1. Úvod

www.uamt.feec.vutbr.cz/~richter
- starší materiály, KPPC
- cvičící a přednášející
- organizace přednášek a cvičení
- konzultace a dotazy
- hodnocení
- literatura

Sestavení programu v C/C++, preprocesor, makra

Opakování jazyka C
- funkce, parametry, typy (velikost, vlastnosti, použití), operace (bitové), ukazatele, pole, alokace paměti

2. Motivace C++

C++
- rozšiřuje programovací možnosti C
- přidává objektové vlastnosti
- C může mít některé vlastnosti navíc
- C přijímá některé vlastnosti z C++

Třída
- nový složený typ (nástupce struct)
- data a práce s nimi
- nejblíže k ní má knihovní celek z jazyka C – rozhraní, data, kód

Návrh třídy
- formulace (definice) problému – slovní popis
- rozbor problému
- návrh datové struktury

- návrh metod
- testování

Formulace problému
- co má třída dělat – obecně
- jak vzniká
- jak zaniká
- jak nastavujeme hodnoty
- jak vyčítáme hodnoty
- jak pracujeme s hodnotami (metody a operátory)
- vstup a výstup

Rozbor problému
- konzultace možných řešení, koncepce
- rozhodneme, zda je možné použít stávající třídu, zda je možné upravit stávající třídu (dědění), zda vytvoříme více tříd (buď výsledná třída bude obsahovat jinou třídu jako členská data, nebo vytvoříme hierarchii – připravíme základ, ze kterého se bude dědit – všichni potomci budou mít shodné vlastnosti). (Objekt je prvkem a objekt dědí z …)
- pohled uživatele (interface), pohled programátora (implementace)

Návrh datové struktury
- zvolí se data (proměnné a jejich typ) které bude obsahovat, může to být i jiná třída
- během dalšího návrhu nebo až při delší práci se může ukázat jako nevyhovující
- Data jsou (většinou) skrytá

Návrh metod
- metoda – funkce ve třídě pro práci s daty třídy
- metody vzniku a zániku
- metody pro práci s daty
- metody pro práci s objektem

- operátory
- vstupy a výstupy
- metody vzniklé implicitně (ošetřit dynamická data)
- zde se (hlavně) zužitkuje C - algoritmy

testování
- na správnost funkce
- kombinace volání
- práce s pamětí (dynamická data)
- vznik a zánik objektů (počet vzniků = počet zániků)
- testovací soubory pro automatické kontroly při změnách kódu

přístup k datům a metodám objektu
přístup k datům uvnitř metod - this
konstruktory
destruktory
operátory

3. Jazyk C++
– jazyk vyšší úrovně
– existuje norma
– objektově orientován
– přenositelný kód
– C je podmnožinou C++

Změny oproti C
- rozšíření možností programovacího jazyka
- objektové vlastnosti
- šablony
- nová klíčová slova

Neobjektové vlastnosti
- přetěžování funkcí a operátorů

- definice proměnné
- reference
- implicitní parametry
- prostory jmen
- typ bool
- alokace paměti (new, delete)
- typově orientovaný vstup výstup
- inline funkce
- šablony

Objektové vlastnosti
- objekty
- dědění
- výjimky

3.1 Objektové programování, základní definice
- nové možnosti programování – nový styl
- sdružování dat a metod (a operátorů) s nimi pracujících
- práva přístupu k datům
- zdrojový kód přípona ”cpp”. (hlavičkové soubory bez přípony nebo ”h”, ”hpp”, ”hxx”)
- výhody: tvorba knihoven, sdílení kódu, údržba programu

Základní pojmy
- ***třída (class)*** - datový celek (datová abstrakce), data + operace, přístupová práva
- ***instance*** - proměnná
- ***objekt*** - instance nějaké třídy
- ***metoda*** - funkce definovaná ve třídě pro práci s daty
- ***zapouzdření (encapsulation)*** – shrnutí logicky souvisejících (součástí programu) do jednoho celku (zde data, metody, přístupová práva) – nového datového typu. Někdy může splňovat i funkce. Někdy je tímto termínem označováno skrytí dat (private) a vytvoření interface

- ***konstruktor a destruktor*** - metody pro inicializaci a likvidaci objektu
- ***rozhraní (interface)*** - co třída nabízí ven
- ***implementace*** - jak to dělá třída uvnitř
- ***dědičnost (inheritance)*** - použití kódu pro podobnou třídu
- ***polymorfismus*** - třídy se stejným rozhraním, a různou implementací, jednotný přístup k instancím

3.2 komentáře (no)

- v C víceřádkové komentáře /* */
- vnořené komentáře (?)
- v C++ navíc jednořádkový komentář: // až po konec řádku
- // již i v C

```
int k;  // nový komentář
// komentář může začínat kdekoli,
int i ;   /* Starý typ  lze stále použít  */
```

3.3 pojem třídy a struktury v C++ (o)

- složený datový typ – jako struct
- prvkem jsou data a metody, přístupová práva
- objektové vlastnosti rozšířeny i pro struct a union
- klíčové slovo class
- deklarace třídy - pouze popis třídy – nevyhrazuje paměť

```
class  jméno_třídy  { parametry, tělo třídy };
struct jméno_struktury {parametry, tělo     };
```

- deklarace jména třídy – pak lze použít pouze adresu
- vlastní popis třídy (=tělo) později

```
class  jméno_třídy;
struct jméno_struktury;
```

- definice proměnné, vyhradí paměť

```
jméno_třídy a, b, *pc; //obdoba int a,b,*pc
// 2x objekt, 1x ukazatel
pc = &a;  // inicializace ukazatele
```

3.4 deklarace a definice proměnných (no)

- v C na začátku bloku programu
- v C++ v libovolném místě (deklarace je příkaz)
- deklarace ve for
- konec s koncem bloku
- deklarace musí mít extern
- snaha nevytvářet neinicializované proměnné

```
for (int i=0;i<10;i++)
 { /* zde je i známo, dd neznámo */
   …
   … // libovolný kod
   double dd=j; // definice s inicializací
   …
 }  // zde končí i a dd
```

3.5 data, metody - práce s nimi (o)

- datové členy – jakýkoli známý typ (jinak ukazatel)
- metody – funkce patřící ke třídě
- členská data a metody
- přístupová práva

```
// deklarace třídy
class Jmeno_třídy { // implicitně private:
   int data1; //datové členy třídy
   float data2;
   Jmeno *j;
   char string[100];
public: // metody třídy
   int metoda1() {…return 2;}
```

```
  void metoda2(int a,float b) {…}
  float metoda3( int  a1,  Jmeno *a2);
 int bb;//nevhodné, veřejně přístupná proměnná
 };

Jmeno_třídy aa, *bb = &aa;
int b = aa.metoda3(34,”34.54”);
int c = bb->metoda3(34,”34.54”);
// obdoba struct pom aa, *bb = &aa;
// aa.i = bb->ii;
```

3.6 přístupová práva (o)

- klíčová slova – private, public, protected
- třída x uživatel, veřejná x privátní,
- přepínače značící začátek práv
- možno vkládat libovolně
- rozdíl mezi class a struct – implicitní přístupové právo

```
struct Komplex { // public: - implicitní
double Re, Im; // public
private:  // přepínač přístupových práv

double Velikost(void) {return 14;}
// metoda (funkce) interní
int pom;  // interní-privátní proměnná

public: // přepínač přístupových práv
// metoda veřejná = interface
double Uhel(double a ) {return a-2;}
};

Komplex a,b;
a.Re = 1; // je možné
b.Uhel(3.14); // je možné
```

```
a.pom = 3; // není možné
b.Velikost(); // není možné


class {           int i   je ekvivalentní
class {private:   int i  …
struct{           int i   je ekvivalentní
struct{public:    int i …
```

3.6 reference (no)

- v C hodnotou (přímou nebo hodnotou ukazatele, pole)
- v C++ reference – odkaz (alias, přezdívka, nové jméno pro stávající proměnnou)
- zápis **Typ&** a musí být inicializován
-

```
T tt, &ref=tt; //  definice s inicializací
extern T &ref; //  deklarace

Typ& pom = p1.p2.p3.p4; // lepší přístup
// zjednodušení zápisu

double Real(T &r) //předání do funkce
{r = 4; return r;}
// r je nové jméno pro volající proměnnou =
// dělí se dvě jména o společné místo v paměti
// dojde k přiřazení do původní proměnné
//možné změny vně,úspora proti volání hodnotou
double ab; Real(ab); // způsob volání
```

- ”splývá” předávání hodnotou a referencí (až na prototyp stejné)
- práce s referencí = práce s původní odkazovanou proměnnou
- nelze reference na referenci, na bitová pole,
- nelze pole referencí, ukazatel na referenci

- vracení parametrů odkazem

```
double& Funkce(double &p1, double *p2) {
double aa;
p1 = 3.14; // pracujem jako s proměnnou
// return aa; // nelze - aa neexistuje vně
if (p1 > *p2)
 return p1; // lze – existuje vně
else
 return *p2; // lze – existuje vně
//vrací se ”hodnota”,referenci udělá překladač
// odkazujem se na proměnnou vně Funkce
}

double bb,cc ; //ukázka volání
dd = Funkce (bb,&cc); // návratem funkce je
// odkaz na proměnnou, s ní se dále pracuje
Funkce(bb,&cc) = dd; // vrací odkaz
```

U ukazatele je jasně vidět z přístupu, že je to ukazatel (& a *)
U reference je předávání a práce jako u hodnoty, liší se pouze v hlavičce

3.7 this (o)

- v každé instanci – ukazatel na aktuální prvek (zajistí překladač)
- `T* const this;`
- klíčové slovo
- předán implicitně do každé metody (skrytý parametr-překladač)

používá se:

- přístup k datům a metodám aktuálního objektu (this je možné vynechat) `this->data = 5, b = this->metoda( a )`
- objekt vrací sám sebe – `return *this`;

- kontrola parametru s aktuálním prvkem **`if (this==&param)`**

- metoda, která vrací maximum

```
class Komplex {
double Re;
public:
Komplex& Max(Komplex &param) // & reference
{ // this je předán implicitně při překladu
// Komplex& Max(Komplex*const this,Komplex &p…
// rozdíl funkce x metoda
if (this == &param) // & adresa
    return *this;// oba parametry totožné –
// nepočítám, rychlý návrat.
if ( this->Re < param.Re ) return param;
else                       return *thís;
//  param i this existují vně -> lze reference
}
};
```

volání:

```
Komplex a,b, c ;
c = a.Max(b); // neboli Max(&a,b)překladem
// -> a se uvnitř metody mění v this
c = b.Max(b); // -> c = b; this je &b
```

alternativní hlavičky - rozdíly při předávání – dočasné proměnné

```
Komplex  Max(Komplex  param)
Komplex & Max(Komplex const &param)
c = a.Max(b);
```

3.8 operátor příslušnosti :: (no)

- odlišení datových prostorů (a tříd)
- přístup ke stejnojmenným globálním proměnným

```
Prostor::JménoProměnné
Prostor::JménoFunkce

float Stav;
fce() {
  int Stav;
  Stav = 5;
  ::Stav = 6.5; //přístup ke ”globální” proměnné
// globální prostor je nepojmenován
}

// při uvádění metody vně třídy
// (bez vazby na objekt dané třídy)
// nutno uvést
int Komplex::metoda(int, int) {}
//při psaní metody u proměnné se odvodí
// z kontextu
Třída a;
a.Metoda(5,6); // Metoda musí patřit ke Třída

Struct A {static float aaa;};
Struct B {static float aaa;};
A::aaa        // proměnná aaa ze struktury A
B::aaa        // proměnná aaa ze struktury B
```

3.9 statický datový člen třídy (o)

- vytváří se pouze jeden na třídu, společný všem objektům třídy

- např. počítání aktivních objektů třídy, zabránění vícenásobné inicializaci, zabránění vícenásobnému výskytu objektu …
- v deklaraci třídy označen jako static

```
class string {
   static int pocet;
// deklarace  nevytváří paměť
}
```

- nemusí být žádný objekt třídy
- vytvoří se jako globální proměnná (nutno inicializovat)

```
int string::pocet = 0;
```

3.10 přetěžování funkcí (no)

- v C jediná funkce s daným jménem
- v C++ více stejnojmenných funkcí – přetěžování
- (přetěžování není ve smyslu překrytí ale přidání)
- funkce odlišené počtem nebo typem parametrů, prostorem,
- typ návratové hodnoty nerozlišuje
- při volání vybírá překladač na základě kontextu
- přednost má ”nejbližší”, jinak uvést celé - Prostor::Jméno
- problém s konverzemi

```
int f(int);
float f(int); // nelze rozlišit – návrat h.
float f(float); // lze rozlišit – jiný param
float f(float, int) // lze – počet param

float ff = f(3); // volání f(int)
f(3.14); // chyba – double lze na int i float
         // – překladač neví
f( (float) 3.14);//v pořádku–volá se f(float)
f(3,4.5); // OK, implicitní konverze parametrů
```

```
// volá se f(float, int) – podle počtu param
```

3.11 implicitní parametry (no)

- parametr, který se dosadí (překladač), není-li uveden při volání
- uvádí se (pouze jedenkrát) v deklaraci (h. soubory)
- v definici se uvádí od poslední proměnné
- při volání se vynechávají od poslední proměnné
- nejen hodnota ale libovolný výraz (konstanta, volání funkce, proměnná …)

```
int f(float a=4,float b=random()); //deklarace
// funkce je použita pro volání
f();
f(22);
f(4,2.3);
// a koliduje s (zastupuje i)
f(void);
f(float);
f (float, float);
// a někdy koliduje i s
f(char); // nejednoznačnost při volání
// s parametrem int – konverze na char i float
```

3.12 přetypování (no)

- explicitní (vynucená) změna typu proměnné
- v C se provádí **(float) i**
- v C++ se zavádí ”funkční” přetypování **float(i)** - operátor
- lze nadefinovat tuto konverzi-přetypování u vlastních typů
- toto platí o “vylepšení” c přetypování. Ještě existuje nový typ přetypování

```
double a; int b = 3;
a = 5 / b; // pravá strana je typu int
```

```
// následně implicitní konverze na typ double
a = b / 5; // pravá strana je typu int
a = 5.0 / b; // pravá strana je double
// výpočet v největší přesnosti
a = (double) b / 5; // pravá strana je double
```

3.13 const, const parametry (no)

1) vytvoření konstantní (neměnné) proměnné
- nelze měnit – kontroluje překladač – chyba
- obdoba **#define PI 3.1415** z jazyka C
- typ proměnné je součástí definice **const float PI=3.1415;**
- obvykle dosazení přímé hodnoty při překladu
- nepředávat odkazem (někdy vytvořena dočasná proměnná)

U použití ve více modulech (deklarace-definice v h souboru)
- u C je **const char a=´b´;** ekvivalentní **extern const char a=´b´;**
- pro lokální viditelnost – **static const char a=´b´;**
- u C++ je **const char a=´b´;** ekvivalentní **static const char a=´b´;**
- pro globální viditelnost – **extern const char a=´b´;**
- výhodné psát včetně modifikátorů extern/static

2) potlačení možnosti změn u parametrů předávaných funkcím (především) ukazatelem a odkazem

```
int fce(const int *i) …
```

- proměnná označená const je hlídána překladačem před změnou
- volání const parametrem na místě neconst parametru (fce) – nelze

shrnutí definicí (typ, ukazatel, reference, const):

`T`	`je proměnná daného typu`
`T *`	`je ukazatel na daný typ`
`T &`	`reference na T`
`const T` `T const`	`deklaruje konstantní T` `(const char a=´b´;)`
`T const *`	`deklaruje ukazatel na`

`const T*`	`konstantní T`
`T const &` `const T&`	`deklaruje referenci na konstantní T`
`T * const`	`deklaruje konstantntí ukazatel na T`
`T const * const` `const T* const`	`deklaruje konstatntní ukazatel na konstantní T`

3.14 alokace paměti (no)

- typově orientovaná (dynamická) práce s pamětí
- klíčová slova **new** a **delete**
- jednotlivé proměnné nebo pole proměnných
- alternativa k **xxxalloc** resp. **xxxfree** v jazyce C
- volá konstruktory resp. destruktory
- lze ve třídách přetížit (volání ”globálních” ::new, ::delete)

jedna proměnná

```
void* :: operator new      (size_t)
void  :: delete            (void *)

char*  pch = (char*) new char;
// alokace paměti pro jeden prvek typu char
delete pch; // vrácení paměti pro jeden char

Komplex *kk;
kk = new Komplex(10,20); // alokace paměti
// pro jeden prvek Komplex s inicializací
// zde předepsán konstruktor se dvěm parametry
```

pole

```
void* :: operator new[ ]   (size_t)
void  :: delete[]          (void *)
```

```
Komplex *pck = (Komplex*) new Komplex [5*i];
// alokace pole objektů typu Komplex, volá se
// implicitní konstruktor pro každý prvek pole
delete[]  pck; // vrácení paměti pole
// destruktory na všechny prvky
delete pck;//destruktor pouze na jeden prvek!!
```

new T = new(sizeof(T)) = T::operator new (size_t)
new T[u] = new(sizeof(T)*u+hlavička) = T::operator new[ ](size_t)
new (2) T = new(sizeof(T),2) - první parametr size_t
new T(v) + volání konstruktoru

```
void T::operator delete(void *ptr)
 { // přetížený operátor delete pro třítu T
   // ošetření ukončení ”života” proměnné
  if (změna) Save(”xxx”);
  if (ptr!=NULL)
   ::delete ptr; //”globální” delete,
// jinak možné zacyklení
…}
```

konstruktory (v novějších verzích) při nenaalokování paměti podle požadavků používají systém výjimek, konkrétně bad_alloc.
“původní” vracení NULL je možné ve verzi s potlačením výjimek

```
char *uk = new(nothrow) char[10]
```

3.15 enum (no)

- v C lze převádět enum a int
- v C je: sizeof(A) = sizeof(int) pro enum b(A);
- v C++ jméno výčtu jménem typu
- v C++ lze přiřadit jen konstantu stejného typu
- v C++: sizeof(A) = sizeof(b) = některý celočíselný typ

3.16 konstruktory a destruktory (o)

```
{//příklad(přiblížení) pro standardní typ int
 int a;
//definice proměnné bez konkrétní inicializace
// = implicitní
 int b = 5.4;
// definice s inicializací. vytvoření,
// konstrukce, int z double hodnoty = konverze
 int c = b;
// konstrukce (vytvoření) proměnné na základě //
proměnné stejného typu = kopie
…
}  // konec platnosti proměnných – zrušení
// je to zrušení bez ošetření – (u std typů)
// zpětná  kompatibilita
```

- možnost ovlivnění vzniku (inicializace) a zániku (úklid) objektu
- volány automaticky překladačem
- konstruktor – první (automaticky) volaná metoda na objekt
- destruktor – poslední (automaticky) volaná metoda na objekt

Konstruktor
- stejný název jako třída
- nemá návratovou hodnotu
- volán automaticky při vzniku objektu (lokálně i dynamicky)
- využíván k inicializaci proměnných (nulování, nastavení základního stavu, alokace paměti, …)

```
class Trida {
```

```
public:
Trida(void) {…} // implicitní kontruktor
Trida(int i) {…} // konverzni z int
Trida(Trida const & a) {…} // kopy konstruktor
 }
```

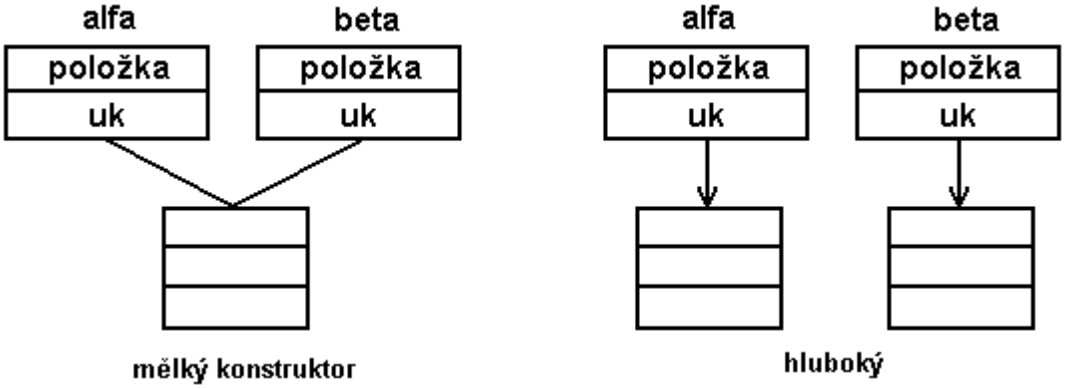
- několik konstruktorů – přetěžování
- implicitní (bez parametrů) – volá se i při vytváření prvků polí
- konverzní – s jedním parametrem
- kopy konstruktor – vytvoření kopie objektu stejné třídy (předávání hodnotou, návratová hodnota …), rozlišovat mezi kopy konstruktorem a operátorem přiřazení

```
Trida(void)   // implicitní
Trida(int i)  // konverzní z int
Trida(char *c) // konverzní z char *
Trida(const Trida &t) // copy
Trida(float i, float j) // ze dvou parametrů
Trida(double i, Trida &t1) //ze dvou parametrů

Trida a, b(5), c(b), d=b, e(”101001”);
Trida f(3.12, 8), g(8.34, b), h = 5;
// pouze pro ”názornost” !!! Překladač přeloží
Trida a.Trida(), b.Trida(5), c.Trida(b),
d.Trida(b) e.Trida(”101001”);
h.Trida((tmp.)Trida(5))), (nebo výjimečně
h.Trida(5) (nebo špatně)
h.operator=((tmp.)Trida(5))), ( (tmp.)~Trida();
)
//”.Trida” by bylo nadbytečné a tak se neuvádí
```

- explicit - klíčové slovo – zakazuje použití konstruktoru k implicitní konverzi

```
explicit Trida(int j)
Trida::Metoda(Trida & a)

int i;
 a.Metoda ( i ); // nelze,
// implicitní konverze se neprovede
 b.Metoda ( Trida(i)); // lze,
// explicitní konverze je povolena
```

- dynamická data ve třídě – problém při rušení , přiřazení, kopii…
- mělké a hluboké kopírování (shallow, deep copy)
- řešením je vlastní kopie nebo indexované odkazy



- u polí se volají konstruktory od nejnižšího indexu
- konstruktor nesmí být static ani virtual
- konstruktory se používají pro implicitní konverze, pouze jedna uživatelská (problémy s typy, pro které není konverze)
- pokud nejsou definovány, vytváří se implicitně bezparametrický jako prázdný
- není-li definován kopykonstruktor, je vytvořen a provádí kopii (paměti) jedna k jedné (memcopy)
- alespoň jeden musí být v sekci public (jinak zákaz pro běžného uživatele)

Destruktor
- stejný název jako třída, předchází mu ~ (proč?)
- je pouze jeden (bez parametrů)
- nemá návratovou hodnotu
- volán automaticky překladačem
- zajištění úklidu (vrácení systémových prvků, paměť, soubory, ovladače, ukončení činnosti HW, uložení dat …)

```
~Trida (void)
```

- destruktory se volají v opačném pořadí jako konstruktory
- je možné ho volat jako metodu (raději ne)
- není-li definován vytváří se implicitně prázdný
- musí být v sekci public

Objekty jiných tříd jako data třídy
- jejich konstruktory se volají před konstruktorem třídy
- volají se implicitní, není-li uvedeno jinak
- pořadí určuje pořadí v deklaraci třídy (ne pořadí v konstruktoru)

```
class Trida {
int i;  // zde je určeno pořadí volání = i,a,b
Trida1 a; // prvek jiné třídy prvkem třídy
Trida2 b;
public:
Trida(int i1,int i2,int x):b(x,4),a(i2),i(i1)
//zde jsou určeny konkretni konstruktory
{ … tělo … }
// vola se konstruktor i, a, b a potom tělo
}
```

```
například
class string {… data …
public:
```

```
string(char *txt) { … }
~string() {…}
}
class Osoba {
int Vek;
string Jmeno;
string Adresa;
public:
Osoba(char*adr, char* name,int ii) :Adresa(adr),
Jmeno(name), Vek(ii) {… tělo konstruktoru …}
}

{ // vlastní kod pro použití
string Add(”Kolejni 8 ”) ;
// standardní volání konstruktoru stringu
Osoba Tonda(Add, ”Tonda”,45);
// postupně volá konstruktor int  (45)
// pro věk, poté konstruktor string pro jmeno
// (tonda)a adresu (Add) (pořadí jak jsou
// uvedeny v hlavičce třídy, a potom
// vlastní tělo konstruktoru Osoba
} // tady jsou volány destruktor Osoba,
// destruktory stringů Adresa a
// Jmeno. A destruktor pro Add
// (v uvedeném pořadí)
```

3.17 inline funkce (no)

- obdoba maker v C
- předpis pro rozvoj do kódu, není funkční volání
- v hlavičkovém souboru
- označení klíčovým slovem inline
- pouze pro jednoduché funkce (jednoduchý kód)

- volání stejné jako u funkcí
- v debug modu může být použito funkční realizace
- některé překladače berou pouze jako “doporučení”

```
inline int plus2(int a) {return a+2;}

int bb = 4,cc;
cc = plus2(bb); // v tomto místě překladač
// vloží (něco jako) cc = bb+2; (+doplňky)
```

3.18 Hlavičkové soubory a třída (o)

- hlavičkový soubor (.h), inline soubor (.inl, nebo .h), zdrojový soubor (.cpp)
- hlavička – deklarace třídy s definicí proměnných a metod, a přístupových práv (”těla” inline metod – lépe mimo) - předpis
- inline soubor – ”těla” inline metod -předpis
- zdrojový soubor – ”těla” metod – ”skutečný” kód
- soubory kde je třída používána
- mimo definici třídy je nutné k metodám uvádět, ke které třídě patří pomocí operátoru příslušnosti T::metoda

hlavička: (soubor.h)

```
class T{
data
metody (bez ”těla”)
};

těla metod přímo v hlavičce, nebo přidat
#include ”soubor.inl”

soubor.inl obsahuje těla inline metod:
T::těla metod
návratová hodnota T::název(parametry) {…}
```

zdrojový kód:

```
#include  hlavička
T::statické proměnné
T::statické metody
T::těla metod
```

soubory, kde je třída používána

```
#include ”hlavička”
použití třídy

//======== konkrétní příklad ==========
// hlavičkový soubor
class POKUS {
int a;
public:
POKUS(void) { this->a = 0;}
//má tělo -> inline, netvoří kód

inline POKUS(int xx);
//označení -> inline, netvoří kód

POKUS(POKUS &cc); // nemá tělo, ani označení
// -> není inline = funkční volání = generuje
// se kód
};
// pokračování hlavičkového souboru
// nebo #include ”xxx.inl” a v něm následující

// je inline, proto musí být v hlavičce
// protože je mimo tělo třídy musí být
// v názvu i označení třídy (prostoru)
POKUS::POKUS(int xx) { this->a = xx; }

// konec hlavičky (resp. inline)
```

```
// zdrojový soubor
#include ”hlavička.h”
POKUS::POKUS (POKUS&cc) { this->a = cc.a; }
```

3.19 inline metody (o)

- obdoba inline funkcí
- rozbalené do kódu, předpis, netvoří kod
- automaticky ty s ”tělem” v deklaraci třídy
- v deklaraci třídy s inline, tělo mimo (v hlavičce)
- pouze hlavička v deklaraci třídy, tělo mimo (ve zdroji) –není inline
- obsahuje-li složitý kód (cykly) může být inline potlačeno (překladačem)

.h soubor	.cpp soubor	pozn.
metoda() {}	-	inline funkce, je definováno tělo v hlavičce
metoda();	metoda:: metoda() {}	není inline. Tělo je definováno mimo hlavičku a není uvedeno inline
inline metoda();	inline metoda:: metoda(){}	je inline ”z donucení” pomocí klíčového slova inline. Definice by ale měla být též v hlavičce (v cpp chyba)
metoda () {}	- neinline	nelze programátorskými prostředky zajistit aby nebyla inline, může ji však překladač přeložit jako neinline (na inline příliš složitá)
metoda ();	inline metoda:: metoda(){}	špatně (mělo by dát chybu) – mohlo by vést až ke vzniku dvou interpretací – někde inline a někde funkční volání
inline metoda()	metoda:: metoda() {}	špatně – mohlo by vést až ke vzniku dvou interpretací – někde inline a někde funkční volání; i když je vlastní kód metody pod hlavičkou takže se o inline ví, nedochází

		k chybě

3.20 shrnutí deklarací a definicí tříd a objektů (o)

- class Trida; - oznámení názvu třídy – hlavička – použití pouze ukazatelem
- class Trida {} – popis třídy – proměnných a metod – netvoří se kód - hlavička
- Trida a, *b, &c=a, d[10]; definice proměnná dané třídy, ukazatel na proměnnou, reference a pole prvků – zdrojový kód
- extern Trida a , *b; deklarace proměnná a ukazatel - hlavička
- platí stejná pravidla o viditelnosti lokálních a globálních proměnných jako u standardních typů
- ukazatel: na existující proměnnou nebo dynamická alokace (->)
- přístup k datům objektu – z venku podle přístupových práv, interně bez omezení (v aktuálním objektu přes this->, nebo přímo)
- pokud je objekt třídy použit s modifikátorem const, potom je nejprve zavolán konstruktor a poté teprve platí const

3.21 operátory přístupu ke členům (o)

- operátory pro přístup k určitému členu třídy – je dán pouze prototyp, reference může být na kterýkoli prvek třídy odpovídající prototypu
- využití například při průchodu polem a prací s jednou proměnnou
- .* dereference ukazatele na člen třídy přes objekt
- ->* dereference ukazatele na člen třídy přes ukazatel na objekt
- nejdou přetypovat (ani na void)
- při použití operátoru .* a -> je prvním operandem vlastní objekt třídy T, ze kterého chceme vybraný prvek použít

```
int (T::*p1) (void);
    // definice operátoru pro přístup k metodě
    // bez parametrů vracející int ze třídy T
p1=&T::f1;
    // inicializace přístupu na konkrétní
```

```
    // metodu   int T::f1(void)

float T::*p2;
     // definice operátoru pro přístup k
     // proměnné
p2=&T::f2;
     // inicializace přístupu na konkrétní
     // proměnnou zatím nemáme objekt ani
     // ukazatel na něj – pouze definici třídy
T tt,*ut=&tt;
ut->*p2=3.14;
(ut->*p1)();//volání fce –závorky pro prioritu
tt.*p2 = 4;
tt.*p1( )
```

3.22 deklarace třídy uvnitř jiné třídy (o)

- jméno vnořené třídy (struktury) je lokální
- vztahy jsou stejné jako by byly definovány nezávisle (B mimo A)
- jméno se deklaruje uvnitř, obsah vně
- použití pro pomocné objekty které chceme skrýt

```
class A {
   class B; // deklarace (názvu) vnořené třídy
   .....
}

class A::B { // vlastní definice těla třídy
   .....
}

A::B x; // objekt třídy B definován vně
```

3.23 const a metody (o)

- const u parametrů – nepředávat hodnotou, ochrana proti nechtěnému přepisu hodnot
- const parametry by neměly být předávány na místě neconst
- na const parametry nelze volat metody, které je změní – kontrola překladač
- metody, které nemění objekt je nutno označit jako const a takto označené metody lze volat na const objekty

```
float f1(void) const {   …  }
//míní float f1(T const * const this) {}
```

3.24 prototypy funkcí (no)

- v C nepovinné uvádět deklaraci (ale nebezpečné) – implicitní definice
- v C++ musí být prototyp přesně uveden (parametry, návrat)

- není-li v C deklarace (prototyp) potom se má za to, že vrací int a není informace o parametrech
- **`void fce( )`** v C++ je bez parametrů tj. **`void fce(void)`**
- **`void fce( )`** je v C (neurčená funkce) – funkce s libovolným počtem parametrů (**`void fce(…))`**
- **`void fce(void)`** v C – bez parametrů

3.25 friend funkce (o)

- klíčové slovo friend
- zaručuje přístup k private členů pro nečlenské funkce či třídy
- friend se nedědí
- porušuje ochranu dat, (zrychluje práci)

```
class Třída {
friend complex;
```

```
friend double f(int, Trida &);
// třída komplex a globální funkce f mohou
// přistupovat i k private členům Třídy.
private:
int i;
}

double f(int a, Trida &tt)
{
 tt.i = a; // jde, protože friend
}
```

- zdrojový kód friend funkce a třídy je mimo a nenese informaci o třídě ke které patří (nemá this …)
- alternativou jsou statické metody

3.26 funkce s proměnným počtem a typem parametrů (no) – výpustka

```
int fce (int a, int b, …);
```

- v C nemusí být ”, …” uvedeno
- v C++ musí být ”, …” uvedeno
- u parametrů uvedených v části ”…” nedochází ke kontrole typů

3.27 typ bool (no)

- nový typ pro reprezentaci logických proměnných
- klíčová slova bool a true a false (konstanty pro hodnoty)
- implicitní konverze mezi int a bool (0 => false, nenula => true, false => 0, true => 1)
- v C pomocí define nebo enum

```
bool test; int i,j;
test = i == j;
// do test se uloží výsledek srovnání i a j
test = i; // dojde ke ”klasické” konverzi
// 0 -> false, ostatní -> true
```

```
j = test; // ”klasická” konverze 0/1
```

3.28 přetížení operátorů (no)

- je možné přetížit i operátory (tj. definovat vlastní)
- klíčové slovo operátor následované typem operátoru
- deklarace pomocí funkčního volání např.
  `int operator +(int ) {}`
- možnost volat i funkčně `operator=(i,operator+( j ))` nebo zkráceně `i=+j`
- operátorem je i vstup a výstup do **stream**u, **new** a **delete**
- hlavní využití u objektů

3.29 operátory (o)

- operátory lze v C++ přetížit stejně jako metody
- správný operátor je vybrán podle seznamu parametrů (a dostupných konverzí), rozliší překladač podle kontextu
- operátory unární mají jeden parametr – proměnnou se kterou pracují, nebo ”this”
- operátory binární mají dva parametry – dvě proměnné, se kterými pracují nebo jednu proměnnou a ”this”
- unární operátory: +, -, ~, !,++,--
- binární +,-,*,/,%,=,^,&,&&,|,||,>,<,>=,==, +=,*=,<<,>>,<<=, …
- ostatní operátory [ ], ( ), new, delete
- operátory matematické a logické
- nelze přetížit operátory: sizeof, ? :, ::, ., .*
- nelze změnit počet operandů a pravidla pro asociativitu a prioritu
- nelze použít implicitních parametrů
- slouží ke zpřehlednění programu
- snažíme se aby se přetížené operátory chovaly podobně jako původní (např. nemění hodnoty operandů, + sčítá nebo spojuje…)
- klíčové slovo operator
- operátor má plné (funkční) a zkrácené volání

```
z = a + b
z.operator=(a.operator+(b))
```

- nejprve se volá operátor + a potom operátor =
- funkční zápis slouží i k definování operátoru

```
T T::operator+(T & param) {}
T operator+(double d, T&param) {}
```

Unární operátory

- mají jeden parametr (u tříd this)
- například + a – , ~, ! ++, --

```
complex       & operator+(void)
complex const & operator+(void)
complex         operator-(void)
```

- operátor plus (+aaa) nemění prvek a výsledkem je hodnota tohoto (vně metody existujícího) prvku – proto lze vrátit referenci – což z úsporných důvodů děláme (výsledek by neměl být měněn=const)
- operátor mínus (-aaa) nemění prvek a výsledek je záporná hodnota – proto musíme vytvořit nový prvek – vracíme hodnotou

- operátory ++ a `--` mají prefixovou a postfixovou notaci

- definice operátorů se odliší (fiktivním) parametrem typu int
- je-li definován pouze jeden, volá se pro obě varianty
- některé překladače obě varianty neumí

```
++( void)   s voláním   ++x
++( int )   s voláním   x++. Argument int se však
při volání nevyužívá
```

Binární operátory

- mají dva parametry (u třídy je jedním z nich)
- například +, -, %, =, +=, <, …

```
complex complex::operator+ (complex & c)
complex complex::operator+ (double c)
complex operator+ (double f,complex & c)

a + b          a.operator+(b)
a + 3.14       a.operator+(3.14)
3.14 + a       operator+(3.14,a)
```

- výstupní hodnota různá od vstupní – vrácení hodnotou
- mohou být přetížené
- lze přetížit i globální (druhý parametr je třída) – často friend
- opět kolize při volání (implicitní konverze)
- parametry se (jako u standardních operátorů) nemění a tak by měly být označeny const, i metoda by měla být const

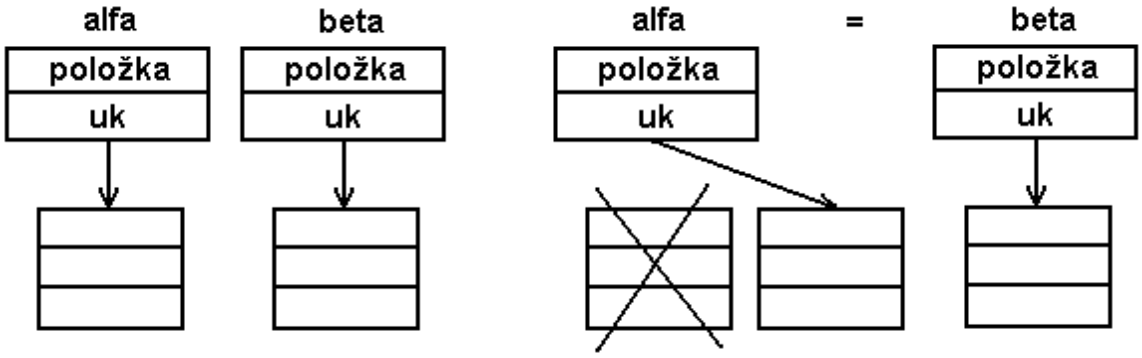
Operátor =

- měl by (díky kompatibilitě) vracet hodnotu
- obzvláště zde je nutné ošetřit případ a = a
- pro činnost s dynamickými daty nutno ošetřit mělké a hluboké kopie
- vytváří se implicitně (mělká kopie – přesná kopie 1:1, memcopy)
- nadefinování zamezí vytvoření implicitního =
- je-li v sekci private, pak to znamená, že nelze použít (externě)
- od kopykonstruktoru se liší tím, že musí před kopírováním ošetřit proměnnou na levé straně

```
T& operator=(T const& r)
```

musí umožňovat

```
a = b = c = d = …;
a += b *= c /= d &= …;
```



Způsoby přiřazení:

```
string * a,* b;
a = new string;
b = new string;
a = b ;
delete a ;
delete b ;
```

- pouze přiřazení ukazatelů, oba ukazatele sdílí stejný objekt (stejná statická i dynamická data)
- chyba při druhém odalokování, protože odalokováváme stejný objekt podruhé

```
string {int delka; char *txt}
string a , b(”ahoj”);
a = b ;
”delete a” ; // volá překladač
”delete b” ;
```

- je vytvořeno a tedy použito implicitní =
- ukazatel txt ukazuje na stejná dynamická data (statické proměnné jsou zkopírovány, ale dále se používají nezávisle)
- pokud je nadefinován destruktor, který odalokuje txt (což by měl být), potom zde odalokováváme pamět, kterou odalokoval již destruktor pro prvek a

```
string {int delka; char *txt;operator =();}
string a , b(”ahoj”);
a = b ;
”delete a” ;
”delete b ”;
```

- použito nadefinované =
- v = se provede kopie dat pro ukazatel txt
- oba prvky mají svoji kopii dat statických i dynamických
- každý prvek si odalokovává svoji kopii

Konverzní operátory

- převod objektů na jiné typy
- využívá překladač při implicitních konverzích
- opačný směr jako u konverzních konstruktorů
- například konverze na standardní typy – int, double…
- nemá návratovou hodnotu (je dána názvem)
- nemá parametr

```
operator typ(void)
T::operator int(void)
```

volán implicitně nebo

```
T aaa;
int i = int (aaa) ;
(int) aaa; // starý typ - nepoužívat
```

Přetížení funkčního volání

- může mít libovolný počet parametrů
- takto vybaveným objektům se říká funkční objekty
- nedoporučuje se ho používat

```
operator ()(parametry )
double& T::operator()(int i,int j) { }
T aaa;
double d = aaa(4,5); // vypadá jako funkce
// ale je to funkční objekt
d  = aaa.operator()(5,5);
aaa(4,4) = d;
```

Přetížení indexování

- podobně jako operátor( ) ale má pouze jeden operand (+ this, je to tedy binární operátor)
- nejčastěji používán s návratovou hodnotou typu reference (l-hodnota)

```
double& T::operator[ ](int )
aaa[5] = 4;
d = aaa.operator[ ](3);
```

přetížení přístupu k prvkům třídy

- je možné přetížit ” -> ”
- musí vracet ukazatel na objekt třídy pro kterou je operátor -> definován protože:

```
TT* T::operator->( param ) { }
x -> m;                  // je totéž co
(x.operator->( ) ) -> m;
```

3.30 statické metody (o)

- pouze jedna na třídu
- nemá this
- ve třídě označená static
- vlastní tělo ve zdrojové části
- nesmí být virtuální
- (jako friend funkce, může k private členům)

- externí volání se jménem třídy bez objektu **`Třída::fce( )`**

```
class Třída {
static int fce (void);
}

int Třída::fce( ) { } // ve zdrojové části
```

3.31 mutable (o)

- označení proměnných třídy, které je možné měnit i v const objektu
- statická data nemohou být mutable

```
class X {
public :
mutable int i ;
int j ;
}

class Y { public: X x; }

const Y y ;
y . x . i ++ ;    může být změněn
y . x . j ++ ;    chyba
```

3.32 prostory jmen (no)

- ”oddělení” názvů proměnných - identifikátorů
- zabránění kolizí stejných jmen (u různých programátorů)
- přidává ”příjmení” ke jménu

prostor::identifikátor
prostor::podprostor::identifikátor

- při použití má přednost ”nejbližší” identifikátor
- klíčové slovo namespace – vyhrazuje (vytváří) prostor s daným jménem
- vytváří blok společných funkcí a dat – oddělení od zbytku
- přístup do jiných prostorů přes celý název proměnné
- klíčové slovo using – zpřístupňuje v daném prostoru identifikátory či celé prostory skryté v jiném prostoru. Umožní neuvádět ”příjmení” při přístupu k proměnným z jiného prostoru
- doporučuje se zpřístupnit pouze vybrané funkce a data (ne celý prostor)
- ”dotažení” proměnné končí s koncem bloku ve kterém je uvedeno

např. je-li **cout** v prostoru **std**, pak správný přístup je **std::cout**
z našeho programu. Použijeme-li na začátku **modulu using namespace std**, pak to znamená, že k proměnným z prostoru **std** můžeme přistupovat přímo a tedy psát **cout** (a ten se najde)
lépe je být konkrétní a tedy **using std::cout**

definice

```
namespace {
proměnné
funkce
třídy
};
```

pomocí **using BázováTřída::g**; lze ve zděděné třídě zpřístupnit prvek který se ”ztratil” (byl překryt či je nepřístupný)

3.33 znakové konstanty, dlouhé literály (no)

- standardně typ char (8bitů – základní znaky)
- znaková sada “interní systému”, překladače, programu
- znaky (univerzální jména znaků) je možné zadávat pomocí ISO 10646 kódu ”F\u00F6rster” (Főrster)

- typ pro ukládání znakových proměnných větších než char (UNICODE-snaha o sjednocení s ISO 10646) – tj. ”dlouhé” znakové konstanty – dostatečně velký celočíselný typ (například unsigned int)
- znaky (char) již proto nejsou konvertovány na int
- v C je sizeof(´a´) = sizeof(int) v C++ je = sizeof(char)
- w_char**, wchar_t**
- wchar_t b = L´a´;
- wchar_t c[20] = L”abcd”;
- existují pro něj nové funkce a vlastnosti například wprintf( ), wcin, wcout, MessageBoxW( )…
- řeší výstup zapsaný v “Unicode” do výstupu na základě lokálních nastavení
- zatím brán jako ”problematický typ”

- třída std:: locale s řešeními pro nastavování a správu informací o národním prostředí (a tedy národní specifika se neřeší přímo ve tvořeném programu)
- setlocale() – nastavení pro program – desetinná tečka, datum, čas
- wcout.imbue(xxx) napojení výstupního proudu na lokální prostředí (locale xxx)

3.34 typ long long (no)

- nový celočíselný typ s danou minimální přesností 64 bitů

```
unsigned  long long int lli = 1234533224LLU;
printf(”%lld”,lli);
```

3.35 restrict (no)

- nové klíčové slovo v jazyce C, modifikátor (jako const…) u ukazatele
- z důvodů optimalizací – rychlejší kód – (možnost umístit do cache či registru)
- říká, že daný odkaz (ukazatel) je jediným, který je v daném okamžiku namířen na data

- to je - data nejsou v daném okamžiku přístupna přes jiný ukazatel – data
- to je - data se nemění
- const řeší pouze přístup přes daný ukazatel, ne přes jiné (lze const i neconst ukazatel ne jednu proměnnou)

--

pulsem

3.36 Vstupy a výstupy v jazyce C++

- jazyk C++ dává možnost řešit vstup a výstup proměnných (na V/V zařízení) podstatně elegantněji než jazyk C. Tyto mechanizmy se postupně vyvíjejí, v poslední době využívají vlastnosti šablon, dědění i přetěžování funkcí. Existují společné vlastnosti operací s proměnnou a V/V, které jsou specializované pro standardní typy.
- přetížení (globálních) operátorů << a >>
- typově orientovány
- knihovní funkce (ne klíčová slova)
- vstup a výstup je zajišťován přes objekty, hierarchie tříd
- knihovny xxxstream
- ”napojení” na soubor, paměť, standardní (seriové) zařízení
- standardní vstupy a výstupy (streamy) – cin, cout, cerr, clog, wcin, wcout, wcerr, wclog
- dříve definováno pomocí metod, nyní pomocí šablon, dříve pracuje s bytem, nyní šablona (tedy obecný typ, např. složitější znakové sady)

práce se streamy

- vstup a výstup základních typů přes konzolu
- formátování základních typů
- práce se soubory
- implementace ve třídách

- obecná realizace streamu

vstup a výstup přes konzolu

- přetíženy operátory << a >> pro základní typy
- výběr na základě typu proměnné
- předdefinován cout, cin, cerr, clog (+ w…)
- knihovna iostream
- zřetězení díky návratové hodnotě typu stream
- vyprázdnění (případného) bufferu – flush, endl (´\n´+flush), ends (´\0´+flush)

```
cin >> i >> j >> k;
cout << i << ”text” << j << k << endl;

xstream & operator xx (xstream &, Typ& p) {  }
```

- načítá se proměnný počet znaků – **gcount** zjistí kolik
- vypouštění bílých znaků – přepínače **ws, noskipws, skipws** (vynechá bílé znaky, nepřeskakuje, přeskakuje BZ na začátku)
- načtení celého řádku **getline(kam,maxkolik)** – čte celý řádek, **get(kam, maxkolik)** – čte řádek bez odřádkování
- načtení znaku **get** – čte všechny znaky, zastaví se na bílém znaku
- **put** – uložení znaku (Pouze jeden znak bez vlivu formátování)
- vrácení znaku – **putback**
- ”vyčtení” znaku tak aby zůstal v zařízení – **peek**

```
int i,j;
 cout << ” zadejte dvě celá čísla \n”;
 cin>> i>> j;

cout<<´\n´<<i<<”/”<<j<<”=”<<double(i)/j<<endl;
```

formátování základních typů

- je možné pomocí modifikátorů, manipulátorů nebo nastavením formátovacích bitů
- ovlivnění tvaru, přesnosti a formátu výstupu
- může být v ”**iomanip**”
- přetížení operátorů << a >> pro parametr typu manip, nebo pomocí ukazatelů na funkce
- přesnost výsledku má přednost před nastavením
- *manipulátory* - funkce pracující s typem stream – mají stream & jako parametr i jako návratovou hodnotu
- slouží buď pro nastavení nové, nebo zjištění stávající hodnoty
- některé působí na jeden (následující) výstup, jiné trvale
- bity umístěny ve třídě ios (staré streamy), nově v **ios_base** – kde jsou společné vlastnosti pro input i output, které nezávisí na templatové interpretaci (ios)
- nastavení bitů – pomocí **setf** s jedním parametrem (vrátí současné)
- **setf** se dvěma parametry – nastavení bitu + nulování ostatních bitů ve skupině (označení bitu, označení společné skupiny bitů)
- nulování bitů – **unsetf**
- někdy (dříve) setioflags, resetioflags, flags
- pro uchování nastavení bitů je předdefinován typ **fmtflags**
- **i = os.width(j)** – šířka výpisu, pro jeden znak, default 0
- **os << setw(j) << i;**
- **i = os.fill(j)** – výplňový znak, pro jeden výstup, default mezera
- **os<<setfill(j) << i;**
- **ios_base::left, ios_base::right, left, right** - zarovnání vlevo vpravo – **fmtlags orig = os.setf(ios_base::left, ios_base::adjustfield)** – manipulátory bity nastaví i nulují
- **ios_base::internal**, internal – znaménko zarovnáno vlevo, číslo vpravo
- bity **left, right, internal** patří do skupiny **ios_base::adjustfield**
- **ios_base::showpos**, manipulátor **showpos, noshowpos** – zobrazí vždy znaménko (+, -)

- **ios_base::uppercase, uppercase, nouppercase** – zobrazení velkých či malých písmen v hexa a u exponentu
- **ios_base:: dec,ios_base::hex, ios_base::oct, dec, oct, hex** – přepínání formátů tisku (bity patří do skupiny – **ios_base::basefield**)
- **setbase** – nastavení soustavy
- **ios_base::showbase, showbase, noshowbase** – tisk 0x u hexa
- **ios_base::boolalpha, boolalpha, noboolalpha** – tisk ”true”, ”false”
- **os.precision(j)** – nastavení přesnosti, významné číslice, default 6
- **os<<setprecision(j) << i**
- **ios_base::showpoint, showpoint, noshowpoint** – nastavení tisku desetinné tečky
- **ios_base::fixed, fixed** – desetinná tečka bez exponentu
- **ios_base::scientific, scientific** – exponenciální tvar
- bity **fixed, scientific** patří do **ios_base::floatfield**
- **eatwhite** – přeskočení mezer, **writes** – tisk řetězce …

práce se soubory

- podobné mechanizmy jako vstup a výstup pro konzolu
- přetížení operátorů >> a <<
- fstream, ofstream, ifstream, iofstream
- objekty – vytváří se konstruktorem, zanikají destruktorem
- první parametr – název otevíraného souboru
- lze otevřít i metodou open, zavřít metodou close
- metoda is_open pro kontrolu otevření (u MS se vztahuje na vytvoření bufferu a pro test otevření se doporučuje metoda fail())

```
ofstream os(”nazev souboru”);
os << ”vystup”;
os.close( );
os.open(”jiny soubor.txt”);
if (!os.is_open()) …
```

- druhý parametr udává typ otevření, je definován jako enum v ios_base
- **ios_base::in** pro čtení
- **ios_base::out** pro zápis
- **ios_base:: ate** po otevření nastaví na konec souboru
- **ios_base::app** pro otevření (automaticky out) a zápis (vždy) za konec souboru
- **ios_base::binary** práce v binárním tvaru
- **ios_base::trunc** vymaže existující soubor
- ios::nocreate - nově nepodporováno - otevře pouze existující soubor (nevytvoří)
- ios::noreplace - nově nepodporováno - otevře pouze když vytváří (neotevře existující)

```
ofstream os(”soub.dat”,
ios_base::out|ios_base::ate|ios_base::binary);
istream is(”soub.txt”,ios_base::in);
fstream iostr(”soub.txt”,
                ios_base::in | ios_base::out);
```

zdroj: www.devx.com
záměna noreplace

```
fstream fs(fname, ios_base::in);
// attempt open for read
if (!fs)
{
 // file doesn't exist; don't create a new one
}
else
//ok,file exists. close and reopen in write mode
{
 fs.close();
 fs.open(fname,ios_base::out);
 //reopen for write
```

```
}
```

záměna nocreate

```
fstream fs(fname, ios_base::in);
// attempt open for read
if (!fs)
{
 // file doesn't exist; create a new one
 fs.open(fname, ios_base::out);
}
else //ok, file exists; close and reopen in write mode
{
 fs.close()
 fs.open(fname, ios_base::out);
// reopen for write
}
```

- zjištění konce souboru – metoda **eof** – ohlásí až po načtení prvního za koncem souboru
- zjišťování stavu – bity stavu – ios_base::io_state
- **goodbit** – v pořádku
- **badbit** – vážná chyba (např. chyba zařízení, ztráta dat linky, přeplněný buffer …) – problém s bufferem (HW)
- **failbit** - méně závažná chyba, načten špatný znak (např. znak písmene místo číslice, neotevřen soubor) – problém s formátem (daty)
- **eofbit** – dosažení konce souboru
- zjištění pomocí metod – **good( ), bad( ),fail( ), eof( )**

- zjištění stavu -

```
if(is.rdstate()&
        (ios_base::badbid|ios_base::failbit)) …
```

- smazání nastaveného bitu (po chybě, i po dosažení konce souboru) pomocí clear(bit)
- při chybě jsou i výjimky **– basic_ios::failure**. Výjimku je možné i nastavit pro **clear** pomocí **exceptions (iostate ist)**
- práce s binárním souborem **write(bufer, kolik), read(bufer,kolik)**
- pohyb v souboru – **seekp** (pro výstup) a **seekg** (pro vstup), parametrem je počet znaků a odkud **(ios_base::cur, ios_base::end, ios_base::beg**)
- zjištění polohy v souboru **tellp** (pro výstup) a **tellg** (pro vstup)
- **ignore** – pro přesun o daný počet znaků, druhým parametrem může být znak, na jehož výskytu se má přesun zastavit. na konci souboru se končí automaticky
-

implementace ve třídách

- třída je nový typ – aby se chovala standardně – přetížení << a >> pro streamy
-

```
istream& operator >> (istream &s, komplex &a ) {

char c = 0;
s >> c; // levá závorka
s >>a.re>>c;//reálná složka a oddělovací čárka
s>>im>>c;//imaginární složka a konečná závorka
 return s;
}

ostream &operator << (ostream &s, komplex &a ) {
s << ‘ ( ’ << a.real << ‘,’ << a.imag << ‘) ‘;
return s;
}
```

```
template<class charT, class Traits>
basic_ostream<charT, Traits> &
operator <<(basic_ostream <charT, Traits>& os,
const Komplex & dat)
```

obecná realizace

- streamy jsou realizovány hierarchií tříd, postupně přibírajících vlastnosti
- zvlášť vstupní a výstupní verze
- ios*_base – obecné definice, většina enum konstant (dříve ios), bázová třída nezávislá na typu – ios*base
- **streambuf** – třída pro práci s bufery – buďto standardní, nebo v konstruktoru dodat vlastní (pro file dědí filebuf, pro paměť strstreambuf, pro konzolu conbuf …)(streamy se starají o formátování, bufery o transport dat), pro nastavení (zjištění) buferu rdbuf
- istream, ostream – ještě bez buferu, už mají operace pro vstup a výstup (přetížené << a >>) – iostream
- iostream = istream + ostream (obousměrný)
- ifstream, ofstream – pro práci s diskovými soubory, automaticky buffer, fstream.h
- istrstream, ostrstream, strstream – pro práci s řetězci, paměť pro práci může být parametrem konstruktoru – strstream.h
- třídy xxx_withassign – rozšíření (istream, ostream) , přidává schopnost přesměrování (např. do souboru,) – cin, cout
- constream – třída pro práci s obrazovkou, clrscr pro mazání, window pro nastavení aktuálního výřezu obrazovky …
- nedoporučuje se dělat kopie, nebo přiřazovat streamy
- stav streamu je možné kontrolovat i pomocí **if (!sout)**, kdy se používá přetížení operátoru !, které je ekvivalentní **sout.fail(),** nebo lze použít **if (sout),** které používá přetížení **operátoru ()** typu **void *operator ()**, a který vrací **!sout.fail().** (tedy nevrací good).

3.37 Shrnutí tříd

```
//======= komplex2214p.cpp - kód aplikace ============

#include "komplex2214.h"

char str1[]="(73.1,24.5)";
char str2[]="23+34.2i";

int main ()
{
Komplex a;
Komplex b(5),c(4,7);
Komplex d(str1),e(str2);
Komplex f=c,g(c);
Komplex h(12,35*3.1415/180.,Komplex::eUhel);
Komplex::TKomplexType typ = Komplex:: eUhel;
Komplex i(10,128*3.1415/180,typ);

d.PriradSoucet(b,c);
e.Prirad(d.Prirad(c));
d.PriradSoucet(5,c);
d.PriradSoucet(Komplex(5),c);

e = a += c = d;
a = +b;
c = -d;
d = a++;
e = ++a;
```

```
if (a == c)    a = 5;
else           a = 4;

if (a > c)     a = 5;
else           a = 4;

if (a >= c)    a = 5;
else           a = 4;

b = ~b;
c = a + b + d;
c = 5 + c;

int k = int (c);
int l = d;

float m = e; // pozor - použije jedinou možnou konverzi a to přes int

//?? bool operator&&(Komplex &p) {}
if (a && c) // musí být implementován - neni-li konverze (např. zde
// se proháčkuje přes konverzi int, kde je && definována)
     e = 8; // u komplex nesmysl

Komplex n(2,7),o(2,7),p(2,7);
n*=o;
p*=p; // pro první realizaci n*=n je výsledek n a p různy i když
// vstupy jsou stejné

if (n!=p)  return 1;
return 0;
}

//======= komplex2214.h - hlavička třídy ===========
// trasujte a divejte se kudyma to chodi, tj. zobrazte *this, ...
```

```
// objekty muzete rozlisit pomoci indexu

#ifndef KOMPLEX_H
#define KOMPLEX_H

#include <math.h>

struct Komplex {
enum TKomplexType {eSlozky, eUhel};
static int Poradi;
static int Aktivnich;
double Re,Im;
int Index;

Komplex(void) {Re=Im=0;Index = Poradi;Poradi++;Aktivnich++; }
inline Komplex
                (double re,double im=0, TKomplexType kt = eSlozky);
Komplex(const char *txt);
inline Komplex(const Komplex &p);
~Komplex(void) {Aktivnich--;}

void PriradSoucet(Komplex const &p1,Komplex const &p2)
                  {Re=p1.Re+p2.Re;Im=p1.Im+p2.Im;}

Komplex Soucet(const Komplex & p)
                 {Komplex pom(Re+p.Re,Im+p.Im);return pom;}

Komplex& Prirad(Komplex const &p)
                  {Re=p.Re;Im=p.Im;return *this;}

double faktorial(int d)
                   {double i,p=1; for (i=1;i<d;i++) p*=i; return p; }

double Amplituda(void)const
```

```
                    {   return sqrt(Re*Re + Im *Im);}

bool JeMensi(Komplex const &p)
                    {return Amplituda() < p.Amplituda();}

double Amp(void) const;

bool JeVetsi(Komplex const &p)
                    {return Amp() > p.Amp();}

// operatory
Komplex & operator+ (void)
                    {return *this;}
// unární +, může vrátit sám sebe, vrácený prvek je totožný s prvkem, //
který to vyvolal

Komplex operator- (void)
                    {return Komplex(-Re,-Im);}
// unární -, musí vrátit jiný prvek než je sám

Komplex & operator++(void)
                    {Re++;Im++;return *this;}
// nejdřív přičte a pak vrátí, takže může vrátit sám sebe
// (pro komplex patrně nesmysl)

Komplex operator++(int) {Re--;Im--;return Komplex(Re-1,Im-1);}

//vrací původní prvek, takže musí vytvořit jiný pro vrácení

Komplex & operator= (Komplex const &p)
                    {Re=p.Re;Im=p.Im;return *this;}
// bez const v hlavičce se neprelozi nektera přiřazeni,
// implementováno i zřetězení
```

```
Komplex & operator+=(Komplex &p)
                    {Re+=p.Re;Im+=p.Im;return *this;}
// návratový prvek je stejný jako ten, který to vyvolal, takľe se dá
// vrátit sám

bool operator==(Komplex &p)
      {if ((Re==p.Re)&&(Im==p.Im)) return true;else return false;}

bool operator> (Komplex &p)
               {if (Amp() > p.Amp()) return true;else return false;}
// můľe být definováno i jinak

bool operator>=(Komplex &p)
               {if (Amp() >=p.Amp()) return true;else return false;}

Komplex operator~ (/*Komplex &p*/ void)
               {return Komplex(Re,-Im);}

// bylo by dobré mít takové operátory dva jeden, který by změnil sám //
prvek a druhý, který by prvek neměnil

Komplex& operator! ()
               {Im*=-1;return *this;}; // a tady je ten operátor
// co mění prvek. Problém je, ľe je to nestandardní pro tento operátor
// a zároveň se mohou plést. Takľe bezpečnějąí je nechat jen ten první
// bool operator&&(Komplex &p) {}

Komplex operator+ (Komplex &p)
                    {return Komplex(Re+p.Re,Im+p.Im);}

Komplex operator+ (float f)
                     {return Komplex(f+Re,Im);}

Komplex operator* (Komplex const &p)
```

```
        {return Komplex(Re*p.Re-Im*p.Im,Re*p.Im + Im * p.Re);}

Komplex &operator*= (Komplex const &p)
// zde je nutno pouľít pomocné proměnné, protoľe
// je nutné pouľít v obou přiřazeních obě proměnné
                 {double pRe=Re,pIm=Im;
                 Re=pRe*p.Re-Im*p.Im;Im=pRe*p.Im+pIm*p.Re;
                return *this;}
// ale je to ąpatně v případě, ľe pouľijeme pro a *= a;, potom první
// přiřazení změní  i hodnotu p.Re a tím nakopne výpočet druhého
// parametru (! i kdyľ je konst !)

// {double pRe=Re,pIm=Im,oRe=p.Re;
// Re=pRe*p.Re-Im*p.Im;Im=pRe*p.Im+pIm*oRe;return *this;}

// verze ve ktere přepsání Re sloľky jiľ nevadí
// friend Komplex operator+ (float f,Komplex &p); //není nutné
// pokud nejsou privátní proměnné

operator int(void)
            {return Amp();}
};

inline Komplex::Komplex(double re,double im, TKomplexType kt )
{
Re=re;
Im=im;
Index=Poradi;
Poradi++;
Aktivnich++;

if (kt == eUhel)
   {Re=re*cos(im);Im = Re*sin(im);}
}
```

```
Komplex::Komplex(const Komplex &p)
{
Re=p.Re;
Im=p.Im;
Index=Poradi;
Poradi++;
Aktivnich++;
}

#endif

//======= komplex2214.cpp - zdrojový kód třídy ===========
// trasujte a divejte se kudyma to chodi, tj. zobrazte *this, ...
// objekty muzete rozlisit pomoci indexu

#include "komplex2214.h"

int Komplex::Poradi=0;
int Komplex::Aktivnich=0;

Komplex::Komplex(const char *txt)
{
/* vlastni alg */;
Re=Im=0;
Index = Poradi;
Poradi++;
Aktivnich++;
}

double Komplex::Amp(void)const
{
return sqrt(Re*Re + Im *Im);
}
```

```
Komplex operator+ (float f,Komplex &p)
{
return Komplex(f+p.Re,p.Im);
}
```

4.1 dědění

- jednou ze základních vlastností objektového programování je myšlenka znovupoužitelnosti kodu – dědění. Nový objekt (třída) může vzniknout na základě jiné, jako její nástavba. Nový objekt získává vlastnosti původní a sám definuje pouze odlišnosti.
- ”znovupoužití” kódu (s drobnými změnami)
- odvození tříd z již existujících
- převzetí a rozšíření vlastností, sdílení kódu
- dodání nových proměnných a metod
- ”překrytí” původních proměnných a metod (zůstávají)
- původní třída – bázová, nová – odvozená
- při dědění se mění přístupová práva proměnných a metod bázové třídy v závislosti na způsobu dědění
- class C: public A – A je bázová třída, public značí způsob dědění a C je název nové třídy
- způsob dědění u třídy je implicitně private (a nemusí se uvádět), u struktury je to public (a nemusí se uvádět) class C: D
- tabulka ukazuje jak se při různém způsobu dědění mění přístupová práva bázové třídy (A) ve třídě zděděné

class Base
{
public:
Metoda1();
Metoda2();
Metoda3();
}
	class Dedeni:public Base {
	public:
	
	Metoda2();
	Metoda3()
	{Base::Metoda3();}
	Metoda4();
	}
		
		// necháme původní metodu z báze
		//vytvoříme novou metodu,původní je skrytá
		// doplníme původní metodu
		// vytvoříme novou metodu

class A	**class B:private A**	**class C:protected A**	**class D:public A**
public a	private a	protected a	public a
private b	-	-	-
protected c	private c	protected c	protected c

- nová třída má vše co měla původní. K ní lze přidat nová data a metody. Stejné metody v nové třídě překryjí původní – mají přednost (původní se dají stále zavolat).
- postup volání konstruktorů - konstruktor bázové třídy, konstruktory lokálních proměnných (třídy) v pořadí jak jsou uvedeny v hlavičce, konstruktor (tělo) dané třídy
- destruktory se volají v opačném pořadí než konstruktory

```
class Base {
   int x;
  public:
  float y;
  Base( int i ) : x ( i ) { };
// zavolá konstruktor třídy a pak proměnné,
// x(i) je kontruktor pro int
}

class Derived : Base {
 public:
 int a ;
  Derived ( int i ) : a ( i*10 ) : Base (a)
{ }
// volání lokálních konstruktoru umožní
// konstrukci podle požadavků, ale nezmění
// pořadí konstruktorů
 Base::y;  // je možné takto vytáhnout
// proměnnou (zděděnou zde do sekce private)
```

```
// na jiná přístupová práva (zde public)
}

------------

class base {
base (int i=10) …
}
class derived:base {
complex x,y; ….

public: defived() : y(2,1) {f()  … }
```

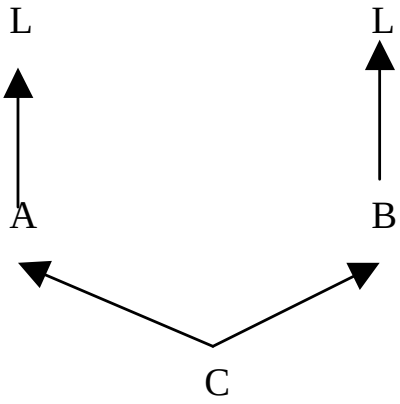
volá se base::base()
complex::complex(void) - pro x
complex::complex(2,1) – pro y
f() … - vlastní tělo konstruktoru

- nedědí se: konstruktory, destruktory, operátor =
- ukazatel na potomka je i ukazatelem na předka
- implicitní konstruktor v private zabrání dědění, tvorbu polí
- destruktor v private zabrání dědění a vytváření instancí
- kopykonstruktor v private zabrání předávání (a vracení) parametru hodnotou
- operátor = v private zabrání přiřazení

4.2 vícenásobné dědění

- v případě, že je výhodné aby objekt dědil ze dvou (či více) C++ toto dovoluje (jedná se ovšem o komplikovaný mechanizmus)
- lze dědit i z více objektů najednou
- problémy se stejnými názvy – nutno rozlišit
- problémy s vícenásobným děděním stejných tříd - virtual
- nelze dědit dvakrát ze stejné třídy na stejné úrovni C:B,B

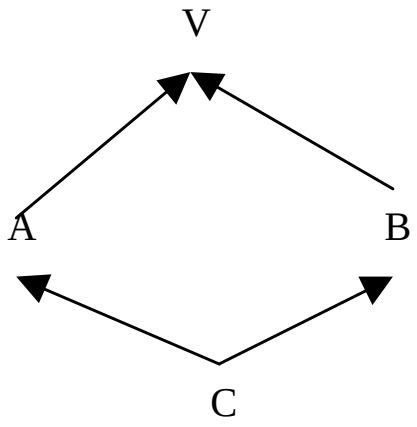
A: public L
B: public L
C: public A, Public B



v C je A::a a B::a

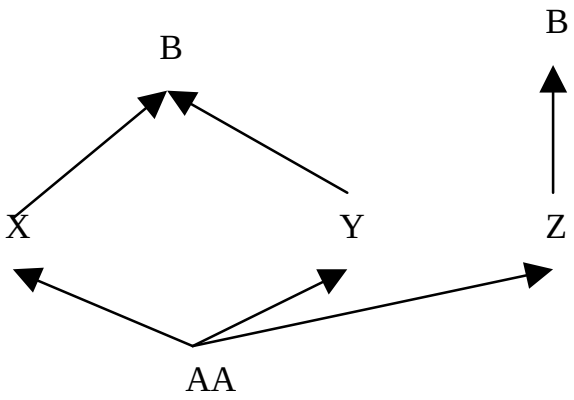
- - - - -

A: virtual V
B: virtual V
C: public A, public B
(konstruktor V se volá pouze jedenkrát)



- - - - - - - -

X: virtual public B
Y: virtual public B
Z: public B
AA: public X, Y, Z



4.3 virtuální metody

Virtuální metody

- v dosud probraných situacích byly vždy volány funkce, které jsou známy již v době překladu. V situaci, kdy v době překladu není známa funkce, která se bude volat (volá se například funkce vykresli grafického objektu, který zadal až uživatel v době chodu programu), je nutný mechanizmus, kdy si funkci v sobě nese objekt a překladač předá řízení na adresu, kterou najde v objektu – k tomu slouží virtuální metody
- zajišťují tzv. pozdní vazbu, tj. zjištění adresy metody až za běhu programu pomocí tabulky virtuálních metod,
- tvm - se vytváří voláním konstruktoru.
- V ”klasickém” programování je volaná metoda vybrána již při překladu překladačem na základě typu proměnné, funkce či metody, která se volání účastní.
- U virtuálních metod není důležité, čemu je proměnná přiřazena, ale jakým způsobem vznikla – při vzniku je jí dána tabulka metod, které se mají volat. Tato tabulka je součástí prvku.
- jsou-li v bázové třídě definovány metody jako virtual, musí být v potomcích identické
- ve zděděných třídách není nutné uvádět virtual
- metoda se stejným názvem, ale jinými parametry se stává nevirtuální, tedy statickou
- pokud je virtual v odvozené třídě a parametry se liší, pak se virtual ignoruje

- virtuální metody fungují nad třídou, proto nesmí být ani static ani friend
- i když se destruktor nedědí, může být virtuální
- Využívá se v situaci, kdy máme dosti příbuzné objekty, potom je možné s nimi jednat jako s jedním (Např. výkres, kresba – objekty mají parametry, metody jako posun, rotace, data … Kromě toho i metodu kresli na vykreslení objektu)

```
class A {
virtual Metoda (){cout << “a”;}
}
class B:A{
virtual Metoda() {cout << “b”;}
}
class C:A{
virtual Metoda() {cout << “c”;}
}

fce () {
A* pole[2];
B b;
C c;
pole [0] = &b; pole [1] = &c;

pole[0].Metoda(); // tiskne b – podle vzniku ne podle toho čemu je přiřazeno
pole[1].Metoda(); // tiskne c
}
```

- Společné rozhraní – není třeba znát přesně třídu objektu a je zajištěno (při běhu programu) volání správných metod – protože rozhraní je povinné a plyne z bázové třídy.

- Virtuální f-ce – umožňují dynamickou vazbu (late binding) – vyhledání správné funkce až při běhu programu.
- Rozdíl je v tom, že se zjistí při překladu, na jakou instanci ukazatel ukazuje a zvolí se virtuální funkce. Neexistuje-li, vyhledává se v rodičovských třídách.
- Musí souhlasit parametry funkce.
- Ukazatel má vlastně dvě části – dynamickou – danou typem, pro který byl definován a statickou – která je dána typem na který v dané chvíli ukazuje.
- Není-li metoda označena jako virtuální – použije se statická (tj. volá se metoda typu, kterému je právě přiřazen objekt).
- je-li metoda virtuální, použije se dynamická vazba – je zařazena funkce pro zjištění až v době činnosti programu – zjistit dynamickou kvalifikaci. vazba (tj. volá se metoda typu, pro který byl vytvořen objekt)
- zavolat metody dynamické klasifikace – přes tabulku odkazů virtuální třídy

- Při vytvoření virtuální metody je ke třídě přidán ukazatel ukazující na tabulku virtuálních funkcí.
- Tento ukazatel ukazuje na tabulku se seznamem ukazatelů na virtuální metody třídy a tříd rodičovských. Při volání virtuální metody je potom použit ukazatel jako bázová adresa pole adres virtuálních metod.
- Metoda je reprezentována indexem, ukazujícím do tabulky.
- Tabulka odkazů se dědí. Ve zděděné tabulce – přepíší se adresy předefinovaných metod, doplní nové položky, žádné položky se nevypouští. Nevirtuální metoda překrývá virtuální
- Máme-li virtuální metodu v bázové třídě, musí být v potomcích deklarace identické. Konstruktory nemohou být virtuální, destruktory ano.
- Virtual je povinné u deklarace a u inline u definice (?).
- ve zděděných třídách není nutno uvádět virtual

- stejný název a jiné parametry – nevirtuální – statická vazba (v dalším odvození opět virtual)
- pokud je virtual v odvozené třídě – a parametry se liší – virtual se ignoruje
- protože virtuální f-ce fungují nad třídou, nesmí být virtual friend, ani static
- i když se destruktor nedědí, destruktor v děděné třídě přetíží (zruší) virtuální
- virtuální funkce se mohou lišit v návratové hodnotě, pokud tyto jsou vůči sobě v dědické relaci

Čisté virtuální metody

- pokud není virtuální metoda definována (nemá tělo), tak se jedná o čistou virtuální metodu, která je pouze deklarována
- obsahuje-li objekt čistou virtuální metodu, nemůže být vytvořena jeho instance, může být ale vytvořen jeho ukazatel.

```
deklarace :    class base {
                   ....
                   virtual void fce ( int ) = 0;
               }
```

- virtuální metoda nemusí být definována – v tom případě hovoříme o čistě virtuální metodě, musí být deklarována.
- chceme využít jednu třídu jako Bázovou, ale chceme zamezit tomu, aby se s ní pracovalo. Můžeme v konstruktoru vypsat hlášku a existovat. Čistější je ovšem, když na to přijde překladač – tj. použít čistě virtuální metody
- deklarace vypadá: virtual void f (int)=0 (nebo =NULL)
- tato metoda se dědí jako čistě virtuální, dokud není definována
- starší překladače vyžadují v odvozené třídě novou deklaraci a nebo definici
- obsahuje-li objekt č.v.m. nelze vytvořit jeho instanci, může být ale ukazatel

```
class B {
public: virtual void vf1() {cout <<”b”;}
void f() {cout<<”b”;}
class C:public B{
void vf1(){cout << ”c”;}   // virtual nepovinné
void f() {cout <<”c”;}}

class D: public B {
void vf1() {cout<<”d”;}
void f() {cout <<”d”;}}

B b; C c;D d;
b.f(); c.f(); d.f(); // vola normální metody třídy / tisk b c d – protože proměnné jsou typu B C D / překladač
b.vf1();c.vf1();d.vf1(); // vola virtuální metody třídy / tisk b c d – protože proměnné vznikly jako B C D/ runtime
B* bp = &c; // ukazatel na bázovou třídu (přiřazení může byt i v ifu, a pak se neví co je dal)
bp->f();  // volá normální metodu třídy B / tisk b, protože proměnná je typu B / překladač
bp -> vf1(); // vola virtuální metodu / tisk c – protože proměnná vznikla jako typ c / runtime
```

4.4 abstraktní bázové třídy

- Při tvorbě tříd někdy potřebujeme, aby bylo možné pracovat se třídami stejným principem, tj. aby se třídy „zvenku“ chovaly stejně. To je aby jejich část volání měla povinně určité metody. K tomu slouží abstraktní bázový typ, který slouží pouze jako základ pro potomky, ale sám se nevyužívá.
- má alespoň jednu čistou virtuální metodu (C++ přístup)
- neuvažuje se o jejím použití (tvorba objektu)

- obecně třída, která nemá žádnou instanci (objektový přístup)
- slouží jako společná výchozí třída pro potomky
- tvorba rozhraní

```
class X {
  public:
  virtual void f()=0;
  virtual void g()=0;
  void h() ;
}

X b; // nelze

class Y: X {
  void f() {}
}

Y b; opet nelze

class Z:  Y{
  void g(){}
}

Z c; uz jde
c.h() z X
c.f() z Y
c.g() z Z
```

4.5 šablony (template)

- dost často napíšeme kód a následně zjistíme, že bychom ho potřebovali několikrát, přičemž jediné čím se liší je typ proměnné se kterou pracuje (například funkce max, lineární seznam ...). Toto může řešit princip šablon, kdy se napíše kod pro obecný typ a

překladač si potom vygeneruje podle něj kod pro typ, který potřebuje.

- umožňují psát kód pro obecný typ
- vytvoří se tak návod (předpis, šablona) na základě které se konkrétní kód vytvoří až v případě potřeby pro typ, se kterým se má použít
- umisťuje se do hlavičkového souboru (předpis, netvoří kód). Do samostatného souboru lze za použití klíčového slova export template …

```
template <typename T> T max ( T &h1, T &h2 )
{
return (h1>h2) ? h1 : h2 ;
}

double d,e,f;
d = max(e,f);
```

- template – klíčové slovo říkající, že se jedná o předpis
- použitelné pro každý typ, který je “schopen” prováděných operací (v příkladu typ, který má definován operátor > a kopykonstruktor)
- <class T> starý zápis nově <typename T>, určuje název typu, pro který se bude vytvářet. T v dalším zastupuje typ
- konkrétní typ T se zjistí při použití, zde double, proto je vytvořeno max, kde na místě T se objeví double
- díky přetížení je možné na základě template vytvoření funkce (či třídy) pro různé typy
- vytvoření je možné i ”silou” – např. deklarací int max(int, int);
- lze i více obecných typů, lze i template pro třídu

```
template < class T, class S >
double max ( T h1, S h2 )
```

```
template <class T, int nn=10> …
```

- výrazový parametr nn, pokud použijeme nn, pak se nahradí zadaným číslem (nebude-li zadání, potom hodnotou 10) <int, 22>

```
template < class T >
class A {
T a , b ;
T fce ( double, T, int )
}

T A<class T>:: fce (double a, T b,int c) {}
```

potom

```
A <double>c, d;
```

bude c,d třídy A, kde a,b jsou double

```
A <int> g, h;
```

bude g,h třídy A, kde a,b jsou int

- specifikace jména typu získaného z parametru šablony

```
template<class T>
class X {
typedef typename T::InnerType Ti;
// synonymum pro typ uvnitř T
int m(Ti o) { ..... }
}
```

- šablony lze i přetěžovat – vytvořit specializaci pro daný typ (pro ostatní typy se volá šablona původní)

```
template <> TSpec <int>
```

4.6 výjimky (exceptions)

- v případě, že nastane chyba, je nutné tuto situaci vyřešit. Ve složitějších algoritmech ovšem můžeme být zanořeni do několika funkcí a řešit (například vypsat dialog) chybu můžeme až o několik funkcí „dál“. K tomu abychom se z místa chyby do místa řešení dostali elegantně slouží mechanizmus výjimek
- nový způsob ošetření chyb (v modulu)

- chyby je nutné řešit, ne vždy je možné to učinit v místě, kde vznikly
- mezi místem chyby a jejím řešením může být několik funkcí (a tedy hodně proměnných)
- výjimka je objekt, který se vytvoří (”hodí”, pomocí klíčového slova throw) v místě chyby (na základě testu chybového stavu if …).
- nese v sobě informaci o typu a příčině chyby (parametr konstruktoru)
- ”legálně” ukončuje funkce (včetně rušení proměnných - destruktory) až do místa kde je ”chycena” (catch)
- po odchycení je možné vybrané výjimky řešit
- blok ve kterém se mají výjimky odchytávat je třeba označit (try-catch)
- catch může být pouze po bloku začínajícím try nebo za jiným catch
- výjimku řeší nejbližší catch
- není-li výjimka zachycena je program ukončen (terminated), pomocí funkce terminate (lze ji předefinovat pomocí set_terminate), která ukončí program
- výjimka se nadefinuje ve třídě, jíž se týká class V {… }
- při chybě se vytvoří objekt třídy výjimky pomocí throw V(), popřípadě s parametrem, který blíže popíše chybu
- výjimka se dá odchytit, je-li v bloku

```
try
   {
…
volání funkce, která hodí/vyvolá výjimku
…
} catch(X:V) {zde je řešení pro výjimku X:V}
catch (X:V2) {zde je řešení pro výjimku X:V2}
```

- výjimku vyřeší první příslušné catch
- lze chytat i více výjimek

- lze chytat i postupně catch(X:V) {… catch(X:V1);}
- catch (…) odchytí všechny výjimky
- catch může mít parametry typu T, const T, T&, const T&, a zachycuje výjimku stejného typu, typu zděděného, pro ukazatel T, musí se dát zkonvertovat na T
- u funkcí je možné napsat které výjimky funkce ”hází” a tím zpřehlednit a zjednodušit psaní void f() throw (v1,v2,v3) {} a jiné nesmí hodit (=abort)
- void f() {} může hodit cokoli
- void f() throw () {} nemůže hodit nic
- provádí se pouze standardní odalokování – neruší tedy objekty vzniklé pomocí new (v metodě. Vzniklé v objektu a rušené v destruktoru ruší). Proto se vytvářejí objekty pracující s pamětí a nealokuje se přímo.
- Při výjimce v konstruktoru se nevolá destruktor
-

4.6 C v C++

- může se stát, že je nutné kombinovat program ze zdrojů v C i C++, předpokládá se volání C z C++, opačná varianta je dosti krkolomná
- různé jazyky mají odlišné volání funkcí (různý způsob a pořadí pro: ”úklid” registrů, předávání parametrů, vytváření lokálních proměnných …)
- jelikož části programů mohou být napsány či přeloženy v různých jazycích (např. knihovny (dll, obj) mohou být pro pascal ….) je nutno při jejich volání zohlednit způsob jejich vytvoření.
- jako parametr v hlavičce funkce musí být pro tyto případy uveden způsob volání
- rozdílný je i způsob funkcí v C a C++
- musíme ošetřit volání funkcí v jazyce C z prostředí v C++

```
#ifdef __cplusplus
extern ”C”
```

```
#endif
{
// celý tento blok bude mít volání jazyka C
float fce(int);
…
}
```

- rozdíly je nutné zohlednit i při definici ukazatelů na funkce v části psané v C++
- C ukazatelům potom musíme přiřazovat C funkce a C++ ukazatelům C++ funkce

int (*pf) (int i) ; - C++ volání v jazyce C++ (nebo C v C)

extern ”C” {typedef int (*pcf) (int) } - C volání v C++

pcf pc; pc = &cfun; (*pc)(10);

Pravidla pro volání konstruktorů a destruktorů (automatické (definice objektů) i dynamické proměnné (vytvoření pomocí new))

1) volání konstruktorů (v každém kroku se začíná od a), pokud byl bod na dané úrovni vyřešen, pokračuje se dalším)

a) konstruktor bázové třídy (existuje-li bázová třída, a zde opět od a))

b) konstruktory objektů třídy (existují-li, v pořadí daném definicí objektů ve třídě. Pro každý objekt se provádí a) b) c)). Předepsané konstruktory v hlavičce konstruktoru neurčují pořadí ale typ.

c) vlastní tělo konstruktoru

2) volání destruktorů – je v opačném pořadí jako volání konstruktorů. Mohou být virtuální (potom se volají v opačném pořadí v jakém došlo ke konstrukci, nehledě na to, čemu je objekt v současnosti přiřazen). Pokud jsou nevirtuální, jsou destruktory volány v opačném pořadí ke konstruktorům, jaké by se volaly pro prvek třídy, které je objekt právě přiřazen. Destruktor je volán na konci definičního bloku pro proměnné v něm definované. Destruktor je volán při delete.

Pravidla pro volání (virtuálních) metod

1) zjistíme, zda-li je volaná metoda virtuální nebo nevirtuální

2) pokud je volaná metoda nevirtuální, rozhoduje “jak to vidí” překladač – neboli je volána taková metoda, která patří k typu (třídě) jak je současný objekt (ukazatel na objekt) definován.

3) pokud je volaná metoda virtuální, nerozhoduje, čemu je aktuálně prvek přiřazen, ale jak “se narodil”. Při vzniku (konstruktor) je mu

totiž virtuální metoda přiřazena na základě vznikajícího typu (a zůstává “majetkem” objektu). U objektu se při běžné činnosti nejedná o problém, protože je známo jak objekt vznikl (podle typu v definici). Důležité je to však u ukazatelů, které mohou ukazovat na cokoli (i když z hlediska daného mechanizmu je nutné dodržet to, že přiřazovat by se měl ukazatel na potomka do ukazatele na předka). Potom musíme najít, jak opravdu vznikl objekt (definice, nebo new), na který se právě ukazuje – nezávisle na množství přiřazení, které se staly.

4) Pokud metoda neexistuje, použije se metoda nejbližší (například u předka).

Zkouška

1) teoretický příkaz na mechanizmus, nebo klíčové slovo C++

2) Složitější příklad z jazyka C (struktura, soubory, pole, řetězce, vázaný seznam, bitové operace …)

3) Třída – úkolem je napsat metody tak, a by šla přeložit daná část kódu

TString obsahuje dynamicky alokované pole charů pro řetězec.
{
TString a(“abc”), b=a,c; // konstruktor z řetězce, kopykonstruktor, implicitní konstruktor
c = a + b; // přiřazení (=) spojených řetězců (+)
a = a;
int i1 = c.Delka();   // vrací délku řetězce
int i2 = VyskytZnaku(a,’b’); // vrátí počet výskytů daného znaku v řetězci
char znak = c[i-1]; // vrátí znak na dané pozici, při indexaci mimo řetězec vrací odkaz na globální proměnnou
c[2] = ‘e’; // index musí fungovat i pro přiřazení
int j = a > b;
cout << a << ” to je a ”;
}

4)
co se vypíše po spuštění tohoto programu. Tištěný text napište k příslušnému řádku funkce main, kterého se týká:

```
class A {
public:
A(void) {cout << 'a';}
virtual ~A(void) {cout << 'b';}
void f(void) {cout << 'c';}
virtual fv(void) {cout << 'd';}
};

class B:public A {
```

```
A a;
public:
B(void) {cout << 'e';}
virtual ~B(void) {cout << 'f';}
void f(void) {cout << 'g';fv();}
virtual fv(void) {cout << 'h';}
};



class C:public A {
B a;
public:
C(void) {cout << 'i';}
virtual ~C(void) {cout << 'j';}
void f(void) {cout << 'k';fv();}
};

int main () {
 A *a;                B b;
 B *c = (B*)new C;
 a = &b;
 a -> f();            a -> fv();
 c -> f();            c -> fv();
 delete c;
 b.f();               b.fv();
}
```

Hodně štěstí, zdraví, a vědomostí v novém roce