, které určují schopnosti objektu provádět přesně určené činnosti. Klíčovým slovem pro definici třídy je **class**.
Příklad deklarace třídy:

```
class NazevTridy
{
  private: //privatní položky třídy
    DatovyTyp promenna;
  public: //veřejné položky třídy
    NazevTridy(); //konstruktor
    DatovyTyp Metoda(DatovyTyp parametr);
     //metoda-členská funkce
    ~NazevTridy(); //destruktor
};
```

Návaznosti na další lekce

Než se budeme věnovat vytváření programů založených na objektových vlastnostech, je potřeba prostudovat kapitolu zabývající se některými zásadními odlišnostmi jazyků C a C++. Tyto odlišnosti a nové vlastnosti jazyka C++ jsou popsány v lekci č.3. Důkladnější vysvětlení a objasnění práce s třídami a objekty se nachází v lekcích č. 4, 7, 8 a 9.

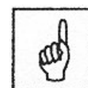
2. NOVÉ PRVKY JAZYKA C++

V této kapitole se dozvíte:

Hlavním cílem lekce je upozornit na nové možnosti jazyka C++ a nové poznatky uplatnit při psaní programů.

Po absolvování lekce by student měl být schopen:

- definovat rozšíření jazyka C++ vzhledem jazyku C
- využívat nové možnosti jazyka C++ při zápisu zdrojových souborů
- využívat ve svých programech přetížené funkce a reference
- předefinovávat operátory



Klíčová slova této kapitoly:
Bool, delete, klíčová slova, konstanty, new, proudový vstup, proudový výstup, přetížená funkce, přetížený operátor, reference, class , friend, inline, operator, private, protected, public, template, this, virtual

Čas potřebný k prostudování učiva kapitoly:
3 hodiny

Obsah lekce:

Odlišnosti programovacího jazyka C++ oproti jeho předchůdci jazyku C. Nová klíčová slova. Datový typ bool. Vstupy a výstupy dat. Reference, předávání parametrů funkcí pomocí referencí. Přetížené funkce. Přetížený operátor. Dynamická alokace paměti.

Nutné předpoklady pro absolvování lekce

Dříve než začnete studovat tuto lekci, projděte si níže uvedené předpoklady pro zahájení studia této lekce.. Zamyslete se nad tím, zda je všechny splňujete. V opačném případě přerušte lekci a proveďte nápravu.

Předpoklady pro zahájení studia lekce:

- Dobrá znalost vlastností jazyka C.
- Mít nainstalován překladač jazyka C++ a mít tak možnost si vyzkoušet vzorové příklady.
- Zvládnutí učiva v první lekci, která se zabývá objektově orientovaným programování v C++. Důležité jsou části, lekce které se zabývají konkrétním zápisem tříd a objektů ve zdrojových souborech.

Předpoklady pro absolvování lekce:

- Nastudovat veškerý učební text
- Provést úkoly a testy v lekci
- Prohlédnout si a pochopit řešené příklady
- Zpracovat programy podle zadání příkladů na konci lekce

Vstupní test

1) Přesuňte šipky na správné rámečky. Pokud si nevíte s úlohou rady, znovu si prostudujte lekci o základech OOP v jazyku C++.

```
class Datum
{
  private:
    int den,mesic,rok;
  public:
    Datum(int,int,int);
    int DejDen() const;
    int DejMesic() const;
    int DejRok() const;
    void ZmenDatum(int,int,int);
}

int Datum::DejDen(int d)
{
  return den;
}

Datum dnes(24.2.2002);
Datum davno(1,1,1);
```

- definice metody třídy
- deklarace třídy
- definice objektů

Nová klíčová slova

Jazyk C++ obsahuje oproti jazyku C další klíčová slova:
class, delete, friend, inline, new, operator, private, protected, public, template, this, virtual

K vysvětlení většiny klíčových slov se dostaneme v příštích lekcích.

Nové prvky

V jazyce C se pro ukazatele, které nemají nikam ukazovat, používá makro NULL. To má obvykle hodnoty 0, 0L nebo (void*)0. V C++ je možné tohoto makra rovněž použít. Existují však situace, kde NULL může působit problémy, proto se doporučuje používat raději 0.

V novějších překladačích jazyka C++ (např. Borland C++ 5.0) se objevuje nov datový typ **bool**, který se řadí mezi celočíselné typy a kter může nabvat hodnot **false** (0) a **true** (1).

Programovací jazyk C++ podporuje komentáře jazyka C a navíc vytváří nov typ.

```
// vše až do konce řádku je bráno jako komentář
```

Konstanty se deklarují následujícím způsobem:

```
const float pi = 3.14159;
//nebo float const pi = 3.14159;
```

Konstantu nelze měnit a tudíž je ji nutné inicializovat na určitou hodnotu. Naše konstanta *pi* představuje hodnotu typu *float*, ale nejedná se o l-hodnotu, to znamená, že nemůže stát na levé straně přiřazovacího výrazu.

```
pi = 3.14;     //Nelze!!!
```

V jazyku C++ je možné napsat:

```
const int M = 500;
double pole[M];//podobný zápis v jazyku C nebyl možný
```

Použití konstant je výhodnější než používání maker. Uvědomme si, že makro nenese žádnou informaci o datovém typu. Jedná se o pouhý řetězec znaků. Naopak konstanty jsou jasně definovány pro konkrétní datový typ. Proto je vhodné se vyhnout častému používání maker, které se naopak v jazyce C používala ve velké míře.

Pomoci rozlišovacího operátoru :: (čtyřtečka) můžeme volat jinak zastíněné globální proměnné.

Příklad:

```
int i=10;      //globální proměnná
void fce( )
{
   int i=20;        //lokální proměnná
```

```
   cout << i << endl << ::i <<endl;
/*nejprve se vypíše hodnota lokální a na druh řádek
globální proměnné*/
}
```

Na obrazovce se vypíše:
20
10

Programovací jazyk C++ umožňuje na rozdíl od jazyka C definovat proměnnou v libovolném místě v bloku, nejen na jeho začátku. Platnost je pak omezená až do konce programového bloku.

Příklad zápisu:

```
void main(void)
{
  int vys=0;
  randomize();
  for (int i=0;i<10;i++)
  {
    vys+=i;
    int x; //definice nové proměnné
    x=random(100)+i;
    printf("x=%d a vys=%d\n",x,vys);
  } //zde končí platnost proměnných 'x' a 'i'
  i=0; //chyba!
  x=0; //chyba!
}
```

Pozor na některé starší verze překladačů, které neuvolňují správně paměť vytvářených proměnných (např. Borland C/C++ 3.1).

Jazyk C++ klade na funkci *main* více omezení než jazyk C.
Funkce *main( )* musí být typu **int** nebo **void**, nelze ji rekurzivně volat, nesmíme získávat a používat její adresu. Funkce může mít až dva parametry přesně určených typů:

```
int maim(int argc, char *argv[ ])
{
...
}
```

Musí se použít volací konvence jazyka C (explicitně uvést identifikátor **_cdecl**).

Výpis a čtení dat

C++ má nové možnosti usnadňující výstup a vstup. Standardní výstupní proud cout nahrazuje stdout a vstupní proud cin, který nahrazuje stdin . Pro chybová hlášení se používá výstupní proud cerr. Proudy a zdroj nebo cíl jsou spojeny s přetíženými operátory << (operátor insertion, pro výstup do proudu) a >> (operátor extration, pro vstup z proudu).

Všechny vstupní a výstupní operátory a manipulátory jsou definovány v externí run-time knihovně, a proto je potřeba provést vložení hlavičkového souboru *iostream.h*.

Příklad:

```
#include <iostrem.h>
//deklaruje základní rutiny pro zpracování proudů
void main(void)
{
 int i;

 cout << "Zadej číslo: "; //  výstupní proud
 cin >> i; //  vstupní proud
 cout << "Číslo je: " << i;
}
```

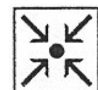
cout - výstupní proud, který zasílá znaky na standardní výstup *stdout* pomocí operátoru <<

cin - vstupní proud připojen na standardní vstup pomocí operátoru >>, umí zpracovávat všechny standardní typy dat

iostream.h - standardní hlavičkový soubor, který nahrazuje řadu funkci ze *stdio.h*

Formátování se provádí pomocí manipulátorů. Jedná se speciální operátory podobné funkcím. Tyto operátory používají jako svůj argument odkaz na proud, který také vracejí. Proto mohou být součástí příkazu vstupu. Manipulátory jsou definovány v hlavičkové souboru *iostream.h* a *iomanip.h*.

Příklad:



```
#include <iostream.h>
#include <iomanip.h>

int main()
{
     int cislo = 200;
     cout << hex << cislo << endl;//vypíše
hexadecimální tvar čísla
     cout << dec << cislo << endl; //vypíše decimální
tvar čísla
     cout << setw(5) << cislo << endl;//nastaví šířku
na 5 pozic     return 0;
}
```

Funkční prototypy

Funkční prototypy v C++ mohou mít nastaveny implicitní hodnoty některých parametrů. Pokud se při volání dané funkce odpovídající argument vynechá, bude za něj dosazena implicitní hodnota.

```
int Funkce(float f=6.1, int i =10);
//......
Funkce(3.14, 25);
```

```
// oba implicitní parametry budou přepsány
Funkce(2.5);        // stejné jako volání
Funkce(2.5,10);
Funkce( );          // stejné jako volání
Funkce(6.1,10);
```

Pozor! Vynechá-li se první parametr, musí se vynechat i všechny následující.

Programovací jazyk C++ zavádí nové klíčové slovo **inline**, které způsobí zkopírování funkce na každé místo v kódu, kde je daná funkce volána. Funkce se bude chovat podobně jako by byla makrem. Na rozdíl od maker však umožňuje typovou kontrolu.

Reference

Jazyk C++ zavádí tzv. *reference*, které představují zvláštní druh proměnné. Na reference se můžeme dívat jako na jiná jména existujících proměnných. Deklarují se podobně jako ukazatele, jen místo znaku * vkládáme znak &. Jakmile však referenci deklarujeme, bude již stále ukazovat na tutéž proměnnou. V C++ rovněž nemůžeme deklarovat ukazatele na reference a pole referencí. Nelze také deklarovat reference na typ *void* (*void&).* Reference se nejčastěji používají při volání funkcí s parametry předávanými odkazem.

```
int prom;
int &ref_prom = prom; //reference
int *uk_prom; //ukazatel

ref_prom = 20; //je to samé jako:  prom=20;
uk_prom = &ref_prom; //je to samé jako:
uk_prom=&prom;
```

Předávání parametrů funkcí

V jazyce C jsou dvě možnosti, jak předávat parametry funkcím:

1. *Volání hodnotou* - předává se samotná proměnná a funkce si vytváří vlastní lokální kopii na zásobníku. Takový způsob není vhodný pro svou časovou a paměťovou náročnost u parametrů s větším datovým typem.

2. Jazyk C neumí předávat parametry odkazem, ale umožňuje *předání adresy na skutečný parametr*, které je pro větší datové struktury výhodnější než první způsob.

V jazyce C++, kromě výše uvedených variant, již existuje možnost *předávání parametrů odkazem* - raději *funkce s parametry volanými referencí*.

Příklad:

```
void swap(int &a, int &b)
{
```

```
 int pom;

 pom=a;
 a=b;
 b=pom;
}

void main(void)
{
 int X=10, Y=20;

 swap(X, Y); // vymění se hodnoty proměnných X a Y
 cout << "X je: " << X <<" Y je: " << Y << endl ;
}
```

Funkce mohou odkazem vracet vypočtený výsledek (funkce vrací referenci). Takovýmto funkcím se říká *referenční*. V příkazu **return** musí být uvedena l-hodnota vraceného typu.

Příklad:

```
int pole[20];
int gl;
int &fce(int i)
{
 if ((i<0) || (i>19)) return gl;
 else return pole[i];
}
//. . .
x = fce(3); // stejné jako: x = c[3];
fce(10) = 150; // stejné jako: x[10] =150;
```

Přetížené funkce

Díky možnosti přetěžovat funkce je program čitelnější. Chceme-li napsat dvě různé funkce se dvěma různými argumenty, mohou mít obě funkce stejný název a různé argumenty.

Příklad čtyř funkcí se stejným jménem, ale různým návratovým typem nebo různými parametry:

```
void fce( ); // funkce č.1
int fce(int); // funkce č.2
float fce(flaot);// funkce č.3
int fce(float, double); // funkce č.4
```

Zavoláme-li v programu funkci *fce(100);*, překladač vyvolá funkci č.2. Je však potřeba dávat pozor na jednoznačnost zápisu.

Příklad použití:

```
int abs(int n)
{
```

```
 return (n < 0) ? n*(-1):n;
}
double abs(double n)
{
 return (n < 0) ? n*(-1):n;
}
```

V případě, že by možnost přetěžovat funkce nebyla, museli bychom napsat různé funkce pro různé datové typy. Například funkce *int abs_i(int n);* a `double` *abs_d(double n);* a podobně.

Dynamická alokace paměti

Jazyk C++ nabízí nové operátory pro alokaci a uvolnění paměti a to operátor **new** a **delete**. Je sice dále možné používat funkcí jazyka C (*malloc, free ...*), ale není to moc vhodné, neboť tyto funkce neví kromě potřebné velkosti nic o dané proměnné. Naproti tomu operátor **new** zná třídu objektu, automaticky volá její konstruktor a také vrací příslušný typ ukazatele. Není třeba přetypovávat, během přiřazení probíhá typová kontrola.

Dealokace paměti, která byla alokována operátorem **new**, se musí provést pomocí operátoru **delete**. Tento operátor automaticky volá destruktor třídy.

Použijeme-li při dynamické alokaci objektu funkce jazyka c (*malloc*), vyhradíme sice potřebný prostor v paměti, ale objekt nevznikne. Funkce *malloc* nezavolá konstruktor třídy! Operátor **new** slouží k dynamické alokaci paměti.

Za klíčové slovo **new** píšeme označení typu proměnné, kterou chceme alokovat. Operátor vybere z volné paměti potřebné místo a vrátí ukazatel na ně. Pokud se operace nepodaří, vrátí se hodnota 0, což nepředstavuje platnou adresu.

Příklad:

```
long double *prom;
prom = new long double;
if (!prom) Chyba( );
//jestliže se alokace nezdařila, voláme funkci Chyba(
)
```

Chceme-li dynamickou proměnnou při alokaci inicializovat na určitou hodnotu, zapíšeme tuto hodnotu do závorek za jméno typu:

```
long double *prom;
prom = new long double(55.66);
if (!prom) Chyba( );
```

Při alokaci pole napíšeme k datovému typu do hranatých závorek počet prvků pole. V tomto případě však nelze použit inicializaci prvků. Jejich hodnoty musíme nastavit dodatečně.

```
int *pole;
pole = new int[100];
```

Operátor **new** může alokovat rovněž vícerozměrná pole. Je potřeba si však uvědomit, že jazyk C++ zná pouze pole jednorozměrná a vícerozměrná pole nahrazuje poli jednorozměrnými, jehož prvky jsou opět pole.

Operátor **delete** je unární a jeho jediným operandem je ukazatel na proměnnou, kterou chceme uvolnit.

```
delete prom;
```

Pozor! Operátor **delete** proměnnou z paměti sice uvolní, ale příslušný ukazatel bude stále ukazovat do stejného místa v paměti, kde se dynamická proměnná nacházela. Doporučuje se po dealokaci přiřadit příslušnému ukazateli hodnotu 0. Dealokace paměti se smí provést pouze jedenkrát, jinak může dojít nekontrolovatelnému chování programu. Ukazatel s hodnotou 0 lze však dealokovat bezpečně bez vedlejších efektů.

Při uvolňování dynamicky alokovaného pole se hranaté závorky píší za operátor **delete**.

```
delete [ ]pole;
```

Přetížený operátor

Dalším rozšířením jazyka je možnost přetížit nejen funkce, ale i operátory. To znamená určit jim činnost v závislosti na kontextu. Toto je možné, neboť operátor je v C++ chápan jako funkce s jedním parametrem (unární operátor) nebo se dvěma parametry (binární operátor). Při definici pak jméno rozšiřujeme o klíčové slovo **operator** @, kde znak @ nahrazuje přetížen operátor.

Pozor! Nelze však přetížit například operátory **?:**, **.***, **::**, **sizeof** a **.** (přístup ke strukturám). U přetížení operátoru **++** a nelze určit zda se jedná o postfixový nebo prefixový přístup.

Příklad:

Vytvořte program, ve kterém přetížíme binární operátor + pro sčítání komplexních čísel. Prohledněte si vzorový zdrojový soubor. Rozšiřte program o přetížený binární
operátor - , který bude odečítat dvě komplexní čísla.

```
// Vzorový příklad pro přetížený operátor
// Binární operátor + bude umět sčítat komplexní
čísla

#include <iostrem.h>
struct complex
{
  double re,im;
};

//definice přetíženého operátoru
complex operator+(complex a, complex b)
{
  complex pom;

  pom.re=a.re+b.re;
  pom.im=a.im+b.im;
  return pom;
}

//přetypování výstupního operátoru
ostream &operator<<(ostream &vys, complex x)
{
 vys << x.re << " + i. " << x.im;
 return vys;
}

int main()
{
  complex VYS, X={1.0,2.0},Y={3.0,4.0};

  VYS=X+Y;
  cout << VYS << endl;

  return 0;
}
```

Příklady:

1. Vytvořte přetížené funkce
   typ MyAbs(typ n);
Funkce budou vracet absolutní hodnotu čísel typu *int, double, long*.

2. Vytvořte přetížené funkce
   void Tisk(typ prom);
Funkce budou vypisovat na obrazovku proměnnou typu *int, double, char, char* *.

3. Vytvořte funkci
   int Suma(int dolni=1, int horni=50, int krok=1);
Funkce bude vracet součet celých čísel od *dolní* hranice do *horní, krok* udává vzdálenost mezi sousedními čísly. Volejte funkci s různě nastavenými parametry.

Všechny příklady si napište a odlaďte ve svém překladači. Krokujte si provádění programu a zamyslete se nad postupem provádění jednotlivých příkazů.

Opakovací test

Označte klíčová slova, pojmy, konstrukce a postupy, které jsou rozšířením jazyka C++ oproti jazyku C.

```
new
<iostream.h>
malloc
<stdio.h>
přetížené funkce
možnost dynamicky alokovat paměť
reference
funkce s proměnným počtem parametrů
```

Shrnutí kapitoly

Jazyk C++ využívá mnoho konstrukcí a postupů známých z jazyka C. Zároveň však zavádí několik nových prvků:
- Rozšiřuje množinu klíčových slov: **class**, **delete**, **friend**, **inline**, **new**, **operator**, **private**, **protected**, **public**, **template**, **this**, **virtual**.
- Novější standardy jazyka C++ zavádí nový datový typ **bool**.
- Vytváří mechanizmus přetížení operátorů a funkcí, které se mohou stejně jmenovat, ale manipulují s různými daty.
- C++ nově definuje vlastnosti konstant. Konstanty mohou nahrazovat makra bez parametrů.
- Novým mechanizmem jsou reference. Referenci můžeme chápat jako dereferencovaný ukazatel. Reference se dají využít k volání parametrů funkcí odkazem.
- Pro dynamickou alokaci paměti zavádí C++ dva nové operátory **new** a **delete**. Ty na rozdíl od funkcí *malloc* a *free* nejen alokují pro vznikající objekt potřebný paměťový prostor, ale navíc volají příslušný konstruktor a destruktor. Tím zajistí vytvoření či zrušení příslušné instance třídy.
- Zavádí se nové proudy pro práci se vstupy a výstupy dat (**cout**, **cin**).

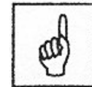
3. TŘÍDY A INSTANCE

V této lekci se budeme zabývat tvorbou tříd. Vysvětlíme si, jak vytvářet datové členy třídy, jak deklarovat a definovat metody. Obsah lekce: třídy a instance. Standardní metody, konstruktor a destruktor. Kopírovací konstruktor. Deklarace a definice metod. Inline funkce, přístupové a změnové metody.

Po absolvování lekce by student měl být schopen:

- umět tvořit třídy a objekty (instance)
- umět definovat *konstruktor* a *destruktor*
- schopni deklarovat a definovat *přístupové* a *změnové* metody (členské funkce)
- schopni definovat *copy konstruktor*

Klíčová slova této kapitoly:
Destruktor, konstruktor, public, private, protecte, přístupové metody, změnové metody



Čas potřebný k prostudování učiva kapitoly:
3 hodiny

Konstrukce třídy

Jak již bylo uvedeno v kapitole o základech objektově orientovaného programování, zavádí jazyk C++ nový typ a to je třída (**class**). Třída je uživatelsky definován typ a obsahuje jak členská data, tak i členské funkce. Pro deklaraci třídy je možné využít klíčová slova **class**, **struct** i **union**. Struktura má všechny své prvky implicitně **public** a přístupová práva lze selektivně změnit. Unie mají přístupová práva implicitně rovněž **public**, ale není je možné změnit. Třída vytvořena pomocí slova **class** má implicitně hodnotu přístupového atributu **privat** a je ji možné selektivně změnit.

public: povoluje vnější přístup k prvkům třídy
private: zakazuje vnější přístup k prvkům třídy
protected: označují se takto prvky nepřístupné vzdáleným přístupem z vnějšku třídy, ale procházející děděním do odvozených tříd.

Nastavení přístupových práv lze v deklaraci provést vícekrát a v různém pořadí.

Datové prvky

Kromě jednoduchých datových typů a polí prvků jednoduchých typů mohou být ve třídě deklarovány také prvky s typem jiné třídy. Při deklaraci prvků je potřeba dbát na některá omezení:

- prvky nesmí být konstantní (např. *const float pi;* - chyba!). Je-li potřeba ve třídě používat symbolicky označené konstantní prvky,

pak se zavedou jako statické konstantní datové prvky a tím jsou společné pro všechny objekty třídy. (v deklaraci třídy se uvede *static const float pi;* a v implementačním textu třídy pak *static const float pi=3.14;*).

- prvky nesmí být přímo inicializovány na určitou hodnotu (např. *int pr=15;* - chyba!). Tato chyba je pochopitelná, když si uvědomíme, že se jedná o *deklaraci*, při níž se prvku ještě nepřiděluje paměť! Inicializace se provádí až prostřednictvím konstruktoru (s výjimkou statických prvků).
- na rozdíl od metod (členských funkcí) nesmíme prvky přetížit, tedy různé datové prvky nesmí mít stejná jména.

Datové prvky by měly mít přístupový atribut **private** a přístup k nim by měly zajišťovat pouze k tomu účelu zavedené metody.

Konstruktor

Jak již bylo dříve uvedeno konstruktor je standardní metoda každé třídy, která se stará o vytvoření objektu. Konstruktor nic nevrací a nesmí být typován ani jako **void**. Při vytváření vlastní třídy máme následující možnosti:

- Nedefinujeme žádný konstruktor. V tom případě si ho kompilátor vytvoří sám (tzv. *implicitní konstruktor*). V implicitním konstruktoru je volán konstruktor bez parametrů bázové třídy a konstruktory bez parametrů pro vytvoření vnořených objektů. V případě, že takové konstruktory neexistují, překladač indikuje chybu.
- Definujeme jeden konstruktor. Ten může mít stejně jako každá jiná metoda parametry včetně jejich inicializace. Jakmile je nějak konstruktor definován, nevytvoří se implicitní konstruktor.
- Přetížíme konstruktor (definujeme více konstruktoru). Tato varianta umožňuje různé způsoby inicializace objektu.

Je možné také vytvořit tzv. *kopírovací (copy) konstruktor*, který dokáže inicializovat objekt podle vzoru realizovaného jiným, již existujícím objektem téže třídy.

Příklad: kopírovací konstruktor

```
class A
{
  private:
     int i;
  public:
     A(int j) {i=j;}
     A(A &vzor) {i=vzor.i;}
}
int main()
{
  A prvni(10);
  A druhy(prvni);   //copy konstruktor
  A treti=prvni;    //copy konstruktor
}
```

Destruktor

Destruktor je rovněž standardní metoda každé třídy, která provádí činnost související s rušením objektu. Není-li ve třídě destruktor explicitně definován, kompilátor vytvoří implicitní destruktor. Explicitní destruktor se jmenuje stejně jako třída a před její jméno se vloží ~, nesmí mít žádné parametry, nic nevrací, nesmí být přetížen, musí být deklarován jako **public**. Překladač volá destruktor automaticky v okamžiku zániku odpovídající proměnné (např. při opuštění příslušného bloku, dané funkce nebo při ukončení programu). Destruktory se volají v obraceném pořadí než konstruktory.

Příklad zápisu:

```
class NejakaTrida
{
  private:
     int a;         //členská data
     int b;         //členská data
  public:
     NejakaTrida( );     // konstruktor bez parametrů
     NejakaTrida(int X, int Y );
// konstruktor s parametry
     ~NejakaTrida( );    // destruktor
     int Vetsi(int X, int Y);
// deklarace nějaké další metody
};
```

Deklarace a definice metod

Zatím jsme si ukazovali hlavně jakým způsobem se jednotlivé metody deklaruji uvnitř třídy. Metody je však potřeba také definovat. Při definici jednotlivých metod nesmíme zapomenout, že identifikátor metody musí být spojen s identifikátorem třídy. Oba identifikátory od sebe oddělujeme :: (čtyřtečkou). Definice metody se pak provádí až za deklaraci třídy. V každé metodě je ještě jeden skryt parametr - ukazatel na instanci, pro niž se daná metoda volala. Lze se na něj odvolat klíčovým slovem **this**. Překladač jej používá k tomu, aby určil, s jakou instancí (objektem) pracuje.

Jazyk C++ umožňuje definovat tělo metody uvnitř deklarace třídy. Takto definované metody se překládají jako vložené (*inline*). Pokud chceme vytvořit *inline* metody a nechceme ji definovat uvnitř deklarace třídy, připojíme v definici metody klíčové slovo **inline**.

Kromě konstruktorů a destruktoru můžeme ostatní metody rozdělit na dvě základní skupiny:

- *změnové* - metody, jejichž účelem je nějakým způsobem změnit objekt.

- *přístupové* - metody, které předávají hodnoty soukromých položek. Klíčové slovo **const** na konci deklarace naznačuje, že daná metoda ponechává objekt beze změn.

Metody, které mají za úkol zajistit komunikaci s daným objektem, je nutné deklarovat v části *public*. Naopak metody, jejichž úkolem je práce pouze uvnitř objektu (např. kontroly hodnot) je většinou vhodnější deklarovat jako *private*.

V deklaraci třídy je možné zařadit prvky, které představují deklaraci typů (např. pomocí konstrukcí **struct, union, enum, class** a **typedef**). Platí však jisté podmínky:

- typ **struct** a **union** je vně třídy použitelný bez ohledu na přístupový atribut (na rozdíl od typů **enum**, **typedef** a **class**, které musí být deklarovány zásadně jako *public*, mají-li přístupny i vně třídy).

Jednotlivé metody můžeme definovat přímo uvnitř deklarace třídy. Tento postup je však vhodný pouze u velmi krátkých kódů. (Pamatujte, že na počátky tvorby programu, však většinou nejsme schopni stoprocentně určit, zda bude funkce krátká či dlouhá.) Mnohem vhodnější je tvořit definice metod mimo tělo třídy.

Příklad zápisu:

```
class JmenoTridy //deklarace třídy
{
  ...
  typ JmenoMetody();
};
//následují definice jednotlivých metod
typ JmenoTridy::JmenoMetody()
{
  //zde příjde kód metody
}
```

Pozor! Nesmíte před jméno metody zapomenout přidat jméno třídy. Následující zápis nedefinuje kód metody třídy, ale pouhou *řadovou funkci* (samostatnou funkci) , která není součástí žádné třídy!

```
typ JmenoMetody()
{
  //zde příjde kód
}
//nejedná se o metodu, ale řadovou funkci
```

Definice objektu

Třída představuje vlastně datový typ. Teprve vytvořením *objektu (instance třídy)* vytvoříme místo v paměti, se kterým může konkrétně provádět příslušné operace.

Příklad zápisu:

```
//deklarace třídy
class JmenoTridy
{
  ...
  typ JmenoMetody();
};

//následují definice jednotlivých metod
typ JmenoTridy::JmenoMetody()
{
```

```
  //zde příjde kód metody
}

void main(void)
{
  //definice dvou objektů třídy "JmenoTridy"
  JmenoTridy objekt1, objekt2;

  objekt1.JmenoMetody(); //volání metody objektu
  objekt2.JmenoMetody(); //volání metody objektu
}
```

Řešené příklady

Příklad č.1

```
/*
 * Příklad č.1 - Třídy a instance
 *
 * R. Fojtík
 */

// začátek deklarace třídy
class Cas
{
  private:
     int sek, min, hod;
  public:
     Cas(int h, int m, int s){sek=s;min=m;hod=h};
               //konstruktor - inline
     void Zmenit(int h,int m,int
s){sek=s;min=m;hod=h};
               //inline změnová metoda
     void NastavHod(int hod);
          // deklarace změnové metody
     void NastavMin(int min);
          // deklarace změnové metody
     void NastavSek(int sek);
          // deklarace změnové metody
     void Tisk() const;  //přístupová metoda
     int DejHod()const;       //přístupová metoda
     int DejMin()const;       //přístupová metoda
     int DejSek()const;       //přístupová metoda
     ~Cas(){ };               // destruktor - inline
};
// konec deklarace třídy

/***** definice metod *****/
void Cas::Tisk()const
{
  cout << hod << ':' << min << ':' << sek << endl;
}

int Cas::DejHod()const
{
  return hod;       //hod označuje this->hod
}

int Cas::DejMin()const
{
  return min;       //min označuje this->min
}

int Cas::DejSek()const
```

```
{
  return sek;       //sek označuje this->sek
}

void Cas::NastavHod(int hod)
{
  this->hod=hod;    //nutné použití parametru this
}

void Cas::NastavMin(int min)
{
  this->min=min;    //nutné použití parametru this
}

void Cas::NastavSek(int sek)
{
  this->sek=sek;    //nutné použití parametru this
}
/*** konec definic ***/


void main(void)
{
  Cas AktualniCas(13,47,55);  //vytvoření instance
třídy Cas

  AktualniCas.Tisk();         //vypis hodnot
  AktualniCas.NastavSek(0);
     //... další metody
  AktualniCas.Zmenit(14,15,16);
     //... další metody
}
```

Při bližší prohlídce programu jste si asi všimli, že konstruktor *Cas(int h,int m,int s)*; a metoda *void Zmenit(int h,int m,int s);* mají vlastně stejný vnitřní kód funkce a jednu z metod by bylo možné vynechat. Není to však vhodné, neboť konstruktor je překladačem automaticky volán ve chvílích, v nichž to považuje za důležité (při vzniku objektu). Obyčejná metoda ke stejnému postupu překladač nikdy nepřiměje. Konstruktor tedy není obyčejnou metodou zastupitelný. V případě, že bychom se snažili využívat jen konstruktor, bychom opět narazili na problém v okamžiku, kdy bychom chtěli již dříve vytvořené instanci změnit hodnoty. Konstruktor totiž vždy vytváří novou instanci.

Všechny metody samozřejmě nemusí být pouze **public**, ale v jistých případech je vhodné, aby byly soukromé pro danou třídu. Pak jejich volání mohou využívat jen ostatní metody dané třídy. Tyto členské funkce pak nejsou přímo přístupné z vnějšku třídy a může je používat jen daná třída.

Příklad č.2

Vytvořte třídu *Datum*, která umožní pracovat s datumovými hodnotami den, měsíc, rok. Vytvořte soukromé metody třídy, které budou kontrolovat správné hodnoty dne, měsíce, roku. Bude-li hodnota špatná, vrátí metoda nejbližší správnou hodnotu.

```
// *** Příklad  - vytvoření třídy Datum ***  //
#include <iostream.h>
#include <string.h>
#include <dos.h>
#include <conio.h>

// zacatek deklarace třidy Datum
class Datum
{
  private:
    int den, mesic, rok;
// Soukromé metody pro kontrolu správných údajů, je-
li hodnota
// nevyhovující, vrátí metoda nejbližší správnou
hodnotu.
    int SpravnyDen(int d);
    int SpravnyMesic(int m);
    int SpravnyRok(int r);
  public:
    Datum();   //konstruktor
    Datum(int d, int m, int r); //konstruktor
    Datum(int d, char *m, int r); //konstruktor
    void VypisDatum() const;  //přístupová metoda
    int DejDen() const;  //přístupová metoda
    int DejMesic() const;     //přístupová metoda
    int DejRok() const;        //přístupová metoda
    void ZmenDatum(int d, int m, int r);
          //změňová metoda
    void ZmenDen(int d);  //změnová metoda
    void ZmenMesic(int m);  //změnová metoda
    void ZmenRok(int r);  //změnová metoda
};
// konec deklarace třidy Datum

// následují definice metod
int Datum::SpravnyDen(int d)
{
  if (d>=1 && d<=28) return d;
   else
    if (d<1) return 1;
     else
     {
if (mesic==1 || mesic==3 || mesic==5 || mesic==7 ||
mesic==8 || mesic==10 || mesic==12)
     if (d>31) return 31;
```

```
       if (mesic==4 || mesic==6 || mesic==9 ||
mesic==11)
     if (d>30) return 30;
       if (mesic==2)
     if ((rok-1980)%4 == 0)
       if (d>29) return 29;
        else return d;
      else
       if (d>28) return 28;
     }
}

int Datum::SpravnyMesic(int m)
{
  if (m<1) return 1;
   else
    if (m>12) return 12;
      else return m;
}

int Datum::SpravnyRok(int r)
{
  // Chceme použít rok pouze v rozmezí 1980 - 2050.
  if (r<1980) return 1980;
   else
    if (r>2050) return 2050;
      else return r;
}


Datum::Datum()
{
   struct date d;

   getdate(&d);
   rok= d.da_year;
   mesic= d.da_day;
   den= d.da_mon;
    //nastavení systémového data
}

Datum::Datum(int d, int m, int r)
{
  rok=SpravnyRok(r);
  mesic=SpravnyMesic(m);
  den=SpravnyDen(d);
}

Datum::Datum(int d, char *m, int r)
{
  rok=SpravnyRok(r);
```

```
  if (strcmp(m,"leden")==0) mesic=1;
  else
   if (strcmp(m,"unor")==0) mesic=2;
   else
    if (strcmp(m,"brezen")==0) mesic=3;
    else
     if (strcmp(m,"duben")==0) mesic=4;
     else
      if (strcmp(m,"kveten")==0) mesic=5;
      else
       if (strcmp(m,"cerven")==0) mesic=6;
       else
     if (strcmp(m,"cervenec")==0) mesic=7;
     else
      if (strcmp(m,"srpen")==0) mesic=8;
      else
       if (strcmp(m,"zari")==0) mesic=9;
       else
        if (strcmp(m,"rijen")==0) mesic=10;
        else
         if (strcmp(m,"listopad")==0) mesic=11;
         else
          if (strcmp(m,"prosinec")==0) mesic=12;
          else mesic=1;
          //hodnota v případě chybného řetězce
  den=SpravnyDen(d);
}

void Datum::VypisDatum() const
{
  cout << den << '.' << mesic << '.' << rok << endl;
}
int Datum::DejDen() const
{
  return den;
}
int Datum::DejMesic() const
{
  return mesic;
}

int Datum::DejRok() const
{
  return rok;
}
void Datum::ZmenDatum(int d, int m, int r)
{
  rok=SpravnyRok(r);
  mesic=SpravnyMesic(m);
  den=SpravnyDen(d);
}
```

```
void Datum::ZmenDen(int d)
{
  den=SpravnyDen(d);
}
void Datum::ZmenMesic(int m)
{
  mesic=SpravnyMesic(m);
}
void Datum::ZmenRok(int r)
{
  rok=SpravnyRok(r);
}
// konec definice metod

int main()
{
   Datum d1;
//objekt vytvořen konstruktorem bez parametrů
   Datum d2(13,2,2005);
//objekt vytvořen konstruktorem se třemi parametry
typu int
   Datum d3(13,"cervenec",2005);
//objekt vytvořen konstruktorem se dvěma parametry
typu //int a jedním char*
   clrscr();
   d1.VypisDatum();
   d2.VypisDatum();
   d3.VypisDatum();

   d1.ZmenDatum(29,2,2003);
   d2.ZmenDatum(29,2,1980);
   d3.ZmenDatum(-1,-1,2008);

   cout << endl;
   d1.VypisDatum();
   d2.VypisDatum();
   d3.VypisDatum();
   getch();
   return 0;
}
// *** Konec příkladu ***  //
```

Příklady

Příklad č.1

Vytvořte třídu *Cas*, která umožní pracovat s časovými hodnotami hodina, minuta, sekunda. Vytvořte soukromé metody třídy, které budou kontrolovat správné hodnoty hodiny, minuty, sekundy. Bude-li hodnota špatná, vrátí metoda nejbližší správnou. Ve funkci main vytvořte alespoň jeden objekt třídy *Cas* a vyzkoušejte metody objektu.

Příklad č.2

Vytvořte třídu *RodneCislo*, která bude obsahovat datový člen *rc*, který bude typu řetězec. Dále třída bude obsahovat metody, které z rodného čísla (*rc*) zjistí den, měsíc, rok narození a pohlaví. Další metoda bude umět vypsat rodné číslo na obrazovku. Vytvořte konstruktor, který inicializuje hodnotu *rc* ze svého parametru.

Opakovací test

Zamyslete se nad následujícím kódem a chybami v něm. Navrhněte správné řešení.

C:\Rosta\ToolBook-kurzy\PROC2\Priklady\4_1\p5.cpp

```
#include <iostream.h>
#include <stdlib.h>
class A
{
   int hod=0;//CHYBA! Datové členy se nesmí inicializovat!
  public:
   A(int h) {hod=h; cout << "Konstruktor A" << endl;}
   void Vypis() const;
   ~A(){cout << "Destruktor A" << endl;}
};
//Následující kód nedefinuje metodu třídy A, ale
//samostatnou řadovou funkci!
void Vypis() const
{
   cout<<"Hodnota proměnné je: "<< hod << endl;
}
int main()
{
  A a;//CHYBA! Není definován konstruktor bez parametrů!
  a.Vypis();//CHYBA! Metoda není definovaná!
  return 0;
}
```

1:1 | Modified | Insertion | 22 lines in file

Příklad správného řešení:

```
#include <iostream.h>
#include <stdlib.h>
class A
{
   int hod;
```

```
  public:
   A(int h) {hod=h; cout << "Konstruktor A" << endl;}
   void Vypis() const;
   ~A(){cout << "Destruktor A" << endl;}
};

void A::Vypis() const
{
   cout<<"Hodnota proměnné je: "<< hod << endl;
}
int main()
{
  A a(10);
  a.Vypis();
  return 0;
}
```

Shrnutí kapitoly

Třída obsahuje členská data (vlastnosti) a metody (členské funkce). Třídu můžeme definovat pomocí klíčových slov **class**, **struct** a **union**. Přístupová práva jednotlivých prvků mohou být **public**, **private** a **protected**.
Metody ve třídě si můžeme rozdělit na:
- *konstruktory* - konstruktory mohou být přetížené a mají stejné jméno jako třída. Nevytvoříme-li explicitní konstruktory, bude vytvořen jeden implicitní konstruktor. Jazyk C++ umožňuje vytvořit *copy constructor*, který se využívá při vytváření kopií daného objektu.
- *destruktor* - je právě jeden (implicitní nebo explicitní)
- *změnové metody* - jejich úkolem je úprava vlastností (členských dat) objektu
- *přístupové metody* - metody, které nemění vlastnosti objektu, pouze zpřístupňují hodnoty těchto vlastností

Kód metod můžeme psát přímo v deklaraci třídy nebo (což je vhodnější) až za ní. Při samostatném definování metody je potřeba do její hlavičky vložit název třídy s přístupovým operátorem čtyřtečky.

```
//definice metody
typ JmenoTridy::JmenoMetody(parametry_metody)
{
    //kód metody - členské funkce
}
```

4. STATICKÉ DATOVÉ ČLENY A FUNKCE. PŘÁTELÉ.

Cílem lekce je objasnit pojmy *statické prvky třídy* a *přátelé* v programovacím jazyce C++ a naučit se je využívat při tvorbě vašich programů. Na praktických příkladech si můžete prohlédnout využití popisovaných mechanizmů.

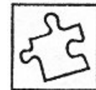
Po absolvování lekce by student měl být schopen:

- umět využívat statické členy a metody při definování tříd
- schopni využívat statické prvky třídy místo méně bezpečných globálních proměnných
- umět vytvářet k vámi definovaným třídám spřátelené řadové funkce, metody jiných tříd nebo celé spřátelené funkce
- schopni vytvářet programy, ve kterých využijete popisované konstrukce

Klíčová slova této kapitoly:
Friend, static, statické prvky třídy, statické metody

Čas potřebný k prostudování učiva kapitoly:
2 hodiny

Vstupní test



1) Označte datové členy, teré nelze uvedeným způsobem definovat jako součást třídy.

```
class NejakaTrida
{
  private:
      const int Max = 100;
      int celkem = 0;
      float f;
      NejakaTrida  *poin;
};
```

2) Které z metod třídy používají u své hlavičky klíčové slovo **const**?

přístupové metody
konstantní metody
změnové metody
konstruktor

3) Které z metod třídy nemohou využívat mechanizmus přetížrní funkcí?

přístupové metody
destruktor
změnové metody
konstruktor

Statické datové členy a funkce

Statické členy třídy definujeme pomocí klíčového slova **static** a jsou sdíleny všemi instancemi dané třídy. Statické prvky jsou uloženy mimo objekty dané třídy a existují nezávisle na jednotlivých instancích. Dokonce i v případě, že neexistuje žádná instance dané třídy. Pozor! Před použitím instancí nesmíme zapomenout inicializovat statická členská data.

Statické datové členy mohou díky svým vlastnostem nahradit používání globálních proměnných!

Statické metody se většinou chovají jako běžné řadové funkce a liší se od nich obvykle pouze přístupovými právy. Můžeme je volat přímo, bez prostřednictví své instance. Statické metody nemohou být *virtuální* a nemohou se přetížit.

Kontrolní úkol:

Pokuste se zdůvodnit, proč není vhodné používat velké množství globálních proměnných.

Příklad

```
#include <iostream.h>

class stromy
{
  private:
    static int celkem;   //celkový počet všech stromů
    int pocet;           //počet stromů určitého
druhu
  public:
    stromy(int p) {celkem+=p;pocet=p;} //konstruktor
    static void VypisCelkem();     //statická metoda
    void VypisDruhu();
    ~Stromy();               //destruktor
};

stromy::stromy(int p)
{
  celkem+=p;
  pocet=p;
}
stromy::~stromy()
{
  celkem-=pocet;
}
void stromy::VypisCelkem()
{
  cout << "Celkový počet všech stromů " << celkem <<
endl;
}
```

```
void stromy::VypisDruhu()
{
  cout << "Počet stromů jednoho druhu " << pocet <<
endl;
}


int stromy::celkem=0;
     //inicializace statických členských dat

int main()
{
   stromy park(10), sad(23);

   park.VypisDruhu();
   sad.VypisDruhu();
   stromy::VypisCelkem(); //volání statické metody
   //vypíše celkem 33
   {
     stromy zahrada(7);
     stromy::VypisCelkem(); //volání statické metody
     //vypíše celkem 40
   } //zavolá se destruktor pro instanci zahrada

   stromy::VypisCelkem(); //volání statické metody
   //vypíše se opět 33
   return 0;
}
```

Všimněte si, že statická položka celkem ve třídě *stromy* nahrazuje globální proměnnou, kterou bychom museli zavést. Tato globální proměnná by sloužila k uchovávání hodnot o celkovém počtu všech stromů. Nebezpečí takto definované proměnné je v jejím osamocení - není zapouzdřená (schovaná) v objektech a tedy je možné ji libovolně měnit. Naproti tomu statický prvek *celkem* je přístupný jen přes metody třídy a nemůže dojít k nesprávnému a neautorizovanému přístupu.

Volání statické metody může být provedeno přes jednotlivé objekty, ale z hlediska přehlednosti (čitelnosti) je vhodnější volat metodu přes jméno třídy. Uvědomte si, že statická metoda nesouvisí je s jedním objektem, ale se všemi existujícími.

```
park.VypisCelkem();
sad.VypisCelkem();
stromy::VypisCelkem();
```

Přátele

V reálném životě jsme obklopeni, kromě běžných, ostatních lidí, také zvláštní skupinou, kterým říkáme přátele a máme k nim výjimečný vztah. Půjčujeme jim osobní věci, mohou nás kdykoliv navštěvovat a jsme pro ně ochotni udělat téměř vše oč požádají.

Podobná filosofie platí i v jazyce C++. Zapouzdření sice jasně vymezuje přístup ke členům třídy, ovšem výjimka z tohoto pravidla jsou právě přátelé - **friend**. Přátelé mají plná přístupová práva ke všem členům dané třídy, i když jimi nejsou. Nemohou však pracovat s ukazatelem **this**. Přáteli se mohou stát jednotlivé funkce, některé metody jiných tříd nebo i celé třídy.

```
class A
{
   friend int f(A a) {......}
//přátelská funkce, která není součásti žádné třídy
   friend class B;
//přátelská třída, je možné využít celou třídu B
   friend int C::metoda(A a);
//přátelská pouze daná metoda třídy C
};
```

Příklad

```
#include <iostream.h>

class rohliky;
class mleko
{
  private:
     int pocet;
  public:
     mleko(int p) {pocet=p;}
     friend int celkem(rohliky r, mleko m);
};
class rohliky
{
  private:
     int pocet;
  public:
     rohliky(int p) {pocet=p;}
     friend int celkem(rohliky r, mleko m);
};
int celkem(rohliky r, mleko m)
{
  return (r.pocet + m.pocet);
}
int main()
{
  mleko ml(50);
  rohliky rh(105);
```

```
  cout << celkem(ml,rh);
  return 0;
}
```

Při prohlídce řešeného příkladu vás jistě napadly jiné možné postupy pro napsání daného programu. Hodnoty lze vracet pomocí přístupových metod, zpracovat a pak následně pomocí změnových metod znovu zapsat do objektu. Výhodou přátel je jednodušší postup a přímý přístup ke členům.

Budete-li vytvářet přátele ke svým třídám, vždy zvažte, zda je tento postup vhodný. Vzhledem k tomu, že přátele obcházejí u zapouzdření přístupové specifikace, přestávají být data v objektu chráněna. Spřátelené funkce se dají samozřejmě obejít například pomocí mechanizmu dědičnosti.

Každá funkce, která je k dané třídě přítelem, musí mít alespoň jeden parametr a tím je odkaz na daný objekt. Tato podmínka je logická, uvědomíme-li si, že objektů dané třídy může existovat několik. Spřátelená funkce (metoda, třída) by bez odkazu "nevěděla", se kterým objektem má manipulovat.

Příklad:

```
#include <iostream.h>
class A
{
 private:
   int hod;
 public:
  A(){hod=0;}
  int VratX() const {return hod;}
  //další metody
  friend void ZmenX(A &a, int x);
};
void ZmenX(A &a, int x)
{
  a.hod=x; //hod je možné přímo měnit
}

int main(int argc, char *argv[])
{
  A ss;
  cout<<ss.VratX();
  ZmenX(ss,20);
  cout<<ss.VratX();
  ZmenX(ss,200);
  cout<<ss.VratX();

  return 0;
}
```

Uvědomte si, že funkce ZmenX je řadovou (samostatnou) funkcí a není součástí třídy A.

Opakovací test

1) Vyberte objekt, jehož součástí je statický člen *pocet*?

```
class A
{
  private:
    static int celkem;
    //další prvky
  public:
   //jednotlivé metody
};
int A::celkem=0;
void main()
{
  A o1;
  A o2,o3;
 //další příkazy
}
```

- objekt o1
- objekty o1 a o2
- všechny objekty o1, o2 i o3
- není součástí žádného objektu

2) Kolik parametrů může mít řadová funkce spřátelená s určitou třídou?
- nesmí mít žádný parametr
- musí mít nejméně tři parametry
- musí mít nejméně jeden parametr
- musí mít dva parametry

3) Napište program, ve kterém vytvoříte třídu *KaroserieAuta*. Třída bude obsahovat číslo barvy karoserie a informaci o celkovém počtu všech vyrobených karoserií.

```
class KaroserieAuta
{
  private:
    static int CelkemKaroserii;
    int CisloBarvy;
  public:
   KaroserieAuta(int);//parametr inicializuje cislo barvy
   ~KaroserieAuta();
   int ZjistiCisloBarvy() const;
   void ZmenBarvu(int);
   static int ZjistiPocetKaroserii();
};
```

V případě, že si s programem nebudete vědět rady, pošlete e-mail s dotazem na adresu tutora.

Shrnutí kapitoly

Prvky třídy, které jsou označeny klíčovým slovem static, mají vzhledem k ostatním zvláštní postavení. Nejsou totiž součástí žádného z objektu a existují i v případě, že žádný objekt nevytvoříme. Static prvek se však musí inicializovat (při tom se vyhradí paměť) před definicí prvního objektu a to v místě, kde se definují globální proměnné.

```
int A::celkem=0;
```

Všechny následně vzniklé objekty dané třídy mají odkaz na static prvek. To znamená, že prvek je sdílen všemi objekty třídy. Dá se pak využít místo samostatných globálních proměnných, jejichž používání není vhodné!

Přátele zavádějí mechanizmus, pomoci kterého můžeme obejit specifikaci přístupu definovanou zapouzdřením. Dříve než podobnou konstrukci použijeme, je potřeba si uvědomit případné následky.

5. DĚDIČNOST

Cílem lekce je naučit se pracovat a využívat dědičnosti při návrhu a tvorbě programů. Lekce je zaměřena hlavně na jednoduchou dědičnost. Bude rovněž vysvětlen rozdíl mezi dědičností a kompozicí.

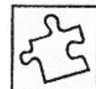
Po absolvování lekce by student měl být schopen:

- umět správně definovat pojmy dědičnost a kompozice
- umět definovat potomky tříd
- schopni vytvářet programy s třídami, které budou odvozeny pomoci jednoduché dědičnosti
- schopni rozlišit v návrhu tříd, zda využijete dědičnosti či kompozici

Klíčová slova této kapitoly:
Dědičnost, inheritance, kompozice, konstruktor, potomek, rodič, specifikace přístupu

Čas potřebný k prostudování učiva kapitoly:
3 hodiny

Vstupní test



1) Jaký typ dědičnosti umožňuje vytvářet programovací jazyk C++? Která odpověď nejsprávnější?

jednoduchou, vícenásobnou i opakovanou
pouze jednoduchou
pouze vícenásobnou
pouze opakovanou
nepodporuje dědičnost

2) Který z následujících řádků kódu vytvoří dynamický objekt třídy A? Máme definovaný pointer **A *p;**

```
p = new A;
p = (A*) malloc(sizeof(A));
p = constructor A;
p = &a;
```

3) Kolik explicitně definovaných konstruktorů musí obsahovat definice třídy?

- nemusí obsahovat žádný
- alespoň jeden
- více než jeden
- méně než deset

Co je to dědičnost?

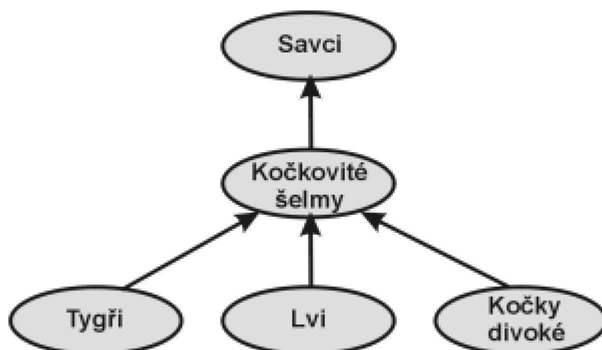
Inheritance - dědičnost znamená možnost přidávat k základní třídě další vlastnosti a schopnosti a vytvořit tak odvozenou třídu. Jsou tři možnosti, jak třídu modifikovat: přidat nové datové členy, přidat nové metody, překrýt metody novou definicí. V této kapitole se budeme zatím zabývat pouze jednoduchou dědičností. Příklad dědičnosti:

základní třída SAVCI - obsahuje vlastnosti společné pro všechny druhy savců

odvozená třída KOČKOVITÉ_ŠELMY - obsahuje vlastnosti společné pro všechny druhy savců a navíc specifické vlastnosti všech druhů kočkovitých šelem

odvozená třída LVI - obsahuje vlastnosti společné pro všechny druhy savců, specifické vlastnosti všech druhů kočkovitých šelem a navíc specifické vlastnosti společné všem lvů

Dědičnost vyjadřuje **konkretizaci** (specifikaci) odvozených tříd. Každý nový potomek je určitým způsobem konkretizací svého předka. Z příkladu je vidět, že třída *SAVCI* je nejvíce abstraktní a



každá další odvozená třída je konkrétnější.

Třída **T2** odvozená od základní třídy **T1** se deklaruje:

```
class T1 : specifikace_přístupu T1 {....};
```

nebo

```
struct T1 : specifikace_přístupu T1 {....};
```

Specifikace přístupu se provádí klíčovým slovem **public**, **private** nebo je prázdná (pak je implicitně **public**, pokud T2 je **struct**, nebo **private**, pokud T2 je **class**).

Co se nedědí:

- konstruktor. Lze jej vyvolat v konstruktoru odvozené třídy.
- destruktor. Automaticky je volán v destruktoru odvozené třídy.
- přetížený operátor =, **new**, **delete**

Atributy přístupu prvků v **T1**	Specifikace přístupu při odvození **T2** je **public**	Specifikace přístupu při odvození **T2** je **private**
Private	--------	--------
Public	*Public*	*private*
Protected	*Protected*	*private*

-------- - znamená, že zděděný prvek je nepřístupný i pro přímý přístup v metodách odvozené třídy.
protected - zděděné prvky jsou využitelné ve vlastních metodách odvozené třídy. Nejsou přímo přístupné vnějším přístupem.
public - zděděné prvky jsou využitelné ve vlastních metodách odvozené třídy. Jsou přímo přístupné také i vnějším přístupem.

Konstruktor třídy potomka

Při vytváření definic potomků je potřeba pamatovat na volání konstruktoru přímého rodiče. Tento konstruktor se nedědí, ani se automatický nevolá, tuto činnost musí zajistit programátor v kódu.

Příklad:

Třída B je dědicem třídy A. Pak při definici konstruktoru třídy B je potřeba volat konstruktor třídy A. Definujte si rovněž destruktory, ve kterých bude výpis. Podívejte se v jakém pořadí se volají konstruktory a destruktory.

```
/*
 * Po spuštění programu se podívejte, v jakém pořadí
 * se volají jednotlivé konstruktory a destruktory!
 * R. Fojtík, 2002
 */
#include <iostream.h>
class A
{
  private:
    int a;
  public:
    A(int);
    ~A();
     //jen pro představu, jak se destruktory volají
```

```
    int DejA() const;
};

class B:public A
{
  private:
    int b;
  public:
    B(int,int);
    ~B();  //jen pro představu, jak se destruktory
volají
    int DejB() const;
};

A::A(int x)
{
  cout << "konstruktor tridy A" << endl;
  a=x;
}
A::~A()
{
  cout << "destruktor tridy A" << endl;
}

int A::DejA() const
{
  return a;
}

B::B(int x, int y):A(x)
//za dvojtečkou voláme konstruktor ze třídy A, NUTNÉ!
{
  cout << "konstruktor tridy B" << endl;
  b=y;
}
B::~B()
{
  cout << "destruktor tridy B" << endl;
}

int B::DejB() const
{
  return b;
}

int main(int argc, char *argv[])
{
  A o1(10);
  B o2(1,2);

  cout <<"Objekt o1 tridy A, a = " <<o1.DejA()<<endl;
```

```
  cout <<"Objekt o2 tridy B, a = " <<o2.DejA()<<endl;
  cout <<"Objekt o2 tridy B, b = " <<o2.DejB()<<endl;

  return 0;
}
```

Rozdíl mezi dědičností a kompozicí

Při návrhu dědičnosti je potřeba postupovat uvážlivě a nevytvářet potomky tam, kde to není vhodné. Mějme například třídu *TDatum* a dále chceme vytvořit třídu osobních údajů *TOsoba*, která obsahuje údaje jako jsou jméno, příjmení, adresa a datum narození. Je asi jasné, že třída *TOsoba* není potomkem třídy *TDatum*, ale pouze obsahuje vložený objekt dané třídy. Jedná se o tzv. *kompozici*. Naopak budeme-li chtít vytvořit třídu *TZaměstnanec*, která obsahuje stejné osobní údaje jako třída *TOsoba* a některá další specifická data a metody, pak *TZaměstnanec* je potomkem *TOsoby*. Jednoduše můžeme říct, že potomky tříd vytváříme, když si můžeme kladně odpovědět na otázku **je?** (*TZaměstnanec* **je** *TOsoba*). Kompozice se vytváří při otázce **má?** (*TOsoba* **má** *TDatum*).

Kompozice vyjadřuje možnost vytváření nových tříd pomocí existujících tříd. Do nové třídy vkládáme instance jiných tříd. Například třída *TOsoba* se skládá (je využito kompozice) z objektů třídy *TDatum* (např. datum_narozeni, datum_vstupu_do_prace...), *TAdresa* (např. adresa_domu, adresa_zamestnani...), *TStrings* (např. jmeno, prijmeni...), *TRodneCislo* (např. rodne_cislo) ...

Kompozice

Jako datové prvky třídy mohou být použity i objekty jiných tříd nebo ukazatele na objekty jiných tříd. Je však samozřejmé, že prvkem třídy nemůže být objekt téže třídy. Takový prvek by totiž rekurzivně volal konstruktor, bez ustanovení hloubky rekurze. Vložené objekty nebo ukazatele na objekty se inicializují v konstruktorech dané třídy.

```
class A
{
  int hod;
  A a; //Chyba!!! Nelze!
  A *p; //je možné
public:
  A(int p){hod=p;p=new A(p);}
  ~A() {delete p;}
};
```

Řešený příklad

Vytvořte program, ve kterém využijete kompozice i dědičnosti. Nejprve vytvořte třídu *RodneCislo*, která umožní operace s hodnotou rodného čísla. Třída zkontroluje správnost rodného čísla, zjisti pohlaví osoby, datum narození.

Následně vytvořte třídu *Osoba*, která bude obsahovat jméno, příjmení, rodné číslo a potřebné metody pro práci s osobou. Od této třídy odvoďte dědice - třídu *Zaměstnanec,* která bude navíc obsahovat výšku platu zaměstnance, číslo provozu, ve kterém pracuje, a příslušné metody.

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>


class RodneCislo
{
  private:
    char rodcis[12];
    int pohlavi;         // 0 - zena, 1 - muz
  public:
    RodneCislo(char * r);
    char *DejRC() const;
    int DejDen() const;
    int DejMes() const;
    int DejRok() const;       // rozmezi let 1900 -
1999
    int DejPohlavi() const;
    void ZmenRC(char *r);
};

class Osoba
{
  private:
    char jmeno[20],prijmeni[20];
  public:
    RodneCislo rc;       // Kompozice
    Osoba(char *,char *, char *);
    char *DejJmeno() const;
    char *DejPrijmeni() const;
    void ZmenJmeno(char *);
    void ZmenPrijmeni(char *);
};

class Zamestnanec:public Osoba     //Dědičnost
{
   private:
     long int plat;
     int provoz;
   public:
```

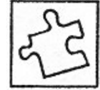
```
     Zamestnanec(char *,char *, char *, long int,
int);
     int DejProvoz() const;
     long int DejPlat() const;
     void ZmenProvoz(int);
     void ZmenPlat(long int);
};

/*********** metody tridy RodneCislo ************/
RodneCislo::RodneCislo(char *r)
{
  strcpy(rodcis,r);      // chybí oprava, zda je
číslo správné
               // součet dvouciferných čísel by měl
být dělitelný 11
  pohlavi=DejPohlavi();
}

char *RodneCislo::DejRC() const
{
  char *pom;
  pom=new char[12];
  strcpy(pom,rodcis);
  return pom;
}

int RodneCislo::DejDen() const
{
   char pom[3];

   pom[0]=rodcis[4];
   pom[1]=rodcis[5];
   pom[2]='\0';
   return (atoi(pom));
}
int RodneCislo::DejMes() const
{
   char pom[3];
   pom[0]=rodcis[2];
   pom[1]=rodcis[3];
   pom[2]='\0';

   return (DejPohlavi()==0)?(atoi(pom)-50):atoi(pom);
}

int RodneCislo::DejRok() const
{
   char pom[3];

   pom[0]=rodcis[0];
   pom[1]=rodcis[1];
```

```
   pom[2]='\0';
   return (atoi(pom)+1900);//rozmezi let 1900 - 1999
}

int RodneCislo::DejPohlavi() const
{
   char pom[3];

   pom[0]=rodcis[2];
   pom[1]=rodcis[3];
   pom[2]='\0';

   return (atoi(pom)>50)?0:1;
}

void RodneCislo::ZmenRC(char *r)
{
  strcpy(rodcis,r);
  pohlavi=DejPohlavi();
}

/*********** metody tridy Osoba ************/
Osoba::Osoba(char *j,char *p, char *r):rc(r)
{
  strcpy(jmeno,j);
  strcpy(prijmeni,p);
}
char *Osoba::DejJmeno() const
{
  char *pom;
  pom=new char[20];
  strcpy(pom,jmeno);
  return pom;
}
char *Osoba::DejPrijmeni() const
{
  char *pom;
  pom=new char[20];
  strcpy(pom,prijmeni);
  return pom;
}
void Osoba::ZmenJmeno(char *j)
{
  strcpy(jmeno,j);
}
void Osoba::ZmenPrijmeni(char *p)
{
  strcpy(prijmeni,p);
}
/*********** metody tridy Zamestnanec ************/
```

```
Zamestnanec::Zamestnanec(char *j,char *p, char *r,
long int pl, int pr)
          :Osoba(j,p,r)
{
  plat=pl;
  provoz=pr;
}
int Zamestnanec::DejProvoz() const
{
  return provoz;
}
long int Zamestnanec::DejPlat() const
{
  return plat;
}
void Zamestnanec::ZmenProvoz(int p)
{
  provoz=p;
}
void Zamestnanec::ZmenPlat(long int p)
{
  plat =p;
}
/******** pretypovani operatoru << ********/
ostream &operator<<(ostream &vys,RodneCislo r)
{
  vys << "Datum narozeni: "
<<r.DejDen()<<'.'<<r.DejMes()<<'.'
          << r.DejRok();
  return vys;
}
ostream &operator<<(ostream &vys,Osoba o)
{
  vys << "Jmeno: " << o.DejJmeno() << endl
      << "Prijmeni: " << o.DejPrijmeni() << endl
      << "Rodne cislo: " << o.rc.DejRC() << endl;
  vys << o.rc << endl
      << "Pohlavi: " ;
  char pom[5];

(o.rc.DejPohlavi()==0)?strcpy(pom,"zena"):strcpy(pom,
"muz");
  vys << pom <<endl;
  return vys;
}
ostream &operator<<(ostream &vys,Zamestnanec o)
{
  vys << "Jmeno: " << o.DejJmeno() << endl
      << "Prijmeni: " << o.DejPrijmeni() << endl
      << "Rodne cislo: " << o.rc.DejRC() << endl;
  vys << o.rc << endl
```

```
      << "Plat : " << o.DejPlat() << endl
      << "Cislo provozu: " << o.DejProvoz() << endl
      << "Pohlavi: " ;
  char pom[5];

(o.rc.DejPohlavi()==0)?strcpy(pom,"zena"):strcpy(pom,
"muz");
  vys << pom <<endl;
  return vys;
}

//************ hlavni funkce *******************
int main()
{

   RodneCislo r("655510/5555");

   cout <<
r.DejDen()<<'.'<<r.DejMes()<<'.'<<r.DejRok()<<endl;
   cout << r.DejPohlavi()<<endl;
   cout << r << endl;

   Osoba Pavel("Pavel","Novacek","671210/4455");
   cout << Pavel;

   Zamestnanec
Jiri("Jiri","Ferenc","641201/2225",120850,2);
   cout << Jiri;

   return 0;
}
/************ KONEC ************/
```

Opakovací test



1) Které z následujících prvků se nedědí do třídy potomka?
- Konstruktor
- Destruktor
- změnová metoda
- privátní prvky rodiče

2) Která z následujících otázek slouží jako pomůcka při rozhodování, zda máme vytvořit novou třídu pomoci dědičnosti?

Je?
Má?
Obsahuje?
Řídí?

3) Který z následujících prvků nesmí být prvkem třídy A?

```
A prvek;
int prvek;
A *prvek;
loat prvek[100];
```

4) Jaká bude specifikace přístupu ve třídě B k privátním prvkům třídy A? Třída B bude odvozena jako veřejný dědic třídy A.
- prvky nejsou přímo přístupné
- prvky budou přímo přístupné
- prvky budou přímo přístupné jen z vnějšku třídy
- prvky se stanou chráněnými

Shrnutí kapitoly

Dědičnost - inheritance patří mezi základní vlastnosti objektově orientovaného programování. Vyjadřuje konkretizaci, neboť potomek je určitým způsobem konkretizován - specifikován oproti rodiči. Potomek využívá kódu svého předchůdce - rodiče, přidává nové vlastnosti a schopnosti, případně staré vlastnosti a schopnosti upravuje.

Dědičnost dělíme na:

- jednoduchou
- vícenásobnou
- opakovanou

Jazyk C++ umožňuje pracovat s všemi druhy dědičnosti.

Při vytváření potomka je potřeba pamatovat na to, že se nedědí ani automatický nevolá konstruktor předka. Je jej potřeba zavolat v kódu programu!

```
B::B(int x, int y):A(x)
{
  //kód konstruktoru třídy B
}
```

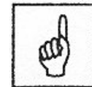
Při návrhu dalších nových tříd je potřeba si rozmyslet, zda budeme využívat *dědičnost* či *kompozici*. Pro bezchybné určení si můžeme pomoci nemotechnickými otázkami. Jednoduše můžeme říct, že potomky tříd vytváříme, když si můžeme kladně odpovědět na otázku **je?** (*TZaměstnanec* **je** *TOsoba*). Kompozice se vytváří při otázce **má?** (*TOsoba* **má** *TDatum*).

Při tvorbě třídy nelze definovat prvek, který by byl objektem tvořené třídy. Došlo by k nekonečné rekurzi při voláni konstruktoru.

6. POLYMORFISMUS

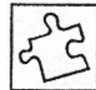
Cílem lekce je vysvětlit význam pojmu *polymorfismus* jako základní vlastnosti objektově orientovaného programování. Lekce objasňuje vztah *časné* a *pozdní vazby* a jejich využití.

Po absolvování lekce by student měl být schopen:
- vědět co je to *polymorfismus*
- schopni rozlišovat a správně využívat *časnou* a *pozdní vazbu*
- umět vytvářet a využívat *virtuální metody*
- schopni definovat *abstraktní* a *instanční třídy*



Klíčová slova této kapitoly:
abstraktní třída, časná vazba, čistě virtuální metoda, instanční třída, pozdní vazba, polymorfní třída, předefinovaná metoda, virtual, virtuální metoda, Virtual Method Table

Čas potřebný k prostudování učiva kapitoly:
3 hodiny



Vstupní test
1) Jakou specifikaci přístupu budou mít ve třídě potomka prvky, které jsou v rodičovské třídě definovány jako public? Potomek je děděn jako private.
private
public
protected
prvek nebude děděn

2) Jakou specifikaci přístupu budou mít ve třídě potomka prvky, které jsou v rodičovské třídě definovány jako protected? Potomek je děděn jako private.
private
public
protected
prvek nebude děděn

3) Jakou specifikaci přístupu budou mít ve třídě potomka prvky, které jsou v rodičovské třídě definovány jako protected? Potomek je děděn jako public.
private
public
protected
prvek nebude děděn

Úvodní příklad

Představme si dvě třídy, ve kterých nadefinujeme metody se stejnou hlavičkou, ale různým vnitřním kódem.

```
class rodic
{
  public:
   void Vypis(){cout <<"rodic"<<endl;}
};

class potomek:public rodic
{
  public:
   void Vypis(){cout <<"potomek"<<endl;}
};
void main()
{
  rodic *p;
  p=new potomek;

  p->Vypis();
 //zavolá se metoda třídy 'rodic' a ne tridy
'potomek'
}
```

Metoda Vypis() se vybere již v době překladu a ne až v okamžiku vytváření dynamického objektu. Aby bylo možné metodu přidělit dynamickému objektu až v okamžiku alokace paměti, je potřeba využít mechanizmu *časné* a *pozdní vazby*.

Časná a pozdní vazba

Překladač za normálních okolností využívá tzv. *časnou vazbu (early binding)*, která při volání metody vyhodnocuje typ instance již v době překladu. Potřebujeme-li však pracovat s potomkem pomocí ukazatele na předka, dostaneme se do problému. V tomto případě se totiž zavolá místo předefinované metody potomka původní metoda předka. Abychom se mohli vyhnout těmto problémům, zavádí jazyk C++ tzv. *virtuální metody*. Třídy, které obsahují takovéto metody se pak označují jako *polymorfní*.

Předefinovaná metoda (někdy nazývaná *překrytá* metoda) je metoda, která je opětovně definovaná ve třídě potomka. Má stejnou hlavičku (včetně parametrů) jako ve třídě rodiče, ale obsahuje jiný kód (provádí jinou činnost). Uvědomte si, že se nejedná o přetížené funkce (viz. lekce Nové prvky jazyka C++), které mají rovněž stejné názvy, ale které se jednoznačně liší typem nebo počtem parametrů.

Chceme-li se přenechat rozhodnutí, která překrytá (předefinovaná) metoda bude volána, až v průběhu programu, musíme metodu označit klíčovým slovem *virtual*. Tímto dáváme překladači najevo, že si přejeme využít *dynamickou* nebo-li *pozdní vazbu (late binding)* před *statickou* nebo-li *časnou vazbou (early binding)*.

Prohlédněte si následující příklad. Pokuste se zdůvodnit, pro které objekty se využije časná a pro které pozdní vazba. Jestliže při volání virtuální metody nepoužijeme ukazatele, nemusí se pozdní vazba uplatnit (např. *bb.Tisk( );*).

Ve výpisu programu se podívejte, v jakém pořadí se volají konstruktory a destruktory.

Příklad č.1:

```
#include <iostream.h>
#include <stdlib.h>


class A
{
      int a;
   public:
      A(int h):a(h){cout << "Konstruktor A"<<endl;}
      virtual ~A(){cout << "Destruktor A"<<endl;}
      virtual void Tisk() const { cout <<"Trida A " << a
<< endl;}
      int DejA() const {return a;}
};

class B:public A
{
      int b;
   public:
      B(int h):A(h),b(h){cout << "Konstruktor B"<<endl;}
      virtual ~B(){cout << "Destruktor B"<<endl;}
      virtual void Tisk() const { cout <<"Trida B " << b
<< " A "<< DejA()<< endl;}
};

int main()
{
  B bb(100);
/* U této instance se bude uplatňovat statická vazba
   a překladač použije správnou metodu Tisk() ze tříd B
*/

  A *pb;
/* U této instance chceme uplatnit dynamickou vazbu
   a překladač použije správnou metodu Tisk() jedině,
   v případě, že je definovaná v rodiči jako virtuální
metoda
*/
  bb.Tisk();

  pb=new B(11); //Až nyní se rozhoduji pro instanci
potomka
  pb->Tisk();   //Použije metodu ze třídy B
  delete pb;    //Uvolní se paměť a zároveň se zavolají
destruktory
  return 0;
}
```

Virtuální metody

Pokud metodu označíme slovem **virtual**, pak se metoda automaticky stává virtuální i ve všech potomcích! Klíčové slovo **virtual** není nutné v deklaraci potomků psát, ale z hlediska přehlednosti vám to jednoznačně doporučuji. Klíčové slovo se nepíše při definici virtuální funkce. Jakmile některou metodu definujeme jako *virtuální*, překladač přidá ke třídě neviditelný ukazatel, který ukazuje do speciální tabulky nazvané *tabulka virtuálních metod* (*Virtual Method Table* dále jen VMT). Pro každou třídu, která má alespoň jednu virtuální metodu překladač vytvoří tabulku virtuálních metod, ve které budou adresy všech virtuálních metod třídy. Tabulka je společná pro všechny instance dané třídy. Adresa VMT se uloží automaticky do instance konstruktorem.

Může definovat i **virtuální destruktor**. Někdy je přímo nutnost definovat destruktor jako virtuální. Máme-li například seznam různých objektů, který chceme vyprázdnit, musí se skutečný typ destruktoru určit až v průběhu programu.

Naopak konstruktory nemohou být virtuální, neboť před jejich voláním není ještě vytvořena VMT. Uvnitř konstruktorů sice můžeme volat virtuální metody, ale ty se budou chovat nevirtuálně. Důvodem je skutečnost, že před voláním konstruktoru potomka, se musí nejprve volat konstruktor předka a ten uloží do odkazu na VMT adresu tabulky virtuálních metod předka. Teprve potom přijde na řadu konstruktor potomka s odkazy na VMT adresu tabulky potomka. Podobná situace s odkazy je i u destruktorů, samozřejmě v opačném pořadí.

Polymorfismus má samozřejmě i své stinné stránky. Potřebou vytvoření VMT se zvyšují nároky na paměť, překladač musí zavést ukazatel VMT, volání virtuálních metod je pochopitelně pomalejší, než volání metod nevirtuálních (někdy se říká *statických*, což není moc vhodné, neboť statickou metodu jsme označovali metodu s klíčovým slovem **static**).

Abstraktní a instanční třídy

Při návrhu hierarchie tříd občas potřebujeme vytvořit základní rodičovskou třídu, od které se budou vyvíjet všichni potomci, ale ze které nechceme vytvářet žádné instance. Pak takovou třídu nazýváme *abstraktní třída*. Jazyk C++ nabízí možnost vytvořit čisté virtuální metodu (*pure virtual method*), která se deklaruje takto:

hlavička_metody=0;

Příklad:

```
class Objekt
{
  public:
    Objekt();
    ~Objekt();
    void Nakresli();
    void Smaz();
    virtual void Zobraz(int barva)=0;
};
```

V případě, že třída obsahuje alespoň jednu čistě virtuální metodu, pak překladač nedovolí definovat instanci této *abstraktní třídy*. Třídám, které neobsahují žádnou čistě virtuální metodu pak říkáme *instanční třídy* a lze od nich definovat instance (objekty).

Jako příkad využití si můžeme představit hierarchii tříd pro tvorbu grafického editoru. Budeme navrhovat třídy pro různé grafické objekty (kružnice, polynom, úsečka...). Všechny tyto objekty mají některé vlastnosti a schopnosti shodné či podobné. Proto se vyplatí navrhnout třídu pro obecný grafický objekt, ve kterém tyto společné vlastnosti a schopnosti částečně definujeme. Obecný grafický objekt je však příliš abstraktní, než aby šel reálně vytvořit (např. nakreslit). Proto pro jeho definici využijeme *abstraktní třídu*.

Příklad č.2

```
#include <iostream.h>

class prarodic
{
  private:
   int cisloPrarodic;
  public:
   prarodic (int);
   virtual ~prarodic();
   virtual void Vypis();
};

class rodic:public prarodic
{
  private:
   int cisloRodic;
  public:
   rodic(int);
   virtual ~rodic();
   virtual void Vypis();
};

class potomek:public rodic
{
  private:
   int cisloPotomek;
  public:
   potomek(int);
   virtual ~potomek();
   virtual void Vypis();
};

//definice metod tridy prarodic
 prarodic::prarodic (int p)
 {
  cisloPrarodic=p;
  cout <<"prarodic ";
 }

 prarodic:: ~prarodic()
```

```
 {
   cout <<"prarodic ";
 }
 void prarodic::Vypis()
 {
   cout <<endl<<"cislo prarodice = "<<cisloPrarodic;
 }
//definice metod tridy rodic
 rodic::rodic (int p):prarodic(p)
 {
  cisloRodic=p;
  cout <<"rodic ";
 }

 rodic:: ~rodic()
 {
   cout <<"rodic ";
 }
 void rodic::Vypis()
 {
   cout <<endl<<"cislo rodice = "<<cisloRodic;
 }
//definice metod tridy potomek
potomek::potomek (int p):rodic(p)
 {
  cisloPotomek=p;
  cout <<"pototmek ";
 }

 potomek:: ~potomek()
 {
   cout <<"potomek ";
 }
 void potomek::Vypis()
 {
   cout <<endl<<"cislo potomka = "<<cisloPotomek;
 }

const int Max=10;

int main(int argc, char *argv[])
{

  prarodic *pole[Max];

  cout<<endl<<"volani konstruktoru:"<<endl;
  cout<<"1.prvek: ";
  pole[0]=new prarodic(1);
  cout<<"2.prvek: ";
  pole[1]=new rodic(2);
  cout<<"3.prvek: ";
  pole[2]=new potomek(3);
  cout<<"4.prvek: ";
  pole[3]=new prarodic(4);
  cout<<"5.prvek: ";
  pole[4]=new potomek(5);
```

```
  cout<<"6.prvek: ";
  pole[5]=new rodic(6);
  cout<<"7.prvek: ";
  pole[6]=new rodic(7);
  cout<<"8.prvek: ";
  pole[7]=new prarodic(8);
  cout<<"9.prvek: ";
  pole[8]=new potomek(9);
  cout<<"10.prvek: ";
  pole[9]=new prarodic(10);

  for(int i=0;i<Max;i++)
   pole[i]->Vypis();

  //uvolneni pameti
  cout<<endl<<"volani destruktoru:"<<endl;
  for(int i=0;i<Max;i++)
   {
     cout<<i+1<<".prvek: ";
     delete pole[i];
   }

  return 0;
}
```

Opakovací test

1) Co musí obsahovat třída, aby mohla být nazývána abstraktní třídou?

čistě virtuální metodu
prvek typu static
dynamický objekt
virtuální metodu

2) U které z následujících metod může být případně uplatněna pozdní vazba?

virtuální metoda
konstruktor
datový prvek třídy
static metoda

3) Která z následujících metod nesmí být virtuální?

změnová metoda
konstruktor
destruktor
přístupová metoda

Shrnutí kapitoly

Polymorfismus patří mezi základní vlastnosti objektově orientovaného programování. Souvisí s dědičností a určuje, které předefinované metody objekt využije.

Pomocí *časné* a *pozdní vazby* můžeme rozlišit, která z předefinovaných metod se přiřadí k vytvářenému objektu.

Předefinovaná metoda je metoda, která je opětovně definována ve třídě potomka. Má stejnou hlavičku jako v rodičovské třídě, ale liší se kódem.

Chceme-li využít u některé metody pozdní vazbu (t.j. vybrat konkrétní metodu až při definici objektu a ne již v době překladu programu), pak metodu musíme označit slovem **virtual**. Virtuální metody jsou ukládány do speciální tabulky *Virtual Method Table* (VMT), která je společná pro všechny instance dané třídy. Virtuální nesmí být konstruktor, neboť před jeho voláním ještě objekt neexistuje, tedy nezná adresu VMT.

Obsahuje-li třída alespoň jednu *čistě virtuální metodu*, pak hovoříme o *abstraktní třídě*. Překladač nám u této třídy nedovolí vytvořit instanci (objekt).

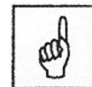
7. VÍCENÁSOBNÁ DĚDIČNOST

Cílem lekce je seznámit se s mechanizmem tvorby tříd pomocí vícenásobné dědičnosti. Objasníme si, jak se definuje *virtuální bázová třída* a jakým způsobem je potřeba tvořit a volat konstruktory v rámci vícenásobné dědičnosti.

Po absolvování lekce by student měl být schopen:

- vědět co je to *polymorfismus*
- umět vytvářet vícenásobnou dědičnost
- schopni vytvářet *virtuální bázové třídy*
- umět správným způsobem vytvářet konstruktory pro třídy vytvářené pomocí vícenásobné dědičnosti
- vědět, jak se volají a zpracovávají konstruktory a destruktory při vícenásobné dědičnosti

Klíčová slova této kapitoly:
vícenásobná dědičnost, virtual, virtuální bázová třída



Čas potřebný k prostudování učiva kapitoly:
3 hodiny

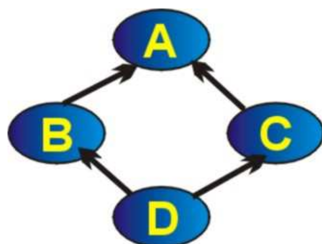
Máme-li třídu vytvořit jako potomka více než jedné třídy najednou, hovoříme o tzv. *vícenásobné dědičnosti*. Téměř každý problém lze sice zvládnout jen pomoci jednoduché dědičnosti, ale mnohdy nám vícenásobná dědičnost zjednoduší a zkrátí dobu návrhu a řešení (viz. datové proudy).

Příklad:

```
class A
{    //........ };
class B
{    //........ };
class C
{    //........ };
class ABC:public A,public B,public C
{
     //........
  public:
     ABC(int a, int b, int c):C(c),B(b),A(a) {//...}
};
```

Pořadí rodičovských tříd při deklaraci třídy určuje pořadí volání konstruktorů rodičovských tříd a opačné volání destruktorů těchto tříd. Je-li však při definici konstruktoru potomka předepsáno pořadí volání rodičovských konstruktorů, pak toto pořadí má přednost před pořadí v deklaraci třídy. Stejně jako u jednoduché dědičnosti může potomek zastoupit předka. Přetypování ukazatele na třídu potomka na ukazatel na třídu předka se v C++ děje automaticky. Obráceným způsobem se přetypování neděje automaticky, ale je potřeba je explicitně zajistit pomocí příslušných operátorů.

Pokusme se vysvětlit, jak by vypadal návrh tříd podle dědičnosti nakreslené na obrázku. Třídu *A* budeme definovat jako *virtuální bázovou třídu*.



Při vícenásobné dědičnosti je však potřeba postupovat v návrhu opatrně. Může se například stát, že dva nebo více předků obsahuje prvky stejného jména. Mějme například rodičovskou třídu *A*, její dva potomky třídu *B* a *C*. Tyto dvě třídy mají pak společného potomka třídu *D*. V takovémto případě je nutné třídu *A* definovat jako *virtuální bázovou třídu*, aby se v instanci třídy *D*, nevolal konstruktor třídy *A* dvakrát.

Příklad:

```
class A
{    //..... };
class B: virtual public A
{    //..... };
class C: virtual public A
{    //..... };
class D: public B, public C
{    //..... };
```

V případě, že třídu *A* zdědíme jak virtuálně, tak i nevirtuálně, se všechny virtuálně zděděné podobjekty dané třídy sloučí. To znamená, že výsledná třída bude obsahovat tolik podobjektů dané třídy *A*, kolikrát je tato třída nevirtuálním předkem, a jeden společný (sloučený) podobjekt *A* za všechny virtuální předky třídy *A*.

Při vytváření konstruktorů platí, že nejprve se volají virtuální konstruktory v pořadí, v němž jsou v deklaraci, a po nich nevirtuální konstruktory v pořadí určeném deklarací. Destruktory se volají v opačném pořadí.

Řešený příklad

Všimněte si volání konstruktorů a destruktorů v následujícím příkladu. Zjistěte, jaká nastane změna, nebude-li třída B a C virtuální potomkem třídy A. V našem příkladu je nutné u třídy D zajistit volání konstruktoru třídy A. Toto volání by nebylo nutné jen v případě, že třída A bude mít implicitní nebo bezparametrický konstruktor, jehož volání při konstrukci objektu třídy D by bylo zajištěno automaticky.

```
#include <iostream.h>

class A
{
```

```
   int hod;
  public:
   A(int h) {hod=h; cout << "Konstruktor A" << endl;}
   void f() const;
   ~A(){cout << "Destruktor A" << endl;}
};

class B:virtual public A
{
   int bb;
  public:
   B(int h):A(h) {bb=h;cout << "Konstruktor B" <<
endl;}
   void g() const;
   ~B(){cout << "Destruktor B" << endl;}
};

class C:virtual public A
{
   int cc;
  public:
   C(int h):A(h) {cc=h;cout << "Konstruktor C" <<
endl;}
   void h() const;
   ~C(){cout << "Destruktor C" << endl;}
};

class D: public B, public C
{
   int dd;
  public:
   D(int h):A(h+1),B(h+2),C(h+3)
{dd=h;cout << "Konstruktor D" << endl;}
   void fce() const;
   ~D(){cout << "Destruktor D" << endl;}
};
void A::f() const
{
  cout << "funkce f" << hod << endl;
}
void B::g() const
{
  cout << "funkce g" << bb << endl;
}
void C::h() const
{
  cout << "funkce h" << cc << endl;
}
void D::fce() const
{
  cout << "funkce fce" << dd << endl;
```

```
}

void main()
{
   D d1(20);

   d1.fce();
   d1.f();
   d1.g();
   d1.h();
}
```

Opakovací test

1) Co říká následující zápis: `class B: virtual public A`

- A je virtuální bazová třída
- A je virtuálním dědicem třídy B
- B je virtuálním rodičem třídy A
- B je virtuální bázová třída

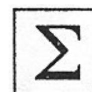
2) V jakém pořadí se zavolají konstruktory, definujeme-li kostruktor třídy D jako:

D(int h):A(h),B(h),C(h){...}
A, B, C, D
A, B, D, C
D, C, B, A
C, A, B, D

3) V jakém pořadí se zavolají destruktory, definujeme-li konstruktor třídy D jako:

D(int h):A(h),B(h),C(h){...}
A, B, C, D
A, B, D, C
D, C, B, A
C, A, B, D

Shrnutí kapitoly



Vytváříme-li třídu jako přímého potomka více než jedné třídy najednou, hovoříme o tzv. *vícenásobné dědičnosti*.

```
class ABC:public A,public B,public C
{
     //........
  public:
     ABC(int a, int b, int c):C(c),B(b),A(a) {//...}
};
```

Třída ABC je přímým potomkem tříd A, B a C. Pořadí rodičovských tříd při deklaraci třídy určuje pořadí volání konstruktorů rodičovských tříd a opačné volání destruktorů těchto tříd. Je-li však při definici konstruktoru potomka předepsáno pořadí volání rodičovských konstruktorů, pak toto pořadí má přednost před pořadí v deklaraci třídy.

Potomek může zastoupit předka. Přetypování ukazatele na třídu potomka na ukazatel na třídu předka se v C++ děje automaticky. Obráceným způsobem se přetypování neděje automaticky, ale je potřeba je explicitně zajistit pomocí příslušných operátorů.

Proto, aby se nevolal konstruktor základní rodičovské třídy vícekrát, je potřeba ji definovat jako *virtuální bázovou třídu*.

```
class A
{    //..... };
class B: virtual public A
{    //..... };
class C: virtual public A
{    //..... };
class D: public B, public C
{    //..... };
```

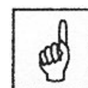
Při vytváření konstruktorů platí, že nejprve se volají virtuální konstruktory v pořadí, v němž jsou v deklaraci, a po nich nevirtuální konstruktory v pořadí určeném deklarací. Destruktory se volají v opačném pořadí.

8. PŘETĚŽOVÁNÍ OPERÁTORŮ

Cílem lekce je seznámit se s mechanizmem přetížení operátorů a s použitím tohoto mechanizmu při návrhu a implementaci programů.

Po absolvování lekce by student měl být schopen:

- umět využívat přetěžování operátorů
- umět definovat přetížené operátory
- umět definovat přetížené operátory jako součástí tříd i samostatně



Klíčová slova této kapitoly:
arita**, **asociativita**, **binární operátor**, **hononyma**, **priorita**, **přetížený operátor**, **přiřazovací operátor**, **unární operátor

Čas potřebný k prostudování učiva kapitoly:
3 hodiny

Vstupní test

1) Která z následujících funkcí je přetížená funkce k následující funkci:

```
int JmenoFunkce(int p);
    int JmenoFunkce(float p);
    int JmenoFunkce(int x);
    int Funkce(int p);
    int JmenoFunkce(int p1, int p2);
```

2) Které klíčové slovo určuje přetěžování operátoru?

operator
define
operatortype
@

3) Který z následujících operátoru nelze přetížit?

?:
++
<<
+

Dalším rozšířením jazyka je možnost přetížit nejen funkce, ale i operátory. To znamená určit jim činnost v závislosti na kontextu. Toto je možné, neboť operátor je v C++ chápan jako funkce s jedním parametrem (unární operátor) nebo se dvěma parametry (binární operátor). Při definici pak jméno rozšiřujeme o klíčové slovo **operator @**, kde znak **@** nahrazuje přetížený operátor.

Pozor! Nelze však přetížit například operátory **?:**, **.***, **::**, **sizeof** a **.** (přístup ke strukturám). U přetížení operátoru **++** a nelze určit zda se jedná o postfixový nebo prefixový přístup.

Hononyma

Programovací jazyk C++ umožňuje přetěžovat operátory, vytvářet *homonyma*. To znamená různé operace provádět pomocí operátoru se stejným jménem.

Rozdělení operátorů z hlediska možnosti přetěžování:
1) ?:, .*, ::, sizeof, . , typeid, dynamic_cast, static_cast, reinterpret_cast a const_cast nelze přetěžovat vůbec
2) ( ) (volání funkce), [ ] (indexy), -> (nepřímý přístup), = a (typ) (přetypování) lze přetěžovat jen jako nestatické metody objektových typů
3) new a delete operátory pro správu paměti

Ostatní operátory můžeme přetěžovat jako nestatické metody objektových typů i jako řadové funkce (ty, které nejsou součástí tříd).

Pravidla pro tvorbu přetížení operátorů

U přetěžovaných operátorů zůstává zachována původní *priorita, asociativita* i *arita* (počet operandů) standardních operátorů. Z toho vyplývá, že i nově definovaný binární operátor * bude mít přednost, tj. vyšší prioritu než binární operátor +. Jazyk neumožňuje prioritu změnit. Asociativita naproti tomu určuje pořadí provádění shodných operátorů. Například výraz:

```
prom1 + prom2 + prom3
```

se interpretuje

```
(prom1 + prom2) + prom3
```

stejně i u všech nově definovaných binárních operátorů +. Jazyk C++ neumožňuje vytvářet zcela nové operátory. Např. **** (druhá mocnina).

Operátorová funkce musí být členem určité třídy nebo musí mít alespoň jeden parametr, který je instancí třídy.

Příklad:

```
class A
{
....
   public:
      A operator++(); //prefixovy operator inkrementace
      A operator++(int);    //postfixovy operator
inkrementace
      A operator&(A); //binarni operator &
      A operator&(); //unarni operator &
      A operator+(A,A); //chyba! Snaha o ternarni operator
};
```

Přiřazovací operátor

```
= != += *= /= %= >>= <<= ^= |=
```

Jednoduchý přiřazovací operátor se může přetěžovat pouze jako nestatická metoda objektového typu. Složené přiřazovací operátory lze přetěžovat i jako řadové funkce.

Binární operátory

```
+ - * / % > < >= <= == != && || & | ^ >> <<
```

V případě, že definujeme binární operátor jako samostatnou, řadovou funkci, musí mít dva parametry, z nichž alespoň jeden musí být objektového nebo výčtového typu. Naproti tomu u binárních operátorů definovaných jako metoda tříd, deklarujeme pouze jeden parametr (pravý operand). Levý operand je vyjádřen instancí, jejichž metodou operátor je.

Unární operátory

```
! ~ + -
```

Tyto operátory se definují buď jako řadové funkce s jedním parametrem objektového nebo výčtového typu nebo jako metoda bez parametrů (operand bude instance dané třídy).

```
++ --
```

U operátorů **++ --** je potřeba rozlišit prefixový a postfixový tvar. Překladač nová homonyma chápe jako prefixové operátory. Pro definování prefixového tvaru musíme operátor vytvořit buď jako metodu bez parametrů nebo jako samostatnou řadovou funkci s jedním parametrem objektového nebo výčtového typu. Naproti tomu pro postfixový tvar musíme operátor vytvořit buď jako metodu s jedním parametrem typu *int* nebo jako samostatnou řadovou funkci s dvěma parametry. První je objektového nebo výčtového typu, druh typu *int*.

Operátor indexování **[ ]** lze přetížit pouze jako nestatickou metodu objektového typu s jedním parametrem.

Příklad

Následující program řeší využití přetížení operátorů **+** a **<<** pro komplexní čísla. První řešení využívá operátorové řadové funkce.

```
#include <iostream.h>

struct complex
{
    double re,im;
};

//definice přetíženého operátoru
complex operator+(complex a, complex b)
{
  complex pom;

  pom.re=a.re+b.re;
  pom.im=a.im+b.im;
  return pom;
}
```

```
//přetypování výstupního operátoru
ostream &operator<<(ostream &vys, complex x)
{
 vys << x.re << " + i. " << x.im;
 return vys;
}

int main()
{
  complex VYS,X={1.0,2.0},Y={3.0,4.0};

  VYS=X+Y;
  cout << VYS << endl;

return 0;
}
```

Následující program řeší využití přetížení operátorů **+** a **<<** pro komplexní čísla. Toto řešení využívá přístupu pomoci třídy a přetížené operátory definujeme jako metody dané třídy nebo spřátelené řadové funkce.

```
#include <iostream.h>
class COMPLEX
{
   double re, im;
  public:
   COMPLEX(double, double);
   COMPLEX() {re=im=0.0;}
//   COMPLEX operator+(COMPLEX);   //meně vhodné
řešeni
   friend COMPLEX operator+(COMPLEX,COMPLEX);
   friend ostream &operator<<(ostream &vys, COMPLEX
x);
   COMPLEX & operator+=(COMPLEX);
};

COMPLEX::COMPLEX(double r, double i)
{
  re=r;
  im=i;
}

/*COMPLEX COMPLEX::operator+(COMPLEX x)
{
     re=re+x.re;     //nevhodne, meni se jeden z
operandu
     im=im+x.im;
     return *this;
} */
```

```
inline COMPLEX & COMPLEX::operator+=(COMPLEX x)
{
   re += x.re;
   im += x.im;
   return *this;
}

// pratelske radove funkce
inline COMPLEX operator+(COMPLEX x1,COMPLEX x2)
{
   return COMPLEX(x1.re + x2.re,x1.im + x2.im);
}

/*  jina moznost
//moznost jak se vyhnout funkci friend,
//musi byt ale definovan operator +=
COMPLEX operator+(COMPLEX x1,COMPLEX x2)
{
  COMPLEX x=x1;
  x +=x2;
  return x;
} */

ostream &operator<<(ostream &vys, COMPLEX x)
{
 vys << x.re << " + i. " << x.im;
 return vys;
}

/*****************************/
int main()
{
  complex VYS,X={1.0,2.0},Y={3.0,4.0};

  VYS=X+Y;
  cout << VYS << endl;

  COMPLEX v, a(11.0,12.0),b(13.0,14.0);
  v = a + b;
  cout << "v= " << v << endl;
  cout << "a= " << a << endl;
  cout << "b= " << b << endl;

  v += a;
  cout << "v= " << v << endl;
  cout << "a= " << a << endl;
  cout << "b= " << b << endl;

  return 0;
}
```

Opakovací test

1) Jaká podmínka platí pro parametry přetížených operátorů?
- alespoň jeden parametr musí být objektového typu
- operátor musí mít nejméně dva parametry
- všechny parametry musí být jednoduchého typu
- oba parametry musí být stejného typu

2) Kolik parametrů bude mít přetížený operátor binární +, je-li operátor definován jako součást třídy?

jeden parametr
žádný parametr
dva parametry
libovolný počet parametrů

3) Kolik parametrů bude mít přetížený operátor binární +, je-li operátor definován jako samostatná řadová operátorová funkce?

jeden parametr
žádný parametr
dva parametry
libovolný počet parametrů

4) Který z následujících zápisů může definovat přetížený operátor pro sčítání prvků. Operátor je součásti třídy s názvem Tclass?

static Tclass operator+(Tclass, Tclass);
Tclass operator+(Tclass);
Tclass operator+(Tclass,Tclass,Tclass);
Tclass operator+(Tclass,Tclass);

Shrnutí kapitoly

Stejně jako funkce, lze v jazyce C++ přetížit i operátory. To znamená, že vytvoříme *hononyma*, naučíme operátory pracovat s novými daty.
Pro přetížení operátorů využíváme klíčové slovo **operator @**. Znak @ nahrazuje přetížený operátor.
Nelze však přetížit například operátory **?:**, **.***, **::**, **sizeof** a **.** (přístup ke strukturám). U přetížení operátoru **++** a nelze určit zda se jedná o postfixový nebo prefixový přístup.
U přetěžovaných operátorů zůstává zachována původní
- *priorita*
- *asociativita*
- *arita* (počet operandů)

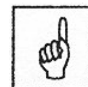
jako standardních operátorů.
Jazyk C++ neumožňuje vytvářet zcela nové operátory. Např. ****** (druhá mocnina).
Není vhodné přetěžovat operátory pro neočekávanou činnost. Například je nevhodné, aby binární operátor + uměl násobit čísla určitých typů.
Operátorová funkce musí být členem určité třídy nebo musí mít alespoň jeden parametr, který je instancí třídy.

9. ŠABLONY

Cílem této lekce je vysvětlit pojem šablon a jejich uživí při tvorbě a návrhu programů.

Po absolvování lekce by student měl být schopen:

- umět definovat šablony pro řadové funkce
- umět definovat šablony pro třídy
- schopni zjednodušit návrh a implementaci programu pomocí návrhu šablon



Klíčová slova této kapitoly:
generické konstrukce, hodnotové parametry, metatřída, parametrizované konstrukce, šablona, template, typename, typové parametry

Čas potřebný k prostudování učiva kapitoly:
2 hodiny

Vstupní test

1) Jak nazýváme dvě funkce, které mají stejný název, ale liší se druhem nebo počtem parametrů?
přetížené funkce
předefinované funkce
funkci nelze definovat
shodné funkce

2) Které z následujících prvků jazyka nelze v programu přetížit?
proměnné
funkce
operátory
metody

Šablona specifikuje, jak definovat skupinu příbuzných tříd. Mějme například vytvořenou třídu zásobník, která uchovává celá čísla. Budeme-li však chtít pracovat v zásobníku s řetězci znaků, je námi vytvořená třída nepoužitelná, přestože operace na zásobníku jsou stejné a jen se změnil datový typ jednotlivých hodnot na zásobníku. Abychom nemuseli psát nový kód pro další datový typ, poskytuje jazyk C++ nástroj umožňující vytvořit abstraktní vzor zvaný šablony (templates). Někdy hovoříme také o generických nebo parametrizovaných konstrukcích. Šablony jsou někdy označovány jako metatřídy. Teprve její "instancí" vzniká běžná třída.

```
template <class typ> class AA{ ....};
```

V lomených závorkách jsou formální parametry, které mohou být buď *typové* nebo *hodnotové*. S hodnotovými parametry se setkáváme u obyčejných funkcí (int, unsigned, atd; nelze použit objektové typy nebo pole). Můžeme u nich předepsat implicitní

hodnoty. Naopak typové parametry jsou uvedeny klíčovým slovem **class** (v novějších překladačích je možné použít i **typename**) a specifikují datové typy.

Příklad šablony řadových funkcí (funkcí, které nejsou metodami tříd):

```
template <class typ> typ Maximum(typ a,typ b);
     //deklarace

template <class typ> typ Maximum(typ a,typ b)
// nebo template <typename typ> typ Maximum(typ a,typ
b)
{
  return (a<b)<b:a;
}
```

Je však potřeba si uvědomit, že při generování instance šablony se neprovádějí ani triviální konverze parametrů, to znamená,že s následujícím příkladem si naše šablona neporadí.

```
const int X=100;
char c = A;
int y = 25;
cout << Maximum(X,y);    //chyba! V Borland C/C++ 3.1
pracuje!
cout << Maximum(c,y);    //chyba!
Cout << Maximum((int)c,y);    //správně
```

Příklad šablony řadových funkcí (funkcí, které nejsou metodami tříd):

```
template <class typ> typ Maximum(typ a,typ b);
     //deklarace

template <class typ> typ Maximum(typ a,typ b)
// nebo template <typename typ> typ Maximum(typ a,typ
b)
{
  return (a<b)<b:a;
}
```

Deklarace šablon objektového typu má tvar:

```
template <class typ> class AA;

template <class typ> class AA
{
   typ h;
  public:
   AA(typ x);
   typ DejA();
};
//definice metod šablony
template <class typ> AA<typ>::AA(typ x)
{
  h=x;
}
```

```
template <class typ> typ AA<typ>::DejA()
{
  return h;
}

int main()
{
  AA<int> a(15);          //instance šablony
  cout << a.DejA();

  return 0;
}
```

Pro většinu šablon je potřebné, aby se daly třídy pro instanci kopírovat a měly bezparametrický konstruktor.

Je vhodné minimalizovat závislost na externích jménech a operacích. Takovou operací může být například výpis pomocí operátorů <<, který je definován jen pro omezenou množinu datových typů. Máme-li v šabloně takovýto operátor, snižujeme počet možnosti využití šablony. Pro eliminaci problému je pak potřeba předefinovat potřebné operace pro další datové typy.

Příklad č.1

Následující kód ukazuje jednoduchou šablonu pro zásobník, který k uložení prvků využívá dynamicky vytvořené pole. Šablona nelze využít pro datové typy, které nemají definován přiřazovací operátor =.

```
#include <iostream.h>

template <typename typ>class Zasobnik
{
  private:
   typ *p;
   int vrchol;
   int max;
   static typ chyba; //implicitní hodnota
  public:
   Zasobnik(int);    //konstruktor
   ~Zasobnik();      //destruktor
   void PridejNaVrchol(typ);
   typ OdeberZVrcholu();
   bool JePrazdny();
};

template <typename typ> Zasobnik <typ>::Zasobnik(int
pocet)
{
  p=new typ[pocet];//vhodné ošetřit, zda se alokovala
paměť!
  max=pocet;
  vrchol=-1;//pole zacina od O
```

```
}

template <class typ> Zasobnik <typ>::~Zasobnik()
{
  delete []p;
}

template <class typ> void Zasobnik
<typ>::PridejNaVrchol(typ x)
{
  if (vrchol<max-1)
  {
    vrchol++;
    p[vrchol]=x;
  }
  else
    cout << "Chyba! Zasobnik je plny! Nelze
pridavat!"<<endl;
}

template <class typ> typ Zasobnik
<typ>::OdeberZVrcholu()
{
   if (vrchol>=0)
   {
     return p[vrchol--];
   }
   else
   {
     cout<<"Chyba! Zasobnik je prazdny! Nelze
odebirat!"<<endl;
     return chyba;
   }
}

template <class typ> bool Zasobnik <typ>::JePrazdny()
{
   return (vrchol<0)? true : false;
}

int Zasobnik<int>::chyba=-1;
double Zasobnik<double>::chyba=-1.0;
char Zasobnik<char>::chyba=' ';

int main(int argc, char *argv[])
{
   //zásobník pro typ int
   Zasobnik<int> s1(10);
   //vytvoří se zásobník maximálně pro 10 prvků
   //přidání desítí prvků do zásobníků
   for(int i=10;i<20;i++) s1.PridejNaVrchol(i);
```

```
    //vybírání prvků
    cout<<"Prvky v zasobniku:"<<endl;
    while (s1.JePrazdny()!=true)
     cout<< s1.OdeberZVrcholu()<<endl;

    //zásobník pro typ double
    Zasobnik<double> s2(10);
    //vytvoří se zásobník maximálně pro 10 prvků
    //přidání prvků do zásobníků
    s2.PridejNaVrchol(1000.11);
    s2.PridejNaVrchol(1456.78);
    s2.PridejNaVrchol(3456.67);
    s2.PridejNaVrchol(4321.43);
    s2.PridejNaVrchol(9876.54);
    //vybírání prvků
    cout<<"Prvky v zasobniku:"<<endl;
    while (s2.JePrazdny()!=true)
     cout<< s2.OdeberZVrcholu()<<endl;

    //zásobník pro typ char
    Zasobnik<char> s3(10);
    //vytvoří se zásobník maximálně pro 10 prvků
    //přidání prvků do zásobníků
    s3.PridejNaVrchol('a');
    s3.PridejNaVrchol('b');
    s3.PridejNaVrchol('c');
    s3.PridejNaVrchol('d');
    //vybírání prvků
    cout<<"Prvky v zasobniku:"<<endl;
    while (s3.JePrazdny()!=true)
     cout<< s3.OdeberZVrcholu()<<endl;

  return 0;
}
```

Příklad č.2

V návaznosti na řešený příklad ze zásobníkem, vytvořte šablonu pro frontu. Prvky budeme opět ukládat do dynamicky vytvořeného pole. Doporučuji nejprve odladit třídu *fronta* pro konkrétní jednoduchý datový typ (např. int). Nezapomeňte ošetřit posun prvku v poli. Prvky se přidávají na konec a odebírají ze začátku. Časem se může stát, že první prvek fronty se vybíráním posune až na konec pole! Nejvhodnější je využít tzv. setřásání pole. Tedy jakmile se nám uvolní na začátku pole prvky. Popřesunujeme (setřeseme) další prvky na začátek pole.

Shrnutí kapitoly

Pomocí šablon specifikujeme, jak definovat skupinu příbuzných tříd. Šablony nazýváme také *generické* nebo *parametrizované konstrukce*, případně jako *metatřídy*. Šablona se definuje:

```
template <class typ> class AA
{
   typ h;
  public:
   AA(typ x);
   typ DejA();
};
//definice metod šablony
template <class typ> AA<typ>::AA(typ x)
{
  h=x;
}
```

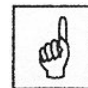
V lomených závorkách jsou formální parametry, které mohou být buď *typové* nebo *hodnotové*. Typové parametry jsou uvedeny klíčovým slovem **class** nebo **typename**.

10. STANDARDNÍ KNIHOVNA ŠABLON

Cílem této lekce je vysvětlit využití standardních knihoven šablon a STL.

Po absolvování lekce by student měl být schopen:

- umět používat šablony, které jsou obsaženy v standardní knihově



Klíčová slova této kapitoly:
generické konstrukce, šablona, template, typename, typové parametry, STL, kontejnery, standardní knihovna

Čas potřebný k prostudování učiva kapitoly:
3 hodiny

Komponenty knihovny STL

STL (Standard Template Library) představuje obecnou knihovnu řešící správu kolekcí dat prostřednictvím moderních algoritmů.

Knihovna je složena z:

- Kontejnery – jsou implementovány jako pole, vázáné seznamy nebou mohou mít pro každý prvek speciální klíč
- Iterátory – umožňují procházení prvků kolekcí objektů (kontejnery nebo jejich podmnožiny).
- Algoritmy – zpracovávají prvky kolekcí

Kontejnery

Kontejnery (kontejnerové třídy) spravují kolekce prvků.

Rozdělení:

- Sekvenční kontejnery – jsou uspořádané kolekce, kde každý prvek má určenou pozici
    - vector
    - deque
    - list
- Asociativní kontejnery – jsou seřazené kolekce, ve kterých skutečná pozice prvk závisí na jeho hodnotě a dané podmínce zařazení
    - set
    - multiset
    - map
    - multimap

Vektory

Vektor (vector) spravuje prvky v dynamickém poli. Odstraňování a přidávání prvků na konci pole relativně rychlé.

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   vector<int> p;
   ...
   p.push_back(hodnota);
   ...
   for(int i = 0; i < p.size(); i++)
      cout << p[i];
}
```

Příklad

Příklad jednoduchého použití kontejneru vector.

```
#include "stdafx.h"
#include <iostream>
#include <vector>
using namespace std;


int _tmain(int argc, _TCHAR* argv[])
{
   vector<int> p;

   for(int i = 0; i < 10; i++)
     p.push_back(i*100);

   for(int i = 0; i < p.size(); i++)
      cout << p[i] << ' ';

   cout << endl << "Vybrane operace:" << endl;
   cout << "p.at(index): " << p.at(9) << endl;
   cout << "p.front(): " << p.front() << endl;
   cout << "p.back(): " << p.back() << endl;


   system("PAUSE");
}
```

Příklad

Příklad jednoduchého použití kontejneru vector.

```
#include "stdafx.h"
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;
```

```
int _tmain(int argc, _TCHAR* argv[])
{

      vector<string> s;

      s.push_back("prvni");
      s.push_back("druhy");
      s.push_back("treti");

      copy(s.begin(),s.end(),
            ostream_iterator<string>(cout," "));
      cout << endl;

      cout <<"max_size(): " <<s.max_size() << endl;
      cout <<"size()> " << s.size() << endl;
      cout <<"capacity(): " << s.capacity() <<endl;

      swap(s[0],s[2]);
      copy(s.begin(),s.end(), ostream_iterator<string>
            (cout," "));
      cout << endl;

      s.back() = "posledni";
      copy(s.begin(),s.end(),
            ostream_iterator<string>(cout," "));
      cout << endl;

      s.insert(find(s.begin(),s.end(),"posledni"),"nove");
      copy(s.begin(),s.end(),
            ostream_iterator<string>(cout," "));
      cout << endl;

      cout <<"max_size(): " <<s.max_size() << endl;
      cout <<"size()> " << s.size() << endl;
      cout <<"capacity(): " << s.capacity() <<endl;

      system ("PAUSE");
      return 0;
}
```

Obousměrné fronty

Obousměrné fronty (deque, nebo-li double-ended queue). Jedná se o dynamické pole, které může růst oběma směry. Tedy vkládání prvků na začátek a konec je rychlé na rozdíl od vkládání doprostřed.

```
#include <iostream>
#include <deque>
using namespace std;

int main()
{
   deque<double> p;
```

```
    ...
    p.push_back(hodnota);
    p.push_front(hodnota2);

}
```

Příklad

Příklad jednoduchého použití kontejneru deque.

```
#include "stdafx.h"
#include <deque>
#include <string>
#include <iostream>
#include <algorithm>
using namespace std;


int _tmain(int argc, _TCHAR* argv[])
{
      deque<string> d;

      d.assign(3,"nejaky text");
      d.push_front("prvni");
      d.push_back("posledni");

      copy(d.begin(),d.end(),ostream_iterator<string>
            (cout,"\n"));
      cout << endl;

      d.pop_front();
      copy(d.begin(),d.end(),ostream_iterator<string>
            (cout,"\n"));
      cout << endl;

      deque<string> d2;
      d2 = d;
      copy(d2.begin(),d2.end(),ostream_iterator<string>
            (cout,"\n"));
      cout << endl;
      d2[1].swap(d2[3]);
      d2.pop_back();
      copy(d2.begin(),d2.end(),ostream_iterator<string>
            (cout,"\n"));
      cout << endl;

      d2.clear();
      cout <<"clear";
      cout << "d2.size()" << d2.size() << endl;
      cout << endl;

      system("PAUSE");
      return 0;
}
```

Seznamy

Seznam (list) je implementován jako obousměrný vázaný seznam. Každý prvek má kromě paměti pro data ukazatele na předchůdce a následovníka. Přístup k určitému prvku se děje procházením seznamu, není možný náhodný přístup. Tedy přístup je pomalejší než v předcházejících případech. Naopak výhodou je vkládání a vybírání v libovolném místě.

```
#include <iostream>
#include <list>
using namespace std;

int main()
{
   list<char> p;

   for(char c='a';c<='z';++c)
     p.push_back(c);
   ...
   while(!p.empty())
   {
      cout << p.front() << endl;
      p.pop_front();
   }
}
```

Příklad

Příklad jednoduchého použití seznamu.

```
#include "stdafx.h"
#include <iostream>
#include <list>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
      typedef list<int> Tlist;
      Tlist list1,list2;
      Tlist::iterator pos;

      for(int i=1;i<=10;i++)
      {
            list1.push_back(i);
            list2.push_front(i);
      }

      cout << endl;
      cout << "list1: ";
      copy(list1.begin(),list1.end(),ostream_iterator<int>
            (cout," "));
      cout << endl;
      cout << "list2: ";
      copy(list2.begin(),list2.end(),ostream_iterator<int>
            (cout," "));
      cout << endl;
```

```
list2.sort();
list1.merge(list2);
cout << "list1.merge(list2); - spojeni list1 a list2
           do list1: ";
copy(list1.begin(),list1.end(),ostream_iterator<int>
      (cout," "));
cout << endl;
cout << "list2: ";
copy(list2.begin(),list2.end(),ostream_iterator<int>
      (cout," "));
cout << endl;

list2 = list1;
cout << "list2 = list1" << endl;
cout << "list1: ";
copy(list1.begin(),list1.end(),ostream_iterator<int>
      (cout," "));
cout << endl;
cout << "list2: ";
copy(list2.begin(),list2.end(),ostream_iterator<int>
      (cout," "));
cout << endl;

list1.remove(10);
cout << "list1.remove(10);" << endl;
cout << "list1: ";
copy(list1.begin(),list1.end(),ostream_iterator<int>
      (cout," "));
cout << endl;

list2.unique();
cout << "list1.unique();" << endl;
cout << "list2: ";
copy(list2.begin(),list2.end(),ostream_iterator<int>
      (cout," "));
cout << endl;

list2.reverse();
cout << "list1.reverse();" << endl;
cout << "list2: ";
copy(list2.begin(),list2.end(),ostream_iterator<int>
      (cout," "));
cout << endl;

list1.clear();
cout << "list1.clear();" << endl;
cout << "list1: ";
copy(list1.begin(),list1.end(),ostream_iterator<int>
      (cout," "));
cout << endl;


list2.unique();
cout << "list2: ";
copy(list2.begin(),list2.end(),ostream_iterator<int>
```

```
            (cout," "));
      cout << endl;

      system("PAUSE");
      return 0;
}
```

Řetězce

basic_string<>
string
wstring

Normální pole

Normální statické nebo dynamické pole – nejedná se však o kontejnery

Společné vlastnosti a operace kontejnerů

Všechny kontejnery poskytují hodnotovou, odkazovou sémantiku. Nevytáříme tedy odkazy ale jijich vnitřní kopie. Proto je nutné, aby prvek kontejneru byl kopírovatelný.

Každá kontejnerová třída obsahuje implicitní konstruktor, kopírovací konstruktor a destruktor.

```
objekt.size()
objekt.empty()
objekt.max_size() – vrací maximální počet prvků
objekt.swap(objekt2)
swap(objekt1,objekt2) – globální funkce
objekt.begin() – vrací iterátor prvního prvku
objekt.end() – vrací iterátor pozice za posledním prvkem
```

Sady a multisady

Kontejnery multisady dovolují duplicity. Sady a multisady jsou implemetnovány prostřednictvím binárních stromů. Což je výhodné při vyhledávání určitého prvku. Sady a multisady neposkytují operátory pro přímý přístup k prvkům.

Prvky sad a multisad mohou ,ít jakýkoliv typ, který je možné přiřazovat, kopírovat a porovnávat podle pravidla řazení. Použité pravidlo řazení definuje druhý – volitelný parametr šablony. V případě, že nepředáme speciální pravidlo řazení, pak je využito implicitní pravidlo `less`. Funkční objekt `less<>()` řadí prvky podle porovnání operátorem <.

```
#indlude <set>

count(prvek) – vrací počet prvků s hodnotou prvek
find(prvek) – vrací pozii prvku s hodnotou parametru nebo
end()
lower_bound(prvek) – vrací první pozici, na kterou byl
prvek vložen
```

```
upper_bound(prvek) – vrací poslední pozici, na kterou byl
prvek vložen
equal_range(prvek) – vrací první a poslední pozici
vloženého prvku
c.insert(prvek) – vkládá kopii prvku a vrací pozici
nového prvku, u sad vrací informaci o úspěšnosti
c.insert(pozice,prvek)
c.insert(zacatek, komec)- vkládá kopie všech prvků
v daném rozsahu
c.erase(prvek)- vyjímá všechny prvky s danou hodnotou a
vrací počet vyjmutývh prvků
```

Mapy a multimapy

Tyto kontejnery pracují s prvky typu páry klíč-hodnota. Multimapy dovolují duplicity.

```
#indlude <map>
```

Mapy a multimapy jsou implementovány pomocí vyvážených binárních stromů. Prvky jsou řazeny podle svých klíčů. Mapy a multimapy nepodporují přímý přístup k prvkům.

Asociativní kontejnery obvykle neposkytují možnost pímého pístupu k prvkům a místo toho využívají iterátory. Výjimkou je *asociativní pole* – je možné využívat operátor indexu, který je tvořen klíčem. Jestliže použijeme index, pro který neexistuje žádný prvek, vloží se automatickz nový pvek do mapy. Tato funkčnost je však možná jen pro prvky z bezparametrickým konstruktorem!

Příklad

Práce s multimapou. Příklad využití multimapy pro implementaci jednoduchého slovníku.

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <map>
#include <string>
#include <algorithm>

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
      typedef multimap<string,string> TSlovnik;
      TSlovnik slovnik;
      TSlovnik::iterator pos;

      slovnik.insert(make_pair("car","auto"));
      slovnik.insert(make_pair("clever","chytry"));
      slovnik.insert(make_pair("clever","bystry"));
      slovnik.insert(make_pair("clever","dumyslny"));
      slovnik.insert(make_pair("intelligent","chytry"));
```

```
	cout << setw(10) << "anglicky" << setw(10)
		<<  "cesky" << endl;
	for(pos =slovnik.begin(); pos != slovnik.end();
++pos)
	 cout << setw(10) <<pos->first.c_str() << setw(10)
		<< pos->second << endl;

	cout << endl << "Preklad slova chytry: " << endl;
	for(pos=slovnik.begin(); pos!= slovnik.end(); ++pos)
		if (pos->second == "chytry")
			cout << pos->first << endl;

	cout << endl << "Preklad slova clever: " << endl;
	for(pos=slovnik.begin(); pos != slovnik.end();
++pos)
		if (pos->first == "clever")
			cout << pos->second << endl;

	pos = slovnik.find("intelligent");
	cout << setw(10)<< pos->first << setw(10) <<
		pos->second << endl;

	cout << "Pocet vyskytu klice clever " <<
		   slovnik.count("clever") << endl;
	cout << endl;
	system("PAUSE");
	return 0;
}
```

Zásobníky

Zásobník (stack) – LIFO (last in, firt out). Prvky vkládáma pomocí členské funkce `push()` a vybíráme pomocí `pop()`- funkce však prvek nevrací. Funkce `top()` vrací další prvek zásobníku bej jeho vyjmutí.

```
#include "stdafx.h"
#include <iostream>
#include <stack>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
	stack<int> s;

	s.push(1);
	s.push(2);
	s.push(3);
	s.push(4);

	cout << endl << "Vyber zasobniku:" << endl;
	while(!s.empty())
	{
		cout << s.top() << endl;
		s.pop(); //Je potřeba volat obě metody!
	}
```

```
	system("PAUSE");
	return 0;
	}
```

Fronty

Fronty (queue) – FIFO.
push() – vkládá prvek do fronty
`front()` – vrací další (první) prvek fronty
`back()` – vrací poslední prvek fronty
`pop()` – vyjímá prvek z fronty

Příklad
Příklad jednoduchého použití zásobníku.

```
#include "stdafx.h"
#include <iostream>
#include <stack>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
	stack<int> s;

	s.push(1);
	s.push(2);
	s.push(3);
	s.push(4);

	cout << endl << "Vyber zasobniku:" << endl;
	while(!s.empty())
	{
		cout << s.top() << endl;
		s.pop();
	}

	system("PAUSE");
	return 0;
}
```

Příklad
Příklad jednoduchého použití fronty.

```
#include "stdafx.h"
#include <iostream>
#include <queue>
#include <string>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
	queue<string> q;

	q.push("prvni");
```

```
	q.push("druhy");
	q.push("treti");

	while(!q.empty())
	{
		cout << q.front() << endl;
		q.pop(); //Je potřeba volat obě metody!
	}


	system("PAUSE");
	return 0;
}
```

Prioritní fronta

Prioritní fronta (priority_queue). Funkce top a pop zpřístupňují a vyjímají další prvek – není to však ten první v pořadí, ale prvek s nejvyšší prioritou.

```
#include "stdafx.h"
#include <iostream>
#include <queue>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
	priority_queue<string> pq;

	pq.push("prvni");
	pq.push("druhy");
	pq.push("treti");
	pq.push("ctvrty");

	//. . . . .
	system("PAUSE");
	return 0;
}
```

Bitové sady

Bitové sady modelují pole bitů nebo booleovských hodnot pevné velikosti. Využívají se pro sparávu sad příznaků.

```
#include <bitset>
```

Řetězce

Základní šablonová třída `basic_string` a její standardní specializace `string` a `wstring`.

```
#include <string>
```

Shrnutí kapitoly

STL (Standard Template Library) představuje obecnou knihovnu řešící správu kolekcí dat. Nejdůležitější částmi knihovny jsou kontejnery, aerátory algoritmy.

Kontejnery můžeme rozdělit na:
- Sekvenční – vector, deque, list
- Asociativní – set, multiset, map, multimap

	vector	**deque**	**list**
Obvyklá implementace - struktura dat	dynamické pole	pole polí	obousměrně vázaný seznam
Povolení duplicit	ano	ano	ano
Kategorie aerátorů	s náhodným přístupem	s náhodným přístupem	obousměrný
Hledání prvků	pomalé	pomalé	velmi pomalé
Vkládání / vyjímání prvků je rychlé	na konci	na konci a na začátku	kdekoliv
Uvolnění paměti při vyjmuti prvků	nikdy	někdy	vždy
Možnost rezervace paměti	ano	ne	-

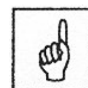
	set	**multiset**	**map**	**multimap**
Obvyklá implementace - struktura dat	vyvážený binární strom	vyvážený binární strom	vyvážený binární strom	vyvážený binární strom
Povolení duplicit	ne	ano	ne u klíčů	ano
Kategorie iterátorů	obousměrný (konstantní prvky)	obousměrný (konstantní prvky)	obousměrný (konstantní prvky)	obousměrný (konstantní prvky)
Hledání prvků	rychlé	rychlé	rychlé u klíčů	rychlé u klíčů
Vkládání / vyjímání prvků je rychlé	-	-	-	-
Uvolnění paměti při vyjmuti prvků	vždy	vždy	vždy	vždy
Možnost rezervace paměti	-	-	-	-

11. DATOVÉ PROUDY

Tato lekce se bude zabývat proudovým vstupem a výstupem dat při práci se soubory.

Po absolvování lekce budete:
- umět pracovat s datovými proudy pro vstup a výstup dat
- umět pracovat se soubory pomocí proudového přístupu

Klíčová slova této kapitoly:
Datové proudy, vstup, výstup



Čas potřebný k prostudování učiva kapitoly:
2 hodiny

Datové proudy. Práce se soubory.

Formátovaný vstup a výstup je praktický shodný i při práci se soubory. Rozdíl je v hlavičkovém souboru, který musíme k programu připojit a také v označení tříd, pomocí kterých se přístup k souboru realizuje. Jsou to *ofstream* pro výstup a *ifstream* pro vstup. Implicitně práce se vstupním nebo výstupním proudem probíhá v textovém režimu.

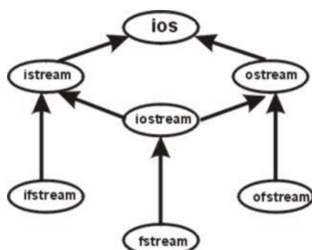
Hierarchie tříd vztahující se k datovým tokům pro soubory:

Otevření souboru je možné dvěma způsoby:

- při vzniku objektu (u konstruktoru je uvedena cesta k souboru) a nebo pomocí členské funkce *open* (v tomto případě je volán implicitní konstruktor bez parametrů).

Uzavření souboru se obdobně provádí dvěma způsoby:

- automaticky destruktorem při zániku objektu nebo členskou funkci *close*.



Otevření pomocí konstruktoru

*ifstream(const char *name, int mode = ios::in, int = filebuf::openprot);*
*ofstream(const char *name, int mode = ios::out, int prot = filebuf::openprot);*
*fstream(const char *name, int mode = ios::in, int prot = filebuf::openprot);*

První parametr je cesta k souboru, druhý parametr jsou atributy otevření souboru (viz. tabulka), třetí parametr je pro sdílení souboru.

Režim	popis činnosti
ios::app	Připojuje data na konec souboru
ios::ate	nastaví se na konec souboru
ios::in	při otevření nastaví režim čtení (implicitní pro ifstream)
ios::out	při otevření nastaví režim zápis (implicitní pro ofstream)
ios::binary	otevře soubor v binárním režimu
ios::trunc	pokud soubor existuje, zruší jeho obsah (implicitní je-li ios::out a není buď ios::ate nebo ios::app)
ios::nocreate	otevření se neprovede, pokud soubor neexistuje
ios::noreplace	existuje-li soubor, zhavaruje otevření pro výstup, není-li nastaveno ios::app nebo ios::ate

Možné parametry pro sdílení:
filebuf::sh_compact - stejné jako implicitní hodnota *filebuf::openprot*, soubor lze sdílet, pokud to povolí operační systém
filebuf::sh_none - soubor nelze sdílet
filebuf::sh_read - soubor lze sdílet jen při čtení

```
/*** příklad otevření souboru pomocí konstruktoru
***/
#include <fstream.h>

void main(void)
{
   ofstram of("soubor.dat",ios::out | ios::binary);

   if (of != 0)
   {
     float f;
          for (int i = 0; i<50, i++)
     {
          f=i*i;
          of.write((const char *)&f, sizeof(f) );
          //neformátovan zápis
     }
     of.close( );
  }
}
/***************** konec ******************/
```

Otevření pomocí členské funkce
Funkce *open* má stejné parametry jako konstruktor.
Deklarace ve třídě *ifstream*:
*void open(const char *name, int mode,int prot=filebuf::openprot);*

Deklarace ve třídě o*fstream*:
*void open(const char *name, int mode,int prot=filebuf::openprot);*
Deklarace ve třídě *fstream*:

*void open(const char *name, int mode,int prot=filebuf::openprot);*
Uzavření souboru se provede členskou funkci *close*, která nemá žádné parametry.
void close( );

```
/***************  příklad ***************/
#include <fstream.h>

void main(void)
{
  int hod=123;
  ofstream os;

  os.open("POKUS.DDD", ios::out); //otevření pro
zápis
  os << hod;
  os.close();

  hod=0;
  ifstream is;
  is.open("POKUS.DDD", ios::in);  //otevření pro
čtení
  is >> hod;
  cout << hod << endl;
  is.close();
}
/**************** konec *****************/
```

Formátovaný a neformátovaný zápis a čtení

Při formátovém zápisu do souboru se používá přetížený operátor << a pro čtení >>. Operátory se používají stejným způsobem jako pro standardní zařízení.

Pro neformátový zápis a čtení se používají funkce:
*ostream &write(const signed char *, int n);*
*ostream &write(const unsigned char *, int n);*
*istream &read(signed char *, int n);*
*istream &read(unsigned char *, int n);*

První parametr je adresa pole obsahující zapisována data, (pole, do kterého se uloží přečtená data). Druh parametr je počet zapisovaných (čtených) bytů.

Pro neformátový zápis jednoho znaku se používá funkce *put*.
ostream put(char);

Funkce *get* slouží pro neformátové čtení řetězce a také jednoho znaku.
istream& get(char, int len, char = '\n');*
istream& get(signed char, int len, char = '\n');*
istream& get(unsigned char, int len, char = '\n')*
istream& get(char&);

istream& get(signed char&);
istream& get(unsigned char&);

Přímý přístup k souborům

Pro zjištění pozice vstupu (čtení) je funkce *tellg* a pro zjištění pozice výstupu (zápisu) je funkce *tellp*.
long tellg( );
long tellp( );

Pro nastavení pozice pro vstup (čtení) slouží funkce *seekg* a pro nastavení pozice pro výstup (zápis) je funkce *seekp*.
ipstream& seekg(streampos pos);
ipstream& seekg(streamoff off, ios::seek_dir);
opstream& seekp(streampos pos);
opstream& seekp(streamoff off,ios::seek_dir);

První parametr udává pozici, druhý může nabývat hodnot, které jsou definované ve třídě *ios*:
beg - hodnota prvního parametru je vztažena k počátku souboru
cur - hodnota prvního parametru je vztažena vzhledem k aktuální pozici v souboru
end - hodnota prvního parametru je vztažena ke konci souboru

Opakovací test

1) Pomocí kterého klíčového slova definujeme šablonu?
- template
- templates
- typename
- class

2) Které z následujícíh pojmů neoznačují šablony?
- generické konstrukce
- parametrizované konstrukce
- metatřídy
- přetížené třídy

3) Které z následujících konverzí parametrů se provede při definování tříd nebo funkcí pomocí šablon?
- pro všechny jednoduché datové typy
- int na float
- typ s menší velikosti na typ s větší velikosti
- neprovedou se žádné konverze datových typů

4) Jak se jmenuje základní třída, od které se definují třídy pro práci se soubory?
- stream
- inout
- iostream
- ios

Shrnutí kapitoly

Programovací jazyk C++ zavádí nový přístup pro prácí se soubory. Hierarchie předem definovaných tříd poskytuje nástroje pro formátovaný i neformátovaný přístup. U formátovaného přístupu využíváme tříd *ofstream* pro výstup a *ifstream* pro vstup.
Pro zápisu do souboru se používá přetížený operátor << a pro čtení >>. Operátory se používají stejným způsobem jako pro standardní zařízení.

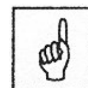
12. VÝJIMKY

Cílem lekce je seznámit se s mechanizmem tvorby výjimek, které slouží k ošetření nestabilních či chybných stavů v programu.

Po absolvování lekce by student měl být schopen:

- umět definovat třídy pro výjimky
- umět ošetřit pomoci výjimek nestabilní či chybné stavy v programu

Klíčová slova této kapitoly:
Výjimky, pokusný blok, zachycení výjimky



Čas potřebný k prostudování učiva kapitoly:
2 hodiny

Prostředky pro práci s výjimkami se objevují až v novějších překladačích jazyka C++ (Borland C++ 4.0, Visual C++ 2.0, Watcom C++ 10.5). Co je vlastně výjimka? Jedná se o situaci, kdy program nemůže pokračovat obvyklým způsobem, nastane vlastně běhová chyba. Ne vždy je možné program ukončit z důvodu běhové chyby. Některé aplikaci vyžadují, aby v případě chyby se přes ní program určitým způsobem přenesl a pokračoval v další části.

Tradiční ošetření chyb se provádí například:
- po zjištění chyby se program ukončí
- vrátí se hodnota, která reprezentuje druh chyby (číslo chyby)
- vrátí se správná hodnota a program zůstává v nepřípustném (chybovém) stavu
- vyvolá se funkce, která se má volat v případě chyby

Mechanizmus ošetření výjimečného stavu je vytvořen hlavně pro ošetření chyb. Jedná se o nelokální strukturu, která se rovněž chápat jako alternativní návratový mechanizmus. Použití výjimečného stavu proto nemusí vždy souviset jen s chybami.

Jazyk C++ umožňuje pracovat pouze s tzv. *synchronními výjimkami*, to znamená výjimkami, které vzniknou uvnitř programu. *Asynchronní výjimky* může vyvolat hardware.

Všechny operace, které jsou jistým způsobem nebezpečné a jejichž provádění by se nemuselo podařit, provádíme v *hlídaném bloku* (*guarded block*), který se skládá s *pokusného bloku* (*try block*) a z jednoho nebo několika *handlerů* (*exception handler*). V pokusném bloku se provádějí operace, které by mohly vyvolat výjimku. Pokud ta nenastane, provedou se všechny příkazy pokusného bloku a část s handlery se přeskočí. Pokud se však některá operace v pokusném bloku nepodaří, zakončí se provádění tohoto bloku a řízení programu převezme některý z handlerů. Nebude-li handlerem program ukončen, pokračuje za hlídaným blokem.

Pro práci s výjimkami slouží tato klíčová slova: **try**, **catch** a **throw**. Klíčové slovo **try** slouží jako prefix pokusného bloku,

handlery uvádí klíčové slovo **catch** a **throw** představuje operátor, který výjimku vyvolá.

Příklad:

Vytvoříme třídu seznam, která bude obsahovat proměnnou udávající počet prvků. Pro zjednodušení si vytvoříme pouze metodu, která zmenšuje počet prvků. Výjimka má nastat tehdy, pokusíme-li se odebírat z prázdného seznamu.

```
#include <iostream.h>
#include <stdlib.h>
const int N=12;
//schvalne je vetsi hodnota, aby se vyvolala vyjimka

class Vyjimka
{
    char *text;
  public:
    Vyjimka(char *t) :text(t){}
    char *DejText() const {return text;}
};
class Seznam
{
    int pocet;
  public:
    Seznam(int x) :pocet(x) {}
    int Odeber() throw (Vyjimka);
};
int Seznam::Odeber() throw (Vyjimka)
{
  if (pocet==0) throw Vyjimka("Seznam je prázdný");
  int p =pocet;
  pocet--;
  return p;
}
int main()
{
  Seznam s(10);

  try               //Pokusný blok
  {
   for (int i=0;i<N;i++)
     cout << s.Odeber();
  }
  catch(Vyjimka v)
  {
    cout << v.DejText();
    exit(1);
  }
  return 0;
}
```

Při předávání handlerů se uplatní je následující konverze:
- přetypování potomka na předka
- přetypování reference na potomka na referenci na předka
- přetypování ukazatele na potomka na ukazatel na předka
- přetypování ukazatele na `void *`

Při definování destruktoru je potřeba si mít na paměti, že by se z něj neměly šířit výjimky. Uvědomme si, že při vzniku výjimky se automaticky volají destruktory lokálních objektů. Šíření výjimky z destruktoru by pak způsobilo vznik nové výjimky ještě před zachycením a ošetřením jiné výjimky. Programovací jazyk C++ však nedovoluje zpracovávat dvě výjimky současně. Systém zavolá funkci `terminate( )` a tím se program ukončí.

Naproti destruktoru může konstruktor skončit výjimkou. Konstruktor však nesmí vytvářet globální objekt. V tomto případě by nešlo výjimku zachytit a program by opět skončil funkcí `terminate( )`. Při vyvolání výjimky lokálního objektu, je však potřeba nedokončený objekt správně uvolnit. Byl-li objekt dynamicky alokován, mohly by nám po něm zůstat v paměti „zbytky", které destruktor neuvolní. Destruktor se totiž nezavolá, pokud objekt nebyl při vytváření zcela dokončen. O uvolnění paměti by se pak měl postarat v handleru výjimky.

Seskupování výjimečných stavů

Výjimečné stavy se v rámci programu často seskupují ve skupiny. Například při práci v poli může nastat stav přetečení či podtečení hodnoty indexu prvku.

```
Enum PoleChyba {Preteceni, Podteceni};

try
{
//….
}
catch (PoleChyba n)
{
  switch (n)
  {
     case Preteceni: //…
     case Podteceni: //…
  }
  //…
}
```

Nároky výjimek

Výjimky přinášejí kromě ošetření nevhodných stavů rovněž zvýšené nároky na paměť a čas a to bez ohledu na to, zda k výjimce došlo či nikoliv. Důvodem je skutečnost, že
- překladač musí ukládat informace o plně zkonstruovaných automatických objektech, pro které je v případě výjimky potřeba destruktor. Po zrušení objektů je potřeba tyto informace zrušit.

- při vstupu do pokusného bloku si program musí uložit informace o handlerech a typech výjimek. Po opuštění bloku opět informace musí odstranit.

Při vyvolání výjimky se program výrazně zpomalí. To je způsobeno tím, že je potřeba uklidit zásobník (zavolat destruktory lokálních objektů) a je potřeba vyhledat handlery.

Příklad č.1

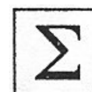
Napište funkci int Read(ifstream& is, queue &q), která otevře soubor, přečte z něj celá čísla a umístí je do fronty. Pokud nastane výjimka (čísla nejde uložit do fronty queue), uzavřete soubor. V případě, že se do fronty vejdou všechna čísla, soubor uzavřete a vraťte počet přečtených čísel. Frontu queue reprezentuje statickým polem celých čísel.

Příklad č. 2

Vyzkoušejte, co se stane, když nastane v předcházejícím příkladu výjimka a vy ji nezachytíte.

Příklad č. 3

Definujte třídu MyInt, která fungovat přesně jako datový typ int s tím rozdílem, že vyvolá výjimku při přetečení nebo podtečení.



Shrnutí kapitoly

Výjimky slouží k ošetření nestabilních stavů a chyb v průběhu programu. Programovací jazyk C++ pracuje pouze s tzv. *synchronními výjimkami*, to znamená výjimkami, které vzniknou uvnitř programu. *Asynchronní výjimky* může vyvolat hardware.

Všechny operace, které jsou jistým způsobem nebezpečné provádíme v *hlídaném bloku* (*guarded block*), který se skládá s *pokusného bloku* (*try block*) a z jednoho nebo několika *handlerů* (*exception handler*).

Pro práci s výjimkami slouží tato klíčová slova: **try**, **catch** a **throw**. Klíčové slovo **try** slouží jako prefix pokusného bloku, handlery uvádí klíčové slovo **catch** a **throw** představuje operátor, který výjimku vyvolá.

```
class Vyjimka
 //kód výjimky
};

class Trida
{
    //atributy třídy
  public:
    //metody třídy
    int NejakaMetoda() throw (Vyjimka);
    //metoda, která může vyvolat výjimku
};

int Trida::NejakMetoda() throw (Vyjimka)
{
  if (podminka) throw Vyjimka();
  //kód metody
}

int main()
{
  Trida o;
  try               //Pokusný blok
  {
    //kód
    o.NejakaMetoda();
  }
  catch (Vyjimka v) //zachycení
  {
    //kód
  }
}
```

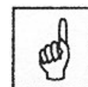
13. VÝVOJ PROGRAMŮ

Cílem lekce je seznámit se s mechanizmem tvorby aplikací, jednotlivými fázemi analýzy a vývoje.

Po absolvování lekce by student měl být schopen:

- vědět v jakých fázích se vyvíjí aplikace
- umět provádět objektovou analýzu a návrh aplikace
- umět lépe navrhovat a vytvářet své programy

Klíčová slova této kapitoly:



Implementace, konečný automat, ladění, objektový model aplikace, objektově orientovaná anylýza, objektově orientovaný návrh, testování

Čas potřebný k prostudování učiva kapitoly:
1 hodina

Vývoj programu se děje v několika základních fázích:

1. Studie
2. Analýza
3. Návrh
4. Implementace
5. Testování
6. Používání

Během všech kroků je potřeba pamatovat na tvorbu důkladné a dostatečně obsáhlé *dokumentace*.

Studie

Studie představuje analýzu požadavků (requirement analysis) na aplikaci. Jedná se o společnou činnost zákazníka a softwarové firmy, která aplikaci vytváří.

Objektově orientovaná analýza (OOA)

Mechanismus abstrakce a hierarchie. Je potřeba nalézt objekty a skupiny příbuzných objektů z reálného světa požadované aplikace. Dále je nutné popsat vazby mezi objekty. Jedná se o *implementačně nezávislou fázi* vývoje aplikace, která dává možnost přenositelnosti. Výsledkem této fáze je **objektový model aplikace**. Model objektu je zobecněný **konečný automat** (state machine), jehož *stav* je dán konkrétními hodnotami atributů. *Vstupy* – odpovídají metodám, které mohou změnit *stav*. *Výstupy* – jsou reprezentovanými hodnotami, které vracejí jednotlivé operace (přístupové metody).

Objekt = je modelem objektu z reálného světa aplikace

Objekty:
- abstraktní
- instanční

Objekty nesmí být ani příliš velké ani naopak malé. Objekty mohou mít následující povahu:
- informační – mají schopnost nést a spravovat informace
- řídící – mají schopnost spravovat a ovládat jiné objekty
- prezentační – mají schopnost prezentovat své vlastnosti a komunikovat s okolím

Podle převažujících rysů můžeme objekty rozdělit na:
- entitní – nést informace a metody pro jejich automatickou správu
- řídící – vedou dialog s okolím programu a reagují na události z okolí
- rozhraní – modelují vstupně-výstupní zařízení a typicky spolupracují s jedním řídícím objektem

Vazby mezi objekty:
Instanční vazba – (používá) – logická vazba (souvislost) mezi jednotlivými objekty, ke které se přidává *kardinalita* vazby. Vztah závislosti je jednosměrný.
Vazba celek-část – (kompozice)
Vazba obecný-speciální – (dědičnost)

Návrh

Nejprve je potřeba zvolit i v návaznosti na vytvořený objektový model aplikace implementační prostředí. Výsledkem této fáze je **objektově orientovaný návrh (Object-Oriented Design - OOD)**, který je upřesněním a doplněním objektového modelu. Tvůrci návrhu by se měli snažit co nejméně narušit objektový model vzniklý při OOA a zachovat tak přenositelnost na jiné systémy. Návrh si můžeme rozdělit na čtyři základní oblasti:
Problémová oblast (Problem Domain Component – PD) - model vzniklý jako důsledek analýzy.
Uživatelské rozhraní (Human Interaction Component – HI) – komunikace s uživatelem
Správa dat (Data Management Component – DM) – zajišťuje ukládání dat do vnější paměti, bezpečnost a konzistenci dat, obnovu dat apod.
Řízení spolupráce (System Interaction – SI) rozhraní na hardware, operační systém a počítačovou síť.

Je potřeba si stanovit:
1) Cíle návrhu
2) Kroky návrhu:
1) Nalezení tříd
2) Specifikovat operace
3) Specifikovat závislost tříd na jiných třídách

4) Specifikovat rozhraní tříd
5) Reorganizovat hierarchii tříd

Doporučení pro analýzu a návrh:

- provést smysluplné zobecnění – při začátku návrhu je možné si pomoci pomůckou, která říká, že podstatné jméno = objekt (třída) a sloveso = operace (metoda).
- minimalizovat vazby mezi třídami – třída je spojena s jinou v případě, že mezi nimi existuje vztah (dědičnosti, používání, kompozice). Osamocené třídy, které nejsou ve vztahu s žádnými jinými třídami, se mohou zrušit. Naopak třída, která je spojena s velkým množstvím tříd, přináší problémy.
- rozdělit třídy, které mají příliš mnoho odpovědností – mnoho operací
- spojit třídy s malými odpovědnostmi
- vyloučit počet jedna – vytvoření mnoha objektů musí být stejně snadné jako vytvoření jediného objektu. Měli bychom dobře zvážit všechna zda jsou smysluplná zahrnutí s počtem jedna.
- opakovanou funkčnost vyjádřit děděním – má-li více tříd stejnou odpovědnost (operaci), pak je vhodné navrhnout lépe dědičnost mezi třídami. To znamená najít základní obecnou třídu a ostatní z ní odvodit.
- úroveň abstrakce – na každé úrovni abstrakce by měla mít třída určitou odpovědnost

Implementace

Jedná se o realizaci programových textů, tedy o vlastní fázi objektově orientovaného programování v konkrétním vývojovém nástroji.

Testování

Testování se provádí až v okamžiku alespoň částečného dokončení projektů. Na rozdíl od ladění, které probíhá již ve fázi implementace a kterou provádí programátoři, provádí testování obvykle speciální tým.

- testování podle scénářů
- testování podle subsystémů

Používání

Pro každý softwarový projekt je důležitá zpětná vazba, kterou poskytují uživatelé aplikace. Získané poznatky jsou důležité pro úpravu nových verzí.

Kontrolní úkol:

Proveďte kompletní vývoj pro druhý úkol v samostatné práci č.3. Zaměřte se hlavně na fázi analýzy a návrhu. Zpracujte textovou a případně i grafickou dokumentaci k jednotlivým fázím vývoje programu.

Shrnutí kapitoly

Správně vytvořit program není jen psaní programového kódu v konkrétním vývojovém nástroji v určitém programovacím jazyku. Jedná se o komplexní činnost, kterou si můžeme rozdělit do několika fází:

1. Studie
2. Objektově orientovaná analýza
3. Objektově orientovaný návrh
4. Implementace
5. Testování
6. Používání

Během všech kroků je potřeba pamatovat na tvorbu důkladné a dostatečně obsáhlé *dokumentace*. Výsledkem analýzy je objektový model aplikace, který by měl být přenositelný a tedy nezávislý na konkrétní hardwarové a softwarové implementaci.
Objektově orientovaný návrh si můžeme rozdělit na čtyři základní části

- *problémová oblast*
- *uživatelské rozhraní*
- *správa dat*
- *řízení spolupráce*

Doporučení pro analýzu a návrh*:*

- provést smysluplné zobecnění
- minimalizovat vazby mezi třídami
- rozdělit třídy, které mají příliš mnoho odpovědností spojit třídy s malými odpovědnostmi
- vyloučit počet jedna
- opakovanou funkčnost vyjádřit děděním

14. PROGRAMOVACÍ JAZYK C#

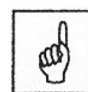
V této kapitole se dozvíte:

- Základní informace o programovacím jazyku C#
- **Tato kapitola je pouze doplňková**

Budete:

- znát základní charakteristiku programovacího jazyka C#
- vědět, jaké jsou rozdíly mezi jazyky C++, Java a C#

Klíčová slova této kapitoly:
C#, C++, Java, framework, .NET



Čas potřebný k prostudování učiva kapitoly:
2 hodiny

Programovací jazyk C# byl vyvinut firmou Microsoft pro platformu .NET. Jedná se o objektově orientovaný jazyk, který vychází z jazyků Java a C++.

Aplikace v C# jsou podobně jako v Javě vyvíjeny pod běhově řízeným prostředím. Jedná se o .NET framework (u Javy je to pak Java Virtual Machine). To sice přináší určité zpomalení běhu aplikace, na druhé straně však lepší přenositelnost.

Podrobnější informace o technologii .NET si můžete přečíst přímo na stránkách http://www.microsoft.com/cze/net/. Informace a soubory ke stažení .NET framework najdete na adrese http://msdn2.microsoft.com/en-us/netframework/default.aspx.

Zdrojový kód je nejprve převeden do mezikódu, který se nazývá MSIL (Microsoft Intermediate Language). Vznikne kód relativních adres, který je interpretován klíčovou součástí .NET frameworku s názvem CLR (Common Language Runtime) - společné běhové prostředí. To je standardizováno organizací ECMA. Podobně jako v jazyku Java, i zde pro práci s operační pamětí použit *Garbage Collector*. Ten se mimo jiné stará o uvolňování nepotřebných objektů z paměti. Díky Garbage Collectoru se již vývojáři nemusejí starat o přiřazování nebo uvolňování operační paměti. Programy pak neobsahují nekorektní práci v oblasti alokace a uvolňování paměti.

Další významným rysem jazyka je CLS (Common Language Specification - Společná jazyková specifikace) a s ní souvisící CTS (Common Type System - společný typový systém). Výsledkem použití CLS a CTS je rovnocennost programovacích jazyků. Díky tomu je možné při vývoji .NET aplikací použít libovolný z několika programovacích jazyků vyšší úrovně (C#, Visual Basic .NET, J#, C++). Dokonce je možné libovolný další jazyk se schopností kompilovat zdrojové kódy do mezikódu MSIL.

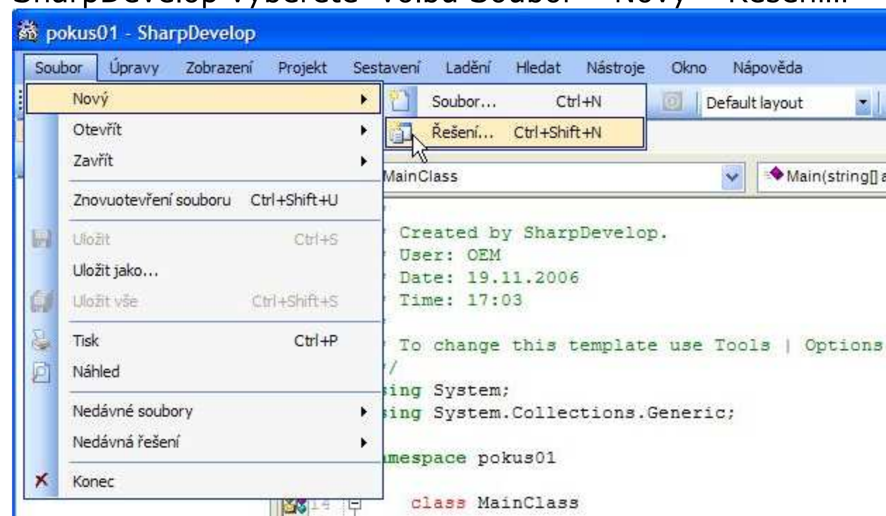
Platforma Microsoft .NET může mít pro široké uplatnění. Je možné tvořit klasické konzolové aplikace, programy využívající Microsoft Win32 API, webové aplikace ASP .NET (nahrazující zastaralé ASP 2.0), webové služby a knihovny tříd.

Základní výhody a nevýhody jazyka C# oproti jazyku Java:
+ obsahuje jednoduchý preprocesor
+ standardizace jazyka
+ rychlejší desktopové aplikace
+ otevřený jazyk

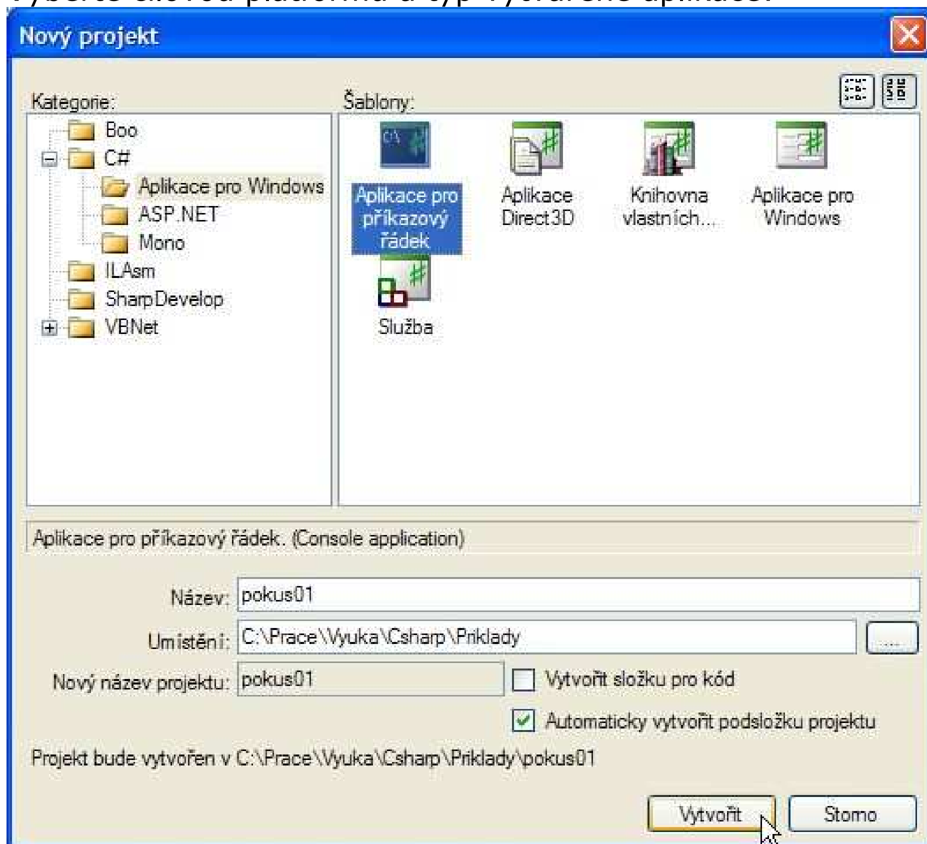
- horší přenositelnost, zatím hlavně jen pro MS Windows platformy (částečně pro Linux)
- méně rozšířen než Java
- v některých konstrukcích složitější

Pro vytváření programů v jazyce C# vhodné použít nějaké integrované vývojové prostředí. Můžete použít Microsoft Visual C# Express Edition, který je k dispozici na adrese: http://msdn.microsoft.com/vstudio/express/visualcsharp/download/ . Další možností je například nástroj SharpDevelop, které si můžete stáhnout na adrese: http://www.icsharpcode.net/OpenSource/SD/.

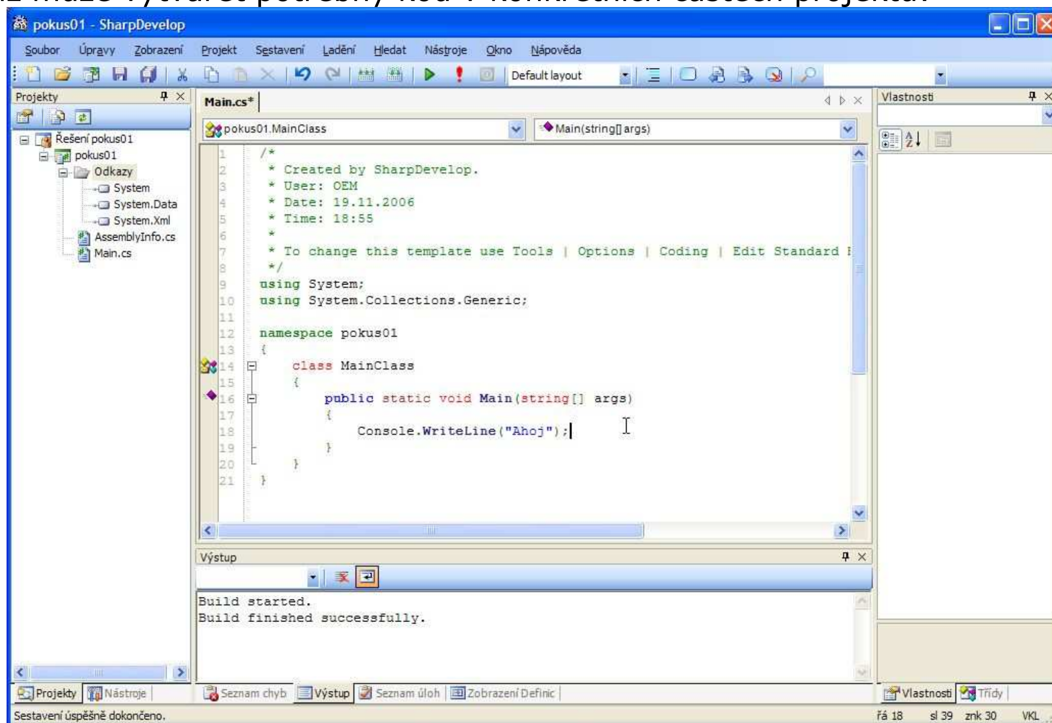
Při vytváření nového projektu (v české verzi nástroje SharpDevelop vyberete volbu Soubor – Nový – Řešení…



Vyberte cílovou platformu a typ vytvářené aplikace:



Dále již může vytvářet potřebný kód v konkrétních částech projektu:



Podívejte se na několik jednoduchých příkladů – ukázek řešení konkrétních problémů v jazyku Java a C#.

Příklad č.1

Jednoduchý výpis textu na obrazovku.

Jazyk Java

```
public class cv01_01
{

    public static void main(String[] args)
    {
       System.out.println("Ahoj");
    }
}
```

Jazyk C#:

```
using System;

class Hello
{
  static void Main()
  {
     Console.WriteLine("Hello World");
  }
}
```

Příklad č.2
Vytvoření jednoduché dědičnosti.
Jazyk Java:

```
package myprojects.cv08_01;
import java.awt.*;
import java.awt.event.*;
class Vozidlo
{
  //položky třídy. . .
}
class OsobniAuto extends Vozidlo
{
  //položky třídy. . .
}

class Cv08_01
{
  public static void main(String args[])
  {
   //práce s objekty
  }
}
```

Jazyk C#:

```
using System;
public class Vozidlo
{
  //položky třídy. . .
}
public class OsobniAuto : Vozidlo
{
  //položky třídy. . .
}

public class Pokus
{
  public static void Main()
  {
    //práce s objekty
  }
}
```

Příklad č.3

Vytvoření abstraktní třídy GrafickyObjekt, která se stane základem pro odvození tříd dalších grafických objektů, jako jsou Obdélník, Elipsa apod.

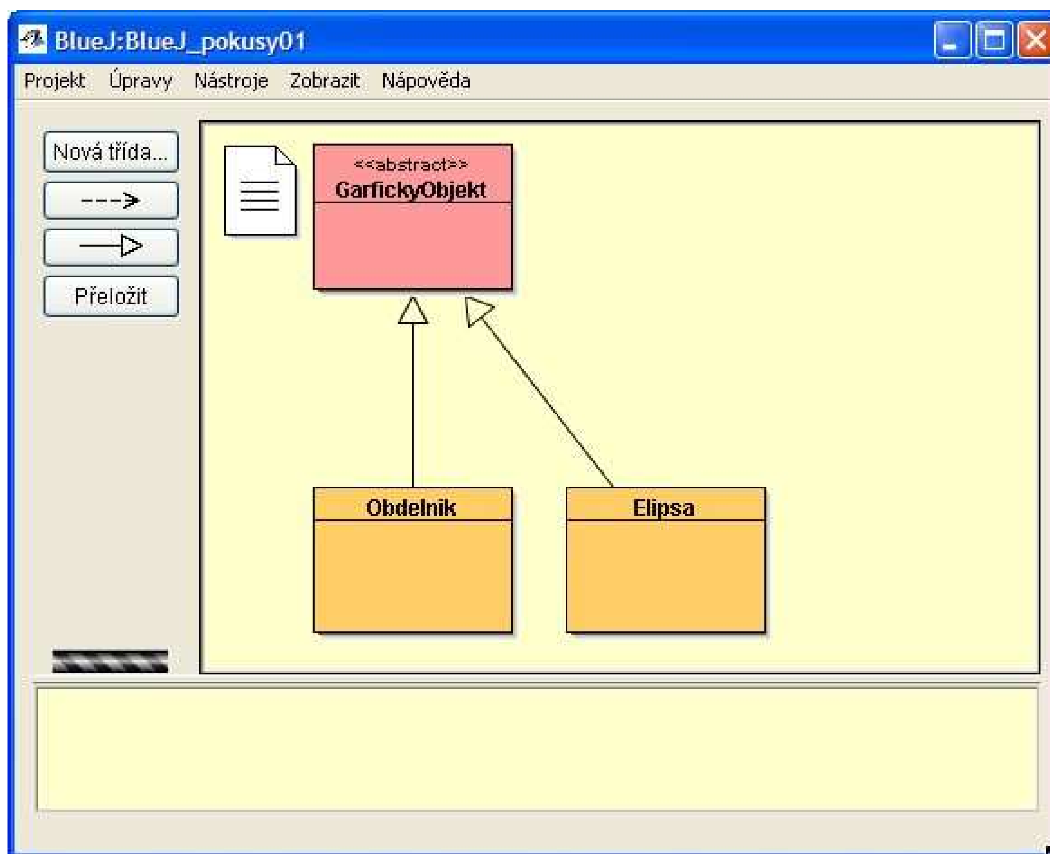
Jazyk Java:

```
package myprojects.cv08_02;
import java.awt.*;
import java.awt.event.*;
abstract class GrafickyObjekt
{
  public abstract void typObjektu();
```

```
  /* Abstraktní metoda, u obecného grafického objektu
   * nejsme schopni říci z jakých údajů budeme vypisovat
   * velikost objektu. */
}

class Obdelnik extends GrafickyObjekt
{
  public void typObjektu()
  {
    System.out.println("Obdelnik");
  }
}

class Cv08_02
{
  public static void main(String args[])
  {
     //práce s objekty
  }
}
```

Jazyk C#:

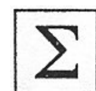
```
using System;
public abstract class Objekt
{
  public abstract void typObjektu();
}

public class Obdelnik : Objekt
{
  public override void typObjektu()
  {
     Console.WriteLine("Obdelnik");
  }
}

public class Pokus
{
   public static void Main()
   {
         //práce s objekty
   }
}
```



Průvodce studiem

Ke zdárnému zvládnutí učiva této a následující kapitoly je potřeba nainstalovat si .NET framework a vývojové prostředí a následně si prakticky napsat alespoň jednoduché příklady. Bez praktického psaní kódu se programovat v konkrétním jazyku nenaučíte.

Další podrobnosti o jazyku C# si vyhledejte odborné literatuře či na internetu. Tento učební text je pouze úvodem do problematiky programování v C# a nikoliv jeho podrobnou příručkou.



Shrnutí kapitoly.

Programovací jazyk C# byl vyvinut firmou Microsoft pro platformu .NET. Jedná se o objektově orientovaný jazyk, který vychází z jazyků Java a C++.

Základní výhody a nevýhody jazyka C# oproti jazyku Java:
+ obsahuje jednoduchý preprocesor
+ standardizace jazyka
+ rychlejší desktopové aplikace
+ otevřený jazyk
- horší přenositelnost, zatím hlavně jen pro MS Windows platformy (částečně pro Linux)
- méně rozšířen než Java

15. JEDNODUCHÉ ŘEŠENÉ PŘÍKLADY V JAZYKU C#

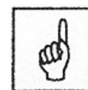
V této kapitole se dozvíte:

- Základní informace o programovacím jazyku C# a jeho konstrukcích
- **Tato kapitola je pouze doplňková**

Budete:

- vědět, jak vytvořit dědice rodičovských tříd
- umět volat konstruktor předka
- umět vytvářet překryté metody
- umět vytvářet abstraktní třídy
- umět pracovat s virtuálními metodami
- umět využívat rozhraní jako náhradu vícenásobné dědičnosti

Klíčová slova této kapitoly:
Abstraktní třída, konstruktor, rodič, vícenásobná dědičnost, virtuální metody



Čas potřebný k prostudování učiva kapitoly:
2 hodiny

Prostudujte si níže uvedené příklady a pokuste se je sami napsat a odladit.

Příklad č.1

```
/*
 * Rostislav Fojtík, 2005
 * Opakování - výpis na obrazovku
 */
using System;
class Hello
{
 static void Main()
 {
   int a=10, b=20;
   Console.WriteLine("Vypis na obrazovku.");
   Console.WriteLine("Soucet {0} a {1} je {2}",a, b,
a+b);
 }
}
```

Příklad č.2

```
/*
 * Rostislav Fojtík, Ostrava 2005
 * Třída a objekt
```

```
 */
using System;
class Object
{
  private int x;
  private int y;
  public Object(int x, int y)
  {
    this.x = x;
    this.y = y;
  }
  public int X
  {
   get
   {
     return x;
   }
   set
   {
     x = value;
   }
  }
  public int Y
  {
   get
   {
     return y;
   }
   set
   {
     y = value;
   }
  }
}

class Pokus
{
  public static void Main()
  {
    Object o1=new Object(12,11);
    Console.WriteLine("x= {0}, y= {1}",o1.X,o1.Y);
    o1.X = 111; Console.
    WriteLine("x= {0}, y= {1}",o1.X,o1.Y);
    Console.WriteLine("Stiskni klavesu");
    Console.Read();
   }
}
```

Příklad č.3

```
/*
 * Dědičnost
 */
using System;
public class ZakladniTrida
{
  private int x;
  protected int px;
  public ZakladniTrida(int a) //Konstruktor
  {
     x = a;
      px = 1;
     Console.WriteLine("Konstruktor ZakladniTrida");
           //jen pro ukázku
  }
 public int X
 {
   get
   {
     return x;
   }
   set
   {
     x = value; //lze ošetřit na správný rozsah
   }
 }
 public void zmenX(int x)
 {
   this.x = x;
 }
 public int dejX()
 {
   return x;
 }
}

public class OdvozenaTrida : ZakladniTrida
{
   private int y;
   public OdvozenaTrida(int a, int b) : base(a)
   {
     y = b;
     Console.WriteLine("Konstruktor OdvozenaTrida");
           //jen pro ukázku
   }
   public int Y
   {
     get
     {
       return y;
     }
   }
   public void metodaZmena(int a)
   {
   y = a;
```

```
    //base.x = a; //NELZE!
    base.X = a;
    // base.zmenX(a); nutno použít veřejný přístup!
    base.px = a;
  }
  public void vypisDat()
  {
    Console.WriteLine("Zakladni trida x = {0}",base.X);
    Console.WriteLine("Zakladni trida px={0}",base.px);
    Console.WriteLine("Odvozena trida y = {0}",y);
  }
}

public class Pokus
{
  public static void Main()
  {
    ZakladniTrida ox;
    ox = new ZakladniTrida(44);
    Console.WriteLine("x = {0}",ox.dejX());
    ox.zmenX(10);
    Console.WriteLine("x = {0}",ox.X);
    // Potomek OdvozenaTrida
    oy = new OdvozenaTrida(1,2);
    Console.WriteLine("x = {0}",oy.dejX());
    //Console.WriteLine("y = {0}",oy.Y);
    //Console.WriteLine("x = {0}",oy.X);
    //oy.zmenX(100);
    //Console.WriteLine("x = {0}",oy.X);
    //oy.metodaZmena(111);
    //Console.WriteLine("x = {0}",oy.X);
    //Console.WriteLine("x = {0}",oy.px);// NELZE!
    //oy.vypisDat();
    Console.WriteLine("Stiskni klavesu");
    Console.Read();
   }
}
```

Příklad č.4

```
/*
 * Rostislav Fojtík, Ostrava 2005
 * Dědičnost, Polymorfismus - virtuální metody
 */
using System;
class Rodic
{
  public virtual void vypis() //virtual
  {
    Console.WriteLine("Trida Rodic");
  }
}

class Potomek : Rodic
{
  public override void vypis() //override
   {
     Console.WriteLine("Trida Potomek");
   }
}

public class Pokus
{
   public static void Main()
   {
     Rodic or;
     or=new Rodic();
     or.vypis();
     Potomek op= new Potomek();
     op.vypis();
     Rodic ox;
     ox = new Potomek();
     ox.vypis();
     Console.Read();
   }
}
```

Příklad č.5

```
/*
 * Rostislav Fojtík, Ostrava 2005
 * Rozhraní - náhrada vícenásobné dědičnosti
 */
using System;
public interface TypObjektu
{
   void jakyObjekt();
}

public interface UdajeObjektu
{
   void vypisUdaje();
}

class Obdelnik : TypObjektu, UdajeObjektu
{
   private int stranaA;
   private int stranaB;
   public Obdelnik(int a, int b)
   {
     this.stranaA = a;
     this.stranaB = b;
   }
   public void jakyObjekt()
   {
     Console.WriteLine("Obdelnik");
   }
   public void vypisUdaje()
   {
     Console.WriteLine("Strana a = {0}, strana b = {1}",
                              stranaA, stranaB);
   }
}
class Kruh : TypObjektu, UdajeObjektu
{
  private int polomer;
  public Kruh(int r)
  {
    this.polomer = r;
  }
  public void jakyObjekt()
  {
    Console.WriteLine("Kruh");
  }
  public void vypisUdaje()
  {
    Console.WriteLine("Polomer = {0}", polomer);
  }
}

public class Pokus
{
  public static void Main()
  {
```

```
    Obdelnik ob = new Obdelnik(10,20);
    ob.jakyObjekt();
    ob.vypisUdaje();

    Kruh kr = new Kruh(22);
    kr.jakyObjekt();
    kr.vypisUdaje();

    Console.WriteLine("Stiskni klavesu");
    Console.Read();
  }
}
```

Příklad č.6

```
/*
 * Rostislav Fojtík, Ostrava 2005
 * Abstraktní třída
 */
using System;
public abstract class Objekt
{
  public abstract void typObjektu();
}

Public class Obdelnik : Objekt
{
  public override void typObjektu()
  {
     Console.WriteLine("Obdelnik");
  }
}

public class Pokus
{
  public static void Main()
  {
   //Objekt oo = new Objekt();
   Obdelnik ob = new Obdelnik();
   ob.typObjektu();
  }
}
```

Příklad č.7

Napište ukázkový příklad, ve kterém vytvoříte třídu Datum. Třída bude umět korektně zvětšit datum o jeden den, zjistit přestupný rok, vypsat datum, vypočíst počet sekund, které uplynuly mezi aktuálním datem a 1.1.1970. Vytvořte parametrický konstruktor pro inicializaci data.

```
/*
 * Created by SharpDevelop.
 * User: Mgr. Rostislav Fojtík, PhD.
 * Date: 10.11.2006
 * Time: 22:40
 *
 * Ukázkový program - vytvoøení tøídy Datum
 */
using System;
using System.Collections.Generic;

namespace datum01
{

   class Datum
   {
       private int den, mesic, rok;

       private bool prestupnyRok()
       {
            //metoda vrátí true, je-li nastavený rok
pøestupný
           if (rok%4==0 || rok%400==0 && rok%1000!=0)
return true;
           else return false;
       }

       public Datum(int d, int m, int r)
       {
             //parametrický konstruktor
           den = d;
           mesic = m;
           rok = r;
       }

       public void pridejDen()
       {
     //Metoda korektnì zvìtší datum nastavené v objektu o
jeden den
          if (mesic==2)
            if (den<28) den++;
            else
            {
              if (den == 28)
                 if (prestupnyRok()) den=29;
                 else
                 {
                   den = 1;
```

```
                 mesic = 3;
               }
           else
                 if (den==29)
                 {
                    den=1;
                    mesic=3;
                  }
          }
          else
           if (mesic==4 || mesic==6 || mesic==9 ||
mesic==11)
             if (den<30) den++;
               else
               {
                  den=1;
                  mesic++;
               }
            else
              if(mesic==1 || mesic==3 || mesic==5 ||
mesic==7 || mesic==8 || mesic==10 || mesic==12)
                 if (den<31) den++;
                 else
                 {
                    den=1;
                      if (mesic<12) mesic++;
                      else
                      {
                          mesic=1;
                          rok++;
                       }
                  }
       }

       public void vypisDatum()
       {
             //metoda vypíše nastavené datum ve formátu
den.mìsíc.rok
          Console.WriteLine(den+"."+mesic+"."+rok);
        }

       public void zvetsitDatum(int d)
       {
        //Metoda zvìtší datum o hodnotu parametru d
          while (d>0)
          {
             pridejDen();
             d--;
          }
       }
       public long prevedCas()
         {
          /* metoda vrátí poèet sekund, které jsou mezi
datem
           * 1.1.1970 a datem, který má objekt nastavený
           */
```

```
          long vysl=0;
         int m=mesic,r=rok;

         vysl=(den-1)*24*3600;

         while(m>1)
          {
          if(m==1 || m==3 || m==5 || m==7 ||
m==8||m==10||m==12)
               vysl=vysl+(31*24*3600);
          else
           if (m==4 || m==6 || m==9 || m==11)
               vysl=vysl+(30*24*3600);
           else
               if (r%4==0 || r%400==0 && r%1000!=0)
                    vysl=vysl+(29*24*3600);
               else
                    vysl=vysl+(28*24*3600);
               m--;
          }

         while(r>1970)
          {
          if (prestupnyRok())
               vysl=vysl+(366*24*3600);
          else
               vysl=vysl+(365*24*3600);
          r--;
          }

         return(vysl);
        }
   }

      class MainClass
      {
           public static void Main(string[] args)
           {
                Datum den1 = new Datum(20,11,2006);
                long sek = 0;

                sek = den1.prevedCas();
                Console.WriteLine(sek);

                den1.vypisDatum();
                den1.pridejDen();
                den1.vypisDatum();

                den1.zvetsitDatum(13);
                den1.vypisDatum();
                Console.Read();
           }
      }
}
```

Multithreading

Velká část současných a dříve vytvořených aplikací multithreading nevyužívá. V rámci programu je kompilátor optimalizován sled instrukcí v rámci jednoho threadu – vlákna. Bylo to výhodné z toho důvodu, že většina procesoru obsahovala jen jedno výpočetní jádro. Vícejádrové procesory nebo systémy s více procesory pak u takových aplikací nepřináší požadované zvýšení výkonu. Aplikace je optimalizována jen pro jedno jádro a další jsou nevyužitá.

Programy, které obsahují více vláken, které jsou na sobě nezávislá, naopak vícejádrové procesory využijí dokonale ke zvýšení výkonu systému.

Práce s thready samozřejmě vyžaduje určitou část výkonu. Je potřeba zajistit vznik, zrušení jednotlivých vláken, případně jejich vzájemnou spolupráci. Příkladem využití více vláken v rámci aplikace může být například rendering grafiky, náročné výpočty v rámci luštění šifer, filtry v grafických editorech, multimediální aplikace, video a zvuk a podobně.

Jako jednoduchý příklad použití více vláken si můžeme ukázat výukovou aplikaci, která porovnává a graficky znázorňuje třídící algoritmy. Pro každý z vybraných algoritmů je přiřazeno jedno vlákno, v rámci kterého je prováděno třídění pole úseček o různých délkách (viz. obrázek).

Následující příklad napsaný v programovacím jazyku C# spustí různými způsoby dvě nezávislá vlákna, jejichž činnost není nijak synchronizována.

```
using System;
using System.Collections.Generic;
using System.Threading;

namespace VlaknaPriklad
{
   class MainClass
   {
```

```
      static void nejakaMetoda1()
      {
       //pomocná metoda, která bude vypisovat čísla od 1
do 10
            for (int i = 0; i <= 10; i++)
                  Console.Write(i + ", ");
      }
       static void nejakaMetoda2()
      {
       //pomocná metoda, která bude vypisovat znaky od 'A'
do 'Z'
            for (char i = 'A'; i <= 'Z'; i++)
                  Console.Write(i + ", ");
      }

      public static void Main(string[] args)
      {
       //první vlákno za pomoci delegáta asociuje metodu,
       //která bude vláknem spuštěna
            ThreadStart vlakno01 = new
ThreadStart(nejakaMetoda1);
      //vytvoření nového vlákna
            Thread vlakno02 = new Thread(vlakno01);
      //spuštění dalšího vlákna
            vlakno02.Start();
      //zavolání metody z aktuálního vlákna
            nejakaMetoda2();

            Console.Read();
      }
   }
}
```

16. CVIČENÍ Č.1

Cíle lekce
Po absolvování lekce budete:
- umět vytvořit jednoduchou třídu a její instanci
- vědět, jak definovat metody třídy
- vědět, jak volat metody třídy

Časová náročnost lekce: **1 hodina**

Příklad č.1

```
/*
 * Mgr. Rostislav Fojtík, 2003
 *
 * Napište si následující příklad, ve kterém tvoříme ¨
 * pokusnou třídu. Ta bude obsahovat dvě položky
 * (int a float) a metody pro práci s nimi.
 */
#include <iostream.h>
#include <stdlib.h>

class PokusnaTrida
{
  private: //soukromé položky
   int i;
   float f;
  public: //veřejné položky
   PokusnaTrida(int,float); //Konstruktor
   void ZmenI(int); //metoda pro nastavení hodnoty 'i'
   void ZmenF(float); //metoda pro nastavení hodnoty 'f'
   int DejI() const; //metoda vrátí hodnotu 'i'
   float DejF() const; //metoda vrátí hodnotu 'f'
}; //Nezapomeňte na středník!

//Následují definice jednotlivých metod
PokusnaTrida::PokusnaTrida(int ih,float fh)
{
  i=ih;
  f=fh;
}

void PokusnaTrida::ZmenI(int ih)
{
  i=ih;
}

void PokusnaTrida::ZmenF(float fh)
{
  f=fh;
}

int PokusnaTrida::DejI() const
{
```

```
  return i;
}

float PokusnaTrida::DejF() const
{
  return f;
}

//Hlavní funkce
int main(int argc, char *argv[])
{
  PokusnaTrida o1(1,1.1);

  //Výpis položek 'i' a 'f'
  cout << "i= "<<o1.DejI()<<", f= "<<o1.DejF()<< endl;

  //Změna položek
  o1.ZmenI(100);
  o1.ZmenF(2.333);

  //Výpis položek 'i' a 'f'
  cout << "i= "<<o1.DejI()<<", f= "<<o1.DejF()<< endl;

  return 0;
}
```

Pokud se nad uvedeným zdrojovým kódem zamyslíte, možná zjistíte, že je zbytečně rozsáhlý a že pomocí neobjektového přístupu by kód byl mnohem kratší. Ale již v příkladu č.2, kde se snažíme využít určité kontrolní mechanizmy zjistíme výhody objektového přístupu.

Podívejme se na příklad č.1 řešený bez definice třídy.

```
#include <iostream.h>

define struct
{
  int i;
  float f;
}T1;

int main(int argc, char *argv[])
{
  T1 p1;

  p1.i=1;
  p2.f=1.1;

  //Výpis položek 'i' a 'f'
  cout << "i= "<<p1.i<<", f= "<<p1.f<< endl;

  //Změna položek
  p1.i=100;
  p1.f=2.333;

  //Výpis položek 'i' a 'f'
  cout << "i= "<<p1.i<<", f= "<<p1.f<< endl;
```

```
  return 0;
}
```

Příklad č.2

```
/*
 * Mgr. Rostislav Fojtík, 2003
 *
 * Upravte minulý příklad tak, aby položka 'i' mohla
nabývat
 * hodnoty jen v uzavřeném intervalu <-100,100> a položka
 * 'f' je hodnoty menší nebo rovno 1000. Pokud budou
hodnoty
 * mimo požadovaný rozsah, nastaví se nejbližší možná
 * hodnota.
 */
#include <iostream.h>
#include <stdlib.h>

class PokusnaTrida
{
  private:
   int i;
   float f;
   int SpravnaI(int); //kontrolní metoda
   float SpravnaF(float); //kontrolní metoda
  public:
   PokusnaTrida(int,float); //Konstruktor
   void ZmenI(int);
   void ZmenF(float);
   int DejI() const;
   float DejF() const;
};

int PokusnaTrida::SpravnaI(int ih)
{
  if (ih<-100) ih=-100;
  if (ih>100)ih=100;
  return ih;
}

float PokusnaTrida::SpravnaF(float fh)
{
  if (fh>1000)fh=1000;
  return fh;
}

PokusnaTrida::PokusnaTrida(int ih,float fh)
{
  i=SpravnaI(ih);
  f=SpravnaF(fh);
}

void PokusnaTrida::ZmenI(int ih)
{
  i=SpravnaI(ih);
```

```
}

void PokusnaTrida::ZmenF(float fh)
{
  f=SpravnaF(fh);
}

int PokusnaTrida::DejI() const
{
  return i;
}

float PokusnaTrida::DejF() const
{
  return f;
}

int main(int argc, char *argv[])
{
  PokusnaTrida o1(1,1.1);

  //Výpis položek 'i' a 'f'
  cout << "i= "<<o1.DejI()<<", f= "<<o1.DejF()<<endl;

  //Změna položek
  o1.ZmenI(1000); //špatná hodnota
  o1.ZmenF(2000.333); //špatná hodnota

  //Výpis opravených položek 'i' a 'f'
  cout << "i= "<<o1.DejI()<<", f= "<<o1.DejF()<<endl;

  return 0;
}
```

Všimněte si, že hlavní program nebylo potřeba nijak zásadně měnit. Ve funkci main nebylo potřeba volat žádné kontrolní mechanizmy, neboť ty jsou automaticky vyvolány již v metodách třídy!

Shrnutí

- pro definování třídy se používá klíčové slovo **class**
- členská data definuje v rámci třídy převážně jako privátní
- metody, pomocí který zjišťujeme nebo měníme stav objektu jsou definovány obvykle jako položky veřejné
- definice metody musí obsahovat odkaz na třídu, ke které patří
  `void `***`PokusnaTrida::`***`ZmenI(int ih)`
- pomocné metody v rámci třídy definujeme obvykle jako soukromé

17. CVIČENÍ Č.2

Cíle lekce

Po absolvování lekce budete:
- umět některé nové vlastnosti jazyka C++
- zdokonalíte se v psaní kódu v jazyku C++
- umět tvořit přetížené funkce
- umět pracovat s referencemi v parametrech funkcí

Časová náročnost lekce: **2 hodiny**

Příklad č.1

```
/*
 * Mgr. Rostislav Fojtík, 2003
 *
 * Funkční parametry
 */
#include <iostream.h>

float Exp(float x=10.0, int n=2);
int Abs(int);

int main(int argc, char *argv[])
{

  cout << Exp()<<endl;
  cout << Exp(2.0)<<endl;
  cout << Exp(2.0,3)<<endl;

  return 0;
}

float Exp(float x, int n)
{
  float vys=1;
  for (int i=1;i<=Abs(n);i++)
    vys *=x;
  if (n<0) vys = 1/vys;

    return vys;
}

int Abs(int x)
{
  return (x<0)?x*(-1):x;
}
```

Příklad č.2

```
/*
 * Mgr. Rostislav Fojtík, 2003
 *
 * Vytvořte přetíženou funkci pro absolutní hodnotu
 */
#include <iostream.h>
#include <stdlib.h>
#include <math.h>

struct Complexni
{
  float re, im;
};

int Abs(int);
long Abs(long);
float Abs(float);
double Abs(double);
float Abs(Complexni);

int main(int argc, char *argv[])
{
  int i=-10;
  long l=-10000000;
  float f=-1.1;
  double d=-1000000000000.1;
  Complexni c={1.1,-2.3};

  cout << Abs(i) << endl;
  /* V případě, že by nebyla definována funkce Abs(int),
nastal
     by problém s přetypováním. Překladač by se neuměl
rozhodnout
     zda použít funkci Abs(long) nebo Abs(float). Bylo by
nutné
     přetypovat explicitně! */
  cout << Abs(l) << endl;
  cout << Abs(f) << endl;
  cout << Abs(d) << endl;
  cout << Abs(c) << endl;

  system("PAUSE");
  return 0;
}

int Abs(int x)
{
  cout <<"int "; // jen pro kontrolu
  return (x<0)?x*(-1):x;
}

long Abs(long x)
{
  cout <<"long "; // jen pro kontrolu
  return (x<0)?x*(-1):x;
```

```
}

float Abs(float x)
{
  cout <<"float "; // jen pro kontrolu
  return (x<0)?x*(-1):x;
}

double Abs(double x)
{
  cout <<"double "; // jen pro kontrolu
  return (x<0)?x*(-1):x;
}

float Abs(Complexni x)
{
  return sqrt(x.re*x.re + x.im*x.im);
}
```

Příklad č.3

```
/*
 * Mgr. Rostislav Fojtík, 2003
 *
 * reference
 *
 */

#include <iostream.h>

void swap(int &, int &);

int main(int argc, char *argv[])
{
  int x=10;
  int &rx=x;

  cout <<"x = "<<x<<endl;
  cout <<"rx = "<<rx<<endl;

  int X=10,Y=20;
  cout<<"X= "<<X<<endl;
  cout<<"Y= "<<Y<<endl;
  swap(X,Y);
  cout<<"X= "<<X<<endl;
  cout<<"Y= "<<Y<<endl;

  return 0;
}

void swap(int &a, int &b)
{
  int pom=a;
  a=b;
  b=pom;
}
```

Příklad č.4

```
/*
 * Mgr. Rostislav Fojtík, 2003
 *
 * alokace paměti pomocí new a delete
 *
 */
#include <iostream.h>

const int Pocet=10;

int main(int argc, char *argv[])
{
  int *p;

  p=new int[Pocet];//alokace paměti

  for (int i=0;i<Pocet;i++)
  {
    p[i]=i;
    cout<<p[i]<<endl;
  }

  delete []p; //dealokace paměti

  return 0;
}
```

Shrnutí

- Proměnné v jazyku C++ je možné definovat kdekoliv, nejen na začátku bloku. Pozor však na nepřehlednost programu!
- Potřebujeme-li vytvořit přetížené funkce, pak se musí jednoznačně lišit počtem nebo typem parametrů.
- Reference lze využít při vytváření parametrů funkcí, které jsou volány odkazem.
- Pro alokaci a dealokaci dynamické paměti raději používejte operátory **new** a **delete**.

18. CVIČENÍ Č.3

Cíle lekce
Po absolvování lekce budete:
- umět definovat třídy a objekty
- umět definovat a používat konstruktory a destruktory
- umět tvořit přístupové a změnové metody

Časová náročnost lekce: 2 hodiny

Příklad č.1

```
/*
 * Mgr. Rostislav Fojtík, 2003
 *
 * Příklad  - vytvoření třídy Datum
 * Ošetření roku stanovíme v rozmezí 1980 až 2099. Chybné
 * hodnoty dne, měsíce i roku nahradíme nejbližšími
možnými
 * hodnotami.
 */
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

// zacatek deklarace třidy Datum
class Datum
{
  private:
    int den, mesic, rok;
// Soukromé metody pro kontrolu správných údajů, je-li
hodnota
// nevyhovující, vrátí metoda nejbližší správnou hodnotu.
    int SpravnyDen(int d);
    int SpravnyMesic(int m);
    int SpravnyRok(int r);
    int PrevedNazev(char *);
  public:
    Datum(int d, int m, int r); //konstruktor
    Datum(int d, char *m, int r); //konstruktor
    void VypisDatum() const;    //přístupová metoda
    int DejDen() const;    //přístupová metoda
    int DejMesic() const;  //přístupová metoda
    int DejRok() const;          //přístupová metoda
    void ZmenDatum(int d, int m, int r); //změňová metoda
    void ZmenDen(int d);  //změnová metoda
    void ZmenMesic(int m);  //změnová metoda
    void ZmenMesic(char *);  //změnová metoda
    void ZmenRok(int r);  //změnová metoda
};
// konec deklarace třidy Datum

// následují definice metod
int Datum::SpravnyDen(int d)
```

```
{
  if (d>=1 && d<=28) return d;
   else
    if (d<1) return 1;
     else
     {
if (mesic==1 || mesic==3 || mesic==5 || mesic==7 ||
mesic==8 || mesic==10 || mesic==12)
      if (d>31) return 31;
        if (mesic==4 || mesic==6 || mesic==9 || mesic==11)
      if (d>30) return 30;
        if (mesic==2)
      if ((rok-1980)%4 == 0)
        if (d>29) return 29;
         else return d;
       else
        if (d>28) return 28;
      }
}

int Datum::SpravnyMesic(int m)
{
  if (m<1) return 1;
   else
    if (m>12) return 12;
      else return m;
}

int Datum::SpravnyRok(int r)
{
  // Chceme použít rok pouze v rozmezí 1980 - 2099.
  if (r<1980) return 1980;
   else
    if (r>2099) return 2099;
      else return r;
}

int Datum::PrevedNazev(char*m)
{
  if (strcmp(m,"leden")==0) return 1;
  else
   if (strcmp(m,"unor")==0) return 2;
   else
    if (strcmp(m,"brezen")==0) return 3;
    else
     if (strcmp(m,"duben")==0) return 4;
     else
      if (strcmp(m,"kveten")==0) return 5;
      else
       if (strcmp(m,"cerven")==0) return 6;
       else
      if (strcmp(m,"cervenec")==0) return 7;
      else
       if (strcmp(m,"srpen")==0) return 8;
       else
        if (strcmp(m,"zari")==0) return 9;
```

```
        else
         if (strcmp(m,"rijen")==0) return 10;
         else
          if (strcmp(m,"listopad")==0) return 11;
          else
           if (strcmp(m,"prosinec")==0) return 12;
           else return 1;   //hodnota v případě chybného
řetězce
}
Datum::Datum(int d, int m, int r)
{
  rok=SpravnyRok(r);
  mesic=SpravnyMesic(m);
  den=SpravnyDen(d);
}

Datum::Datum(int d, char *m, int r)
{
  rok=SpravnyRok(r);
  mesic=PrevedNazev(m);
  den=SpravnyDen(d);
}

void Datum::VypisDatum() const
{
  cout << den << '.' << mesic << '.' << rok << endl;
}
int Datum::DejDen() const
{
  return den;
}
int Datum::DejMesic() const
{
  return mesic;
}

int Datum::DejRok() const
{
  return rok;
}
void Datum::ZmenDatum(int d, int m, int r)
{
  rok=SpravnyRok(r);
  mesic=SpravnyMesic(m);
  den=SpravnyDen(d);
}
void Datum::ZmenDen(int d)
{
  den=SpravnyDen(d);
}
void Datum::ZmenMesic(int m)
{
  mesic=SpravnyMesic(m);
}
void Datum::ZmenMesic(char *m)
{
```

```
  int pom=PrevedNazev(m);
  mesic=SpravnyMesic(pom);
}
void Datum::ZmenRok(int r)
{
  rok=SpravnyRok(r);
}
// konec definice metod

int main(int argc, char *argv[])
{

   Datum d1(1,1,1),d2(3,3,2003); //objekt vytvořen
konstruktorem se třemi parametry typu int
   Datum d3(13,"cervenec",1998);
        //objekt vytvořen konstruktorem se dvěma
parametry typu int a jedním char*

   d1.VypisDatum();
   d2.VypisDatum();
   d3.VypisDatum();

   d1.ZmenDatum(29,2,1998);
   d2.ZmenDatum(29,2,1980);
   d3.ZmenDatum(-1,-1,1980);

   cout << endl;
   d1.VypisDatum();
   d2.VypisDatum();
   d3.VypisDatum();

   system("PAUSE");
   return 0;
}
```

Příklad č.2

```
/*
 * Mgr. Rostislav Fojtík, 2003
 *
 * Vytvořte třídu RodneCislo, která bude obsahovat datový
člen rc,
 * který bude typu řetězec. Dále třída bude obsahovat
metody,
 * které z rodného čísla (rc) zjistí den, měsíc, rok
narození
 * a pohlaví. Vytvořte konstruktor, který inicializuje
hodnotu rc
 * ze svého parametru.
 */

#include <iostream.h>
#include <string.h>

class RodneCislo
{
  private:
    char rc[11];//bez lomítka
    int den,mes,rok; //datum narození
    //zde bychom mohli využít již existující třídy Datum
    bool SpravnaHodnota(char*);
  public:
    RodneCislo(char*);
    bool JeMuz() const;
    int DejDenNarozeni() const;
    int DejMesicNarozeni() const;
    int DejRokNarozeni() const;
    void VypisRodnehoCisla() const;
};

bool RodneCislo::SpravnaHodnota(char *r)
{
  int pom=0;
  int p1,p2,p3,p4,p5;
  /* Kontrola - rodné číslo se rozdělí po dvojici cifer,
z nich
     vznikne pět dvouciferných čísel, která se sečtou.
Suma by
     měla být dělitelná bezezbytku 11 */
  p1= ((r[0]-'0')*10) + (r[1]-'0');
  p2= ((r[2]-'0')*10) + (r[3]-'0');
  p3= ((r[4]-'0')*10) + (r[5]-'0');
  p4= ((r[6]-'0')*10) + (r[7]-'0');
  p5= ((r[8]-'0')*10) + (r[9]-'0');
  pom= p1+p2+p3+p4+p5;
  if (pom%11) return false;
    else return true;
}
RodneCislo::RodneCislo(char *r)
{
  if (SpravnaHodnota(r)) strcpy(rc,r);
   else strcpy(rc,"0000000000");
```

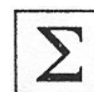
```
   /* Toto ošetření je pouze modelové a v případě
seriozního
      programu bychom museli zvolit jiné, dokonalejší! */
}

bool RodneCislo::JeMuz() const
{ //true=muž, false=žena
  char str;
  str=rc[2];
  if (str>='5') return false;
    else return true;
}
int RodneCislo::DejDenNarozeni() const
{
  char str[3];
  str[0]=rc[4];
  str[1]=rc[5];
  str[2]='\0';
  return atoi(str);
}
int RodneCislo::DejMesicNarozeni() const
{
  char str[3];
  int m;
  str[0]=rc[2];
  str[1]=rc[3];
  str[2]='\0';
  m = atoi(str);
  if (JeMuz()) return m;
   else return (m-50);
}
int RodneCislo::DejRokNarozeni() const
{
  char str[3];
  str[0]=rc[0];
  str[1]=rc[1];
  str[2]='\0';
  return atoi(str);
  /* Vrátí jen dvojčísli roku, pro návrat celého roku
     by bylo potřeba přičíst letopočet, např. 1900 */
}
void RodneCislo::VypisRodnehoCisla() const
{
  if (strcmp(rc,"0000000000")==0) cout<<"Chybne rodne
cislo!"<<endl;
   else cout << rc <<endl;
}


int main(int argc, char *argv[])
{
  RodneCislo Eva("8952045993");
  RodneCislo Pavla("8752045994");
  cout<<"Pohlavi muz "<<Eva.JeMuz()<<endl;
  cout<<"Datum narozeni: "
```

```
        <<Eva.DejDenNarozeni()<<"."<<Eva.DejMesicNaroze
    ni()
        <<"."<<Eva.DejRokNarozeni()<<endl;
  Eva.VypisRodnehoCisla();
  Pavla.VypisRodnehoCisla();

  return 0;
}
```

Příklad č.3

Vytvořte třídu Zásobník, pro celá čísla. Jednotlivé prvky budou ukládány ve statickém poli, které bude mít maximální velikost 100 prvků. Uvažte, jaké metody budeme potřebovat.

Shrnutí



- Třídy definujeme pomocí klíčových slov **class**, **struct** a **union**.
- Členská data zavádíme většinou jako privátní.
- Konstruktory jsou buď implicitní nebo explicitní. Explicitní konstruktory mohou být přetížené. Jména konstruktorů se shodují se jménem třídy.
- Destruktor musí být ve třídě právě jeden a to buď implicitní nebo explicitní. Destruktory nemají parametry. Konstruktory i destruktory patří mezi veřejné metody.
- Metody třídy si můžeme rozdělit na přístupové (zjišťují stav objektu) nebo změnové (mění stav objektu).

19. CVIČENÍ Č.4

Po absolvování lekce budete:

- umět definovat třídy a objekty
- umět definovat a používat členská a data a metody typu *static*
- umět tvořit *friend* metody a třídy

Časová náročnost lekce: **2 hodiny**

Příklad č.1

```
/*
 * Rostislav Fojtík, 2003
 *
 * Vytvořte třídu Fronta pro celá čísla. Prvky fronty
 * budou ukládány do statického pole o maximální
velikosti
 * 100 prvků.
 *
 */
#include <iostream.h>

const int Max=100;

class Fronta
{
  private:
    int pole[Max];
    int prvni,posledni;
  public:
    Fronta();
    bool PridejNaKonec(int);
    int OdeberZCela();
    bool JePrazdna() const;
};

Fronta::Fronta()
{
  prvni=0;
  posledni=-1;
}

bool Fronta::PridejNaKonec(int hod)
{
  if (posledni<Max)
  {
    posledni++;
    pole[posledni]=hod;
    return true; //povedlo se zařadit prvek
  } else return false; //nelze zařadit prvek, pole je
obsazeno
}

int Fronta::OdeberZCela()
{
  if (posledni>-1)
```

```
  {
    int pom=pole[prvni];
    for (int i=0;i<posledni;i++)
      pole[i]=pole[i+1];
    posledni--;
    return pom;
   }
   return -1;
}
bool Fronta::JePrazdna() const
{
  if (posledni==-1) return true;
    else false;
}
int main(int argc, char *argv[])
{
  Fronta f;

  for (int i=1; i<=2; i++)
    f.PridejNaKonec(i);

  cout<<"Vybrani fronty: "<<endl;
  while (f.JePrazdna()==false)
   cout<<f.OdeberZCela()<<endl;

  return 0;
}
```

Příklad č.2

```
/*
 * Rostislav Fojtík
 *
 * Vytvořte třídu PevnyDisk, která bude obsahovat
 * kapacitu, volnou kapacitu a počet otáček.
 * Dále umožní zjišťovat celkovou kapacitu a celkovou
 * volnou kapacitu všech disků.
 */
#include <iostream.h>

class PevnyDisk
{
  private:
    static long celkovaKapacita;
    static long celkovaVolnaKapacita;
    long kapacita;
    int pocetOtacek; //za minutu
    long volnaKapacita;
  public:
    PevnyDisk(long,int);
    ~PevnyDisk();
    long ZjistiKapacitu() const;
    long PevnyDisk::ZjistiVolnouKapacitu() const;
    int ZjistiPocetOtacek() const;
    bool UlozitData(long);
    static long ZjistiCelkovouKapacitu();
    static long ZjistiCelkovouVolnouKapacitu();
};

PevnyDisk::PevnyDisk(long n, int ot)
{
    kapacita = volnaKapacita = n;
    celkovaKapacita += n;
    celkovaVolnaKapacita += n;
    pocetOtacek = ot; //za minutu
}
PevnyDisk::~PevnyDisk()
{
   celkovaKapacita -= this->kapacita;
   celkovaVolnaKapacita -= this->kapacita;
}
long PevnyDisk::ZjistiKapacitu() const
{
  return this->kapacita;
}
long PevnyDisk::ZjistiVolnouKapacitu() const
{
  return this->volnaKapacita;
}
int PevnyDisk::ZjistiPocetOtacek() const
{
  return this->volnaKapacita;
}
bool PevnyDisk::UlozitData(long n)
{
```

```
  if (n<=volnaKapacita)
  {
   volnaKapacita -=n;
   celkovaVolnaKapacita -= n;
   return true; //povedlo se uložit data
  }
  else return false;//nepovedlo se uložit data

}
long PevnyDisk::ZjistiCelkovouKapacitu()
{
  return celkovaKapacita;
}
long PevnyDisk::ZjistiCelkovouVolnouKapacitu()
{
  return celkovaVolnaKapacita;
}

long PevnyDisk::celkovaKapacita=0;
long PevnyDisk::celkovaVolnaKapacita=0;

int main(int argc, char *argv[])
{
  PevnyDisk hdd1(10000000,7200),hdd2(80000000,7200);
  PevnyDisk hdd3(30000000,5600);

  cout<<"Kapacita hdd1="<<hdd1.ZjistiKapacitu()<<"
B"<<endl;
  cout<<"Kapacita hdd2="<<hdd2.ZjistiKapacitu()<<"
B"<<endl;
  cout<<"Kapacita hdd3="<<hdd3.ZjistiKapacitu()<<"
B"<<endl;

  { PevnyDisk hdd4(30000000,7200);

    cout<<"Celkova kapacita vsech disku ="
     <<PevnyDisk::ZjistiCelkovouKapacitu()<<" B"<<endl;
    cout<<"Celkova volna kapacita vsech disku = "
     <<PevnyDisk::ZjistiCelkovouVolnouKapacitu()<<"
B"<<endl;

    cout<<"Volna kapacita
hdd1="<<hdd1.ZjistiVolnouKapacitu()<<" B"
     <<endl;
    cout<<"Volna kapacita
hdd2="<<hdd2.ZjistiVolnouKapacitu()<<" B"
     <<endl;
    cout<<"Volna kapacita
hdd3="<<hdd3.ZjistiVolnouKapacitu()<<" B"
     <<endl;
    cout<<"Volna kapacita
hdd4="<<hdd4.ZjistiVolnouKapacitu()<<" B"
     <<endl;

    hdd1.UlozitData(5000000);
    hdd2.UlozitData(20000000);
```

```
    cout<<"Volna kapacita
hdd1="<<hdd1.ZjistiVolnouKapacitu()<<" B"
      <<endl;
    cout<<"Volna kapacita
hdd2="<<hdd2.ZjistiVolnouKapacitu()<<" B"
      <<endl;
    cout<<"Volna kapacita
hdd3="<<hdd3.ZjistiVolnouKapacitu()<<" B"
      <<endl;
    cout<<"Volna kapacita
hdd4="<<hdd4.ZjistiVolnouKapacitu()<<" B"
      <<endl;

    cout<<"Celkova kapacita vsech disku = "
      <<PevnyDisk::ZjistiCelkovouKapacitu()<<" B"<<endl;
    cout<<"Celkova volna kapacita vsech disku = "
      <<PevnyDisk::ZjistiCelkovouVolnouKapacitu()<<"
B"<<endl;
  }
 cout<<"Po zruseni disku hdd4"<<endl;
 cout<<"Celkova kapacita vsech disku = "
      <<PevnyDisk::ZjistiCelkovouKapacitu()<<" B"<<endl;
 cout<<"Celkova volna kapacita vsech disku = "
      <<PevnyDisk::ZjistiCelkovouVolnouKapacitu()<<"
B"<<endl;

 return 0;
}
```

20. CVIČENÍ Č.5

Po absolvování lekce budete:
- umět definovat dědice (potomky)
- umět pracovat s jednoduchou dědičností
- umět definovat konstruktory u potomků
- umět pracovat s pozdní vazbou pomocí virtuálních metod

Časová náročnost lekce: 2 hodiny

Příklad č.1

Vytvořme třídu *Vozidlo* a jejich dědice *OsobniAuto* a *NakladniAuto*. Návrh tříd si co nejvíce zjednodušíme. Naším cílem bude pouze si vyzkoušet mechanizmus dědičnosti. V reálném programu by návrh tříd určitě vypadal jinak.

Nadefinujte si sami třídu *Motorka* jako dědice třídy *Vozidlo*.

```
#include <iostream.h>
#include <stdlib.h>
#include <string.h>

class Vozidlo
{
  private:
    char spz[8];
    long obsahMotoru;
    long hmotnostVozidla;
    int maxRychlost;
    int maxOsob;
  public:
    Vozidlo(long om);
    Vozidlo(long om, long hv, int mr, int mo);
    char *zjistiSpz() const;
    long zjistiObsahMotoru() const;
    long zjistiHmotnostVozidla() const;
    int zjistiMaxRychlost() const;
    int zjistiMaxOsob() const;
    bool zmenSpz(char *);
    void zmenObsahMotoru(long);
    void zmenHmotnostVozidla(long);
    void zmenMaxRychlost(int);
    void zmenMaxOsob(int);

};

Vozidlo::Vozidlo(long om)
{
  obsahMotoru = om;
  //vynulovat ostatní hodnoty, aby v nich nebyla náhodná
čísla
  hmotnostVozidla = maxRychlost = maxOsob = 0;
  spz[0]='\0';
}
Vozidlo::Vozidlo(long om, long hv, int mr, int mo)
```

```
{
  obsahMotoru = om;
  hmotnostVozidla = hv;
  maxRychlost = mr;
  maxOsob = mo;
  spz[0]='\0';
}
char *Vozidlo::zjistiSpz() const
{
  char *p=new char[8];
  strcpy(p,spz);
  return p;
}
long Vozidlo::zjistiObsahMotoru() const
{
  return obsahMotoru;
}
long Vozidlo::zjistiHmotnostVozidla() const
{
  return hmotnostVozidla;
}
int Vozidlo::zjistiMaxRychlost() const
{
  return maxRychlost;
}
int Vozidlo::zjistiMaxOsob() const
{
  return maxOsob;
}
bool Vozidlo::zmenSpz(char *s)
{
  if (strlen(s)>8) return false;
   else strcpy(spz,s);
  return true;
}
void Vozidlo::zmenObsahMotoru(long om)
{
  obsahMotoru = om;
  //bylo by vhodné ošetřit, zda daný parametr je správný
}
void Vozidlo::zmenHmotnostVozidla(long p)
{
  hmotnostVozidla = p;
  //bylo by vhodné ošetřit, zda daný parametr je správný
}


void Vozidlo::zmenMaxRychlost(int p)
{
  maxRychlost = p;
  //bylo by vhodné ošetřit, zda daný parametr je správný
}
void Vozidlo::zmenMaxOsob(int p)
{
  maxOsob = p;
  //bylo by vhodné ošetřit, zda daný parametr je správný
```

```
}
/*** konec třídy Vozidlo ***/

enum OsobniTyp{sportovni,tridverovy,klasicky,kombi};
class OsobniAuto:public Vozidlo
{
  private:
    int pocetDveri;
    long objemZavazadlovehoProstoru;
    OsobniTyp typ;
  public:
    OsobniAuto(long om, OsobniTyp ot, long ozp);
    int zjistiPocetDveri() const;
    long zjistiObjemZavazadlovehoProstoru() const;
    OsobniTyp zjistiTyp() const;
    void zmenObjemZavazadlovehoProstoru(long);
};
OsobniAuto::OsobniAuto(long om, OsobniTyp ot, long
ozp):Vozidlo(om)
{
    objemZavazadlovehoProstoru = ozp;
    typ = ot;
    if (typ <=1) pocetDveri=3;
    if (typ >=2) pocetDveri=5;
}
int OsobniAuto::zjistiPocetDveri() const
{
  return pocetDveri;
}
long OsobniAuto::zjistiObjemZavazadlovehoProstoru() const
{
  return objemZavazadlovehoProstoru;
}
OsobniTyp OsobniAuto::zjistiTyp() const
{
  return typ;
}
void OsobniAuto::zmenObjemZavazadlovehoProstoru(long p)
{
  objemZavazadlovehoProstoru = p;
}
/*** konec třídy OsobniAuto ***/

enum NakladniTyp{sklapecka,skrinova};




class NakladniAuto:public Vozidlo
{
  private:
    long maxNosnostNakladu;
    long hmotnostNakladu;
    long maxObjemNakladu;  //maximální objem nákladu
    NakladniTyp typ;
  public:
```

```
    NakladniAuto(long om, NakladniTyp t, long mn, long
mo);
    long zjistiHmotnostNakladu() const;
    long zjistiMaxNosnostNakladu() const;
    long zjistiMaxObjemNakladu() const;
    NakladniTyp zjistiTyp() const;
    bool zmenHmotnostNakladu(long);
};
NakladniAuto::NakladniAuto(long om, NakladniTyp t, long
mn, long mo):Vozidlo(om)
{
    maxNosnostNakladu = mn;
    hmotnostNakladu = 0;
    maxObjemNakladu = mo;
    typ = t;
}
long NakladniAuto::zjistiHmotnostNakladu() const
{
  return hmotnostNakladu;
}
long NakladniAuto::zjistiMaxNosnostNakladu() const
{
  return maxNosnostNakladu;
}
long zjistiMaxObjemNakladu() const
{
 return maxObjemNakladu;
}
NakladniTyp NakladniAuto::zjistiTyp() const
{
 return typ;
}
bool NakladniAuto::zmenHmotnostNakladu(long p)
{
  if (p<=maxNosnostNakladu) hmotnostNakladu=p;
  else return false;
  return true;
}
/*** konec třídy NakladniAuto ***/


int main(int argc, char *argv[])
{
  Vozidlo v1(5);

  v1.zmenSpz("1T12345");
  cout<<v1.zjistiObsahMotoru()<<endl;
  cout<<v1.zjistiSpz()<<endl;

  v1.zmenSpz("1T51212");
  cout<<v1.zjistiSpz()<<endl;

  OsobniAuto oa1(2,kombi,100);
  oa1.zmenSpz("5T53233");
  cout<<"Osobni auto:"<<endl;
  cout<<oa1.zjistiSpz()<<endl;
```

```
  cout<<oa1.zjistiTyp()<<endl;

  NakladniAuto na1(2,sklapecka,100);
  na1.zmenSpz("2T45566");
  cout<<"Nakladni auto:"<<endl;
  cout<<na1.zjistiSpz()<<endl;
  cout<<na1.zjistiTyp()<<endl;

  return 0;
}
```

21. CVIČENÍ Č.6

Po absolvování lekce budete:

- umět definovat dědice (potomky) v rámci vícenásobné dědičnosti
- umět definovat virtuální bázovou třídu
- umět definovat konstruktory u potomků
- umět pracovat s pozdní vazbou pomocí virtuálních metod

Časová náročnost lekce: 2 hodiny

Příklad č.1

Vytvořte třídu *Osoba* a její potomka třídu *Student* a *Zaměstnanec*. Následně vytvořte třídu *Doktorand* jako potomka tříd *Osoba* a *Zaměstnanec*.

Pro zjednodušení budeme ve třídě *Osoba* uvažovat jen vlastnosti jméno a věk. Ve třídě *Student* nás bude zajímat pouze počet kreditů a studijní ročník. U třídy *Zaměstnanec* uvažujeme pouze plat a jméno katedry. U *Doktoranda* jen téma doktorské práce.

Všimněte si při výpisu programu pořadí volání konstruktorů a destruktorů.

```
/*
 * Mgr. Rostislav Fojtík, 2003
 * Virtuální bázová třída
 *
 * Program zjednodušujeme na co největší míru.
 */
#include <iostream.h>
#include <stdlib.h>
#include <string.h>

class Osoba
{
  protected:
    int vek;
    char jmeno[20];
  public:
    Osoba(char*,int);
    int zjistiVek()const;
    bool zmenVek(int);
    void zmenJmeno(char *);
    char *zjistiJmeno() const;
    virtual void vypisUdaju();
    virtual ~Osoba();
};

class Student:virtual public Osoba
{
  private:
    int pocetKreditu;
    int rocnik;
  public:
    Student(char*jm,int ve,int pk,int ro);
```

```
    int zjistiPocetKreditu() const;
    int zjistiRocnik() const;
    void zvysPocetKreditu(int);
    void zvysRocnik();
    virtual void vypisUdaju();
    virtual ~Student();
};
class Zamestnanec:virtual public Osoba
{
  private:
    long plat;
    char katedra[20];
  public:
    Zamestnanec(char*jm,int ve,char *kat, long pl);
    long zjistiPlat() const;
    char *zjistiKatedru() const;
    void zvysPlat(int);
    virtual void vypisUdaju();
    virtual ~Zamestnanec();
};

class Doktorand:public Student, public Zamestnanec
{
  private:
    char temaDoktPrace[200];
  public:
    Doktorand(char*jm,int ve,char *kat, long pl, int
pk,int ro,char*te);
    virtual ~Doktorand();
    virtual void vypisUdaju();
    char *zjistiTema() const;
};




/*** definice metod ***/
Osoba::Osoba(char *j, int v)
{
  cout << "Parametricky konstruktor tridy Osoba"<<endl;
  //předcházejí řádek je jen pro kontrolu
  strcpy(jmeno,j);
  vek=v;
}
Osoba::~Osoba()
{
  cout <<"Destruktor tridy Osoba"<<endl;
  //předcházejí řádek je jen pro kontrolu
}
int Osoba::zjistiVek()const
{
  return vek;
}
char *Osoba::zjistiJmeno() const
{
  char *p=new char[20];
```

```
  strcpy(p,jmeno);
  return p;
}
bool Osoba::zmenVek(int v)
{
  if (v>=0 && v<=130)
  {
    vek = v;
    return true;
  }
  else
  {
    vek=0;
    return false;
  }
}
void Osoba::zmenJmeno(char *p)
{
  strcpy(jmeno,p);
}
void Osoba::vypisUdaju()
{
  cout <<"Osoba:"<<endl;
  cout <<"Jmeno: "<<jmeno<<endl;
  cout<<"Vek: "<<vek<<endl;
}
/*** konec třídy Osoba ***/

Student::Student(char*jm,int ve,int pk,int
ro):Osoba(jm,ve)
{
  cout << "Parametricky konstruktor tridy Student"<<endl;
  //předcházejí řádek je jen pro kontrolu
  pocetKreditu = pk;
  rocnik = ro;
}
int Student::zjistiPocetKreditu() const
{
  return pocetKreditu;
}
int Student::zjistiRocnik() const
{
  return rocnik;
}
void Student::zvysPocetKreditu(int kr)
{
  pocetKreditu +=kr;
}
```

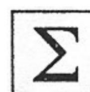
```
void Student::zvysRocnik()
{
  rocnik++;
}
void Student::vypisUdaju()
{
  cout <<"Student:"<<endl;
  cout <<"Jmeno: "<<jmeno<<endl;
  cout <<"Vek: "<<vek<<endl;
  cout <<"Pocet kreditu: "<<pocetKreditu<<endl;
  cout <<"Rocnik: "<<rocnik<<endl;
}
Student::~Student()
{
   cout <<"Destruktor tridy Student"<<endl;
  //předcházejí řádek je jen pro kontrolu
}
/*** konec třídy Student ***/

Zamestnanec::Zamestnanec(char*jm,int ve,char *kat, long
pl)
                :Osoba(jm,ve)
{
  cout << "Parametricky konstruktor tridy
Zamestnanec"<<endl;
  //předcházejí řádek je jen pro kontrolu
  strcpy(katedra,kat);
  plat=pl;
}
long Zamestnanec::zjistiPlat() const
{
   return plat;
}
char *Zamestnanec::zjistiKatedru() const
{
  char *p=new char[20];
  strcpy(p,katedra);
  return p;
}
void Zamestnanec::zvysPlat(int p)
{
  plat=p;
}
void Zamestnanec::vypisUdaju()
{
  cout <<"Zamestnanec:"<<endl;
  cout <<"Jmeno: "<<jmeno<<endl;
  cout <<"Vek: "<<vek<<endl;
  cout <<"Plat: "<<plat<<endl;
  cout <<"Katdra: "<<katedra<<endl;
}
Zamestnanec::~Zamestnanec()
{
  cout <<"Destruktor tridy Zamestnanec"<<endl;
  //předcházejí řádek je jen pro kontrolu
}
```

```
/*** konec třídy Zamestnanec ***/

Doktorand::Doktorand(char*jm,int ve,char *kat, long pl,
int pk,
           int ro, char*tem):
      Osoba(jm,ve),Zamestnanec(jm,ve,kat,pl),Student(jm,ve
      ,pk,ro)
{
  cout << "Parametricky konstruktor tridy
Doktorand"<<endl;
  //předcházejí řádek je jen pro kontrolu

  strcpy(temaDoktPrace,tem);
}
Doktorand::~Doktorand()
{
  cout <<"Destruktor tridy Doktorand"<<endl;
  //předcházejí řádek je jen pro kontrolu
}

void Doktorand::vypisUdaju()
{
  cout <<"Doktorand:"<<endl;
  cout <<"Jmeno: "<<jmeno<<endl;
  cout <<"Vek: "<<vek<<endl;
  cout <<"Plat: "<<zjistiPlat()<<endl;
  cout <<"Katedra: "<<zjistiKatedru()<<endl;
  cout <<"Pocet kreditu: "<<zjistiPocetKreditu()<<endl;
  cout <<"Rocnik: "<<zjistiRocnik()<<endl;
  cout <<"Tema doktorske prace: "<<temaDoktPrace<<endl;
}
char *Doktorand::zjistiTema() const
{
  char *p=new char[200];
  strcpy(p,temaDoktPrace);
  return p;
}
/*** konec třídy Doktorand ***/


int main(int argc, char *argv[])
{
  Osoba *po;


  po=new Student("Pavel",22,103,3);
  po->vypisUdaju();
  delete po;

  po=new Doktorand("Karel",25,"Informatika",5000,111,2,
                    "Vliv C++ na spanek");
  po->vypisUdaju();
  delete po;

  return 0;
}
```

Poznámky k programu:

- V reálném programu by bylo lepší se vyhnout vícenásobné dědičnosti, která mnohdy přináší zbytečné komplikace. V našem zadání bychom mohli třídu *Doktorand* definovat pouze jako dědice třídy *Student*, neboť se nejedná o plnohodnotného zaměstnance (např. nezvyšuje se mu platová třída).
- Všimněte si nešikovně definovaných konstruktorů, kterým se v rámci dědičnosti neustále zvyšuje počet parametrů. Když si uvědomíme, že naše třídy jsou velice zjednodušeny a neobsahují další údaje (rodné číslo, adresa, telefon, studijní obor, platová třída, kvalifikace, tituly…), pak bychom se dostali k obrovskému počtu parametrů u konstruktorů. Je potřeba si uvědomit, že není důležité většinu z vlastností objektu ihned nastavovat a inicializovat při konstrukci objektů!

Shrnutí



- Nezapomeňte, že se nedědí konstruktory a destruktory. U konstruktoru potomka je navíc nutné zavolat konstruktor předka! Máme-li virtuální bázovou třídu, je potřeba zavolat i konstruktor této třídy.
- Nevytvářejte konstruktory s mnoha parametry. V dalších potomcích by konstruktory měly neúnosné množství parametrů. Ne vždy je nutné inicializovat všechny vlastnosti na určité hodnoty již při konstrukci objektu. Na druhé straně se vyhýbejte neinicializovaným proměnným, které pak v sobě mají „náhodné" hodnoty.
- Při návrhu programu vždy dobře zvažme, zda je skutečně nutné definovat vícenásobnou dědičnost. Zda není možné využít vhodnější a méně komplikované jednoduché dědičnosti.

22. CVIČENÍ Č.7

Po absolvování lekce budete:
- umět definovat a používat přetížené operátory

Časová náročnost lekce: 2 hodiny

Příklad č.1

Vytvořte třídu *Komplex* pro práci s komplexními čísly a předefinujte si některé operátory.

```
/*
 * přetížení operátorů
 *
 * Rostislav Fojtík
 */

#include <iostream.h>
#include <stdlib.h>

class COMPLEX
{
   double re, im;
  public:
   COMPLEX(double, double);
   COMPLEX();

   friend COMPLEX operator+(COMPLEX,COMPLEX);
   friend COMPLEX operator-(COMPLEX,COMPLEX);
   friend ostream &operator<<(ostream &vys, COMPLEX x);

   COMPLEX & operator+=(COMPLEX);
   COMPLEX & operator-=(COMPLEX);
};
COMPLEX::COMPLEX(double r, double i)
{
  re=r;
  im=i;
}
COMPLEX::COMPLEX()
{
  re=im=0.0;
}
inline COMPLEX & COMPLEX::operator+=(COMPLEX x)
{
   re += x.re;
   im += x.im;
   return *this;
}
inline COMPLEX & COMPLEX::operator-=(COMPLEX x)
{
   re -= x.re;
   im -= x.im;
   return *this;
}
```

```
// pratelske radove funkce
inline COMPLEX operator+(COMPLEX x1,COMPLEX x2)
{
   return COMPLEX(x1.re + x2.re,x1.im + x2.im);
}

inline COMPLEX operator-(COMPLEX x1,COMPLEX x2)
{
   return COMPLEX(x1.re + x2.re,x1.im + x2.im);
}

ostream &operator<<(ostream &vys, COMPLEX x)
{
 vys << x.re << " + i. " << x.im;
 return vys;
}
/******************************/
int main()
{

  COMPLEX v, a(11.0,12.0),b(13.0,14.0);

  v = a + b;
  cout << "v =a + b" << endl;
  cout << "v= " << v << endl;
  cout << "a= " << a << endl;
  cout << "b= " << b << endl;

  v = b - a;
  cout << "v = a - b" << endl;
  cout << "v= " << v << endl;
  cout << "a= " << a << endl;
  cout << "b= " << b << endl;

  v += a;
  cout << "v +=a" << endl;
  cout << "v= " << v << endl;
  cout << "a= " << a << endl;
  cout << "b= " << b << endl;

  v -= b;
  cout << "v -=b" << endl;
  cout << "v= " << v << endl;
  cout << "a= " << a << endl;
  cout << "b= " << b << endl;

  system("PAUSE");
  return 0;
}
```

Úkol: Doplňte si další přetížené operátory pro práci s komplexními čísly.

23. CVIČENÍ Č.8

Po absolvování lekce budete:
- umět definovat šablony pro řadové funkce
- umět definovat metatřídy - šablony pro třídy

Časová náročnost lekce: 2 hodiny

Příklad č.1

```
/*
 * Mgr. Rostislav Fojtík, 2003
 *
 * Šablony řadových funkcí
 *
 */
#include <iostream.h>

template <typename Typ> Typ mensi(Typ a, Typ b)
{
  return (a<b)?a:b;
}

/* Následující řešení pomocí referencí je výhodnější
   pro rozsáhlejší datové typy, neboť se netvoří
   lokální kopie. */
template <typename Typ> Typ& vetsi(Typ &a, Typ &b)
{
  return (a<b)?a:b;
}

int main(int argc, char *argv[])
{
  int x=12,y=23;

  cout<<mensi(x,y)<<endl;
  cout<<vetsi(x,y)<<endl;
  return 0;
}
```

Příklad č.2

```
/*
 * Mgr. Rostislav Fojtík, 2003
 *
 * Vytvořte šablonu pro třídění pole
 *
 */
#include <iostream.h>
#include <string.h>

template <typename Typ> void Sort(Typ pole[],int pocet)
{
  for(int i=0;i<pocet-1;i++)
    for(int j=0;j<pocet-1;j++)
      if (pole[j]>pole[j+1])
      {
        Typ pom=pole[j];
        pole[j]=pole[j+1];
        pole[j+1]=pom;
      }
}


int main(int argc, char *argv[])
{
  int pole1[]={12,4,6,23,54,6,87,26};
  float pole2[]={1.2,4.4,6.6,1.23,5.4,66.7,8.7,0.26};

  Sort(pole1,8);
  for(int i=0;i<8;i++)
    cout<<pole1[i]<<" ";
  cout<<endl;

  Sort(pole2,8);
  for(int i=0;i<8;i++)
    cout<<pole2[i]<<" ";
  cout<<endl;


  return 0;
}
```

Doplňte program o možnost třídění pole řetězců.

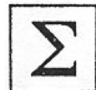
Příklad č.3

```
/*
 * Mgr. Rostislav Fojtík, 2003
 *
 * Vytvořte šablonu pro frontu
 *
 */
#include <iostream.h>

template <typename Typ> class Fronta
{
  private:
    Typ *pole;
    int vrchol;
    int max; //maximální možný počet prvků
  public:
    Fronta(int);
    void pridejNaKonec(Typ);
    Typ odeberZCela();
    bool jePrazdna();
    ~Fronta();
};

template <typename Typ> Fronta<Typ>::Fronta(int pocet)
{
  max=pocet;
  vrchol=-1;
  pole=new Typ[max];//ošetřete alokaci paměti
}
template <typename Typ> Fronta<Typ>::~Fronta()
{
  delete []pole;
}
template <typename Typ> void
Fronta<Typ>::pridejNaKonec(Typ hod)
{
  if (vrchol<max)
  {
    vrchol++;
    pole[vrchol]=hod;
  }
}
template <typename Typ> Typ Fronta<Typ>::odeberZCela()
{
  if (jePrazdna()==false)
  {
    Typ pom=pole[0];
    for(int i=1;i<=vrchol;i++)
      pole[i-1]=pole[i];
    vrchol--;
    return pom;
  }
}


template <typename Typ> bool Fronta<Typ>::jePrazdna()
```

```
{
  if (vrchol<0) return true;
   else return false;
}


int main(int argc, char *argv[])
{
  Fronta<int> f1(5);
  for(int i=10;i<15;i++)
    f1.pridejNaKonec(i);

  while (f1.jePrazdna()!=true)
    cout<<f1.odeberZCela()<<endl;

  return 0;
}
```

Kontrolní úkol:

Zamyslete se nad omezeními, která přináší výše uvedený způsob řešení. Které části kódu mohou způsobit problémy, případně nemožnost využít šablonu pro určité datové typy?



Shrnutí

- Mechanizmus šablon nám může usnadnit návrh programů. V mnohých případech je vhodnější vytvořit pro daný algoritmus šablonu než přetíženou funkci.
- Během definice šablon je potřeba mít na zřeteli, že ne všechny operace a oprátory lze využít u všech typů.
- Parametry šablon mohou být *typové* nebo *hodnotové*.