Jazyk C++

Jan Cvejn,
Technická univerzita v Liberci, Fakulta mechatroniky
2001-2002

- Moderní jazyk založený na jazyku C
  - obsahuje četná vylepšení oproti jazyku C
  - rozšíření jazyka C o rysy objektově orientovaného programování
  - navržen tak, aby konstrukce jazyku C byly jeho podmnožinou (tj. kód v C byl bez jakýchkoliv modifikací i kódem v C++)
    - v současné podobě neplatí zcela stoprocentně (ale u naprosté většiny kódu ano)
    - zpětná kompatibilita s C je určitým omezením
  - navržen tak, aby byl přeložený kód co do efektivity i velikosti srovnatelný s kódem v C

- Původně speciální preprocesor, generující kód v jazyku C
  - dnes samostatné kompilátory C++ (moderní rysy jazyka (šablony, výjimky)) nelze implementovat preprocesorem, problematické ladění)

- Dnes samostatná norma ANSI jazyka C++ podobně jako u jazyka C

- Objektově orientované jazyky
  - Simula 67 – jeden z prvních obj. orient. jazyků
    - objekty zde odpovídají objektům reálného světa – obsahují kromě atributů a metod i příkazovou část, která se vykonává zdánlivě paralelně (kvaziparalelně) s ostatními objekty
    - objekty se alokují pouze dynamicky, uvolnění paměti je automatické (Garbage Collector)

- U současných jazyků – obj. orientované programování implementované většinou pouze pro zjednodušení a zpřehlednění práce s daty
  - zpřehlednění zápisu programu sdružením datových struktur a souvisejících algoritmů (=objekty)
  - vytváření knihoven univerzálních objektů pro manipulaci s jinými objekty – zjednodušení zápisu programu
  - prevence chyb z neznalosti vnitřní implementace objektů
    - ochrana přístupu k atributům objektů – lze pevně definovat rozhraní pro manipulaci s objekty (většinou = podmnožina metod objektu)
  - snadnější údržba programu
    - každý objekt definuje rozhraní mezi implementační částí a okolním kódem

    - implementační část lze při zachování rozhraní modifikovat bez změn programu

  - V praxi používané jazyky:
    - C++, Objective C, Java, SmallTalk, Python, Visual Basic a další
    - Akademické jazyky: např. Eiffel

- C++ ...jazyk s podporou objektů (OOP je implementováno jen jako rozšíření, s určitým omezením)
- Čisté obj. orient. jazyky – založené na principech obj. orientovaného programování

- Objektově orientovaný přístup v C++
  - Vhodný zejména pro tvorbu rozsáhlých programů

Neobjektová rozšíření oproti C

- komentář //
  - pro jeden řádek
  - původní /* */ zůstává
- přípona souborů .cpp
  - přípona .cpp zpravidla vyvolá kompilátor C++, zatímco .c kompilátor C
- definice proměnných na místě
- přesná hlavička funkce vždy před použitím
  - v ANSI C bylo pouze jako doporučení
- implicitní argumenty funkcí

```
void addMul ( float *pa, float *pb, float ka=1, float
kb=1);
```

- lze:

```
addMul(p1,p2);
addMul(p1,p2,2);
addMul(p1,p2,2,3);
```

`addMul(p1,p2);` - dosadí implicitní hodnoty ka=1, kb=1
`addMul(p1,p2,2);` - dosadí jen kb=1

- implicitní argumenty – buďto pouze v deklaraci hlavičky fce nebo jen v definice fce
- implicitní argumenty musí být na konci seznamu arg.

- inline funkce
  - překládá se jako makro (vkládá se kód na místě volání), ale definuje se jako normální fce
  - zpravidla v hlavičkovém souboru .h, nikoliv v modulu .cpp

- struct lze bez typedef
  - struktura je v C++ přímo třídou (viz dále), nikoliv pouze datovým typem

- reference
  - vytvoří proměnnou sdílející stejné umístění v paměti s jinou proměnnou

```
int a=10;
int &ra=a;
```

  - využívá se jako alternativa k pointerům pro předávání proměnných odkazem

```
addVect ( Vect &v1, Vect &v2);
```

- na rozdíl od pointeru ref. proměnná musí být inicializovaná (tj. při vytvoření vždy ukazuje na existující objekt)

- bylo nutno zavést kvůli přetěžování operátorů (viz dále)

- typ `bool`
  - rovněž nová klíčová slova `true/false`
  - ale lze i přiřazovat hodnoty 0, !=0
  - automatická konverze na typ int a zpět

- operátory `new, delete`
  - bylo třeba zavést pro dynamickou alokaci objektů
    - tj. C++ už není tak nízkoúrovňový jako C

```
Data *pData=new Data;
…
delete pData;
```

- lze využít pro alokaci polí, i vícerozměrných

```
float *pole;
pole=new float[100];
…
delete[] pole;
```

- mez pole může být proměnná
- u vícerozměr. polí může být proměnná pouze první mez

- přetížené funkce
  - lze definovat více funkcí stejného jména pro více typů argumentů
  - překladač podle typu argumentů pozná o kterou funkci jde
    - nejprve se hledá přesná shoda typů, pak se zkouší std. konverze

- přetížené operátory
  - podobně jako u přetížených funkcí
  - využívá se klíčové slovo `operator`

např. : `Complex operator + (Complex &c1, Complex &c2);`

může i vracet referenci na stále existující proměnnou (`static`) :

```
Complex& operator + (Complex &c1, Complex &c2)
{
static Complex s;
 …
return s;
}
```

- ušetření násobného kopírování dat
- ale je třeba s opatrností

  - operátory zachovávají priority

- nelze přetížit všechny operátory, ale většinu:
  - běžně přetěžují se např.: `+,-,*,/,  [], <<,>>, <,>,=,==, !, (), new, delete`

- znakové konstanty jsou typu `char`, ne `int`
- nový význam `const` – nahrazuje `#define`
  - lze použít i pro meze polí, parametry šablon
  - funguje jako statická proměnná v modulu – lze vkládat pomocí `#include`

Objektově orientované programování v C++

- třídy v C++
  - definice shodná s definicí struktury (bez `typedef`)
  - klíčové slovo `struct` nebo `class` (rozdíl v implicitních přístup. právech k členům třídy – viz dále)
  - kromě datových členů rovněž funkce (=tzv. metody), popř. operátory, které manipulují s datovými členy třídy:

```
struct Complex
{
float re; float im;
void print() {printf("%f,%f",re,im);}
};
```

  - vytvoření instance – stejně jako u struktury nebo jednoduchých typů

```
Complex a,b;
```

  - Složitější metody – většinou definovány odděleně
  - operátor příslušnosti `::` - pro definování oddělených metod

```
struct Complex
{
float re; float im;
void print();
};

…

void Complex::print()
{  printf("%f,%f",re,im); }
```

  - přístup k datovým členům třídy – jako u struktur (ale pouze, pokud jsou přístupné – viz dále)
  - volání metod třídy:

```
Complex a;  a.print();
```

- většinou oddělené umístění :
  - soubor `.h` - jen deklarace třídy, vkládá se do jiných modulů pomocí `#include`
  - soubor `.cpp` – definice metod třídy

- metody definované v těle funkce – překládány jako `inline`
- ukazatel `*this`

- ukazuje v definici metody na aktuální instanci
- je implicitní, tj. není ho nutné psát – bylo by správně také :

```
void Complex::print()
{  this->printf("%f,%f",re,im); }
```

- operátor :: se rovněž používá pro vyvolání globální funkce (jejíž jméno a parametry by mohly kolidovat s některou metodou)

  `::print()` …vyvolá globálně definovanou funkci print()

  - definice operátoru jako metody třídy:

```
struct Complex
{
   float re; float im;
   Complex operator+(Complex &a);
};

Complex Complex::operator+(Complex &a)
{
   Complex c;
   c.re=re+a.re;
   c.im=im+a.im;
   return c;
};
```

- u operátoru se 2 arg. je prvním argumentem `*this`, druhý arg. se předává jako parametr

- konstruktor
  - speciální metoda vyvolaná automaticky při vytvoření instance
  - má stejné jméno jako třída, nevrací žádnou hodnotu
  - standardní (implicitní) konstruktor nedělá nic

```
struct Complex
{
   float re; float im;
   Complex() {re=im=0;}
};
```

  - může mít argumenty
  - může být více přetížených konstruktorů pro různé typy argumentů:

```
struct Complex
{
   float re; float im;
   Complex() {re=im=0;}
```

```
   Complex(float r, float i) {re=r; im=i;}
};

Complex u;          //-> vyvolá první konstruktor
Complex v(1,2);    //-> vyvolá druhý konstruktor
```

- copy konstruktor – speciální konstruktor pro zkopírování obsahu instance do jiné instance
    - je automaticky předdefinovaný
        - provádí zkopírování obsahu paměti
    - lze předefinovat – nutné větš. pouze pro objekty s dynamicky alokovanými dat. členy

```
struct Complex
{
   float re; float im;
   Complex() {re=im=0;}
   Complex(Complex &a) {re=a.re; im=a.im;}
};

Complex b;
Complex a(b);
```

- konstruktor s parametrem lze využít pro definování konverze mezi dvěma typy objektů
  - automaticky se vyvolá pro konverzi objektů, zejména při použití operátorů

- destruktor
  - speciální metoda automaticky vyvolaná před destrukcí objektu
  - nemá žádné parametry, nevrací žádný typ
  - většinou se definuje pouze u tříd s dynamicky alokovanými datovými členy

```
struct Complex
{
   float re; float im;
   Complex() {re=im=0;}
   ~Complex() {}
};
```

- copy konstruktor – použití pro typové přizpůsobení:

```
Complex a,b;
a=b+2.0  // pokud je definován copy konstr. complex(&double)
```

- operátor =
  - podobně jako copy-konstruktor je implicitně definován (= mezi strukturami existuje i v C)

- je ho ale možné předefinovat – je třeba pokud třída obsahuje dynamicky alokovaná data

```
struct Complex
{
   …
   Complex &operator=(Complex &c) {re=c.re; im=c.im;}
};
```

- na rozdíl od copy-konstruktoru je třeba dealokovat původní data objektu
- pozn: při inicializaci objektu pomocí = se volá copy-konstruktor a nikoliv operátor =

```
Complex a;
Complex b=a;   //volá copy-konstruktor, totez co Complex b(a);
   …
b=a;    //volá operator=
```

- operátor pro přetypování:

```
Complex::operator double()  // nepíše se návratová hodnota
```

- vnořené třídy
  - lze vnořovat stejně jako struktury
  - pro oddělenou definici metod se použije vícenásobně operátor ::

```
struct A
{
   struct B {  void m(); }
};

void A::B::m() {…}
```

- výčtové členy třídy
  - zvnějšku se na ně lze odkazovat nezávisle na instanci pomocí operátoru příslušnosti `::`

- statické datové členy
  - společné proměnné pro všechny instance třídy
  - klíčové slovo `static`
  - musí být globálně definovány v některém modulu
  - lze k nim přistupovat bez konkrétní instance, pomocí operátoru `::`

př.: třída s počítadlem instancí

```
/* Complex.h */
```

```
struct Complex
{
   float re; float im;
   static int cnt;          //počítadlo instancí - static.

   Complex() {re=im=0; ++cnt;}
   ~Complex() {--cnt;}
};

/* Complex.cpp */

int Complex::cnt=0;     //nutné - definice static. atributu
…

Complex a;

int n=a.cnt;            //lze obojí
int n=Complex::cnt;
```

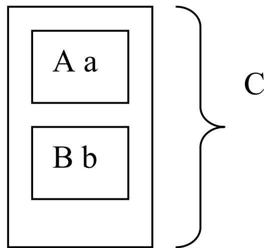
- statické metody třídy
  - pracují pouze se statickými členy
  - klíč. slovo `static`

- metody s označením `const`
  - nemodifikují atributy

```
struct C
{
   int a;
   void print() const {}
};
```

Kompozice

- princip využití a rozšíření existujícího kódu
- kompozice: třída C obsahuje atributy, které jsou instancemi tříd A, B

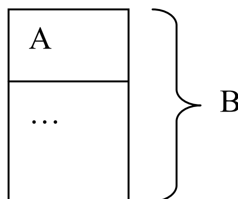


- vlastně definice jednoduché třídy je kompozicí objektů element. typů
- konstruktory vnořených objektů při kompozici:
  - píšou se za hlavičku konstruktoru nadřazeného objektu, oddělené ,
    - pořadí volání konstruktorů je dáno pořadím dat. členů v deklaraci třídy
  - pokud nejsou uvedeny, volá se jejich konstruktor bez parametru

```
struct Complex
{
      float re; float im;
      Complex() : re(0), im(0)
      {}
};
```

Dědění (odvozování)

- třída B obsahuje všechny atributy a metody třídy A a přibírá některé další (popř. předefinuje metody)




```
struct A
{
      int x,y;
      void print() { printf("%d,%d",x,y); }
};

struct B: A
```

```
{   int z;   };

A a; B b;

b.x=1;b.z=2;
b.print();
```

- přebírá všechno kromě konstruktoru, destruktoru, operátoru =
- konstruktory při dědění
  - implicitně se před vykonáním kódu nejprve vyvolá konstruktor bez parametrů rodičovské třídy
  - je ale možné místo toho vyvolat konstruktor s parametry rodič. třídy
    - uvede se za hlavičkou konstruktoru odvozené třídy (podobně jako při kompozici)

- konverze pointerů:
  - pointer na odvozenou třídu je současně pointerem na předka (probíhá automatická konverze)
    - protože odvozená třída začíná na stejné adrese a má větší délku
    - ale ne naopak
    - explicitní konverze lze použít vždy

```
A *pa=&a;
B *pb=&b;

pa=pb;
```

ale ne: `pb=pa`
šlo by: `pb=(B*)pa;`

- `pb=(B*)pa` … explicitní konverze – zde se předpokládá, že programátor ví, co dělá

- zakrývání jmen při dědění
  - pokud se v odvozené třídě objeví symbol, který už byl definován v rodičovské třídě, je význam zakryt symbolem definovaným v nové třídě
  - je ale stále možné přistupovat i k zakrytým členům pomocí operátoru `::`

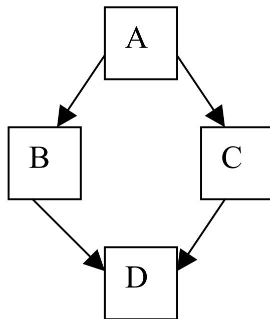
```
struct A
{
   int x,y;
   void print() { printf("%d,%d",x,y); }
};

struct B: A
{
   int z;
   void print()
```

```
   { A::print(); printf("%d\n",z );}   //využití metody print()
                                        // rodičovské třídy
};
```

- vícenásobná dědičnost, virtuální dědičnost
  - rys specifický pro C++
  - třída může být odvozena od více tříd

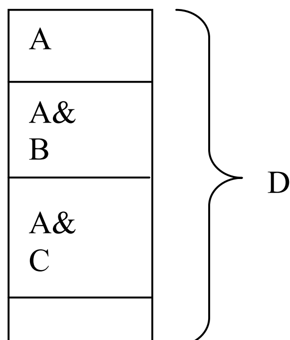
```
struct D: B,C
{
     ...
};
```

  - problém vzniká, pokud třídy B,C jsou odvozeny od společného předka
    - v instanci D by byla instance A dvakrát, nejednoznačnost při překladu



  - aby se toto nestalo – je třeba použít tzv. virtuální dědění od bázové třídy A
    - překladač umístí instanci A do instance D pouze jednou
    - lze vyložit tak, překladač umístí do tříd B,C reference na datové členy A

```
struct A {...};

struct B: virtual A {...};
struct C: virtual A {...};

struct D: B,C  {... };
```



Přístupová práva k členům třídy

- v praxi je někdy výhodné zabránit přímé manipulaci s některými datovými členy tříd (na úrovni zdroj. kódu)
  - vhodné zejm. při vytváření knihoven a sdílených částí kódu
  - rozhraní tříd je v ideálním případě definováno výhradně pomocí metod, datové členy jsou zvnějšku nepřístupné – nelze je přímo měnit

- klíčová slova – lze libovolněkrát za sebou opakovat v libovol. pořadí
  - `public`
    - předchází deklaraci veřejně přístupných symbolů
    - implicitní při deklaraci třídy klíčovým slovem „`struct`"
  - `private`
    - předchází deklaraci veřejně nepřístupných symbolů, nejsou přístupné ani v odvozené třídě
    - implicitní při deklaraci třídy klíčovým slovem „`class`"
  - `protected`
    - předchází deklaraci veřejně nepřístupných symbolů, které jsou však přístupné v odvozené třídě

```
struct A
{
      private:
      int x,y;
public:
      void print() { printf("%d,%d",x,y); }
};
```

- při deklaraci pomocí `class` je `private` implicitní:

```
struct A
{
      int x,y;
      public:
      void print() { printf("%d,%d",x,y); }
};
```

- při dědění je možné ovlivnit přístup. práva k členům rodičovské třídy
  - přístupová úroveň specifikovaná u rodičovské třídy udává nejvyšší úroveň, jakou členy rodič. třídy v odvozené třídě mohou mít

```
class A {};
class B: public A  {};     //zde zůstane přístup. úroveň členů A beze změny
class B: protected A  {};   //zde budou public členy A změněny na protected
```

- pokud úroveň přístup. práv k rodič. třídě při dědění není specifikována, je:

- u struct - `public`
- u class - `private`

Friend funkce

- V některých výjimečných případech je třeba, aby třída umožnila přístup některé konkrétní funkci (nebo třídě), aby mohla pracovat s jejími soukromými (private nebo protected) členy
- v definici třídy je možné deklarovat tzv. spřátelenou funkci (resp. třídu) pomocí klíčového slova `friend` (resp. `friend class`)

```
class C
{
	private:
	int a, b;
	friend int sumC(C& c);
};

int sumC(C& c)
{
	return  c.a+c.b;
}
```

- V tomto příkladu ale není vhodné takto zjišťovat součet členů třídy C – bylo by přirozené sum() definovat místo toho jako metodu třídy C.

Ukazatele do tříd

- bežný ukazatel lze použít pro data, nelze ale použít pro metody

```
int *p=&C.x;
```

- lze definovat tzv. ukazatele do tříd, které nejsou svázány s konkrétní instancí

```
int C::*p;
p=&C::x;       //nepotřebuje instanci, ale jen offset
```

- získání/modifikace dat – musíme dodat instanci:

```
C obj, *pobj;
obj.*p=5;
pobj->*p=10;
```

- lze použít i pro volání metod
  - *., ->. se chápe jako samostatný operátor, který má nižší prioritu než ()
  - proto vyvolání metody : `(obj.*p)()` – nutno dát do závorek

Dynamická vazba

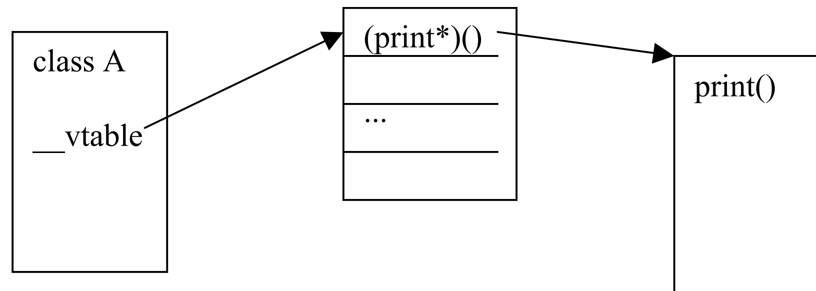
- statická vazba volání metod – definovaná pevně už při překladu:

```
struct A
{     int x,y;
      void print() { printf("%d,%d",x,y); }
};

struct B: A
{   int z;
void print()
{ A::print(); printf("%d\n",z );}   //využití metody print()
                                     //rodičovské třídy
};

A *p;
B b;
..
p=&b;    //lze, protože b je současně objektem A
p->print();  //vyvolá se A::print(), i když p ukazuje vlastně na
              //objekt třídy B
```

- dynamická vazba – je třeba zaručit, aby se vyvolala metoda skutečné třídy instance, na kterou p ukazuje
  - lze řešit pomocí tzv. virtuálních metod
  - klíčové slovo `virtual`
    - nutné pouze v deklaraci bázové třídy
    - v odvozených třídách být už nemusí
  - virtuální metody v odvozených třídách se musí shodovat co do počtu a typu argumentů i návratové hodnoty s virt. metodami bázové třídy

```
struct A
{     int x,y;
      virtual void print() { printf("%d,%d",x,y); }
};

struct B: A
{   int z;
virtual void print()    //zde být virtual už nemusí - ale může
{ A::print(); printf("%d\n",z );}
};

A *p;
B b;
..
p=&b;    //lze, protože b je současně objektem A
p->print();  //vyvolá se B::print() - dynamická vazba
```

- mechanismus dynamické vazby implementován tak, že každá instance třídy obsahuje skrytý ukazatel na tabulku pointerů na metody. Pointery v tabulce jsou nastaveny při vytvoření instance. Takže volání virtuální metody vždy prochází přes 2 pointery (vyšší režie než u normální metody):



- *polymorfismus*
  - práce s instancemi různých tříd (předem neznámých) jednotným způsobem
  - lze implementovat s využitím virtuálních metod za předpokladu, že všechny třídy dotyčných objektů jsou odvozeny od společné rodičovské třídy (ale i přes více generací)

- příklad - dynamický seznam, který nevlastní prvky (obsahuje pouze pointery)

```
class Node   //bázová třída pro všechny objekty vkládané do
              //seznamu
{
      Node *pnext;
      virtual void write(){};
      Node() {pnext=NULL;}
};

class List   //seznam
{
      Node *phead;

      List(){phead=NULL;}
      ~List();
      void addHead(Node &n);
      Node &popHead();
      void write();
};

class MyData : public Node    //uživatelská třída, odvozená od
                              //Node
{
      int i;
      MyData() {i=0;}
      MyData(int p) {i=p;}
```

```
	void write() { printf("%d\n",i);}
};

void main()   //hlavní program
{
	MyData d1(1),d2(2),d3(3);
	List list;

	list.add(d1);list.add(d2);list.add(d3);
	list.popHead().write();
	list.popHead().write();
	list.write();
}
```

- pozn.: nutnost provést přetypování, když chceme vyjmout prvek ze seznamu a normálně ho používat (přitom se předpokládá, že známe skutečný typ):

```
MyData &r=(Mydata&)popHead();
r.i=10;
```

- čistá virtuální funkce
    - virtuální funkce, která není v bázové třídě definována a musí být definována v každé odvozené třídě
    - tj. programátor je donucen tuto funkci definovat
    - deklaruje se pomocí =0 za jménem metody

```
class Node
{
	Node *pnext;
	virtual void write()=0;   //čistá virtuální funkce –
//programátor musí v odvoz. třídě předefinovat

	Node() {pnext=NULL;}
};
```

    - pokud třída obsahuje nějakou čistou virt. fci, nelze vytvořit její instanci (lze pouze od ní něco odvodit) =tzv. abstraktní třída

- virtuální destruktor
    - destruktor může být a vlastně by měl být vždy virtuální
    - pokud není, může se stát, že operátor `delete` aplikovaný na pointer na bázovou třídu nevyvolá správný destruktor

- Rozšíření předcházejícího příkladu: seznam, který vlastní celé prvky (ne jen pointery)
    - tj. prvky sám alokuje v dyn. paměti
    - oproti předchozímu výhoda, že seznam své prvky likviduje sám a nemůže se stát, že to udělá někdo jiný „bez jeho vědomí“
    - každý objekt musí předefinovat metodu `dynamicCopy()`, která ho „naklonuje“

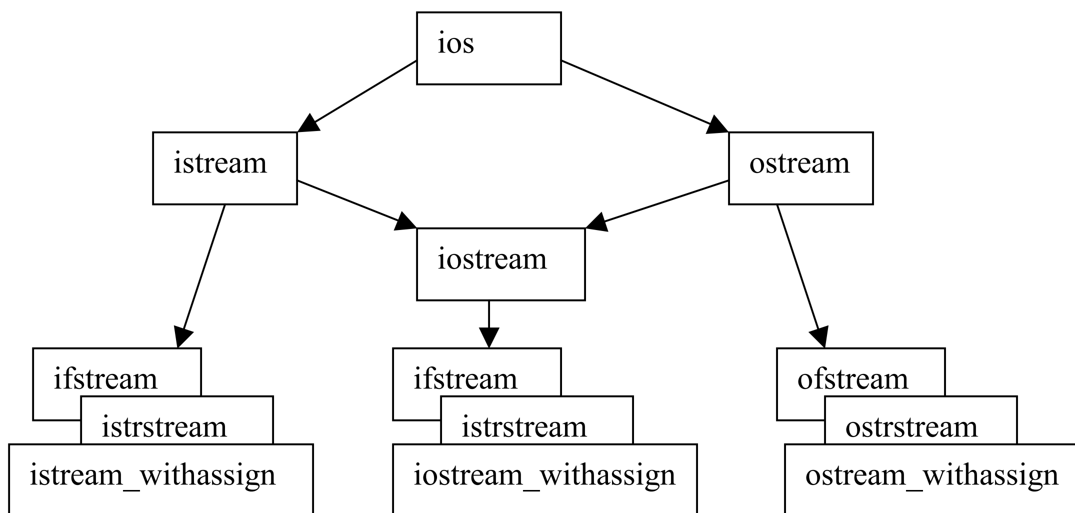
```
struct Node
{
      Node *pnext;
      Node() {pnext=NULL;}
      virtual ~Node()
      virtual Node* dynamicCopy()=0;
};

struct Bod
{
      float x,y;
      Node* dynamicCopy() { return new Bod(*this); }
};

struct Bod3 : Bod
{
      float z;
      Node* dynamicCopy() { return new Bod3(*this); }
};

class List   //seznam
{
      Node *phead;
public:
      List(){phead=NULL;}
      ~List();
      void addHead(Node &n);
      void removeHead();
      void write();
};

List::addHead(Node &n)
{
      Node *pn=n.dynamicCopy();
      pn->next=phead;
      phead=pn;
}

void List::removeHead()
{
      assert(phead!=NULL);  //pokud není splněno, program
                            //skončí a nahlásí č. řádku
      Node *pn=phead;
      phead=phead->pnext;
      --count;
      delete pn;
}
```

```
List::~List()
{
     Node *pn=phead;
     while(phead) removeHead();
}
```

standardní V/V v C++

- prostředky V/V běžně používané v C lze v C++ beze změny používat
- snaha o elegantnější řešení s využitím možností C++

- objektová knihovna proudů v C++
    - definuje několik tříd, které nabízí metody a operátory pro V/V



- `ios` – společná bázová třída
    - zahrnuje společné příznaky pro práci s proudy (způsob formátování, chybové příznaky) – většinou se testují pomocí metod:
    - stav – `enum io_state {goodbit,eofbit,failbit,badbit,hardbit}`
    - zjištění stavu – `bad(), eof(), fail(), good(), operator !, (void*)`
    - formátování :
      `x_flags -enum {skipws,left,right,internal, dec,oct,hex, showbase, showpoint,uppercase,showpos, scientific,fixed,unitbuf,stdio} -> metoda int flags(newflags)`
      `x_precision,x_width,x_fill -> int width(int)`

- `ostream, iostream` – obecné třídy pro V/V (<iostream.h>)
    - používají se zejm. přetížené operátory pro vstup a výstup <<, >>, které jsou přetížené pro elementární datové typy:
    - `istream`
        - `istream &istream::operator >> (Typ&);`

  - `get(char&);  //přečte 1 znak`
  - `get(char*,int,char), getline(char*,int,char),  // čte po řádcích`
  - `read()  //- pro binární, blok.V/V`
  - `ignore(count,delim=EOF),  //přeskočí znaky`
  - `peek(),  //otestuje znak`
  - `eatwhite()  //přeskočí mezery`
  - `tellg(), seekg()   // získá/nastaví pozici v souboru`
  - `ostream`
  - `ostream &ostream::operator>> (Typ&);`
  - `put() - zapíše 1byte`
  - `write() - pro binár. blokový V/V`
  - `flush() - vyprázdní buffer`
  - `tellp(), seekp()   // získá/nastaví pozici v souboru`

- `fstream, ifstream, ofstream` (<fstream.h>)
  - V/V z/do souboru (většinou textový, ale lze i binární)
  - `fstream`:
    - konstuktor `fstream(char*,int mode);`
    - `mode = os::in,out,ate,app,trunc,nocreate,noreplace,binary` – lze kombinovat pomocí |
    - bez konstruktoru – metoda `open(..)`- stejné param. jako konstr.,
    - `close()` – zavře, automaticky zavře rovněž destruktor
- `ofstream(const char* szName, int nMode = ios::out);` // standardně pro výstup
- `istream(const char* szName, int nMode = ios::out);` // standardně pro vstup

- `strstream, istrstream, ostrstream` (<strstream.h>)
  - V/V z/do řetězce – pro zpracování textu
  - `strstream(char* pole, int len, int mode);`
    - len >0 ..delka, len =0 ..řetězec zakončený /0, len<0 ..nekonečná délka
    - `mode= in, out, ate=app`

- `istream_withassign, ostream_withassign`
  - mají definovaný operátor = , který umožňuje přesměrování (zejm. pro standardní V/V – cin, cout, cerr)

- standardně otevřené proudy:
  - `cin` - std. vstup (`istream_withassign`)
  - `cout` – std. výstup (`ostream_withassign`)

  např.:

```
int a=5; float x;
```

```
cout << ″text=″ << a << ″,″ << x << ″\n″;
```

- operátory je možné zřetězit, díky tomu že `ostream::operator>>(Typ&)` je definován tak, že vrací `ostream&`.

- parametry mohou být různého typu – operátor je přetížený pro elementární typy. Je však možné jej globálně přetížit i pro uživatelské typy:

např.:

```
ostream &operator<<(ostream &os, Complex &c)
{
   os << x << ″,″ <<y;
   return os;
}

Complex z;

cout << z;
```

- čtení ze souboru:

```
ifstream  is(″soubor.txt″);
while(is.good())
{
   int a;
   is >> a;
   cout << a <<endl;   //endl je manipulátor – viz dále
}
```

- pro formátování výstupu lze použít a definovat tzv. manipulátory

  např.: `cout << hex << a <<endl;` //vypíše hex. hodnotu a, ukončí řádek a //vyprázdní buffer

- standardně definované manipulátory:

- `dec, hex,oct   // nastaví číselnou soustavu`
- `setbase(n)       // nastaví číselnou soustavu`
- `endl                // endline+flush`
- `ends    // /0 +flush`
- `flush   //vyprázdní buffer`
- `setiosflags(n), resetiosflags(n)   //nastaví příznaky ios`
- `setfill(n), setprecision(n)    //nastaví znak pro výplň mezer a přesnost`
- `setw(n)   // šířka pole (minimální počet znaků, které vystoupí)`

- `ws  // přeskoč mezery`

- Implementace manipulátorů
  - buď jako funkce, které vrací `ostream&`
    - `ostream& fun(ostream&);`
    - je přetížen operátor << pro tento typ v <istream.h>
  - s více parametry – manipulátor je objekt, který má konstruktor s parametrem ostream a má vlastní přetížený operator <<

Genericita

- mechanismus umožňující pracovat s třídami podobně jako s proměnnými
  - vzhledem k tomu, že v C++ je třída strukturou, tj. datovým typem, je implementace poněkud atypická a s určitými omezeními oproti některým „čistým“ objektově orientovaným jazykům
  - řešení spočívá ve využití tzv. šablon
- genericita umožňuje vytvářet algoritmy a třídy pracující s předem neznámými typy objektů, které nemusí být odvozeny od společné rodičovské třídy

- šablony
  - podobně jako makra v C existují pouze na úrovni zdrojového souboru

Šablony funkcí

  - používají se tam, kde je třeba definovat třídu obdobných funkcí pro různé typy argumentů, které však mají stejný zápis kódu

např.:

```
double sqr(double x) {return x*x;}
int  sqr(int x) {return x*x;}
short sqr(short x) {return x*x;}
```

- třebaže zápis kódu je stejný, jedná se o zcela rozdílné funkce. V tomto případě by bylo možné použít pouze double sqr(double x) – pro ostatní typy argumentů by se provedly konverze, ale výpočet by např. pro int trval mnohem déle než u varianty int sqr(int x).

- v C++ lze elegantně řešit pomocí šablon funkcí:

```
template <class T> T sqr(T x) {return x*x;}

double x; int k;
Complex c;

sqr(x);  sqr(k);
sqr(c);
```

- v každém případě se vykoná jiná funkce. U `sqr(c)` se předpokládá, že `Complex` má definovaný operátor `*`.
- pokud se objeví volání šablony, vygeneruje se na daném místě pro daný typ argumentu příslušná funkce (jestliže už nebyla vygenerovaná)
- podobně jako makra jsou šablony většinou v hlavičkových souborech
- místo klíč. slova `class` lze `typename`

- parametry šablony funkce
  - pouze typové, lze použít i implicitní argumenty

- implicitní generování instance
  - šablona automaticky generuje danou funkci pro typy argumentů při volání fce
  - zde musí přesně odpovídat typy argumentů, neprovádí se žádné konverze
  - např. `y=sqr(x)`

- explicitní generování instance
  - někdy je výhodné generovat šablonu po konkrétní typ, který je jiný, než typ argumentu a využít konverze typů
  - např.: `y=sqr<int>(x)` -zde se vygeneruje instance pro `int` a provede se konverze x na `int` – je-li možná
  - lze dopředu explicitně generovat instanci : `template double sqr(double);` - vytvoří se, i když se nepoužije

- zastínění šablony
  - lze napsat normální funkci, která má parametry odpovídající šabloně – zastíní pak generování šablony pro konkrétní typ argumentu a použije se jako instance
  - umožňuje použít šablonu, i když pro některý typ se zápis funkce odlišuje od ostatních

Šablony tříd

```
template <class T,int delka=100>
class Vector
{
      T *pData[delka];
      T& operator[](int i) {return pData[i];}
};

typedef Vector<float, 20> Flt_vect;
flt_vect v1;
Vector<int> v2;
…
v2[8]=1;
```

- u šablon tříd mohou být typové parametry i hodnotové parametry
- hlavička šablony se přesně opakuje i při oddělené definici metod

```
template <class T,int delka=100>
T& min() { T m; … return  m;}
```

- lze zastínit celou třídu pro daný typ nebo i jednotlivou metodu
- explicitní generování : `template class Vector<int,2>`

- statické atributy šablony
    - `template<class R> R vektor<R>::pocet=0;` …vlastně samostatná šablona statického atributu
    - dále třeba explicitně vytvořit instance (aby někde existovaly): `vektor<int> r; vector<float> r;`
    - také je možné při generování šablony zvlášť dopsat: `double vektor<double>::pocet=0;`

přes šablony – lze rovněž definovat i `friend` funkce

Výjimky

- často je třeba ošetřit chybové situace v programu, aniž by se program ukončil
- chyby v univerzálních knihovnách – nutno informovat a současně přinutit uživatele něco s tím dělat
- možnosti:
    - systematické testování, zda došlo k chybě – velmi obtížné, nepřehledné
    - možnost vracet speciální návratovou hodnotu – nelze ale použít vždy (např. `operator[]` nemůže takovou hodnotu vracet)
    - zápis chybové hodnoty do globální proměnné
        - ale jak přinutit uživetele, zby stav zjistil?
    - využití jazyka s podporou výjimek
        - moderní přístup

- Výjimky
    - Synchronní – generovány v programu, v C++ pouze
    - Asynchronní- vzniknou v OS – např. stisk Ctrl+break, dělení 0 atd.
        - Lze použít tzv. Strukturované výjimky v Microsoft C/C++

- Hlídaný (pokusný) blok
    - blok kódu, ve kterém se může vyskytnout výjimka
      `try {  …  }`
- Handler – blok pro ošetření výjimky
    - za blokem `try{}` následuje 1 nebo více handlerů `catch(typ) výraz;`

    - Jestliže nastane výjimka uvnitř pokusného bloku, ukončí se pokusný blok (+vyvolají se destruktory všech objektů lokálních v `try{}`) a zavolá se `catch(typ)`, který odpovídá typu výjimky, pak se normálně pokračuje za posledním `catch()` tohoto hlídaného bloku (hlídaných bloků může být ve funkci víc).
    - Jestliže `catch()` pro daný typ neexistuje v dané funkci, vrátí se do funkce, která ji vyvolala a hledá tam, atd. – nenajde-li, pokračuje výš, jestliže se takto dostane až nad úroveň `main()`, program se ukončí = tzv. neošetřená výjimka
        - vyvolá se fce `terminate()` - i ta lze předefinovat

    - Vyvolání výjimky
        - `throw(objekt)` – objekt je třída nebo jednoduchý typ

- Jestliže v `catch` je uveden argument, ne jen typ, lze hodnotu využít jako u funkce
  `catch(typ proměnná)`

- Univerzální handler : catch(...)
    - zpracuje všechny nedefinované typy výjimek

- Konverze typů v handleru

  - neprovádí se, až na konverzi objektu na předka a standardní konverzi ukazatelů na objekty

- Jestliže výjimka vznikne ve funkci, vyvolají se destruktory lokál. objektů. Jestliže vznikne v konstruktoru, destruktory se nevolají. Z destruktoru se nesmí šířit výjimka. – v tom případě se vyvolá `terminate()`

- V handleru `catch()` lze výjimku poslat dál (výš) příkazem `throw` bez parametru – vhodné, když má být v dané fci výjimka ošetřena jen částečně

Př.:

```
main()
{
      try
      {
            funkce1();
            funkce2();
            funkce3();
      }
      catch(double) { cout << "vyjimka double"; }
      catch(Vyjimka vyj)  { exit(1);}
      …

}
```

- operátor `new` – standardně hází výjimku typu `bad_alloc`, pokud nelze alokovat paměť
  - vetšinou lze předefinovat

Dynamická identifikace typů (RTTI)

- operátor `typeid` – třeba jen výjimečně
    - smyl má jen v případě, že není k dispozici virtuální metoda pro daný typ
      `type_info typeid(výraz nebo jméno typu, třídy)`
    - objekt `type_info` má:
        - přetížené operátory `==`, `!=`
        - metoda `name()` – označení
        - metoda `before()` – pro lexikální porovnání typů

- zpravidla nutno povolit v kompilátoru, jinak lze typeid stále použít, ale jen staticky
    - RTTI vyžaduje přídavnou režii, proto standardně není zapnuto

- někdy existuje klíčové slovo (např. `_rtti`), používá se před deklarací třídy, která má používat rtti.

Operátory pro bezpečnější přetypování

- Zavedeny hlavně z důvodu zvýšení přehlednosti, protože přetypování se v C++ často používá i ve zcela standardních situacích

- Klasický operátor přetypování `(typ)` zachován
    - umí všechno až na dynamické přetypování v případě virtuál. dědění, dynamickou kontrolu typu a (asi) převod `const` a `volatile`.

- `T dynamic_cast<T> (V)`
    - V musí být ukazatel na plně definovaný objektový typ, nebo reference
        - T může být v případě ukazatele `void*` - musí být ale polymorfní typ
    - Používá se pro legální přetypování polymorfních typů, používá rtti
    - Pro reference, resp. ukazatele
    - Převod z potomka na předka a naopak,
    - funguje i v případě virtuální dědičnosti – na to využívá rtti (jinak nelze)
        - pozná, jestli ukazuje skutečně na podobjekt nebo ne
    - přitom kontroluje, jestli převod má smysl (jestli ukazatel skutečně ukazuje na správný typ) – jinak se neprovede
    - u virtuální dědičnosti lze i přetypovat z jednoho podobjektu na druhý(!).
        - pokud ale není v rámci jednoho objektu, neprovede se
    - Při přetypovani nahoru se musí jednat o polymorfní typ (musí mít alesp. 1 virtualní fci)
    - Neuspěšné přetypování ukazatelů – vrátí `NULL`
    - Neuspěšné přetypování referencí – hodí výjimku `bad_cast`

- `T static_cast<T>(V)`
    - Pro běžné konverze typů z předka na potomka a naopak, bez dynamické kontroly typů (tj. ne u virtuálního dědění)
    - Lze i pro ostatní běžné konverze, které lze provádět pomocí `(typ)`, kdy správnost lze jasně určit už při překladu

- `reinterpret_cast`
    - pro špinavou práci – převod ukazatele na číslo, převod ukazatele z jedné třídy na jinou nesouvisející třídu

- `const_cast<T>(V)`
    - `T` se od `V` liší jen slovem `const`, resp. `volatile`
    - pro převod const a volatile na normální
    - nekdy je např. potřeba v konstantní (zejm. virtuální metodě) ještě s objektem něco udělat - modifikovat

Prostory jmen

- pro oddělení jmen identifikátorů v různých knihovnách
- užitečné pro rozsáhlé programy

```
namespace jméno { seznam všech deklarací }
```

- symboly z namespace jsou pak přístupné zvnějšku pomocí jména
    - namespace :: jmeno symbolu – obdobně jako u tříd, např.

```
std::vect<int> v;
```

- `::`symbol – pro zastíněný symbol, který není uvnitř žádného namespace
- lze do sebe vnořovat , jako třídy

- vytvoření přezdívky

```
namespace alias=jmeno
```

- deklarace lze po částech:
    - `namespace A {…}`
    - `namespace B {…}`
    - `namespace A {…}`

- anonymní prostor jmen
    - chybí jméno
    - chová se jako statické proměnné
        - tj. jsou viditelné (bez udání jména) jen v daném modulu

- direktiva `using` :
    - `using namespace jméno;`
        - pak není nutné psát jmeno::symbol

```
using namespace std;

vect<int> v;
```