

Praktické programování v jazyce C++

Přednášející a cvičící:

Richter Miloslav

zabývá se zpracováním signálu, především obrazu. Realizoval několik průmyslových aplikací na měření nebo detekci chyb při výrobě. Řízení HW a zpracování naměřených dat realizoval převážně pomocí programů v jazyce C/C++, který je v těchto oblastech využíván díky svým vlastnostem jako je přenositelnost, rychlost, dostupnost dat, kvalitní překlad ...

Petyovský Petr

zabývá se zpracováním obrazu, především v dopravních aplikacích (počítání a detekce aut, sledování přestupků, ...). Účastnil se řady projektů, které byly ukončeny praktickou realizací v průmyslu. Programoval v různých jazycích a má rozsáhlou praxi v implementaci programů na různé platformy.

Obecné informace a organizace kurzu BPPC

Rozsah výuky:

- 6 kreditů
- přednáška - 2hod
- cvičení na počítači - 2hod
- projekt
 - zadání a jednoduché konzultace na počítačových cvičeních,
 - hlavní práce doma
 - větší konzultace v pátek v numerických cvičeních (po domluvě)
- státní svátky – přednáška odpadá, cvičení se posunou
- přesuny na cvičeních – první dva týdny – pouze výměnou

Obecné informace a organizace kurzu BPPC

Stránky kurzu a materiály:

- z e-learningu link na stránky kurzu (BPPC)
- materiály ke kurzu [www.uamt.feec.vutbr.cz/~richter \(/vyuka/XPPC/bppc\)](http://www.uamt.feec.vutbr.cz/~richter/vyuka/XPPC/bppc)
- dvoje materiály společně: denní BPPC, kombinovaný KPPC
- starší materiály stále na síti,
- materiály (skripta, doporučená literatura) v sekci „literatura a studijní materiál“
- literatura – skripta, knihy, přednášky, www
- materiály k přednáškám: v sekci „literatura ...“ jako celek, nebo v sekci „náplň/plán přednášek“ po jednotlivých tématech
- materiály ke cvičením: (výhledově) v sekci „literatura ...“ jako celek, nebo v sekci „náplň/plán cvičení“ po jednotlivých příkladech

Obecné informace a organizace kurzu BPPC

Literatura a programovací prostředky:

- materiály ke kurzu (stránky kurzu)
- „papírové“ zdroje (knihy) – velké množství; najděte si knihu, která vám „sedí“; v knize by se mělo vyskytovat v příkladech slov class (cca ve 2/3 či více knihy)
- www zdroje – nutno hodnotit jejich kvalitu (některé jsou nevyhovující)
- kromě programování i témata: kultura programování, objektové programování, algoritmy, STL, programovací styly ...

Překladač:

- MSVC Professiona 2013
- ke stažení na <http://msdnaa.feec.vutbr.cz/> <http://msdnaa.feec.vutbr.cz/>

Obecné informace a organizace kurzu BPPC

Povinnost výuky:

- přednášky – „povinné“, přítomnost nekontrolována
- přednáška s půlsestrální testem – nepovinné, nutno předem zaregistrovat v IS
- cvičení – povinné, kontrolováno, jedna neúčast omluvena automaticky
- projekt – povinné, pravidelná práce (kontakt se subversion) povinnou součástí
- DU – příklady neprobrané na cvičeních

Obecné informace a organizace kurzu BPPC

Konzultace výuky:

- konzultace a dotazy
přednášky,
cvičení,
e-learning/chat,
mail,
jabber
konzultační hodiny
- cvičící a přednášející – rovnocenní

Obecné informace a organizace kurzu BPPC

Hodnocení výuky:

- hodnocení kurzu BPPC (body a termíny)
zkouška - 50b, zkouškové období, papírově
projekt - 20b (2+3+3+6+6); doma+cvičení, průběžně, počítač),
půlsemestrální test - 30b, termín po domluvě, cca 9 týden, papírově

- hodnocení kurzu KPPC
DU + projekty - doma 35b (7x5b), odevzdání IS, počítač,
půlsemestrální test -15b, termín po domluvě (X.XX), papírově
zkouška – společně s BPPC

www stránky

http://www.uamt.feec.vutbr.cz/~richter/vyuka/XPPC/bppc/bppc_main.html

Dotazy k organizaci kurzu

??

??

???????

???????

???????

Dotazy k látce předchozích kurzů BPC1A, BPC2A

????????
???????? ????????
?? ??

Náplň kurzu

- rozsah je přizpůsoben minulým letům
- týdenní plán přednášek – orientační
- neobjektové vlastnosti – rozšíření vlastností oproti jazyku C
- objektové vlastnosti – objektové programování, dědění, polymorfismus
- „nástroje“ – výjimky, šablony, streamy, STL,

Náplň kurzu

- opakování a rozšíření jazyka C
- programátorské dovednosti
- jazyk C++

Náplň kurzu

Opakování a rozšíření jazyka C, neobjektové vlastnosti

- především na cvičeních než bude možné začít s objektovým C++
 - překlad programu, hlavičkové a zdrojové soubory
 - rozdíly „čistého C“ oproti možnostem C++ a objektovému programování
 - nové datové typy,
 - ukazatele x reference
-
- vstup a výstup - streamy
 - knihovny jazyka C, C++

Náplň kurzu

Programátorské dovednosti

- obecná pravidla programování (návrh, tvorba, překlad a testování programu)
- základní programátorské dovednosti a návyky – programátorská kultura, trasování, ... použití objektů
- nástroj pro správu (verzí) projektů svn pro spolupráci více autorů na jednom projektu
- nástroj doxygen (+graphviz) pro komentování programů a pro tvorbu dokumentace

Náplň kurzu

Jazyk C++ a objektové programování

- základní teorie, rozdíly oproti C stylu programování
- objektové vlastnosti,
- dědění,
- polymorfismus
- šablony
- STL
- ...

Výuka programovacích jazyků na tomto oboru

... a počátek devadesátých let

- Analogové programování (modelování dynamických soustav), logické obvody a programovatelné automaty, Assembler (práce s programovatelným HW), Pascal (vědecké výpočty), Prolog, Lisp (umělá inteligence)
- podle aktuální situace a oboru činnosti (dostupné překladače, drivery, programová prostředí a jejich možnosti) se některé způsoby programování a jazyky opouštějí a jiné se dostávají do popředí (programování hradlových polí, mikroprocesorů, inteligentních periferií, zpracování dat, komunikace na sítích ...)
- základem je stále logické myšlení, znalost základních nástrojů programování a jejich využití

Polovina devadesátých let

- výuka jazyka C/C++ navazující na Pascal. V **jednom semestru** výuka jazyka C, obsluha základního HW (čítače, časovače, přerušení, DMA ...), C++.

Dvacáté první století

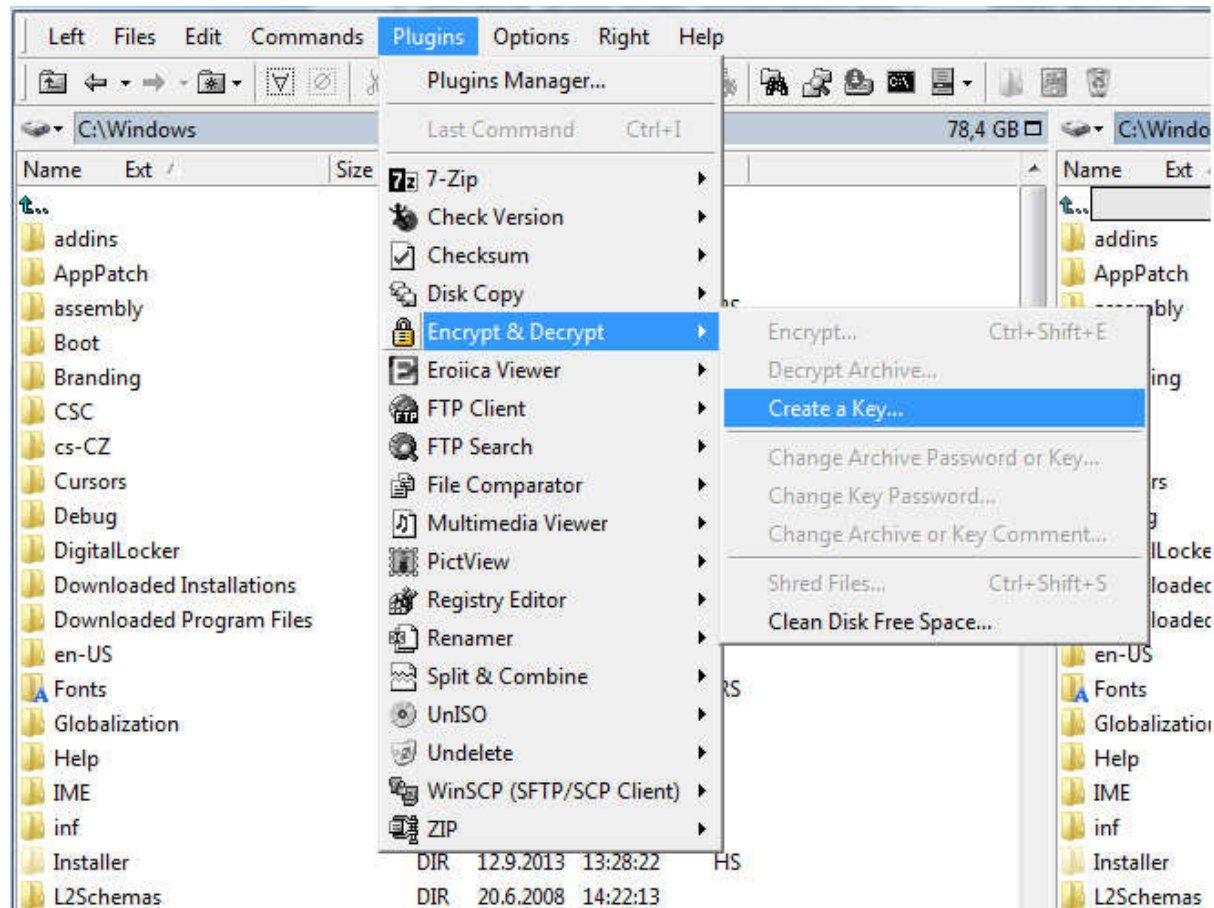
- po ukončení výuky Pascalu se C/C++ učí ve dvou semestrech.
- počátek grafického programování – "... pište aplikaci bez znalosti jazyka ... v našem prostředí napíšete aplikaci bez programování ..." – stačí pro "běžného uživatele", náš absolvent by však měl uvažovat i v souvislostech (jak a proč je implementováno, ...) a znát mechanismy hlouběji

Příklady využití jazyka C++

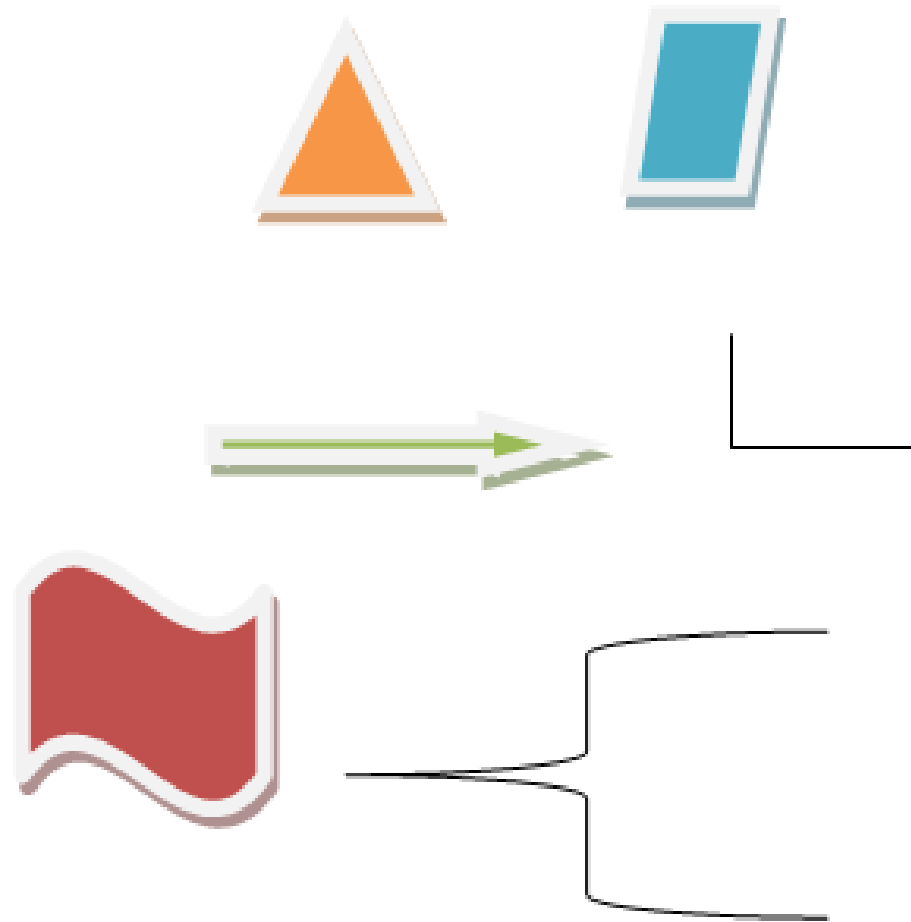
- vhodné využití: složitější projekty, znovupoužití kódu (dědění, polymorfismus, šablony)
- projekty realizované v C++:
 - Adobe Systems (Photoshop, Illustrator, Adobe Premier ...)
 - Google (Google Chromium, Google file system)
 - Mozilla (Firefox, Thunderbird)
 - MySQL (využito: Yahoo!, Alcatel-Lucent, Google, Nokia, YouTube, Wikipedia ...)
 - Autodesk Maya 3D (3D modeling, games, animation, modeling ...)
 - Winamp Media Player
 - Apple OS X (některé části a některé aplikace)
 - Microsoft (většina na bázi C++)
 - Symbian OS

Konkrétní realizace – projekty vhodné pro objektové programování

- *menu* – je struktura obsahující položky (menu, podmenu, akce ...). Struktury pro menu, položky, texty, barvy ...



- *grafické objekty* – mají společný interface (rozhraní/ přístup). I když jsou různého typu, je možné s nimi pracovat jednotně – program si "zjistí" jakého jsou typu a provede správnou akci pro daný typ (vykreslení, rotace, posun, inverze barev, zjištění který objekt je nejbližší dané pozici (kliknutí myši), ...)



- vytváření nových datových typů s operacemi jako typy standardní (ale vlastním chováním)
- například komplexní čísla, zlomky, matice
- znovupoužití kódu pro různé typy ...

ukázka kódu pro vlastní typ MATRIX (matice). "Umí" operace jako základní typy, a také je možné ho "naučit" funkce. Pro výpočty je možné zvolit přesnost výpočtu – zde double

```
MATRIX <double> y,x(5,5),a(5,1),A,x1(10,5), ykontrola;  
int i,j;
```

```
// řešení rovnice  $y = x * A$  pro neznámé A
```

```
A = SolveLinEq(x,y);
```

```
// kontrola správnosti
```

```
ykontrola = x * A;
```

```
// vypočtení kvadrátu chyby řešení
```

```
chyba = (ykontrola - y);
```

```
chyba *= chyba;
```

```
// obecné výpočty se standardními operátory a vlastními funkcemi pro
```

```
// nový typ (MATRIX)
```

```
x1 = x * A * Transp(y) * Inv(x);
```

Opakování „programování“ – předchozí kurzy

HW návaznost

- procesor – sběrnice, instrukční sada, optimalizace rychlosti, datové typy, operace (matematické, logické, podmínky, skoky, podprogram ...)
- paměti a periferie
- adresování

Tvorba programu

- návrh
- kriteria hodnocení

Programové prostředky (editor, překladače, ladící prostředky, sestavení programu)

Jazyk

- klíčová slova
- datové typy
- základní mechanizmy jazyka

Procesor

- má určitý počet instrukcí (příkazy)
- instrukce říká, co a s čím se má udělat
- instrukce trvá určitý počet cyklů (času, přístupu k paměti ...)
- obsahuje registry (vnitřní paměti)
- akumulátor – výpočetní jednotka (ALU)
- je schopen pracovat s určitými datovými typy
- čítač instrukcí říká, kde leží další instrukce (ovlivňují ho instrukce skoků (podmíněné/nepodmíněné)), cykly
- podprogram (call/return) – zásobník
- interrupt (přerušeni) - volatile proměnné
- registr příznaků – výsledky operací (nulovost, kladnost, přetečení ...)

Paměť

- v paměti jsou uloženy instrukce a data programu
- program obsahuje instrukce, které se vykonávají
- datová oblast příslušná programu – základní data pro proměnné programu
- zásobník – lokální data, adresy při podprogramech
- statické a globální proměnné v datové části programu (inicializace)
- „volná“ datová oblast – je možné o paměť z ní požádat „systém“
- mapování periférií do paměti – data se mění "nezávisle" – volatile proměnné
- cache paměť na čipu – podstatně rychlejší přístup k datům

Datové typy (vázané na procesor, nebo emulované v SW)

- celočíselné – znaménkové x bezznaménkové (zápis binárně, oktalově, dekadicky, hexadecimálně)
- s desetinnou čárkou
- podle typu procesoru a registru (spojení registrů) je dána přesnost (velikost typu v bytech)
- adresa x ukazatel
- pro adresování (segment:offset, indexovaný přístup ...)
- pro vyjádření znaku se využívá celočíselná proměnná – teprve její interpretací (například na tiskárně) „vznikne“ znak.
- Základní znaková sada (ASCII, EBCDIC) je osmibitová
- Rozšířená znaková sada UNICODE

Matematické operace

- Sčítání, odčítání – základ (celočíselné)
- Násobení, dělení
- Mocniny, sinus, cos, exp ... jsou většinou řešeny podprogramy, nebo pomocí tabulek (a interpolací). Jsou součástí knihoven ne jazyka.

Boolovské operace

- použití pro vyhodnocování logických výrazů
- Tabulka základních logických funkcí pro kombinace dvou proměnných

0	0	1	1	vstup A - nabývá hodnoty
0	1	0	1	vstup B - nabývá hodnoty
0	0	0	0	nulování
0	0	0	1	AND
0	0	1	0	přímá inhibice (negace implikace) - Nastane-li A, nesmí nastat B.
0	0	1	1	A
0	1	0	0	zpětná inhibice
0	1	0	1	B
0	1	1	0	XOR nonekvivalence (jsou-li proměnné různé je výsledkem 1, jsou-li stejné, pak 0)
0	1	1	1	OR
1	0	0	0	negace OR
1	0	0	1	negace XOR (výsledek je 1, pokud jsou proměnné stejné, pokud jsou různé pak je výsledek 0)
1	0	1	0	negace B
1	0	1	1	zpětná implikace
1	1	0	0	negace A
1	1	0	1	přímá implikace (nastane-li stav A, je výsledek řízen stavem B. Z nepravdy A nemůžeme usoudit na stav B – mohou být platné oba stavy (nebude-li přšet, nezmoknem). Pokud platí A je možné z výsledku usuzovat na B (B je stejné jako výsledek) pokud A neplatí nelze o vztahu výsledku a B nic říci.
1	1	1	0	negace AND
1	1	1	1	nastavení do jedničky

Způsoby „adresování“

- Součást instrukce - INC A (přičti jedničku k registru A) – registr, se kterým se pracuje je přímo součástí instrukce
- Přímý operand – JMP 1234 – skoč na danou adresu – je uvedena v paměti za instrukcí. Může mít i relativní formu k současné pozici
- Adresa je uvedena jinde (v jiné proměnné) – PUSH B – registr B se uloží na zásobník, LD A, BC – do registru A se načte hodnota z adresy ve dvojici registrů BC
- Indexové adresování MOVI A,[BC+IX] – do registru A se načte hodnota z paměti, která je posunuta o IX (index) od adresy v registru BC (báze).

Programování

- Rozbor úlohy – které funkce patří k sobě (knihovny), rozhraní funkcí (předávané a návratové hodnoty), datové typy pro proměnné
- Algoritmy – řešení daného úkolu ve funkci
- Zapsání kódu
- překlad – „jazyková“ správnost
- Ladění kódu – debugging – „funkční“ správnost
- Testovací databáze

Postup programování

- požadované vlastnosti
- návrh činnosti
- návrh datových struktur
- návrh funkčních volání

Hodnocení programu

- Výkon a efektivita – čas, využití zdrojů
- Spolehlivost – HW, SW (na podněty musí správně reagovat)
- Robustnost – odolnost proti rušení (HW, SW, uživatel)
- Použitelnost – jak je „příjemný“ pro uživatele, jak snadno se zapracovává do programu
- Přenositelnost – jak velké úpravy je nutné dělat při překladu na jiné platformě (jiným překladačem) – jazyk, použité funkce, návaznost na OS, velikost datových typů, endiány ...
- Udržovatelnost – dokumentace, komentáře, přehlednost
- Kultura programování – programátorský styl, komentáře (popisují proč je to tak), dokumentace

Programovací prostředí

- Editor – vytvoření zdrojových a hlavičkových souborů (co to je, jaká je mezi nimi vazba)
- Překladač + preprocesor – direktivy preprocesoru #xxx, překlad do mezikódu, kontrola syntaktických chyb
- Linker – spojení částí programu (.o, .obj, .lib, .dll, ...) do jednoho celku (.exe, .lib, .dll, ...)
- knihovny (.c, .cpp, .o, .obj, .lib, .dll) předpřipravené části kódu, které zjednodušují psaní programu. Jejich rozhraní je oznámeno v hlavičkovém souboru.
- Debugger – je možné hledat chyby v programu. Trasování – procházení programu po krocích nebo částech s možností zobrazení hodnot proměnných nebo paměťových míst
- Projekt – sada souborů, jejichž zpracováním vznikne výsledek (exe, dll, ...)
- Řešení (solution) - sada společných projektů
- Překlad – kompilace (zpracování zdrojových souborů); linkování (sestavení programu), build (kompilace změněných souborů a linkování); build all, rebuild (kompilace všech souborů a linkování); reset, clean, clear (smazání všech souborů (meziproduktů) překladu)

Opakování jazyka C

Imperativní programování – popisujeme kroky, které má program vykonat

Strukturovanost programu – „grafická“ v rámci funkcí, programátorský styl, (firemní) kultura programování, program realizován pomocí funkcí (předávání parametrů),

Klíčová slova - cca 35 klíčových slov

void

char, short (int), int, long (int)

signed, unsigned

float, double, (long double)

union, struct, enum

auto, register, volatile, const, static

extern, typedef

sizeof

if, else, switch, case, default, break – (podmíněné větvení)

goto

return

for, while, do, continue, break - cykly a skoky

operátory (matematické a logické , přiřazení (možnost zřetězení), ternární operátor)

Datové typy

- datové typy – udávají přesnost, typ, znaménko, modifikátory
- velikost vázána na platformu (sizeof)
- celočíselné neznaménkové – pro logické operace (bitové, posuny...)
- složené datové typy (struktury, union)
- ukazatel – adresa spojená s typem, který na ní leží, ukazatelová aritmetika
- pole – návaznost na ukazatel, řetězce C (typ string)
- alokace paměti – automatické proměnné, dynamické proměnné (kde leží), globální proměnné, definice a deklarace (inicializace)
- výčtový typ enum

- psaní konstant (znaková 'a'; celočíselná -12, 034, 0xFA8ul; neceločíselné 23.5, 56e-3; pole "ahoj pole"
- escape sekvence \n \r \t \a \0 \0x0D
- konverze datových typů implicitní a explicitní
- typedef

Boolovská logika

- použití logické (proměnná je brána jako celek nula/nenula) x matematické (po bitech)
- využití pro maskování
- spojeno s neznaménkovými celočíselnými typy
- hodnoty používané k vyjádření logické proměnné
- operace bit po bitu (nad bitovými řezy, bitwise) a s celým číslem (konverzí na bool)

Funkce

- návratová hodnota – jak se předává
- parametry funkce – lokální parametry, předávání hodnotou (využití ukazatele), předávání polí (v závislosti na definici)
- lokální parametry funkcí
- funkce main – musí mít návratovou hodnotu int, může mít parametry
- funkce pro (formátovaný) vstup a výstup – standardní vstupy a výstupy stdin, stdout, stderr;

Úvod C++

jazyk C++ základní popis

- jazyk vyšší úrovně
- nová klíčová slova a mechanismy
- přenositelný kód
-

jazyk C++ programovací mechanizmy

- rozšiřuje programovací možnosti C (neobjektové vlastnosti)
- přidává objektové vlastnosti (program objektově orientován, lze psát i "standardně")
- dědění – znovupoužití a rozšíření/modifikace kódu
- šablony – znovupoužití kódu (pro různé datové typy), STL
- výjimky – nový způsob ošetření chyb

jazyk C++ návaznost na jazyk C

- existují nezávislé normy C a C++ - C (C99, C11) a C++ (C++98/03, C++11, C++14), embedded C
- C může mít některé vlastnosti navíc, až na výjimky lze říci, že C je podmnožinou C++
- v některých vlastnostech je C++ přísnější (trvá na přesném dodržování pravidel)
- C přijímá některé (neobjektové) vlastnosti z C++

jazyk C++ zdrojové soubory

- zdrojový kód (nejčastěji) přípona ".cpp".
- díky dvou normám dva překladače. Jeden pro C, druhý pro C++ (u MSVC GUI rozhoduje o použití přípona zdrojového souboru)
- hlavičkové soubory bez přípony nebo ".h", ".hpp", ".hxx",
- „staré“ hlavičkové soubory (například stdio.h) mají obdoby s předponou c a bez rozšíření .h (stdio.h -> cstdio). Při použití nového souboru je jeho obsah součástí jmenného prostoru std.

Neobjektové vlastnosti - popis

- lze použít i samostatně,
- lze použít v „C programovacím stylu“ pro zlepšení komfortu programování
- hlavní využití u práce s objekty (vlastními typy)

Neobjektové vlastnosti - základ

- definice proměnné
- přetěžování funkcí
- přetěžování operátorů (operátor = speciální funkce)
- inline funkce
- implicitní parametry
- reference
- prostory jmen
- typ bool

„Neobjektové“ vlastnosti – složitější mechanismy

- výhoda jejich použití vynikne při práci s objekty
- lze s nimi pracovat pro standardní proměnné i bez znalosti objektů, i když mohou pracovat s objekty, nebo jsou dokonce na objektech postaveny
- alokace paměti (new, delete) – návaznost na inicializaci a rušení objektu
- typově orientovaný vstup, výstup - streamy
- šablony
- výjimky

Objektové programování

- přináší nové možnosti a styl programování
- vystavěn na složeném datovém typu (struct)
- spojení dat a funkcí/metod pro práci s nimi
- ochrana dat (přístupová práva)
- výsledný mechanismus (spojení dat, metod a práv) nazýváme zapouzdřením
- zlepšuje "kulturu" programování – automatická inicializace, ukončení života proměnné, chráněná data ...

Komentáře (no = NeObjektová vlastnost)

- v původním C pouze víceřádkové komentáře /* */
- vnořené komentáře nelze používat - /* /* */ */
- v C++ navíc jednořádkový komentář: // až po konec řádku
- // již i v C99

```
int k;      // nový řádkový komentář
// komentář může začínat kdekoli,
int i ;    /* původní víceřádkový typ lze stále použít pro
rozsáhlejší komentáře */
```

Napište funkci pro výpočet obvodu čtverce a okomentujte ji

```
/** \brief Vypocet obvodu ctverce na zaklade delky strany.  
\param[in] aStrana Delka jedne strany  
\return      Vraci vypocteny obsah ctverce  
\attention Tato funkce byla napsana pro hru XXX, ktera ma  
ctvercovy rastr (celá čísla) a proto vypocty probihaji nad  
typem int.  
*/  
int ObvodCtverce(int aStrana)  
{  
    int /*takhle ne*/ Vysledek;  
  
    // ctverec ma ctyri stejne strany  
    Vysledek = 4 * aStrana;  
    return Vysledek;  
}
```

přetěžování funkcí (no)

co se stane v jazyce C při překladu následujícího kódu?

```
int ObvodCtverce(int aStrana)
{
    int Vysledek;
    Vysledek = 4 * aStrana;
    return Vysledek;
}
```

```
double ObvodCtverce(double aStrana)
{
    double Vysledek;
    Vysledek = 4 * aStrana;
    return Vysledek;
}
```

přetěžování funkcí (no)

- v C může být jen jediná funkce s daným jménem
- v C++ může být více stejnojmenných funkcí – přetěžování (viz. minulý příklad)
- přetěžování není myšleno ve smyslu překrytí (znemožnění přístupu k původní, ale přidání další funkce se stejným jménem na stejné úrovni)

```
int f(int);           // první z funkcí  
int f(int, int);     // přetížení – stejné jméno,  
// jiný počet parametrů
```

```
// použít lze obě  
a = f(b);  
c = f(d,e);
```

přetěžování funkcí (no)

- při volání vybírá překladač z přetížených funkcí/metod na základě kontextu (překladač si funkce interně „pojmenuje“ tak, že k názvu přidá typ parametrů např. @fYYXX, takže funkce pro něj potom nemají stejný název)
- funkce musí být odlišené: počtem nebo typem parametrů, prostorem,
- typ návratové hodnoty se nerozlišuje (nelze rozlišit pouze návratovou hodnotou)

```
int f(int);      // první z přetížených funkcí  
float f(int);   // nelze - návratová hodnota neodlišuje  
float f(float); // lze rozlišit - jiný typ parametru  
float f(float, int) // lze rozlišit - jiný počet parametrů
```

přetěžování funkcí (no)

- při vyhledávání má přednost "nejbližší", jinak musíme uvést celý přístup - `Prostor::Jméno`
- problém s konverzemi

```
int f(int);    // první z přetížených funkcí
float f(int);  // nelze - návratová hodnota neodlišuje
float f(float); // lze rozlišit - jiný typ parametru
float f(float, int) // lze rozlišit - jiný počet parametrů
```

```
float ff = f(3); // volání f(int)
```

```
f(3.14); // chyba - parametr double lze převést na int i float
          // a překladač nemůže jednoznačně rozhodnout
```

```
f( (float) 3.14); // s konverzí v pořádku - volá se f(float)
```

```
f(3,4.5); // OK, implicitní konverze parametrů
// volá se f(float, int) - rozlišení počtem parametrů
```

implicitní parametry (no)

- pokud je funkce volána velice často se stejným parametrem, je možné využít jeho implicitní uvedení v deklaraci (hlavičkový soubor)
- při deklaraci funkce uvedeme hodnotu parametru, která se doplní (překladačem) při volání, ve kterém není uvedena
- implicitní parametry se v deklaraci funkce používají od posledního parametru
- při volání se parametry přestávají uvádět od posledního parametru

```
int f(float a=4,float b=random()); //deklarace  
// tato funkce je použita pro volání
```

```
f(); // překladač doplní chybějící a volá f(4,random())
```

```
f(22); // překladač doplní chybějící a volá f(22,random())
```

```
f(4,2.3); // je-li zadán následující parametr různý  
// od implicitního, musí se uvést i předchozí,  
// i kdyby byl stejný jako implicitní
```

implicitní parametry (no)

- implicitní parametry se uvádí (pouze jedenkrát) v deklaraci (h. soubory)
- jako implicitní parametr nemusí být použita hodnota ale libovolný výraz (konstanta, volání funkce, proměnná ...)

```
int f(float a=4, float b=random()); //deklarace  
// tato funkce je použita pro volání
```

```
// předchozí definice funkce koliduje s (=zastupuje i)  
// následující funkce.  
// Tyto funkce nemohou být již definovány.  
// protože překladač je nedokáže odlišit.
```

```
typ f(void);  
typ f(float);  
typ f (float, float);  
// a někdy koliduje i s  
typ f(char); // nejednoznačnost při volání s parametrem int -  
// je možná konverze na char i float
```

prostory jmen (no) - úvod

- "oddělení" názvů proměnných/identifikátorů v rámci různých částí programu
- zabránění kolizí stejných jmen (u různých programátorů) - jsou-li proměnné se stejným názvem v různých jmenných prostorech, potom jejich názvy nekolidují
- použitím jmenného prostoru se přidává "příjmení" ke jménu (jméno prostoru se přidá k názvu proměnné/funkce jako příjmení a tedy celek příjmení::jméno je různý i pro stejná jména)

prostor::identifikátor

prostor::podprostor::identifikátor

- například proměnné v nové knihovně cstdio leží v prostoru std a tedy při tisku je musíme psát

std::printf("text"); // jsme-li mimo prostor std

- pokud jsme ve stejném prostoru jako volaná proměnná, nemusíme jméno prostoru používat

printf("text"); // jsme-li v prostoru std

prostory jmen (no) – použití volání

- při použití má přednost "nejbližší" identifikátor – prvně se hledá v aktuálním prostoru, potom v nadřazeném. Do prostorů se stejným jménem se bez uvedení pomocí using nedostaneme
- 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 = totální porušení ohraničení=ochrany).
- Zároveň se nedoporučuje používání using (zpřístupňování) v hlavičkových souborech = globální dosah zpřístupnění/otevření. Lépe zpřístupnit v c/cpp souborech jen tam kde je skutečně potřeba
- zpřístupnění/"dotažení" proměnné končí s koncem bloku, ve kterém je uvedeno

nechceme-li psát std:: při volání printf

```
std::printf("text");
```

musíme před použitím prostého printf uvést některý z příkazů:

```
using std::printf(char *); // pouze funkce
```

```
using namespace std; // celý prostor – horší varianta
```

prostory jmen (no) - vytvoření

- vlastní prostor definujeme pomocí klíčového slova namespace – vyhrazuje (vytváří) prostor s daným jménem
- vytváří blok společných funkcí a dat – oddělených od zbytku (jménem prostoru)
- definičních bloků se stejným názvem prostoru může být v programu (obsah prostoru může být tedy definován na více místech).
- přístup do jiných prostorů přes celý název proměnné

definice:

```
namespace XX { // začátek prostoru s daným jménem XX  
proměnné // definice proměnných patřících do prostoru XX  
funkce  
třídy  
};
```

prostory jmen (no) – použití definice

Hlavičkový soubor

```
namespace XX {  
int funkce(void); // funkce v prostoru  
struct AA { // struktura definovaná ve jmenném prostoru  
double x;  
};  
}
```

Zdrojový soubor (vlození všeho stejně jako u hlavičky, nebo lze i jednotlivě):

```
int XX::funkce(void) // nutno uvést prostor do kterého patří  
{  
AA a; // definice proměnné struktura „uvnitř“ prostoru  
  
    return 1;  
}  
  
XX::AA b; // definice proměnné struktura mimo prostor
```

prostory jmen (no) - poznámka

- pomocí **using NadřazenýProstor::x; using BázováTřída::g;** lze zpřístupnit prvek, který se "ztratil" (byl překryt či je přímo nepřístupný)
- pomocí namespace můžeme zkrátit i název vnořených prostorů

```
namespace zkratka = Petr::Prostor1::Oddíl2;
```

```
// přístup k proměnným v prostoru Oddíl z Prostor1 z Petr  
x = zkratka::promenna;
```

operátor příslušnosti :: (no)

- Jsme-li uvnitř prostoru, můžeme se odkazovat na jeho členská data, funkce a metody přímo (protože jsou v prohledávání „nejblíž“). Třída se svým obsahem je vlastně prostorem.
- pokud přistupujeme k datům nebo metodám uvnitř prostoru, či přes objekt, určuje tento prostor (třída), primární oblast, ve které hledat typ použitého objektu
- pokud nelze prostor, ve kterém se data nebo metody nacházejí jednoznačně odvodit z kontextu, musíme tento prostor explicitně uvést
- odlišení datových prostorů (a tříd) přístupem přes operátor příslušnosti ::
- použití i pro přístup k (překrytým) globálním proměnným

Prostor :: JménoProměnné

Prostor :: JménoFunkce ()

(statická) proměnná Pocet uvnitř prostoru/třídy Komplex je přístupná pomocí:

Komplex :: Pocet = 10;

bez uvedení Komplex by se jednalo o „obyčejnou“ globální proměnnou

Napište funkci, která bude mít lokální proměnnou se stejným názvem jako je název proměnné globální a ukažte v této funkci jak provést přístup k lokální i globální proměnné.

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

```
// při uvádění metody vně prostoru/třídy
// (bez vazby na objekt daného prostoru / 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
```

výjimky (exceptions)

- mechanismus ošetření chyb
- jak se dá ošetřit chyba v programu?

výjimky (exceptions)

- chyby zřejmé za překladu – test, který vyřeší kompilátor
- chyby vzniklé za chodu programu – některé je schopen kompilátor (makro `assert`, nedefinovaná proměnná...), dělení nulou, odmocnina záporného čísla, některé pomocné programy (VLD, checker), testování hodnoty proměnné pomocí knihovního makra – většinou končí ukončením programu, „výskokem“ dialogového okna – nelze s tím moc dělat z pozice uživatele, někdy ani programátora. Vhodné spíše pro ladění.
- představte si účetní, jak může reagovat na hlášku „`sqrt(-)`“
- chyby vzniklé za chodu programu ošetřené programátorem – provedení testu před rizikovými operacemi (dělení, odmocnina, test na přidělení paměti, souboru ...) – následně „dopravit“ chybu do „rozumného“ místa a zde se snažit 1) zachránit data co se dá a uložit je, 2) převést (pravděpodobnou) chybu do srozumitelného jazyka – například místo „`sqrt(-)`“ uvést „Nedostatek dat (položky ř. ...) pro vyřešení rovnice stanovující ...“.

Nevýhody posledně uvedeného řešení?

výjimky (exceptions)

nevýhody řešení:

- k chybě může dojít „hluboko“ v programu (například přes více funkcí)
- pro specifikaci chyby je nutný složitější/složený typ
- je nutné řešit ukončení každé funkce (odalokovat paměť, zavřít soubory, ...)
- cesta chyby do „rozumného“ místa tak může spočívat i v několika returnech:

```
struct SChyba chyba = funkce ();  
if (chyba.err)  
{  
    ošetří ukončení funkce  
    return chyba;  
}
```

- řešení, které nabízí C++, které řeší výše uvedené – mechanismus výjimek

výjimky (exceptions) - úvod

- oddělení řešení chyb od kódu programu
- při použití výjimky se oddělí parametry a návratové hodnoty od hlášení chyb
- při neošetření výjimky program „padne“ – nepokračuje tedy ve výpočtu s chybnými daty
- přenést řešení chyby do místa, kde je to možné a vhodné, aniž by bylo nutné se věnovat řešení mechanismu přenosu a ošetření cesty mezi těmito body (odalokování paměti, zavření souborů ...)
- možnost odlišit typ chyby a řešit chyby podle typu
- možnost popsat typ a příčinu chyby (k přenosu informace lze použít složený datový typ, který může obsahovat: typ chyby, místo chyby, hodnoty parametrů při vzniku chyby...)
-

výjimky (exceptions) – vytvoření výjimky

- výjimka je objekt, který se vytvoří ("hodí", pomocí klíčového slova throw) v místě chyby (na základě testu chybového stavu if ...).
- throw je příkaz pro vytvoření objektu přenášejícího informace o výjimce
- následně se "legálně" ukončují funkce (včetně rušení proměnných - destruktory) až do místa kde má být chyba reprezentovaná daným typem "chycena" (catch)
- při chybě se vytvoří objekt třídy výjimky pomocí throw V(), popřípadě s parametrem, který blíže popíše chybu

SChyba xxx; // složený objekt pro popis chyby

```
if (a ==0) // "hození" jednoduchého textového objektu char*  
    throw "chyba číslo x v místě y";  
if (spatne){  
    Napln(xxx, a, b, c ,d); // naplnění aktuálními daty  
    throw xxx; // "hození" složitějšího objektu  
}
```

výjimky (exceptions) – definice oblasti ze které výjimky zpracováváme

- blok, ve kterém se mají výjimky odchytávat, je třeba ohraničit/označit (try-catch)
- 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.
- výjimka se dá odchytit, pouze je-li v označeném 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}
```

výjimky (exceptions) – zpracování výjimek

místa kde má být chyba reprezentovaná daným typem "chycena" (catch)

- po odchycení je možné vybrané výjimky řešit
- catch může být pouze po bloku začínajícím try nebo za jiným catch
- výjimku vyřeší první příslušné catch, na které se narazí
- lze chytat i více výjimek
- lze chytat i postupně catch(X:V) {... catch(X:V1);}
- catch (...) odchytí všechny výjimky (je tedy uváděna jako poslední z bloků catch)
- je možné poslat výjimku dál pomocí throw;

```
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}  
catch(...){zde je řešení pro (všechny) ostatní výjimky}  
throw; //nedokáži vyřešit, přepošlu dál}
```

výjimky (exceptions) – dokončení

- 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
- 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 { ... }
- u funkcí je možné napsat které výjimky funkce "hází" a tím zpřehlednit a zjednodušit psaní a zrychlit program díky možným optimalizacím:

void f() throw (v1,v2,v3) {} a jiné nesmí hodit (=abort)

void f() {} nebo void f() noexcept(false) {} může hodit cokoli

void f() throw () {} nebo void f() noexcept(true) {} nemůže hodit nic

- Při výjimce v konstruktoru se nevolá destruktork

výjimky (exceptions) – knihovní, předdefinované

- existuje společný základ pro výjimky – třída `std::exception`
- z této základní třídy jsou odvozeny další třídy, např. `logic_error`, `runtime_error` a dále z nich `invalid_argument`, `out_of_range`, `underflow_error`, ...
- knihovny jazyka vyvolávají pouze výjimky z dané množiny odvozené od `exception`

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, zůstává možnost jejich použití (nevhodné)
- new resp delete volá konstruktory resp. destruktory (což malloc a free nedělají !!!)
- lze ve třídách přetížit (volání "globálních" ::new, ::delete)

```
char* pch = (char*) new char;
```

```
delete pch;
```

dynamická alokace pro jednu proměnnou (objekt). Lze předefinovat konstruktor

```
void* :: operator new    (size_t)
void  :: operator 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 předefinován konstruktor se dvěma parametry
```

dynamická alokace pro pole hodnot – implicitní konstruktory

```
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
```

Vysvětlete rozdíl mezi delete a delete[]

Obě dvě verze zajistí odalokování paměti.

První verze zavolá destruktory na každý prvek v poli.

Druhá verze volá destruktor pouze na jeden (první) prvek.

Každý z destruktorů se tedy chová odlišně. Zavoláme-li obyčejné delete na pole, nemusí dojít k volání destruktorů na prvky (a odalokování jejich interní paměti, vrácení zdrojů...).

V případě zavolání „polního“ delete na jeden prvek může dojít k neočekávaným výsledkům (například může očekávat před polem hlavičku s informacemi o počtu prvků pole. Pokud zde hlavička nebude, může dojít k chybné interpretaci dat, které zde budou).

```
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 třídu T  
    // ošetření ukončení "života" proměnné  
    if (změna) Save("xxx");  
    if (ptr!=nullptr)  
      ::delete ptr; //"globální" delete,  
    // jinak možné zacyklení  
    ...}
```

konstruktory při nenaalokování paměti podle zadaných požadavků používají systém výjimek, konkrétně `std::bad_alloc` z knihovny `<new>`.

“původního” vracení `nullptr` (ve starších překladačích `NULL`) je možné docílit ve verzi s potlačením výjimek pomocí volby `nothrow` (opět nutno přidat knihovnu - `#include <new>`). Nepoužívat není-li to vyloženě nutné

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

```
try{ // alokace paměti generuje výjimky při neúspěchu  
    // musíme řešit výjimku bad_alloc  
    mat = new TType*[aY]; // vlastní alokace s výjimkou  
    for(y = 0; y < aY; ++y)  
        mat[y] = new TType[aX]; // další alokace s výjimkou  
    }  
catch(std::bad_alloc)  
{  
    if(mat) // pokud došlo v minulém bloku k výjimce  
        deallocate(mat, y - 1); // odalokujem co se naalokovalo  
        throw(ENOMEM); // pošlem výjimku dále, změníme její typ  
    }
```

základy vstupu a výstupu znaků v C++

- neřeší klíčová slova, ale knihovní funkce (součást normy)
- složitý mechanismus, řešeno pomocí třídy, šablon
- použití (přetížení) operátorů bitových posunů << a >>

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

```
istream & operator >> (istream &, Typ& p) { }
```

```
ostream & operator << (ostream &, Typ& p) { }
```

- díky přetížení operátoru není nutné při volání specifikovat/kontrolovat typy (správný operátor hledá překladač)
- pro námi definované typy je nutno tyto operátory napsat

```
struct TPole2D {};
```

```
istream & operator>> (istream &is, TPole2D & p){... return is;}
```

```
ostream & operator<< (ostream &os, TPole2D & p){... return os;}
```

Pozn.: znak & je symbol pro referenci, nový způsob předávání parametrů

základy vstupu a výstupu znaků v C++

- realizace pomocí streamů – objekt propojující program a V/V zařízení
- pro práci s konzolou slouží předdefinované objekty **cin**, **cout**, **cerr** uvedené v knihovně **<iostream>**
- objekty jsou v prostoru **std::**. Použití **std::cout**, **std::endl**.
- endl (manipulátor) slouží k vynucení vytištění textu a odřádkování
- jelikož prvním operandem je "cizí" objekt, jedná se o (friend) funkce

```
int i;  
double j;  
char k[]="Ahoj";  
SPole2D p;
```

```
cin >> i >> j >> k;  
cout << i << "text" << j << k << p << endl;
```

cout, **cin**, **endl** jsou v prostoru **std**, pak správný přístup z našeho programu je **std::cout**. Použijeme-li na začátku modulu **using namespace std**, pak to znamená, že k proměnným z prostoru **std** můžeme přistupovat přímo a tedy psát pouze **cout**. Lépe je být konkrétní a tedy použít **using std::cout; using std::cin; using std::endl;**

Deklarace a definice proměnných (no)

- v (původním) C na začátku bloku programu tj. za ihned za {
- v C++ v libovolném místě (deklarace je příkaz)
- deklarace ve for je lokální pro cyklus
- konec života proměnné je s koncem bloku, ve kterém je definovaná
- deklarace musí mít extern (převážně v hlavičkovém souboru)
- důvodem je hlavně snaha nevytvářet neinicializované proměnné (jejichž použití vede k chybám). Proměnnou tedy definujeme, až v okamžiku kdy ji můžeme inicializovat

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

Reference (no)

- v C je možné předávání parametrů pouze hodnotou
- v případě, že v C potřebujeme další „odkaz“ na již existující proměnnou (v tom samém bloku, nebo předat jako parametr funkci), řešíme pomocí ukazatele – lze i v C++ ukazatel je lokální hodnota
- nakreslete umístění proměnných z následujícího příkladu v paměti

```
// dvě proměnné předané hodnotou, první typu int,  
// druhá typu ukazatel  
// obě jsou definice lokálních proměnných s inicializací  
void funkce(int aParam, int *aProm)  
{  
    int *prom; // neinicializovaná proměnná  
    prom = &aParam; // druhý parametr pro práci s aParam  
    *aProm = 3; // práce s předanou proměnnou (zde bbb)  
}
```

```
int bbb, a = 3;  
funkce (a, &bbb); // předání - inicializace parametrů
```

Reference (no)

- v C++ reference – odkaz (alias, přezdívka, nové jméno pro stávající proměnnou)
- zápis proměnné typu reference je **Typ&** a musí být inicializována v definici
- obdobně jako předávání ukazatelem, reference šetří čas a paměť při předávání parametrů do a z funkcí

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

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

Varianty použití operátoru &:

```
int aa, bb, cc, *pc;  
cc = aa & bb; // & značí operátor logické and bit po bitu  
pc = &cc; // & značí operátor získání adresy prvku  
int &rx = aa; // & značí že je definována proměnná jako reference  
// reference je to v definici proměnné na levé straně =  
int *pa = &aa; //=získání adresy, je v definici, ale napravo =
```

Reference (no)

Napište funkci Real, která má parametr typu reference double a vrací double. Funkce nastavuje předanou hodnotu na 4. Ukažte volání této funkce a popište co se děje s proměnnými a jejich hodnotami

Napište funkce Real, která má parametr typu reference double a vrací double. Funkce nastavuje předanou hodnotu na 4. Ukažte volání této funkce a popište co se děje s proměnnými a jejich hodnotami

```
double Real(double &r) //předání proměnné referencí do funkce  
{r = 4;  
return r;}
```

```
double ab; Real(ab); // způsob volání
```

```
// r je nové jméno pro volající proměnnou =  
// dělí se dvě jména o společné místo v paměti.  
// Proměnné ab a r leží na stejné paměti (dělí se o ni).  
// Při přiřazení do r dojde i ke změně původní proměnné ab.  
// Reference umožní změny vně, úspora proti volání hodnotou.
```

- "splývá" předávání i práce při předání 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
- je možné vrácení parametrů odkazem (je možné vrátit pouze parametry (proč?))

Reference (no)

Napište funkci, která má dva parametry. Jeden předávaný referencí, druhý pomocí ukazatele. Funkce vrátí prvek z větší hodnotou pomocí reference. Vysvětlete způsob předávání proměnných („jak to vypadá v paměti“ během programu). Ukažte a vysvětlete použití funkce „na levo“ od rovná se funkce() = proměnná;

Funkce má dva parametry. Jeden předávaný referencí, druhý ukazatelem. Funkce vrátí prvek z větší hodnotou pomocí reference. „Jak to vypadá v paměti“ během programu?

```
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, dd ; //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

const, const parametry (no)

klíčové slovo `const` slouží k vytvoření konstantní proměnné, nebo k ochraně proměnných před nechtěnou změnou (především při předávání parametrů ukazatelem nebo referencí)

vytvoření konstantní (neměnné) proměnné

- nelze ji měnit – proměnná označená `const` je hlídána překladačem před změnou
- konstantní proměnná - obdoba **`#define PI 3.1415`** z jazyka C
- typ proměnné je součástí definice **`const float PI=3.1415;`**
- pokud lze, použít `const` typ místo `#define`
- obvykle dosazení přímé hodnoty při překladu
- `const int` a `int` jsou dva různé/rozlišitelné typy
- při snaze předat (odkazem) `const` proměnnou na místě nekonstantního parametru funkce dojde k chybě (starší překladače vytvoří dočasnou proměnnou)
- volání `const` parametrem na místě `nonconst` parametru (fce) – nelze
- už je i v C

const, const parametry (no)

U použití ve více modulech (deklarace-definice v h souboru) – rozdíl v C a C++

- 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

const, const parametry (no)

Potlačení možnosti změn u parametrů předávaných funkcím (především) ukazatelem a referencí (odkazem)

```
int fce(const int *i)  
int fce1(int const &ii)  
const int & fce2(double aa)...
```

const, const parametry (no)

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

T reprezentuje konkrétní typ (int, double, TKomplex, CString ...)

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

inline funkce (no)

- obdoba maker v C, oproti kterým mají typovou kontrolu (zajistit nejrychlejší volání)
- dávat přednost inline, pokud lze
- předpis pro rozvoj do kódu, není funkční volání
- umisťuje se v hlavičkovém souboru
- označení funkce klíčovým slovem inline (v deklaraci)
- pouze pro jednoduché funkce (jednoduchý kód)
- „volání“ stejné jako u funkcí
- v debug modu může být použita funkční realizace
- některé překladače berou pouze jako “doporučení”
- je i v C

```
inline int deleno2(double a) {return a/2;}
```

```
int bb = 4;
```

```
double cc;
```

```
cc = deleno2(bb); // v tomto místě překladač
```

```
// vloží (něco jako) cc = (int)(double(bb)/2);
```

typ bool (no)

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

typ long long (no)

- nový celočíselný typ s danou minimální přesností 64 bitů

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

- je i v C

restrict (no)

- nové klíčové slovo v jazyce C99, modifikátor (jako const...) u ukazatele
- z důvodu optimalizací – rychlejší kód – (možnost umístit data 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 (jinak než prostřednictvím daného ukazatele)
- const řeší pouze přístup přes daný ukazatel, ne přes jiné (lze const i nonconst ukazatel na jednu proměnnou)

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
-
- makra `va_start`, `va_arg`, `va_end` `<cstdarg>`

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

přetypování (no)

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

```
double aa; int b = 3;  
// nejdříve se vyčísluje pravá strana, potom operátor =  
aa = 5 / b; // pravá strana (od =) je typu int  
// následně implicitní konverze na typ double při zápisu do aa  
aa = b / 5; // pravá strana je typu int  
aa = 5.0 / b; // pravá strana je double  
// výpočet v největší (přítomné) přesnosti  
aa = (double) b / 5; // pravá strana je double
```

enum (no)

- typ definovaný výčtem hodnot (v překladači reprezentovány celými čísly)
- 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

```
enum EBARVY {RED, YELLOW, WHITE, BLACK}
```

```
enum EBARVY barva = RED; // použití enum v C  
EBARVY barva = RED; // v C ++ je enum datový typ
```

```
int i;  
i = barva; barva = i; // v C lze, v C++ nelze
```

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” (Förster)
 - 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)
 - **klíčové slovo wchar_t** – dostatečně „široký“ typ pro uložení „nejširšího“ locale znaku
 - char16_t, char32_t – typy pro znaky jejichž „šířku“ známe
 - respektuje pravidla stejně velkého celočíselného typu (zarovnání, znaménko...)
-
- 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
 - dán normou, ale stále v překladačích rozdílné implementace

znakové konstanty, dlouhé literály (no)

- 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)

Objektové programování

- přináší nové možnosti a styl programování
 - vytváří nový datový typ, který „umí“ vše co standardní datové typy + to co ho naučíme
 - překladač se k tomuto typu chová stejně jako k typům standardním
 - vystavěn na složeném datovém typu (struct)
 - spojení dat a funkcí/metod pro práci s nimi
 - ochrana dat (přístupová práva)
 - výsledný mechanismus (spojení dat, metod a práv) nazýváme zapouzdřením
 - zlepšuje "kulturu" programování – automatická inicializace (konstruktor), ukončení života proměnné (destruktor), chráněná data ...
-
- nejbližší k ní má knihovný celek z jazyka C – rozhraní, data, kód
 - výhody: tvorba knihoven, sdílení kódu, údržba programu

Třída - základ objektového programování

- možnost vytvořit nový složený typ (odvozena od struct)
- třída class respektuje nové přístupy, ale zůstává i struct (zpětně kompatibilní vlastnosti)
- třída je obdobně jako struktura dána **popisem** vlastností a velikosti nového typu (v hlavičkovém souboru).
- konkrétní realizace funkcí/metod třídy je ve zdrojovém souboru
- prvky/proměnné třídy se vytváří až při definici objektu/proměnné
- definujeme-li proměnnou daného typu, je jí rezervováno místo v paměti. U třídy nehovoříme o proměnné ale spíše o objektu, nebo o instanci
- velikost objektu (třídy, struktury) může být větší než součet velikostí jejích proměnných/atributů
- funkce přístupné/nabídnuté k použití uživateli tvoří rozhraní třídy (přes které se s ní komunikuje)
- standardní typy mají i operátory. Jelikož se objekt chová jako nový typ podobný standardnímu, je možné pro něj nadefinovat i operátory (mající podobné chování jako u standardního typu)

Objektové programování – znovupoužití kódu

- C++ umožňuje znovupoužití kódu, tvorbu společných rozhraní (šablony, dědění, polymorfismus...)

šablony

- naprosto stejný kód pro různé typy,
- napsáno pouze jednou
- lze i pro neobjektové funkce

dědění

- odvození třídy z již existující třídy
- nová třída má vše co původní + drobné změny a rozšíření

polymorfismus

- dědění, které využívá tzv. virtuálních metod
- tvoření skupin tříd, které mají společné rozhraní – jde k nim přistupovat stejně, i když jsou to různé třídy
- při volání metody se vyhledá metoda pro typ aktuálního prvku – lze tedy dát do společné skupiny prvky různých tříd (se společnou bází – rozhraním)

Rozbor úlohy pro objektové programování

- podobný jako u „normálního“ programování
- stanovení logických celků a jejich vazeb
- stanovení objektů a jejich rozhraní

- formulace (definice) problému – slovní popis
- vznik (inicializace) a zánik (zrušení) objektu
- rozbor problému – vstupní a výstupní data, operace s daty
- návrh dat (vnitřní datové struktury)
- návrh metod (vstupy, výstupy, "výpočty"/operace, vzájemné volání, rozhraní, předávaná data)
- testování (modelové případy, hraniční případy, vadné stavy ...)

Rozbor problému – koncepce programu

- konzultace možných řešení, koncepce
- možnost znovupoužití kódu – stávající řešení nebo nové řešení; šablona, dědění (vztah „je“), prvek jiné třídy (vztah „má“)
- 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 ...) – relace má (jako prvek) a je (potomkem-typem)
- pohled uživatele (interface), pohled programátora (implementace)
- použití výjimek

Formulace problému – konkrétnější specifikace prvků programu

- co má třída dělat – obecně
- určení požadované přesnosti pro vnitřní data
- jak vzniká (->konstruktor)
- jak zaniká (->destruktor)
- jak nastavujeme a vyčítáme hodnoty (->getter a setter – data jsou soukromá/nedosažitelná pro uživatele)
- jak pracujeme s hodnotami (->metody a operátory)
- vstup a výstup

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

Navrhněte datovou strukturu (členské proměnné/atributy) pro třídu komplexních čísel.

Pro třídu komplexních čísel se nabízí dvě realizace dat:

- Reálná a imaginární složka
- Amplituda a fáze (délka a úhel, ...)

První verze je výhodná pro operace jako sčítání, druhá pro násobení.

Budeme více násobit nebo sčítat? Obecně nelze říci -> reprezentace jsou rovnocenné.

Dále ve třídě/objektu můžeme mít datový člen pro signalizaci chybového stavu – minulý výpočet se nezdařil (dělení nulou ...)

Návrh metod

- metoda – funkce ve třídě pro práci s daty třídy
- metody vzniku a zániku = konstruktor a destruktory
- metody pro vstup a čtení dat = gettery a settery
- metody pro práci s objektem
- operátory
- vstupy a výstupy
- metody vzniklé implicitně (ošetřit dynamická data)
- vnitřní realizace (implementace) metod - zde se (hlavně) zužitkuje C – algoritmy
- to jak je třída napsaná (jak vypadá ona a metody uvnitř) nazýváme implementací

Navrhněte metodu/funkci, která vrátí reálnou část komplexního čísla (pro obě reprezentace dat)

```
double Real()  
{  
    return iReal;  
}
```

```
double Real()  
{  
    return iAmpl * cos( iUhel );  
}
```

V případě, že uživatel nepracuje přímo s datovými atributy třídy (iReal, iAmpl, iUhel), potom při změně (implementace/realizace) vnitřních parametrů uživatel rozdíl nepozná, protože s třídou komunikuje přes metody/funkce, které jsou veřejně přístupné a které tvoří rozhraní/interface mezi atributy třídy a uživatelem

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

Příklad:

Napište testovací soubor `tester.bat` pro testování programu `progzk.exe` tak, že mu předložíte vstupní soubor a jeho výstup porovnáte s výstupem očekávaným. V případě chyby vytiskněte hlášení na konzolu

Část testovacího souboru (**tester.bat** - Windows) pro jedny vstupní parametry

```
progzk.exe <input1.dat >output1.dat  
fc output1.dat vzor1.dat  
if ERRORLEVEL == 1 echo "Program s input1 vrátil chybu"
```

nebo pro UNIX/LINUX vložte následující obsah do souboru: **tester.sh**

```
#!/bin/sh  
progzk.exe <input1.dat >output1.dat  
diff output1.dat vzor1.dat  
if [ $? -ne 0 ] ; then  
    echo "Program s input1 vrátil chybu."  
fi
```

Nastavte soubor jako spustitelný...

```
chmod u+x tester.sh
```

A spusťte soubor **tester.sh** z lokálního adresáře

```
./tester.sh
```

Základní pojmy Objektového programování (shrnutí):

Třída (Class) - Nový datový celek (datová abstrakce) obsahující: data (atributy / složky) a operace (metody), přístupová práva

Instance (Instance) - realizace (výskyt / exemplář) proměnné daného typu.

Objekt (Object) - instance třídy (proměnná typu třída).

Atribut (Member attribute / member variable) - data / proměnné definované uvnitř třídy. Často se používá i pojem složka.

Metoda (Method / Member function) - funkce definovaná uvnitř třídy. Má vždy přístup k datům třídy a je určena pro práci s nimi.

Implementace (Implementation) - Těla funkcí a metod (tj. kód definovaný uvnitř funkce / metody).

Zapouzdření (Encapsulation) – shrnutí logicky souvisejících (součástí programu) do jednoho celku (zde atributy, metody, přístupová práva) – nového datového typu třída. Ve volnější pojetí lze používat i pro funkce a proměnné definované v rámci jednoho .c souboru. Někdy je tímto termínem označováno skrytí přímého přístupu k atributům a některým metodám třídy (private members). Volně dostupné vlastnosti se označují jako veřejné (public members).

Rozhraní (Interface) - Seznam metod, které jsou ve třídě definovány jako veřejné a tvoří tak rozhraní mezi vnitřkem a vnějškem třídy.

Životní cyklus objektu (Object live cycle) - Objekt jako každá proměnná má definováno místo vzniku (definice/inicializace) a místo zániku.

Konstruktor / Destruktor (Constructor / Destructor / c'tor / d'tor) - metody které jsou v životě objektu volány jako první resp. jako poslední. Slouží k inicializaci atributu resp. k jejich de inicializaci objektu.

Operátor (Operator) - Metoda umožňující zkrácený zápis svého volání pomocí existujících symbolů. (součet +, podíl /, apod.)

Dědičnost (Inheritance) - Znovupoužití kódu jedné třídy (předka) k vytvoření kódu třídy nové (potomka). Nová třída (potomek) získá všechny vlastnosti (atributy, metody) z potomka a může definovat libovolnou novou vlastnosti nové.

Mnohotvarost (polymorfism) - Třídy se stejným rozhráním, a různou implementací, jednotný přístup k instancím. Mechanismus umožňující volání metod potomka z metod předka.

Pojem třídy a struktury v C++ (o – Objektová vlastnost)

- struct a class jsou složené datové typy – vychází ze struct jazyka C
 - pomocí struct a class se definuje nový datový typ (má „stejné“ vlastnosti a možnosti co do použití jako standardní typy – int, double ...)
 - struct v C++ je rozšířena o (objektové vlastnosti) možnost přidat metody a přístupová práva
 - struct a class v C++ se liší pouze minimálně (leč v důležité věci)
 - nebude-li dále řečeno jinak, platí uvedené pro struct i class
 - třída tvoří jmenný prostor
-
- v jazyce C struct obsahuje pouze data
 - v jazyce C++ obsahuje data, metody (funkce pracující s třídou)
 - v jazyce C++ lze nastavit přístupová práva pro data a metody (lze je používat pouze „uvnitř“ třídy, nebo je může používat i „uživatel“ třídy)
 - v C++ platí, že „data jsou to nejcennější, co máme“ a proto jsou v class data implicitně „schována“ – private
 - struktura kvůli zpětné kompatibilitě s jazykem C musí implicitně umožňovat uživatelský přístup k datům - public

Pojem třídy a struktury v C++

- pro deklaraci/definici slouží klíčové slovo class/struct
- deklarace třídy - pouze popis třídy – nevyhrazuje paměť – uvádíme v hlavičkovém souboru
- deklarací struktury/třídy vzniká nový datový typ, který se chová (je rovnocenný) jako základní datové typy (pokud autor třídy odpovídající činnosti popíše)
- objektové vlastnosti rozšířeny i pro union

Na základě znalostí o struktuře jazyka C napište pro jazyk C++:

- oznámení jména struktury, třídy
- definice struktury, třídy s vyznačením, kde budou parametry (data a metody)
- nadefinujte dva objekty (proměnné) dané třídy a ukazatel na objekt dané třídy, který inicializujete tak aby ukazoval na první z objektů

Pozn.: postupujte na základě znalostí o strukturách

Pro nový datový typ platí stejná pravidla jako pro základní typy

- deklarace - oznámení názvu/nového typu. Vlastní definice dat a metod je později.
- následně lze použít pouze ukazatel (ne instanci), jelikož není známa velikost typu
- netvoří kód -> je v hlavičkovém souboru

```
// deklarace typu - netvoří kód  
class jméno_třída;  
struct jméno_struktury;
```

- členská data a metody jsou uvedeny za názvem třídy v {} závorkách, **musí** být zakončeno středníkem
- nepoužívat typedef – většinou není nutný
- popis dat a hlaviček metod netvoří kód -> píšeme do hlavičkového souboru

```
// definice typu - netvoří kód - předpis  
class jméno_třída { parametry, tělo třídy };  
struct jméno_struktury {parametry, tělo      };
```

- definice proměnné, vyhradí paměť -> v cpp zdrojovém souboru
- klíčové slovo class, struct není nutné uvádět, stačí název typu
- zápis totožný jako pro int, nejste-li si zápisem jistí, napište pro typ int a následně typ změňte

```
// definice proměnných daného typu a práce s nimi  
jméno_třída a, b, *pc; // zabere paměť pro proměnné  
//obdoba int a,b,*pc  
// 2x objekt, 1x ukazatel  
pc = &a; // inicializace ukazatele
```

Pojem třídy a struktury v C++ - postup při psaní programu

- vytvoříme dva soubory pro třídu – hlavičkový a zdrojový
- součástí vývoje třídy by měl být i testovací kód (testovací funkce) – může být součástí zdrojového souboru („odstranitelnou“ například pomocí #if - #endif), lépe vytvořit soubor zvláštní (například s funkcí main)
- hlavičkový soubor ošetřit proti vícenásobnému načtení
- ve zdrojovém souboru třídy (a souborech kde budeme třídu používat) naincludovat hlavičkový soubor
- v hlavičkovém souboru napsat deklaraci třídy včetně tělíčka (nezapomenout ukončit středníkem)

hlavičkový soubor třídy

```
#ifndef CCOMPLEX_H123  
#define CCOMPLEX_H123  
class CComplex {
```

```
};
```

```
#endif
```

zdrojový soubor třídy

```
#include "ccomplex.h"
```

Přístupová práva (o)

- nastavuje kdo má k datům/metodám přístup: pouze třída (private/privátní/soukromá) x uživatel i třída (public, veřejná)
- klíčová slova pro nastavení přístupových práv – private, public, protected
- klíčová slova jsou přepínače - označují začátek bloku stejných přístupových práv
- přepínače možno vkládat libovolně
- rozdíl mezi class a struct (jeden z mála) – implicitní přístupové právo private x public.
To je právo, které je implicitně nastaveno na „začátku“ těla definice – platí do první změny, do prvního uvedení přepínače na jiná přístupová práva

class X	{		int i	je ekvivalentní
class X	{ private:		int i	...
struct Y	{		int i	je ekvivalentní
struct Y	{ public:		int i	...

Přístupová práva - (o) postup při psaní programu

- uvádí se pouze v definici – tj. v hlavičkovém souboru
- říkají, které z proměnných může používat uživatel (mimo třídu)
- lze je používat pro každou metodu/proměnnou zvlášť ale z důvodu přehlednosti je lepší volit větší celky (shlukovat členská data se stejnými právy)

```
// definice třídy CComplex
class CComplex {
    double iReal; // privátní datová proměnná/složka
public:
    double iImag; // veřejně přístupná proměnná
private: // uživateli nepřístupné proměnné
    double iAlpha; // privátní
    double iAmplitude; // privátní
}; // konec definice třídy CComplex
```

```
// definice proměnné uživatelem (mimo prostor třídy)
class CComplex prom;
prom.iImag = 5; // lze - iImage je public
prom.iReal = 2; // nelze - iReal je private
prom.iAlpha = 2; // nelze - iAlpah je private
```

Data, metody - práce s nimi (o)

- datové členy – jakýkoli známý typ (objekt nebo ukazatel)
- s datovými členy pracujeme stejně jako s daty uvnitř struktury (z důvodu „bezpečnosti dat“ se snažíme přístupu uživatele k datům zabránit)
- rozlišujeme přístup k datům objektu = operátor „.“ K datům „ukazatele“ =operátor “->“
- metody – členské funkce patřící ke třídě, pracujeme s nimi stejně jako s daty a „funkční volání“ naznačíme přidáním „()“, mezi kterými jsou uvedeny parametry metody
- uživateli přístupné metody tvoří interface
- přístupová práva – **private, protected, public**

Nadefinujte třídu, která bude mít privátní členská data (po jednom) typu int, float, class Jmeno, C-řetězec. Dále bude mít veřejné členské metody a jeden datový člen int:

metoda1 – bez parametrů, vrátí int

metoda2 – dva parametry int a float, bez návratové hodnoty

metoda 3 – vrátí float a má parametry int a ukazatel na Jmeno

Napište část programu, který nadefinuje objekt dané třídy, ukazatel inicializovaný tímto objektem. Dále ukažte přístup ke členům a metodám a řekněte, které budou fungovat.

```
// deklarace (popis) třídy - v souboru jmeno_tridy.h
class Jmeno_tridy { // implicitně private:
    int data1; //datové členy třídy
    float data2;
    Jmeno *j; // class není nutné uvádět
    char string[100];
public: // metody třídy
    int metoda1() {...return 2;}
    void metoda2(int a,float b) {...}
    float metoda3( int  a1,  Jmeno *a2);
    int dd;//nevhodné, veřejně přístupná proměnná
};
```

```
Jmeno_tridy aa, *bb = &aa;
// obdoba struct pom aa, *bb = &aa;
// přístup jako k datovým prvkům struct
aa.dd = bb->dd;
int b = aa.metoda3(34,"34.54"), c = bb->metoda3(34,"34.54");
aa.metoda1(); // přístup přes proměnnou/objekt
bb->metoda1(); // přístup přes ukazatel na proměnnou/objekt
// aa.data1 = bb->data1; // nelze přistupovat k privátním datům
```

```
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, *pc = &b;
```

```
a.Re = 1;      pc->Re = 3;      // je možné
```

```
b.Uhel(3.14); pc->Uhel(0);      // je možné
```

```
a.pom = 3;     pc->pom = 5 ;     // není možné
```

```
b.Velikost(); pc->Velikost(); // není možné
```

Data, metody – postup při psaní programu

- datovou reprezentaci a metody si dobře promyslíme a rozvrhneme
- pokud není speciální důvod, potom data jsou v sekci private
- metody „nebezpečné“ nebo s omezenými kontrolami jsou private (aby je nemohl volat uživatel, který by mohl použít tyto metody, aniž by domyslel důsledky)
- metody tvořící interface (přístupné uživateli) – uvedeme v sekci public
- z hlediska programu obvykle jako první píšeme „speciální“ metody pro vznik a zánik proměnné – konstruktory a destruktory

```
class CComplex { // implicitně private:
    double iReal, iImag; //datové členy třídy
    double iApha, iAmplitude;
// použít všechny čtyři členy? nebo jen dva? které?

public: // metody třídy
    void Set(double aRe, double aIm) {...}
    double GetReal() {return iReal;}

    int error;//nevhodné, veřejně přístupná proměnná
};
```

ukazatel this (o)

- základ mechanismu volání metod – metodám se předá ukazatel na aktuální objekt, který metodu zavolal (zajistí překladač)
- this - klíčové slovo
- **T* const this; // „součást“ každého objektu**
- předán implicitně do každé metody (skrytý parametr-překladač)
- při použití metody **aa.Metoda()**; se vlastně „volá“ **Metoda(&aa)** – objekt, který chce „svou“ metodu zavolat je do ní překladačem skrytě předán jako parametr.
- Prototyp metody je napsán v hlavičkovém souboru **void Metoda(void)** ale je „přeložen“ se skrytým parametrem jako **void Metoda(T *const this)** a potom v těle lze používat členské metody a data aktuálního objektu {**this->data1 = 4;this->metodax();**}

Použití this při psaní programu:

- přístup k datům a metodám aktuálního objektu (this je možné vynechat, proměnná třídy je první v rozsahu prohledávání – třída tvoří jmenný prostor)
this->data = 5, b = this->metoda(a)
- objekt vrací sám sebe – **return *this;**
- kontrola parametru s aktuálním prvkem **if (this==¶m)**

ukazatel this

Napište třídu Komplex a v ní metodu, která vrátí maximum ze dvou proměnných typu Komplex.

```

class CComplex {
double Re,Im;
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 a tak
// nepočítám, => rychlý návrat.
if ( this->Re < param.Re ) return param;
else                return *this;
// param i this existují vně -> lze reference
} // konec funkce
}; // konec třídy

```

volání:

```

Komplex aa,b, c ;
c = aa.Max(b); // neboli Max(&aa,b). Překladem
// se aa uvnitř metody mění v this
c = b.Max(b); // mezivýsledek c = b; this v Max je &b

```

Alternativní hlavičky pro metodu Max. Vysvětlete rozdíly při předávání (kde vznikají dočasné proměnné).

```
Komplex Max(Komplex param)  
Komplex & Max(Komplex const &param)
```

```
c = a.Max(b);
```

c = a.Max(b);

společné volání pro obě hlavičky. Z volání není vidět rozdíl pro předávání hodnotou a referencí.

Komplex Max(Komplex param)

předaný parametr b z volání se stane předlohou pro lokální proměnnou param, která vznikne jako (plnohodnotná lokální) kopie – musí tedy vzniknout a zaniknout objekt. Změní-li se param, nemění se původní objekt b.

Vracený parametr je vrácen hodnotou – musí tedy vzniknout (jako plnohodnotná kopie prvku z return xxx;) a zaniknout.

Komplex & Max(Komplex const ¶m)

pokud se předává objekt pomocí reference, nevytváří se žádný nový objekt, odkazujeme se na objekt původní. Manipulací s prvkem param pracujeme přímo s objektem b. Aby nedošlo k nechtěné změně b, můžeme param označit jako const a potom ho nelze změnit. Vracet referenci můžeme, pouze pokud proměnná existuje ve funkci volající i volané.

Jelikož je předávání parametrů pomocí reference úspornější (časově i paměťově), dáváme mu přednost (v situacích kdy to jde!).

inline metody (o)

- obdoba inline funkcí pro třídu
 - slouží pro zápis krátkých metod
 - rozbalené do kódu, předpis, netvoří kod
-
- automaticky jsou to metody s „tělem” v deklaraci třídy (hlavičkový soubor)
 - inline jsou i metody v deklaraci třídy s klíčovým slovem inline, tělo mimo (v hlavičce)
 - pouze hlavička metody v deklaraci třídy a tělo mimo (ve zdroji) – není inline
-
- obsahuje-li složitý kód (cykly) může být inline potlačeno (překladačem)
 - může být bráno překladačem pouze jako doporučení

.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

inline metody (o) – postup při psaní programu

- podle „délky“ metody rozhodneme, zda je vhodné, aby byla inline (pokud funkce či metoda obsahuje cykly, nebo volání jiných metod, nemá smysl, aby byla inline)
- definice třídy by měla být přehledná – měla by obsahovat pokud možno jen deklarace

inline metody (o) – postup při psaní programu

- těla extrémně krátkých inline metod můžeme psát přímo do „těla“ definice třídy -> metoda je implicitně inline (bez označení)
- delší těla inline metod je výhodné kvůli přehlednosti psát mimo „tělo“ definice třídy. Před prototypem se uvede klíčové slovo inline, vlastní tělo metody se napíše za definici třídy do hlavičkového souboru
- metody s funkčním voláním (nejsou inline) mají v definici třídy pouze prototyp. Tělo metody se uvede ve zdrojovém souboru

hlavičkový soubor:

```
class CComplex{ public:  
    int Metoda1(void){return 1;} // implicitně inline  
    inline int Metoda2(); // implicitně inline díky klíčovému slovu  
    int Metoda3(); // není inline ani tělo => funkční volání  
};  
// jsme-li mimo třídu, musí být CComplex::  
int CComplex::Metoda2(){...; return 2;}
```

zdrojový soubor (include hlavička):

```
int CComplex::Metoda3(){...;...;...;return 3;}
```

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, netvoří kód
- inline soubor – "těla" inline metod – předpis. Jelikož jsou těla mimo definici třídy, musí obsahovat jméno třídy, do které patří (a operátor přístupu). Mimo definici třídy je nutné k metodám uvádět, ke které třídě patří pomocí operátoru příslušnosti T::metoda
- zdrojový soubor – "těla" metod – "skutečný" kód. Jelikož jsou těla mimo definici třídy, musí obsahovat jméno třídy, do které patří (a operátor přístupu)
- V souborech, kde je třída používána, musí být vložen hlavičkový soubor třídy.

hlavička: (soubor.h)

```
class T{  
data  
metody (bez "těla"); // těla v cpp  
inline metody (bez "těla"); // tělo je v h souboru  
metody (s „tělem) {} // inline metody  
};
```

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; }
```

statický datový člen třídy (o)

- vytváří se pouze jeden na třídu, společný všem objektům třídy
- např. počítání aktivních objektů třídy, společné nastavení pro všechny prvky třídy, zabránění vícenásobné inicializaci, zabránění vícenásobnému výskytu objektu ...
- v deklaraci (*.h) třídy označen jako static

```
class string {  
    static int pocet;  
    // deklarace statické proměnné nerezervuje paměť  
};
```

- existuje i v okamžiku, kdy neexistuje žádný objekt třídy
- není součástí objektu, vytvoří se jako globální proměnná (nutno definovat a inicializovat ve zdrojovém souboru *.cpp)

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

statické metody (o)

- pouze jedna na třídu
- je možné ji volat aniž by existoval (jediný) objekt dané třídy
- nemá this (= „normální“ funkce s přístupem k privátním datům)
- ve třídě označená static
- vlastní tělo ve zdrojové části
- nesmí být virtuální
- může k private členům (obdobně jako friend funkce),
- externí volání se jménem třídy bez objektu **Třída::fce()**

statické členy – použití v programu

// v *.h popis

```
class Třída { public:  
    static int Pocet; //staticka data  
    static int Kolik (void); //staticka metoda  
    // příklady použití/přístupu v rámci třídy  
    int Prvku1(void){return Pocet;}  
    int Prvku2(void){return Kolik();}  
}
```

// v souboru *.cpp kód – pouze jednou, plná cesta

```
int T::Pocet = 0; // místo v paměti a inicializace  
int T::Kolik() {return Pocet;} // kód funkce
```

// způsob použití v programu – celá cesta

```
T::Pocet = 1; // pro blokování, chceme-li jen jeden prvek  
int p1 = T::Pocet; // je-li public  
int p2 = T::Kolik(); // je-li public  
T a;  
int p3 = a.Prvku1(); // pro metodu musí existovat prvek T
```

konstruktory a destruktory (o)

- slouží k ovlivnění vzniku (inicializace) a zániku (úklid) objektu
- základní myšlenkou je, že proměnná by měla být inicializována (nastavena do počátečního stavu) a zároveň by se při zániku proměnné neměly ztratit získaná data nebo obdržené zdroje (soubory, paměť ...)
- jsou to metody se specifickými vlastnostmi (rozšíření/omezení)
- jsou volány automaticky překladačem (nespoléhá se na uživatele)
- konstruktor – první (automaticky) volaná metoda na objekt. Víme, že data jsou (určitě) neinicializovaná.
- místo volání konstruktorů určíme definicí nebo vytvořením proměnné pomocí new
- destruktor – poslední (automaticky) volaná metoda na objekt
- místo volání destrukturu určí místo konce života proměnné (konec bloku kde byla nadefinována) nebo delete

- „konstruktory“ vlastně známe již z jazyka C z definicí s inicializací
- některé „konstruktory“ a „destruktor“ jsou u jazyka C prázdné (nic nedělají)

```
{//příklad(přiblížení) pro standardní typ int
    int a;
//definice proměnné bez konkrétní inicializace = implicitní

    int b = 5.4;
// definice s inicializací. vytvoření, konstrukce proměnné int
// z hodnoty double = konverze (z double na int)

    int c = b;
// konstrukce (vytvoření) proměnné int na základě proměnné
// stejného typu = kopie
...
} // konec životnosti proměnných - zánik proměnných
// je to prosté zrušení bez ošetření - (u std typů)
// zpětná kompatibilita
```

Destruktor

- slouží k „úklidu“ proměnné – uchování dat, vrácení zdrojů (vrácení systémových prvků, paměť, soubory, ovladače, ukončení činnosti HW, uložení dat ...)
- má stejný název jako třída, názvu předchází ~ (proč?)
- na celou třídu je pouze jeden (bez parametrů)
- nemá návratovou hodnotu
- volán automaticky překladačem

~Třída (void) {} // destruktor

- 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
- při dědění je výhodné aby byl virtual

{ Třída aa, bb, *pc = new Třída;

**delete pc; // volá se destruktor
} // volají se destruktory pro prvky bb a aa (v tomto pořadí)**

Konstruktor

- je to metoda, která má některé speciální vlastnosti
- metoda konstruktor má stejný název jako třída
- nemá návratovou hodnotu
- volán automaticky při vzniku objektu (lokálně i dynamicky)
- konstruktor je využíván k inicializaci proměnných (nulování, nastavení základního stavu, alokace paměti, ...)
- možnost (funkčního zápisu) konstrukce je přidána i základním typům

```
class Trida {  
    int ii;  
    public:
```

```
    Trida(void) {...} // implicitní konstruktor
```

```
    Trida(int i): ii(i) {...} // konverze z int  
    // ii(i) je konstruktor proměnné ii na základě i
```

```
    Trida(Trida const & a) {...} // kopy konstruktor  
}
```

- třída může mít 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
```

- konstruktor může použít i překladač ke konverzi (například voláním metody s parametrem int, když máme metodu jen s parametrem Trida, ale máme konverzní konstruktor z int)
- konstruktory se používají pro implicitní konverze, pouze jedna uživatelská (problémy s typy, pro které není konverze)
- explicit - klíčové slovo – zakazuje použití konstruktoru k implicitní konverzi

```
class Trida {public:  
explicit Trida(int j); // konverzní konstruktor z int  
Trida::Metoda(Trida & a);  
}
```

```
int i;
```

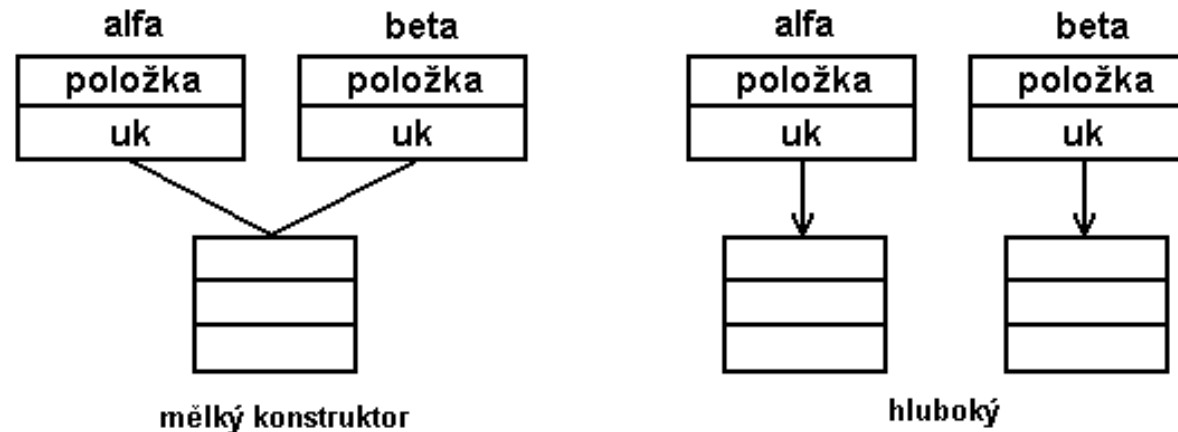
```
a.Metoda ( i ); // nelze, implicitní konverze se neprovede  
// kdyby nebylo uvedeno explicit, pak by překladač  
// využil konstruktor k implicitní konverzi int->Trida
```

```
b.Metoda ( Trida(i)); // lze, explicitní konverze je povolena
```

- u polí se volají konstruktory od nejnižšího indexu
- konstruktor nesmí být static ani virtual
- alespoň jeden musí být v sekci public (jinak zákaz pro běžného uživatele)
- implicitní konstruktor (kvůli zpětné kompatibilitě) vzniká automaticky jako prázdný
- je-li nadefinován jiný konstruktor, implicitní automaticky nevznikne
- není-li programátorem definován kopykonstruktor, je vytvořen překladačem (kvůli zpětné kompatibilitě) a provádí kopii (paměti) jedna k jedné (memcpy)

Vysvětlete jaký je problém při vzniku automatického kopykonstruktoru je-li ve třídě ukazatel?

- automatický kopykonstruktor se chová jako prostá kopie prostoru jedné proměnné do druhé
- pokud objekt nevlastní dynamická data (ukazatel na ně), dojde ke kopii hodnot což je v pořádku
- jsou-li dynamická data ve třídě (tj. jsou odkazována ukazatlelm) dojde ke kopii ukazatele. Potom dva objekty ukazují na stejná data (a neví od tom) → problém při rušení dat – první rušený objekt společná data zruší ...
- nazýváme je mělké (memcpy) a hluboké kopírování (shallow, deep copy – vytváří i kopii dat na které se odkazují ukazatele. Ukazatele mají tedy různé hodnoty.)
- řešením je vlastní kopie nebo indexované odkazy



Konstruktory a destruktory – použití při psaní programu

- promyslíme, na základě jakých dat může objekt vznikat (implicitní vznik je častý, kopykonstruktor se používá i při předávání hodnotou, ...)
- promyslíme, zda je nutné ošetřit zánik proměnné (data, zdroje)
- můžeme vytvořit i privátní konstruktory, které nemůže používat uživatel (tomu bychom měli ale alespoň jeden nechat)
- u kratších konstruktorů promyslíme možnost inline
- atributy třídy s jednoduchým přiřazením inicializujeme mezi hlavičkou a tělem konstr.
- nejprve se volají konstruktory atributů v pořadí atributů uvedeném v definici. Typ inicializace je implicitní, pokud není v inicializační části metody naznačeno jinak.
- po definici atributů se provádí tělo konstruktoru

```
class Trida {  
  CComplex a,b,c;  
public: Trida(int i, int j = 0): c(i), a(i,j)  
  // inicializují se nejprve atributy v pořadí (definice)=>a,b,c  
  // pro a je volán konstr. se dvěma parametry, pro b implicitní,  
  // pro c je volán konstruktor CComplex s jedním parametrem  
  {b.iRe = ...; b.Im=...;} // po inicializaci atributů jde tělo  
}; // pro b nejdříve implicitní konstr., potom druhé přiřazení  
// stejným proměnným v těle, nevhodné
```

Konstruktory novinky:

move sémantika – move konstruktor

- obsah jedné proměnné se přesune do druhé (první se „smaže“)
- slouží v situacích, kdy je proměnná pouze dočasná, k přesunu zdrojů bez jejich získávání a pouštění
- přesun hodnot je rychlejší než kopie, „prázdný“ objekt má jednodušší činnost destrukturu
- volá se, je-li parametr r-hodnota (například objekt vrácený hodnotou z funkce, nebo reference na takový objekt)
- operátor && = „reference na r-hodnotu“

Trida(Tridat && x) {}

možnost „volání“ konstruktoru stejné třídy v rámci jiného konstruktoru

Trida(int i, int j) :Trida(){}

**Trida(int i, int j) :Trida(), ii(i+2*j){} // chyba, nesmí
// být žádná další inicializace**

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 konkrétní konstruktory  
    { ... tělo ... }  
    // volá 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("Kolejní 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  
} // tedy jsou volány destruktory Osoba,  
// destruktory stringů Adresa a  
// Jmeno. A destruktory pro Add  
// (v uvedeném pořadí)
```

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`

const a metody (o)

- const u parametrů – parametry nepředávat hodnotou (paměťově a časově náročné), const slouží jako ochrana proti nechtěnému přepisu hodnot v metodě (funkci)
- const u návratové hodnoty – návratovou hodnotu nelze měnit (či použít na místě ne-const argumentu metody/funkce)
- const parametry by neměly být předávány na místě nonconst parametrů
- na const parametry nelze volat metody, které je změní – kontrola překladač
- metody, které nemění objekt je nutno označit (programátorem) jako const a pouze takto označené metody lze volat na const objekty

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

```
float f2(void) {}
```

```
const T a;
```

```
T b;
```

```
a.f1(); b.f1(); b.f2(); // lze, převod na const je v pořádku  
a.f2(); // nelze konstantní proměnnou na místě nekonstantní
```

const a metody – použití v programu

- u parametrů metod/funkcí se doporučuje ty, které se nemění označit const
- u metod, které nemění parametr, který je vyvolal, je nutné za definici přidat const,
- případný pokus o změnu „this“ v metodě označené const hlídá překladač
- T a const T jsou dva různé typy. Proto můžeme mít dvě metody stejného jména, jednu volanou pro konstantní proměnné a druhou pro nekonstantní

friend funkce (o)

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

```
class Třída {  
    friend complex;  
    friend double f(int, Třída &);  
    // třída komplex a globální funkce f mohou  
    // přistupovat i k private členům Třídě.  
private:  
    int i;  
}
```

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

friend funkce

- zdrojový kód friend funkce je mimo třídu a nenese informaci o třídě, ke které patří (nemá this ani jinou „zpětnou vazbu“ na třídu, která ho označila friend ...)
- alternativou jsou statické metody
- použití pro nečlenské operátory – první parametr nepatří ke třídě a proto není možné použít metodu

T operátor*(double d, T &t)

s voláním

T a, b;

b = 5 * a;

- použití pro metody, které mají více parametrů, jejichž typy nepatří mezi standardní (int, double ...)

ve třídě Y a Z označíme třídu X jako friend, potom

X X::Metoda(double d, Y &y, Z &z)

má přístup k privátním prvkům třídy Y a Z (a samozřejmě X). Volání je

X a, b; Z c, Y d;

b = a.Metoda(2.1, d, c);

přetížení operátorů (o)

- pro vlastní typy je možné přetížit i operátory (tj. definovat vlastní)
 - pro definici slouží klíčové slovo `operator` následované typem/znakem operátoru
 - operátor je speciální metoda/funkce, mající plné a zjednodušené volání
 - deklarace pomocí „funkčního“ volání např. unární a binární `+` pro typ `int` by šlo psát:
-
- **`int operator +(int a1) {}`**
`int operator +(int a1, int a2) {}`

s „funkčním“ voláním

`operator +(i);`
`operator+(i, j);`

ve zkrácené formě

`+i;`
`i + j;`

možnost volat i „funkčně“

`operator=(i, operator+( j ))`

nebo zkráceně

`i=+j`

operátory - obecné shrnutí (o)

- operátory lze v C++ přetížit
- správný operátor je vybrán podle seznamu parametrů (a dostupných konverzí), výběr rozliší překladač podle kontextu
- operátory unární mají jeden parametr – u funkcí proměnnou, se kterou pracují, nebo "this" u metod
- operátory binární mají dva parametry – dva parametry funkce, se kterými pracují nebo jeden parametr a "this" u metod
- 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
- operátorem je i **new** a **delete**,
- pro vstup a výstup do **streamu** je operátor využít,
- hlavní využití u vlastních tříd (objektů)

operátory – definice a použití (o)

- slouží ke zpřehlednění programu
- snažíme se, aby se přetížené operátory chovaly podobně jako původní (např. nemění hodnoty operandů, operátor + 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

metoda patří ke třídě (T::) první parametr je this (tj. a + b pro a , b stejného typu)

T T::operator+(T & param) {}

(friend) funkce pro případ, že prvním operandem je „cizí“ typ (tj. například 5 * a)

T operator+(double d, T¶m) {}

Unární operátory

- mají jeden parametr (this)
- například + a − , ~, !, ++, --

complex	operator+(void)	// zbytečně vrátí objekt
complex	& operator+(void)	// vrácený objekt lze změnit
complex	operator+(void)	const
complex const	& operator+(void)	const

- 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 (tj. odlišná) hodnota – proto musíme vytvořit nový prvek – vrátíme hodnotou
- pokud nejsou operandy měněny (a většina standardních operátorů je nemění), potom by měly být označeny const (pro this i parametr). návrat referencí potom musí být také const.

Unární operátory

- 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
- při volání dáváme přednost ++a (netvoří tmp objekt)

++(void) s voláním ++x

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

T& operator ++(void)

T operator ++(int)

operátor ++aa na rozdíl od aa++ netvoří dočasný prvek pro návratovou hodnotu a proto by měl v situacích, kdy lze tyto operátory zaměnit, dostávat přednost

Binární operátory

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

```
complex complex::operator+ (const complex & c) const  
complex complex::operator+ (                double c) const  
complex                operator+ (double f, const 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 – více stejných operátorů v jedné třídě
- lze přetížit i jako funkci (globální prostor, druhý parametr je třída) – často friend
- opět může nastat 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 =

- pokud není napsán, vytváří se implicitně (mělká kopie – přesná kopie 1:1, memcopy)
- měl by (díky kompatibilitě) vracet hodnotu (musí fungovat zřetězení `a = b = c = d;`)
- obzvláště zde je nutné ošetřit případ `a = a`
- pro činnost s dynamickými daty nutno ošetřit mělké a hluboké kopie
- nadefinování zamezí vytvoření implicitního `=`
- je-li v sekci `private`, pak to znamená, že nelze použít (externě)
- od kopykonstrukturu 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 &= ...;`

Vysvětlete rozdíl mezi mělkým a hlubokým kopírováním.

Vysvětlete rozdíl mezi kopykonstruktorem a operátorem `=`.

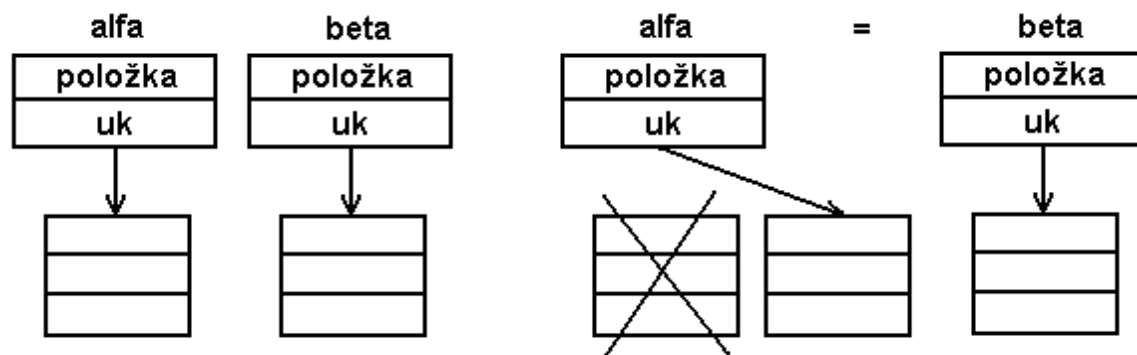
Operátor =

Problém nastává v případě, že jsou v objektu ukazatele na paměť, kterou je nutné při zániku objektu odalokovat.

Na rozdíl od kopykonstruktoru je nutné si uvědomit, že v cílovém objektu jsou platná data. Před jejich naplněním novými hodnotami je nutné odalokovat původní paměť.

Při mělkém kopírování (i implicitně vytvářený operátor) dojde ke sdílení paměti (přiřazení ukazatele na paměť) aniž by o tom objekty věděly. Při odalokování dojde k problémům.

Při hlubokém kopírování (musí napsat tvůrce třídy) se vytvoří kopie dat paměti, na kterou ukazuje ukazatel, nebo se použije mechanismus čítání odkazů na danou paměť.



Vysvětlete rozdíly mezi operátorem = pro:

- ukazatele
- objekty s členskými objekty bez ukazatelů
- objekty obsahující ukazatele bez operátoru =
- objekty obsahující ukazatele s operátorem =

Operátor =

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

Objekty bez ukazatelů mohou využívat implicitní operátor = (memcpy), pokud po nich není požadován nějaký postranní efekt (nastavení čítače, kontrola dat...)

```
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 destruktory, který odalokuje txt (což by měl být), potom zde odalokováváme paměť, kterou odalokoval již destruktory 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 (zákaz pomocí klíčového slova explicit)
- opačný směr jako u konverzních konstruktorů (jiný typ -> můj typ/ můj typ -> jiný typ)
- například konverze na standardní typy – int, double...
- nemá návratovou hodnotu (je dána názvem)
- nemá parametr (jen this)
- v C++ lépe používat konverze pomocí "cast" (dynamic, static ...)

operator typ(void)

T::operator int(void)

volán implicitně nebo

T aaa;

int i = int (aaa) ;

(int) aaa; // starý typ - nepoužívat

Operátory vstupu a výstupu

- nejedná se o zvláštní typ operátoru, jedná se o využití standardního operátoru
- knihovní funkce
- řešeno pomocí třídy
- použití (přetížení) operátorů bitových posunů << a >>
- díky přetížení operátoru není nutné kontrolovat typy (správný hledá překladač)
- pro vlastní typy nutno tyto operátory napsat
- pro práci s konzolou předdefinované objekty cin, cout, cerr v knihovně <iostream>
- objekty jsou v prostoru std::.. Použití using: std::cout, std::endl.
- jelikož prvním operandem je "cizí" objekt, jedná se o (friend) funkce

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

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

```
istream & operator >> (istream &, Typ& p) { }
```

```
ostream & operator << (ostream &, Typ& p) { }
```

Přetížení indexování []

- podobně jako operátor() ale [] má pouze jeden operand (+ this, je to tedy binární operátor s indexem jako jediným „viditelným“ parametrem)
- nejčastěji používán s návratovou hodnotou typu reference (l-hodnota)

double& T::operator[](int)

```
aaa[5] = 4;  
d = aaa.operator[ ](3);
```

Přetížení funkčního volání ()

- může mít libovolný počet parametrů
- takto vybaveným objektům se říká funkční objekty
- nedoporučuje se ho používat (plete se s funkcí)

```
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 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;

operátory a STL

- je-li nutné používat relační operátory, stačí definovat operátory == a <
- ostatní operátory jsou z nich odvozeny pomocí template v knihovně <utility>

```
namespace rel_ops {  
    template <class T> bool operator != (const T& x, const T& y){return !(x==y);}  
    template <class T> bool operator > (const T& x, const T& y){return y<x;}  
    template <class T> bool operator <= (const T& x, const T& y){return !(y<x);}  
    template <class T> bool operator >= (const T& x, const T& y){return !(x<y);}  
}
```

zdroj: http://www.cplusplus.com/reference/utility/rel_ops/

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 práci 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);  
p1=&T::f1;
```

```
T tt;  
tt.*p1( )
```

```
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( );
```

dědění

- jednou ze základních vlastností objektového programování je myšlenka znovupoužitelnosti kódu – 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 – bazová, nová – odvozená

dědění

- 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).

<pre>class Base { public: Metoda 1(); Metoda 2(); Metoda 3(); }</pre>	<pre>class Dedeni:public Base { public: Metoda2(); Metoda3() {Base::Metoda3 ();} Metoda4(); }</pre>	<pre>// necháme původní metodu z báze //vytvoříme novou metodu,původní je skrytá // doplníme původní metodu // vytvoříme novou metodu</pre>
---	---	---

dědění

- při dědění se mění přístupová práva proměnných a metod базové třídy v závislosti na způsobu dědění
- class C: public A – A je базová 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 базové třídy (A) ve třídě zděděné

class A
public a
private b
protected c

class B:private A
private a
-
private c

class C:protected A
protected a
-
protected c

class D:public A
public a
-
protected c

- postup volání konstruktorů - konstruktor báze 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 konstruktor 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
using Base::y; // je možné takto vytáhnout
// proměnnou (zděděnou zde do sekce private)
// na jiná přístupová práva (zde public)      };
```

dědění

- - - - příklad 2 - - - - -

```
class base {  
public:  
base (int i=10) {...}  
}
```

```
class derived:base {  
complex x,y; ...
```

```
public: derived() : y(2,1) {f() ... }  
  
}
```

volá se base::base()

complex::complex(void) - pro x

complex::complex(2,1) – pro y

f() ... - vlastní tělo konstruktoru

dědění

- 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

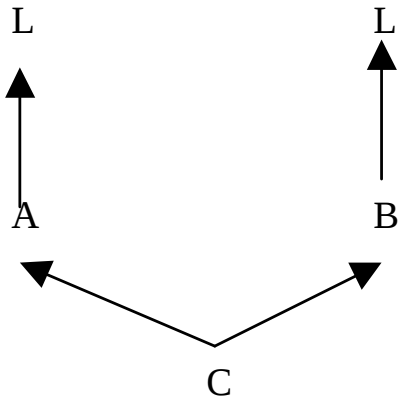
vícenásobné dědění

- v případě, že je výhodné aby objekt dědil ze dvou (či více) C++ toto dovoluje (jedná se ovšem o komplikovaný mechanismus)
- 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

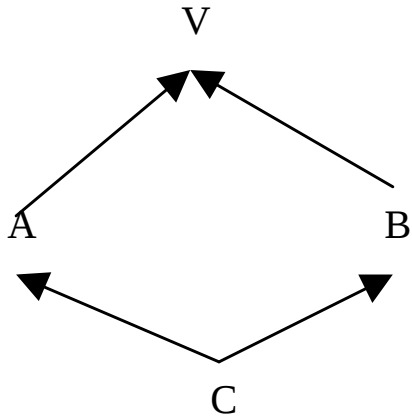
vícenásobné dědění

A: virtual V

B: virtual V

C: public A, public B

(konstruktor V se volá pouze jedenkrát)



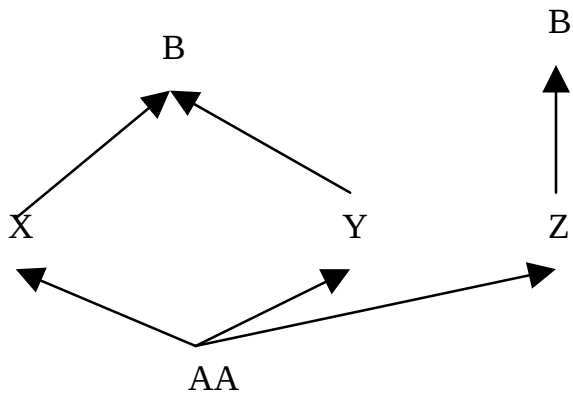
vícenásobné dědění

X: virtual public B

Y: virtual public B

Z: public B

AA: public X, Y, Z



Virtuální metody - polymorfismus

- potomka lze použít v místě, kde je možné použít předka
- 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 vykreslí grafického objektu, který zadal až uživatel v době chodu programu), je nutný mechanismus, 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,
- tvrn - 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.

Virtuální metody

- jsou-li v bázevé 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, a překryje původní. Tomu lze zamezit použitím klíčového slova final za definicí
virtual int Metoda(int i) const final {}
- 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 destruktory nedědí, může/musí být virtuální (je-li dědění)
- Využívá se v situaci, kdy máme dosti příbuzné objekty, potom je možné s nimi jednat jako s jedním – jednotný interface (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  
}
```

Virtuální metody

- Společné rozhraní – není třeba znát přesně třídu objektu a je zajištěno (při běhu programu) volání správných metod – protože rozhraní je povinné a plyne z báze 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 (tvm) a statickou – která je dána typem na který v dané chvíli ukazuje (překladač).
- Není-li metoda označena jako virtuální – použije se nevirtuální (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. Dynamická/pozdní vazba znamená, že se volá metoda typu, pro který byl vytvořen objekt

Virtuální metody

- 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 – přepíše se adresy předdefinovaných 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 (?).

Virtuální metody

- 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 destruktory nedědí, destruktory v děděné třídě přetíž (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 vypsát 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/nullptr)
- tato metoda se dědí jako čistě virtuální, dokud není definována
- starší překladače vyžadují v odvozené třídě novou deklaraci anebo definici
- obsahuje-li objekt č.v.m. nelze vytvořit jeho instanci, může být ale ukazatel

```

class B {
public: virtual void vf1() {cout << "bv"; }
        void f()          {cout << "bn"; }
class C:public B{
        void vf1()        {cout << "cv";}    // virtual nepovinné
        void f()          {cout << "cn";}}

class D: public B {
        void vf1()        {cout << "dv";}
        void f()          {cout << "dn";}}

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 báзовou třídu
// (přiřazení může být i v ifu, a pak se neví co je dál)
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

```

abstraktní базové 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 базový 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
```

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 kód pro obecný typ a překladač si potom vygeneruje podle něj kód 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
- umísťuje 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;  
int i;  
d = max(e,f);  
d = max(e,i); //nelze, parametry jsou různého typu  
d = max<float>(e,i); // typ T stanoven explicitně
```

- 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>
```

„chytré“ ukazatele – `unique_ptr`, `shared_ptr`, `weak_ptr`

- „obaly“ pro ukazatele, které mají „vylepšené“ vlastnosti (například při konci života odalokují naalokovanou paměť)
- „bezstarostné“ ukazatele – umožňují volněji programovat a zamezit leakům v paměti
- výhoda při výjimkách – odalokuje se paměť
- ve standardní knihovně `<memory>`
- `unique_ptr` pro unikátní vlastnictví paměti – je to „handle“ na ukazatel, při přiřazování mezi těmito `_ptr` se objekt přesouvá „move“
- `shared_ptr` pro společné sdílení paměti – obsah se „kopíruje“ – vytváří se odkazy a k odalokování dochází až při rušení posledního prvku, který na paměť odkazuje

```
{  
int *pu = new int;  
unique_ptr<double> dp = new double;
```

```
if ( ) return 1; // neodalokuje pu, dp se odalokuje  
if ( ) throw "chyba"; // neodalokuje pu, dp se odalokuje
```

```
delete pu; // odalokuje pu  
} // zanikne dp  
ukazatele auto/unique/ shared_ptr  
move_ptr counted/ staré názvy
```

„chytré“ ukazatele 34.3.1 strous

- "chytré" ukazatele
- ukazatele jsou v <memory>, -> v prostoru std
- automatické odalokování naalokované paměti přístupné pomocí tohoto objektu
- "garbage collector" - automatické odalokování v místě zániku "nosiče"
- pozor v některých částech (výjimka v konstruktorech ...) je nutné ošetřit jinak

auto_ptr (před C++11)

- starý mechanismus nahrazený `unique_ptr` (podobná - funkce, bezpečnější, umí pole...).
- stávají se vlastníkem předaného objektu
- "garbage collection" funkce - odalokuje vnitřní ukazatel při svém zániku.
- každý ukazatel může vlastnit pouze jeden `auto_ptr` (jinak by byl dvakrát odalokován)
- přiřazení dvou `auto_ptr` je realizováno pomocí přesunu (move přiřazení) - původní `auto_ptr` má nyní ukazatel `nullptr`
- přístup k hodnotám pomocí operátoru `*` (vrací typ pointeru podle typu `auto_ptr`) a `get()` (vrací ukazatel na "skutečný" uložený typ). V případě, že je `auto_ptr<base>` a je do ní vložen ukazatel na potomka, potom přístup přes `*` vrací `base`, zatímco `get()` umožní pracovat s potomkem.
- metoda `release` vrátí objekt, a `auto_ptr` "vlastní" `nullptr`

```
{
auto_ptr<double> // parametr double, ale jedná se o ukazatel
    apd (new double); // vytvoří se nový objekt,
// ukazatel se vloží, apd je jediným vlastníkem objektu
    *apd.get() = *apd; // dva způsoby přístupu (get lepší)
    double *up = &*apd ; // zjištění adresy, operátor * vrátí
// dereferencovaný vnitřní prvek,
// operátor & zjistí jeho adresu

} // zaniká objekt apd a součástí jeho zániku je i zrušení
// vnitřního objektu - pomocí ukazatele na double
```

unique_ptr

- stávají se (jedinými) vlastníky objektu vloženého pomocí ukazatele
- při svém zániku odalokuje odkazovaný objekt
- má specializaci pro pole
- move přiřazení
- podporuje operátory *(vhodné pro základní typy), ->(vhodné pro přístup do struktury), [] (bez pointerové aritmetiky)

pro pole

```
unique_ptr <double []> x(new double[xx]);
```

přístup

```
x[i] nebo x.get()[i]
```

shared_ptr

- ukazatel se sdílí (spoluvlastní) mezi několika shared_ptr
- součástí je počítadlo, které počítá, kolik objektů na vložený prvek odkazuje
- "odalkování" nebo zrušení probíhá tak, že se odečítá počítadlo. Poslední prvek zruší i odkazovaný objekt
- make_shared - vytvoří shared pointer tak, že z dodaného objektu si vytvoří kopii

weak_ptr

- realizuje dočasné vlastnictví
- nevlastní vložený ukazatel
- umí sdílet ukazatele s shared_ptr, aniž by je vlastnil
- pro přístup nutno zkonvertovat na shared_ptr pomocí lock (tím se "zablokuje" případné zrušení původního shared_ptr v této době)

```
weak_ptr<double> xx;  
shared_ptr<double>bb (new double);  
xx = bb;  
...  
shared_ptr aa = xx.lock();  
if (xx)  
    // ukazatel je v pořádku=bb ještě existuje  
else  
    // ukazatel není v pořádku = bb již zaniklo  
  
// jednodušeji  
if (xx.expired())  
    // bb je zrušeno  
else  
    // bb je ještě validní
```

Dynamická identifikace typu (Run Time Type Identification)

- slouží ke zjištění "skutečného" typu nebo srovnání "stejnosti" dvou typů
- třída `std::type_info`
- operátory (klíčová slova) – **`typeid`**, **`xxx_cast`**
-

`typeid`

- slouží ke zjištění typu výrazu za chodu programu nebo porovnání typů dvou proměnných
- vrací objekt třídy **`std::type_info`**
- za chodu umí určit pouze typ třídy s virtuální metodou (polymorfizmus)

`typeid`

- v hlavičce **`typeid`**
- porovnání "tříd" pomocí v ní definovaných operátorů (`==` a `!=`)
- metoda **`name`** pro zjištění jména typu (`char *`)

`typeid(aaa).name`

dynamic_cast<typ>(prevadeny_vyraz)

- přetypování mezi potomky (jinak vrátí nullptr)
- pro "virtual"
- pokud máme v poli ukazatele na bázi, přetypujeme na "skutečný" typ (potomka), nebo naopak
- přetypovává se ukazatel nebo reference

static_cast

- obyčejná přetypování
- dále využití k přetypování mezi "nevirtuálními" potomky
- nekontroluje převáděné typy (musí však existovat platný konverzní mechanismus)

const_cast

- manipulace s const a volatile u proměnných
- typ zůstává zachován

reinterpret_cast

- převody čísel na ukazatele a zpět
- převody ukazatelů na různé typy mezi sebou
(nebezpečné převody)

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 mechanismy 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 "**io**manip****"
 - 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 mechanismy jako vstup a výstup pro konzolu
- přetížení operátorů >> a <<
- fstream, ofstream, ifstream, iostream
- 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("navez 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

```
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);
```

- ios::nocreate - nově nepodporováno - otevře pouze existující soubor (nevytvoří)
- ios::noreplace - nově nepodporováno - otevře pouze když vytváří (neotevře existující)

zdroj: www.devx.com

záměna nocreate

```
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 noreplace

```
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::badbit | 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), базová třída nezávislá na typu* – *iosbase*
- **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 řetězcí, paměť pro práci může být parametrem konstruktoru – strstream.h
- třídy xxx_withassign – rozšíření (istream, ostream) , přidává schopnost přesměrování (např. do souboru,) – cin, cout
- cstream – 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 vrátí **!sout.fail()**. (tedy nevrací good).

deklarace třídy uvnitř jiné třídy (o)

- jméno vnořené třídy (struktury) je lokální
- vztahy jsou stejné jako by byly definovány nezávisle (To je, ve třídě A máme jednodušší zápis přístupu k B, ale přístupová práva jsou stejná, jako by byla B definována mimo A.)
- jméno se deklaruje uvnitř, obsah vně
- použití pro pomocné objekty, které chceme skrýt

```
class A {  
    class B; // deklarace (názvu) vnořené třídy  
    B y; // objekt třídy B definován ve třídě A  
    .....  
}
```

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

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

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
```

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 - není-li konverze (např. zde
```

```
// se prohlašuje 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 různé i když
```

```
// vstupy jsou stejné
```

```
if (n!=p) return 1;
```

```
return 0;
```

```
}
```

```
//===== komplex2214.h - hlavička třídy =====
```

```
// trasujte a divejte se kudyma to chodí, tj. zobrazte *this, ...
```

```
// objekty můžete rozlišit pomocí 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 přiřazení,

// implementováno i zřetězení

Komplex & operator+=(Komplex &p)

{Re+=p.Re;Im+=p.Im;return *this;}

```
// návratový prvek je stejný jako ten, který to vyvolal, takže 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;}  
// může 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, že je to nestandardní pro tento operátor
```

```
// a zároveň se mohou plést. Takže bezpečněji 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 použít pomocné proměnné, protože
// je nutné použít 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 špatně v případě, že použijeme pro a *= a;, potom první
// přiřazení změní i hodnotu p.Re a tím nakopne výpočet druhého
// parametru (! i když 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 složky jil' 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);  
}
```

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 destruktů (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 báze třídy (existuje-li bázeová 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 destruktů – 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 mechanismu 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 příklad na mechanismus, 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(); // vrátí 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 vrátí 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