Programovací jazyk C++

Mgr. Rostislav Fojtík
Ostrava, 1998

Obsah

1. Úvod ... 3
2. Základy objektově orientovaného programování v jazyku C++ ... 4
2.1. Třídy ... 4
2.2. Dědičnost - inheritance ... 6
3. Nové prvky jazyka C++ ... 8
3.1. Proudový vstup a výstup ... 9
3.2. Funkce ... 10
3.2.1. Nové prvky ... 10
3.2.2. Odkazy ... 10
3.2.3. Přetížené funkce ... 11
3.2.4. Funkce main ... 11
3.3. Práce se soubory ... 12
3.3.1. Otevření a uzavření souboru ... 12
3.3.2. Formátový a neformátový vstup a výstup ... 14
3.3.3. Přímý přístup k souboru ... 14
3.4. Dynamická alokace paměti ... 15
3.4.1. Operátor new ... 15
3.4.2. Operátor delete ... 15
3.5 Přetížený operátor ... 16
4. Objektově orientované programování v jazyku C++ ... 17
4.1. Třídy a instance ... 17
4.1.1. Konstrukce třídy ... 17
4.1.2. Datové prvky ... 17
4.1.3. Standardní metody - konstruktor a destruktor ... 17
4.1.4. Deklarace a definice metod ... 18
4.1.5. Statické datové členy a funkce ... 23
4.1.6. Přátelé ... 24
4.1.7. Kompozice objektů ... 25
4.2. Dědičnost - inheritance ... 26
4.3. Polymorfismus ... 31
4.3.1. Virtuální metody ... 31
4..3.2. Abstraktní a instanční třídy ... 32
4.3. Vícenásobná dědičnost ... 33
5. Šablony ... 34
6. Výjimky ... 36
7. Přetypování ... 38
Literatura ... 39

1. Úvod

Tyto učební texty slouží pro studenty III. ročníku studijních oborů se zaměřením na informatiku. Učivo navazuje na znalosti ze zimního semestru o programování v jazyku C.

Jazyk C++ je odvozen od jazyka C. Vnáší do jazyka nové možnosti, upravuje a rozšiřuje některé prvky standardního jazyka C. Jazyk C++ navrhl Bjarne Stroustrup z Bellových laboratoří AT&T. Ten pro svou práci na simulaci modelu pro distribuované systémy zjistil, že se mu velmi dobře hodí objektový přístup. Nejprve použil jazyk Simula, který se příliš neosvědčil, proto se zaměřil na návrh nového jazyka, který by nejen podporoval objektový přístup, ale byl i dostatečně rychlý. Jako základ použil existující jazyk C. Nový jazyk nejprve dostal název "C with Classes" - C s třídami. Později se prosadil název C++ ("následník C"). Standardizace jazyka C++ není doposud uzavřena.

K nejznámějším překladačům C/C++ patří překladače firem Borland (Borland C/C++ 5.0), Microsoft (Visual C/C++ 2.0), Watcom C/C++ 11, Zortech, IBM ...

2. Základy objektově orientovaného programování v jazyku C++

Programovací jazyk C vychází z koncepce, která rozděluje program na data a algoritmické struktury (reprezentované např. funkcemi). Obě skupiny mohou být zpracovávány teoreticky nezávisle na sobě (algoritmy musí být pouze vhodné pro vstup, zpracování a výstup určitých dat).

Jazyk C++ může využívat většiny postupů jazyka C, ale navíc principů *objektově orientovaného programování*, které vnímají data i příslušné algoritmy jako jeden celek. Data a algoritmy jsou sdruženy v objektech. S daty se v objektech manipuluje pomocí metod (v jazyku C++ se metody často označují *členskými funkcemi*), které jsou součásti daného objektu. Často se hovoří o tom, že "objekty si posílají zprávy". Toto posílání zpráv objektu je realizováno jako vyvolání některé z jeho metod.

Obecné vlastnosti charakterizující OOP:

1. Zapouzdření (Encapsulation)

Spojení prvků dat s metodami, které mají pracovat s daty.

2. Dědičnost (Inheritance)

Možnost odvozovat nové třídy, které dědí data a metody z jedné nebo více tříd. V odvozených třídách je možno přidávat nebo předefinovávat nová data a metody.

3. Polymorfismus

Česky možno vyjádřit přibližně pojmem "vícetvarost". Umožňuje jediným příkazem zpracovávat "podobné" objekty.

Tyto vlastnosti umožňují vytvářet lépe strukturovaný a udržovatelný program.

2.1. Třídy

Srovnání klíčových pojmů rozdílných jazyce C a C++:

uživatelem definovaný datový typ	třída
proměnná	objekt (instance třídy)
funkce	metoda (členská funkce)
volání funkce	zpráva, událost

Základem OOP 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 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*.

Příklad deklarace:

```
class rozpocet{// deklarace třídy, není nutné používat typedef
            private:    // seznam soukromých členů
           int hotovost;      // soukromý člen třídy - data
            public:     // seznam veřejných členů třídy
           int plat(int velikost);// metoda, neboli členská funkce
           int najemne(int n);      // deklarace metody
           int pujcky(int p); // deklarace metody
            };
```

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

```
      rozpocet me_penize;     // rozpocet je třída, me_penize
                              // představuje objekt
```

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

struktura v C	struktura v C++	třída v C++
typedef struct{	struct hodnota{	class hodnota{
int a;	int a;	public:
float b;	float b;	int a;
}hodnota;	};	float b; };

Rozdíl mezi strukturami v C a třídami v C++ je v úrovni přístupu ke členům. Ten se určuje pomocí **public:**, **privat:** a **protected:** (veřejné, soukromé a chráněné členy).

Veřejné členy - na ně se můžeme 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 inicializaci členských dat. jedná se o speciální funkci, která je automaticky volána při vytvoření objektu. 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. Nelze však přetížit a nemá žádné parametry.

Příklad

Příklad výpisu textu na obrazovku pomocí projektu a objektově orientovaného přístupu:

```
//*** 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 metod
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    ***/
//*** Konec příkladu ***//
```

2.2. Dědičnost - inheritance

Inheritance umožňuje přidat k základní (rodičovské nebo bázové) 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 inheritance - třída má jen jednoho předka (rodiče). Vytváříme stromovou hierarchii tříd. Třídu v nejvyšší úrovní označujeme jako kořenovou třídu.
Vícenásobná inheritance - třída má více předků.
Opakovaná inheritance - 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
            }
```

3. Nové prvky jazyka C++

Jazyk C++ obsahuje oproti jazyku C další klíčová slova:

class	**delete**	**friend**	**inline**	**new**	**operator**
private	**protected**	**public**	**template**	**this**	**virtual**

V jazyce C se pro ukazatele, které nemají nikam ukazovat používá makro NULL, která 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 nový datový typ **bool**, který se řadí mezi celočíselné typy a který může nabývat hodnot **false** (0) a **true** (1).

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

```
// vše od dvou lomítek až do konce řádku je bráno jako komentář
```

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 funkci k předávání parametrů odkazem.

```
int prom;
int &ref_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;
```

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 vhodnější než používání maker jako v jazyce C. 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 druhý řádek globální proměnné  }
```

Struktury (**struct**) a unie (**union**) patří mezi objektové typy (položky mohou být datové typy i metody). 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. V C++ mohou unie také představovat objektové typy, ale s omezeními: nemohou mít předky ani potomky, jejich prvky nesmí být instance objektového typu s konstruktorem, destruktorem nebo přetíženým operátorem "=". Samotné unie své konstruktory mít mohou.

3.1. Proudový vstup a výstup

C++ má nové možnosti usnadňující vstup a výstup. 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 připojený 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 výstupu. Manipulátory jsou definovány v hlavičkové souboru *iostream*.h.

manipulátor	**zápis**	**činnost**
dec	outs << dec ins >> dec	nastaví vlajku desítkové konverze
hex	outs << hex ins >> hex	nastaví vlajku šestnáctkové konverze
oct	outs << oct ins >> oct	nastaví vlajku osmičkové konverze
ws	ins >> ws	odstraňuje bílé znaky
endl	outs << endl	vloží konec řádku a vyprázdní bufer
ends	outs << ends	přidá k řetězci ukončovací nulu
flush	outs << flush	vyprázdní bufer výstupního proudu
setbase(int)	outs << setbase(n)	nastaví číselnou bází na *n* (0,8,10,16) 0 představuje desítkový základ

resetiosflags(long)	ins >> resetiosflags(l) outs << resetiosflags(l)	zruší specifikované formátovací bity
setiosflags(long)	ins >> setiosflags(l) outs << setiosflags(l)	nastaví specifikované formátovací bity
setfill(int)	ins >> setfill(n) outs << setfill(n)	nastaví znak výplně na *n*
setprecision(int)	ins >> setprecision(n) outs << setprecision(n)	nastaví fp přesnost na *n* číslic
setw(int)	ins >> setw(n) outs << setw(n)	nastaví šířku výstupu na *n* pozic

Příklad:

```
int cislo = 200;
cout.fill($);
cout.width(4);
cout << cislo;  //zobrazí se $200.
```

3.2. Funkce

3.2.1. Nové prvky

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.

3.2.2. Odkazy

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, 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 referenci.*.

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;
```

3.2.3. Přetížené funkce

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

```
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 nešlo, museli bychom napsat různé funkce pro různé datové typy. Například funkce *int abs_i(int n);* a `double` *abs_d(double n);* a podobně.

3.2.4. Funkce main

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**).

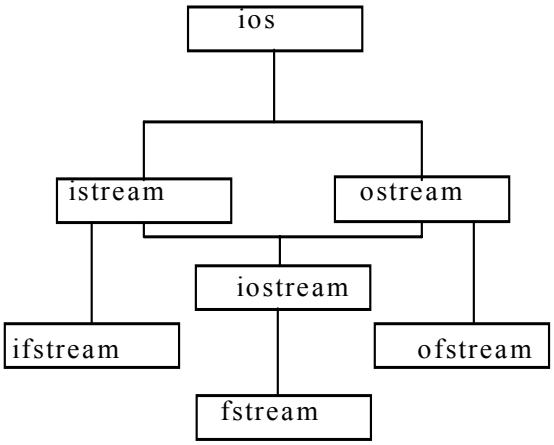
Příklady

1. Vytvořte přetížené funkce *typ mabs(typ n);*, které budou vracet absolutní hodnotu čísel typu *int, double, long*.
2. Vytvořte přetížené funkce *void Tisk(typ prom);*, které budou vypisovat na obrazovku proměnnou typu *int, double, char, char **.
3. Vytvořte funkci *int Suma(int dolni=1, int horni=50,int krok=1);* , která bude vracet součet celých čísel od *dolní* hranice do *horní, krok* udává vzdálenost mezi sousedními čísly. Volejte funkci s různě nastavenými parametry.

3.3. 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:



3.3.1. Otevření a uzavření souboru

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 ****************/
#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) );//neformátovaný 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.

v*oid 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 ******************/
```

3.3.2. Formátový a neformátový vstup a výstup

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). Druhý 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&);

3.3.3. Přímý přístup k souboru

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 v 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

3.4. 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.

3.4.1. Operátor new

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 hodnotu 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.

```
int matice[10][20];
matice = new int[10][20];
```

3.4.2. Operátor delete

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;

3.5 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 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.

Příklad:

```
#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;
}

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

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

  return 0;
}
```

4. Objektově orientované programování v jazyku C++

4.1. Třídy a instance

4.1.1. 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 definovaný 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.

4.1.2. 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.

4.1.3. Standardní metody - konstruktor a destruktor

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 nějaký 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 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
      }
```

4.1.4. 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 skrytý 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.

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).

Příklad:

```
      // 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);* dělají vlastně stejnou činnost 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. 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 přístupné z vnějšku třídy a může je používat jen daná třída.

Příklad

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.

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

```
// zacatek deklarace tridy Datum
class Datum
{
  private:
    int den, mesic, rok;
    // Privat metody pro kontrolu spravnych udaje, je-li hodnota nevyhovujici,
    // vrati metoda nejblizsi spravnou hodnotu.
    int SpravnyDen(int d);
    int SpravnyMesic(int m);
    int SpravnyRok(int r);
  public:
    Datum();
    Datum(int d, int m, int r);
    Datum(int d, char *m, int r);
    void VypisDatum() const;
    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);
    ~Datum() {}
};
// konec deklarace tridy Datum

// 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 pouzit rok pouze v rozmezi 1980 - 1999.
```

```
  if (r<1980) return 1980;
   else
    if (r>1999) return 1999;
      else return r;
}

Datum::Datum()
{
   struct date d;

   getdate(&d);
   rok= d.da_year;
   mesic= d.da_day;
   den= d.da_mon;
}

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 pripade chybneho retezce
  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;
   Datum d2(13,2,1998);
   Datum d3(13,"cervenec",1998);

   clrscr();
   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();
   getch();
   return 0;
}
// *** Konec příkladu ***  //
```

4.1.5. 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. Před použitím instancí nesmíme zapomenout inicializovat statická členská data.

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.

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();
};

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 smrk(10), jedle(2), borovice(8);

   smrk.VypisDruhu();
   jedle.VypisDruhu();
   borovice.VypisDruhu();

   stromy::VypisCelkem();     //volání statické metody

   return 0;
}
```

4.1.6. Přátelé

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 nebo i celé třídy.

```
class A
{
   friend int f() {......}    //přátelská funkce
   friend class B;                  //přátelská třída
   friend int C::metoda();    //přátelská metoda jiné třídy
}
```

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;
}
```

4.1.7. Kompozice objektů

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.

Příklad:

```
class AAA
{
  int pom;
 public:
  AAA(int p){pom=p;}
  int DejA()const {return pom;}
};
```

```
class CCC
{
   int hod;
   AAA a;
   AAA *b;
  public:
   CCC(int h);//:a(12),hod(h){}
   void Vypis();
   ~CCC(){delete b;}
};
CCC::CCC(int h):a(55),hod(h)
{
  b=new AAA(44); //alokace paměti s inicializaci hodnoty
}
void CCC::Vypis()
{
  cout << "hod= "<<hod<<endl;
  cout << "a.pom= "<<a.DejA()<<endl;
  cout << "b.pom= "<<b->DejA()<<endl;
}

int main()
{
  CCC c(1000);

  c.Vypis();

return 0;
}
```

Pokud je ve třídě vložený objekt tvořen dynamicky (viz. Prvek *b*), je potřebné definovat explicitní konstruktor, ve kterém bude alokována paměť, a destruktor, v němž se paměť uvolní.

Součástí třídy může být rovněž ukazatel na objekt, který je však vytvořen nezávisle na dané třídě (např. nějaký globální objekt). V tomto případě konstruktor nepřiděluje a destruktor neuvolňuje paměť pro vložený objekt.

4.2. Dědičnost - inheritance

Inheritance znamená možnost přidávat k základní třídě další vlastnosti 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 LEVHARTI - obsahuje vlastnosti společné pro všechny druhy savců, specifické vlastnosti všech druhů kočkovitých šelem a navíc specifické vlastnosti společné všem levhartům

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**).

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. Zděděné metody takový prvek využívají stejným způsobem jako v základní třídě.
protected - zděděné prvky jsou využitelné ve vlastních metodách odvozené třídy. Nejsou přístupné vnějším přístupem.
public - zděděné prvky jsou využitelné ve vlastních metodách odvozené třídy. Jsou přístupné také i vnějším přístupem.

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

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

Příklad:

```
#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);
  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 ************/
```

4.3. Polymorfismus

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, neboť se 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*.

4.3.1. Virtuální metody

Chceme-li se přenechat rozhodnutí, která překrytá 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 raději *časnou) vazbou (early binding)*.

Příklad:

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

  getch();
  return 0;
}
```

Jestliže při volání virtuální metody nepoužijeme ukazatele, nemusí se pozdní vazba uplatnit (např. *bb.Tisk( );*).

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 se to doporučuje. 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 nazývané *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 to 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, 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 metod nevirtuálních (někdy se říká *statických*, což není moc vhodné, neboť statickou metodu jsme označovali metodu s klíčovým slovem **static**).

4..3.2. 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*. Ostatním třídám říkáme někdy *instanční třídy*.

4.3. Vícenásobná dědičnost

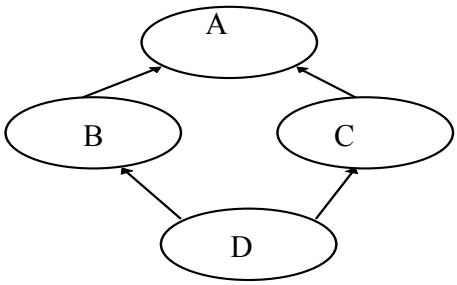
Máme-li třídu vytvořit jako potomka více než jedné třídy najednou, hovoříme o tzv. *vícenásobné dědičnosti*. Téměř každý problém lze sice zvládnout jen pomoci jednoduché dědičnosti, ale mnohdy nám vícenásobná dědičnost zjednoduší a zkrátí dobu návrhu a řešení (viz. datové proudy).

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.

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.

5. Šablony

Šablona specifikuje, jak definovat skupinu příbuzných tříd. Mějme například vytvořenou třídu zásobník, který 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*.

```
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
```

Deklarace šablon objektového typu má tvar

```
      template <class typ> class AA;

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

6. Výjimky

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.

Jazyk C++ umožňuje pracovat pouze s tzv. *synchronními výjimkami*, to znamená výjimkami, které vzniknou uvnitř programu. 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);
```

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

  return 0;
}
```

7. Přetypování

Kromě běžné operace **(část)** zavádí C++čtyři nové operátory k přetypovaní:

dynamic_cast - používá se pro bezpečné přetypování polymorfních tříd, k přetypování mezi potomky a předky a jako jediný z těchto operátorů může využívat dynamické identifikace typů.

static_cast - slouží k běžnému přetypování z předka na potomka nebo naopak bez dynamické kontroly typů

reinterpret_část - umožňuje konverze jejichž výsledek může být závislý na implementaci, cílovéplatformě nebo paměťovém modulu.

const_část - jako jediný z operátorů umožňuje vytvořit z nekonstanty konstantu a naopak, popřípadě přidávat či odebírat modifikátor **volatile**.

Příklad použití:

```
class A{....};
class B:public A
{.....};

int main()
{
  A a;
  A *pa;
  B b;
  B *pb;

  pa=dynamic_část<A*>(pb);
  ....
}
```

Text není dokončen!

Literatura

R. Pecinovský, M. Virius	Objektové programováni I	Grada, 1996
R. Pecinovský, M. Virius	Objektové programováni II	Grada, 1996
M. Virius	Pasti a propasti jazyka C++	Grada, 1997
B. Stroustrup	C++ Programovací jazyk	BEN, 1997
S. Racek	Objektově orientované programování v C++	KOPP, 1994
K. Nenadál, D. Václavíková	Borland C++	Grada, 1992
G. Renner	Borland C++ kompendium	UNIS, 1992
A. Večerka	Jazyk C++	UP Olomouc, 1996
I. Vondrák, P. Šaloun	Objektově orientované programování	VŠB-TU Ostrava, 1995
D. Kačmář	Programování v jazyce C++	VŠB-TU Ostrava, 1995
M. Virius	C++ pro nás ostatní, kurs v časopise Softwarové noviny, ročník 1996 a výš	