Výuka jazyka C++ pomocí realizace příkladu s využitím třídy pro komplexní čísla formou otázek a odpovědí

(vývojová verze 0.08)

Struktura textu je taková, že úvodní texty a návodné texty jsou psány italikou. Otázky jsou číslovány X.X a jsou modré. Následují odpovědi. Snažte se zobrazit pouze otázku a odpovědět. Až potom si přečtěte odpověď.
Jelikož programování je svým způsobem umění vytvořit celek z dostupných částí, existuje více správných řešení a tedy vaše řešení může být správné, i když se od uvedeného liší.
Text se zaměřuje především na základní použití mechanizmů C++ (na praktickou stránku). Pro získání detailnějšího a komplexnějšího přehledu doporučujeme použít dostupnou literaturu.

Označení (PC) znamená, že by mělo být Programováno v C – a tedy realizováno na výpočetní technice.

Na konci kapitol je souhrnný projekt, který svou obtížností odpovídá v té době probrané látce a má za úkol procvičit tuto látku.

Tento text by měl být doplněn/podpořen studiem s učebnicí C++ (Pratta, Virius, Herout, …)

Projekt komplexní čísla, vykreslení grafu

Navrhněte program pro vykreslení frekvenčních charakteristik ze standardního přenosu v komplexní rovině a logaritmických souřadnicích.
Potřebujeme typ komplexní číslo -> zkusíme ho vytvořit.

1 Rozbor úlohy - Funkce pro vykreslení charakteristiky - přenos

Přenos je komplexní funkce komplexní proměnné, F(p) = F(a + jω) = … nebo F(jω) =…. Znamená to tedy, že se vypočte funkce pro měnící se komplexní parametr a výsledky (komplexní čísla) se vykreslí do komplexní roviny, nebo ve formě dvou grafů: amplituda a fáze.

1.1 Napište funkci, která bude mít v čitateli polynom prvního řádu, ve jmenovateli polynom třetího řádu. Koeficienty čitatele budou b, koeficienty jmenovatele a, proměnná polynomu p. Zapište matematicky jako vzorec a v notaci pro MATLAB. F(p) = ?

$$F(p) = \frac{b_1 p + b_0}{a_3 p^3 + a_2 p^2 + a_1 p + a_0}$$

F1 = (b1 * p + b0) / (a3 * p^3 + a2 * p^2 + a1 * p + a0)

Koeficienty a,b jsou v daném případě reálná čísla, parametr p je komplexní. Výsledek F1 je také komplexní. Z hlediska frekvenční charakteristiky jsou koeficienty a,b konstanty, p = jω je proměnná.

Znak ^ se v MATLABu používá pro mocninu. !! Pozor v jazyce C znamená něco jiného a pro mocninu se používá funkce pow (tj. power=mocnina) !!
Bylo by výhodné, kdyby náš program „uměl“ MATLABovskou notaci.

2 Třída – nový datový typ - Objektově Orientované programování

V jazyce C++ je možné vytvořit nový datový typ, který „umí“ (je schopen) provést stejné operace jako základní datové typy. Novým datovým typem tedy pro nás bude typ pro komplexní čísla.

Název nového typu může být libovolný. Většinou se volí tak, že začíná T (Type/Typ), případně C (Class/třída), S (Struct/Struktura). Pro nás tedy (například) TKomplex.

Překladač umí nový typ přeložit (ve smyslu „volání“, tj. hlavičky) v situacích ve kterých umí přeložit základní typy. Překladač ale neví, co mají dělat (ve smyslu činnosti, tj. tělíčka), pokud ho to nenaučíme (nenapíšeme danou funkci).

Pro práci při tvorbě tříd platí, že pokud to umím se základním datovým typem, musí to jít (po změně názvu typu) i s novým datovým typem (pokud napíšu příslušné tělíčka, aby překladač věděl co dělat).

2.1 Se kterým datovým typem je lepší představy provádět? S typem int nebo double?
Při úvahách o operátorech přihlédněte k zápisu vzorce F1 = (b1 * p + b0) / (a3 * p^3 + a2 * p^2 + a1 * p + a0), který zároveň později povede k požadavku: „aby šlo používat vzorce stejným způsobem jako v MATLABu“.

Pro představu je lepší použít datový typ int, pro který dává smysl použití většího počtu operátorů (například zmíněný operátor ^, který pro double nedává smysl. I když i pro int je jeho činnost značně odlišná od plánované mocniny).

Z hlediska přesnosti by byl vhodnější typ double, ale ani int ani double nestačí k reprezentaci komplexního čísla (takže pro představu jsou rovnocenné).

3 Tvorba třídy – co by měl typ umět – vycházíme z kódu

Při tvorbě třídy můžeme vycházet z toho, co právě potřebujeme. Napíšeme tedy program (nový typ můžeme pro začátek zaměnit typem int, nejlépe pomocí typedef int TKomplex, abychom po změně nemuseli dělat v programu změny typu (mimo typedef). Tento postup je praktický pro okamžitou činnost psaní programu a pro úvodní úvahy. Je ale nepraktický pro budoucnost, protože kvalitní návrh třídy je možné provést jen z komplexního rozboru (ne z omezeného aktuálního výběru) – viz následující kapitola.

Napište funkci FrChar, která bude mít jako vstup koeficienty čitatele, jmenovatele (stejného řádku), řád, a která vypočte hodnoty pro frekvence od 0.01 až 1000, vypočtené hodnoty vrátí pomocí parametru. (???)

3.1 Jaká bude hlavička funkce? Definujte vstupy a výstupy

int FrChar(double aCitatel[], double aJmenovatel[], int aRad, TKomplex *aData, double *aFrekvence, int aPocetFr)

Funkce vrátí počet naalokovaných dat/prvků, nebo kód chyby.

Všimněte si, že díky typedef jsme odlišili „skutečné“ celočíselné hodnoty od hodnot, které se změní na komplexní čísla.

3.2 Jaké bude tělo funkce? Tělo funkce tvořte spíše jako tvorbu algoritmu s tím, že správné hodnoty budou vypočteny později (až se vytvoří datový typ pro komplexní čísla).
Použijte zatím typ int. Pro výpočet výsledku přenosové funkce použijte formát zápisu jako pro MATLAB (i když zatím nedává smysl). (PC)

Vypadá Vaše funkce nějak takhle?

```
int FrChar(double aCitatel[], double aJmenovatel[], int aRad, TKomplex *aData, double *aFrekvence, int aPocetFr)
{
naalokujte pole pro uložení výsledku
pro každou frekvenci {
    vypočtěte hodnotu čitatele (pro danou frekvenci)
    vypočtěte hodnotu jmenovatele (pro danou frekvenci)
    vypočtěte výslednou hodnotu přenosu (pro danou frekvenci) a výsledek uložte
}
return chyba nebo délka vektoru;
}
```

3.3 V jazyce C/C++ by kód mohl být například (PC):

```
double pom; int i,j;
for (i=0;i<Počet;++i) {
 TKomplex p = frekvence[i];
   {citatel = jmenovatel = 0; // naše = bude muset mít návratovou hodnotu, aby se chovalo standardně
  for (j=0;j<rad;j++) {citatel += a[j]*p^j; jmenovatel = jmenovatel + = b[j]*p^j ;}
  výsledek[i] = citatel / jmenovatel;
}
}
```

všimněte si řádku TKomplex p = frekvence[i];. Zde bychom potřebovali, aby se frekvence zapsala do imaginární složky. Ukážeme si i jiné způsoby, pro začátek bychom mohli uvažovat, že máme k dispozici ryze komplexní proměnnou i_komplex (předdefinovanou na 0 + 1i) a tedy můžeme zatím psát

TKomplex p; p = frekvence[i] * i_komplex;.
Zároveň zjišťujeme, že budeme kromě operátorů potřebovat i způsob/funkci pro vytvoření komplexního čísla, nastavení hodnot komplexního čísla a pro zjištění hodnot komplexního čísla.

3.4 Které funkce/operátory bude muset třída umět? S jakými datovými typy budou operátory pracovat. (Slovně/teoreticky)

Při úvahách přihlédněte k následujícímu kódu a výsledkům:

```
int i,j; double x,y;
xx = i  /  j; // jakého typu jsou parametry, a jaký je typ výsledku?
xx = x / y; // jakého typu jsou parametry, a jaký je typ výsledku?
xx = i  / x; // jakého typu jsou parametry, a jaký je typ výsledku?
xx = x /  i; // jakého typu jsou parametry, a jaký je typ výsledku?
```

(jedná se vlastně o čtyři operátory dělení, mající různé parametry)

Při rozboru kódu zjistíme, že budeme potřebovat operátory:
double * TKomplex pro násobení koeficientů s komplexním operátorem p, výsledkem by měl být přesnější z typů, tedy TKomplex (stejně jako v následujících případech)
TKomplex^int pro mocnění (porušujeme zde pravidlo, že operátor by měl pokud možno dělat v nové třídě věci stejné (nejhůře pak podobné) jako u standardních tříd – aby byl program čitelný. Na druhou stranu musíme splnit podmínku kompatibility zápisu s MATLABEM, které zde dáme přednost.)
alternativně: TKomplex^double – je univerzálnější než ^int, možná pomalejší (?)
TKomplex + TKomplex (nebo TKomplex +=TKomplex v závislosti na použitých operátorech – zkuste čitatel jedním a jemnovatel druhým operátorem) pro sčítání prvků v čitateli a jmenovateli
TKomplex = TKomplex - pro přirazení výsledku výpočtu na pravé straně (a = b * c, kde výsledkem b *c je TKomplex)
TKomplex / TKomplex (nebo TKomplex/=TKomplex) pro výpočet podílu čitatele a jmenovatele

4 Tvorba třídy – co by měl typ umět – vycházíme z rozboru

Tvorba třídy se skládá z několika úvah (které je nutné řešit současně (nebo v návaznosti) jako celek)

4.1 Jaká data má komplexní číslo (a tedy i třída pro tento typ)? Proveďte diskuzi nad dvojím možným vyjádřením komplexního čísla, který formát je lepší? Jaký standardní datový typ pro tato data použijete?

Komplexní číslo má dvě složky. Existuje složkový zápis (reálná a imaginární část) a exponenciální (Amplituda a fáze). První zápis je vhodnější pro sčítání, druhý pro násobení (mocniny…). Jelikož obecně nevíme, které operátory se budou vyskytovat častěji, můžeme oba zápisy považovat za rovnocenné.

Druhou otázkou je zda používat jeden z formátů (dvě proměnné na typ), nebo oba zaráz (čtyři proměnné na typ). Patrně lepší přístup (z hlediska orientace) je lepší mít dvě proměnné a v případě potřeby přepočítávat, než po každém výpočtu (nezapomenout) dopočítat druhý typ.

Pro reprezentaci dat (složek) je vhodný reálný datový typ – double (pokud nepotřebujeme extrémní přesnost).

U tříd dále mohou existovat pracovní/režijní/pomocné proměnné. Například proměnná index s jednoznačnou identifikací proměnné, nebo proměnná pro uložení stavu proměnné (neinicializovaná/inicializovaná/neplatná (po chybě)/…)

Další částí je vnik a zánik typu. Jedná se o stav, který známe u standardních typů jako definici s inicializací. Zánik standardních typů realizuje překladač a programátora se netýká.

4.2 Co se děje při následujících definicích? int aa, bb = 3, cc = bb, dd = 8.5;

Pro prvek aa se pouze vytvoří paměť na uložení, inicializace konkrétní hodnotou se neprovede. Jedná se o nejjednodušší vytvoření prvku, které se používá, není-li uveden parametr – proto se nazývá implicitní konstrukce.
Paměť rezervovaná pro prvek bb naplní hodnotou 3. Z hlediska mechanizmu překladu znak '=' neznamená přiřazení ale inicializaci při vytvoření prvku (konstrukci hodnoty nového prvku na základě hodnoty). Jelikož se jedná o přiřazení int hodnoty do typu int, vytváří se kopie. Jedná se tedy o konstrukci kopírovací.
Paměť prvku cc se naplní hodnotou v prvku bb. Jelikož oba prvky jsou typu int, jedná se o předchozí případ, kdy se vytváří kopie.
Paměť prvku dd se naplní hodnotou 8.5 (?). Hodnota 8.5 je ovšem typu double. Proto musí dojít ke změně = konverzi na cílový typ, tj. typ int. Proto se inicializace (každá inicializace s jedním parametrem jiného typu) nazývá konverzní.

U nových datových typů můžeme činnost i typ a počet parametrů nadefinovat. Znamená to, že i implicitní konstrukce může provádět inicializační činnost (na rozdíl od standardních typů). Dále můžeme vytvářet inicializaci s více hodnotami. Proto je zavedena (i pro standardní typy) možnost inicializaci zapisovat jako funkci, ve tvaru proměnná a v závorce parametry, ze kterých bude vytvořena. Pro náš příklad je možné psát i: int aa, bb(3), cc(bb), dd(9.5);

4.3 Jaké způsoby inicializace byste navrhli pro typ TKomplex a co by dělaly? (Zatím pouze ekvivalenty inicializace z minulého zadání). Nejprve popište slovně bez návaznosti na jazyk C++. Uveďte zápis definice proměnných s inicializacemi, které navrhnete. Nejprve použijte předchozí definici pro int, kde zaměníte typ na TKomplex a popište, jaká by byla činnost.

Slovně (všimněte si, že tato část přesně popisuje, co se stane, ale vychází pouze ze znalosti komplexních čísel a formulací činností. Neříká, jak se bude realizovat v programu.): Proměnná pro komplexní číslo se bude vytvářet bez dodaných dat – výsledkem bude komplexní číslo s nulovými hodnotami. Proměnná se bude vytvářet jako kopie jiného komplexního čísla. Proměnná komplex se bude vytvářet z jiného základního datového typu (hodnotového, mimo ukazatelů …) tak, že se tato hodnota stane reálnou složkou, imaginární složka bude nulová.
TKomplex aa, bb(3),cc(bb),dd(9.5);
Zde se odrazíme od pravidla, že co funguje pro int, musí fungovat pro náš typ. Zbývá „pouze" říci co se bude

dít.
Pro aa se vytvoří „implicitní“, „prázdný“, … prvek. Názor na to, co to je, a jak to vypadá, se může lišit. My zvolíme možnost, že vznikne komplexní číslo s nulovými složkami. Všimněme si, že navrhujeme implicitní inicializaci, která u standardních typů nic nedělá, ale u nových typů je možné ji vytvořit/předepsat.
Pro bb(3) se jedná o konverzi z typu int. Komplexní číslo má ale dva parametry. Opět můžeme realizovat prakticky cokoli nás napadne, a dokážeme to naprogramovat. Ale nejrozumnější patrně bude přiřadit hodnotu do reálné složky (což se děje velice často) a složku imaginární vynulovat.
Pro dd(9.5) se opět jedná o konverzi, tentokrát z typu double. Otázkou je, zda potřebujeme konverzi z int i z double, které dělají totéž. Výsledkem úvahy by asi bylo, že konverzi z int nepotřebujeme, protože při volání s typem int ho dokáže překladač konvertovat na double a zavolat inicializaci s typem double.
Pro cc(bb) se jedná o vytvoření kopie. Činnost je jasná – reálná složka nové proměnné cc bude kopií reálné složky původní proměnné bb, a podobně pro složky imaginární.

4.4 Jaké další způsoby inicializace byste navrhli pro typ TKomplex a co by dělaly? Uveďte textový popis, zápis definice proměnných s inicializacemi, které navrhnete.

Obecně slovně: Proměnná typu TKomplex půjde vytvořit na základě řetězce ve formátu …, dále půjde vytvořit ze dvou hodnot, kde první hodnota bude reálná složka. Proměnná půjde také vytvořit tak, že načte data z otevřeného souboru (kde budou uložena ve stejném formátu jako pro inicializaci z řetězce).
TKomplex ee(“10+56i“),ff(“<12;13.5>“), gg(5,8.3), hh(6.2,7,0), mm(otevreny_soubor);
Z řetězce (může být i více typů formátů řetězců, pokud je uvnitř jediné funkce rozpoznáme a zpracujeme)
Z reálné a imaginární složky.
Z amplitudy a fáze. Nutno odlišit od předchozího - typem to rozumně nelze, takže počtem parametrů (poslední parametr slouží k odlišení, ale jeho hodnota se nevyužije).
Načtení hodnot z otevřeného souboru.
a další …

4.5 Stejně jako vznik proměnné je možné ovlivnit i zánik proměnné – k tomu slouží funkce nazývaná destruktor, která řekne, co se má dít při zániku proměnné. Vyjmenujte činnosti, které je nutné udělat před zánikem prvku (nápověda: soubory, paměť, vypočtená data …).

Jelikož se zánikem prvku „zmizí“ i jeho data, je nutné ošetřit situace, kdy by tato ztráta mohla být „nepříjemná“. To jsou například následující situace:
- zmizí vypočtená data – je nutné je nejprve uchovat/uložit,
- zmizí unikátní odkaz na získané zdroje (například ukazatel na alokovanou paměť – nutno odalokovat, odkaz na otevřený soubor – nutné uzavřít, …)
Při „záchraně“ dat je ale nutné uvažovat o vlivu na obsluhu – „grafický“ dotaz na uložení dat nemůžeme použít u typu, jehož proměnných jsou v programu vytvořeny desítky a více (komplexní číslo, řetězec, matice …). Toto je možné pro jednu či dvě hodnoty (například dokument word, excel …)

Pro datové typy s více proměnnými je nutné navrhnout i zadávání a čtení hodnot. Při zadávání je nutné uvažovat i způsoby kontroly správnosti dat (například, že délka nemůže být záporná …).
Stejně je nutné vyřešit i čtení hodnot. Zadávané/čtené hodnoty nemusí být přístupné přímo, ale mohou být přepočítávány (například z více „vnitřních" hodnot bude jedna výsledná). Může také existovat více výstupních reprezentací (pohledů) na data.

4.6 Popište/navrhněte způsoby nastavování a čtení dat pro typ TKomplex.

Nastavování je vhodné uvažovat pro jednotlivé složky a jako celek – odpovídaly by tomu názvy funkcí SetReal, SetImag, Set (=SetKomplex). Otázkou je, jak by se první dvě funkce zachovaly ke „druhé" složce (zda by ji nulovaly, nebo ji ponechaly nezměněnou). Podobný problém vyvstane zřetelněji při druhém způsobu, kdy dává smysl patrně pouze společné zadání amplitudy a fáze SetAF a následný přepočet na vnitřní data reálné a imaginární složky.
Můžeme také vytvořit proměnnou z řetězce Set("10+5i"). Funkce se může jmenovat stejně jako jiná, pokud má jiné parametry (zde má funkce set jeden parametr char*, a tedy se výrazně liší od Set uvedené výše, která má dva parametry typu double).
Pro čtení by bylo vhodné volit GetReal, GetImag. Také pro druhou reprezentaci GetAmpl, GetFaze. Představené by mohly vracet přímo dotazované hodnoty. Pokud by byl požadavek na získání obou parametrů zaráz, musely by být vráceny pomocí dvou parametrů funkce (např.) GetRI, GetAF.

V případě implementace servisní nebo chybové proměnné by bylo nutné uvažovat i o manipulaci s těmito proměnnými.

4.7 Navrhněte funkce pro práci s komplexními čísly. (Mimo operátory zmíněné dříve)

Abs
Doplněk
Inverze
Mocnina
Porovnávání podle různých kritérií (Problémem je, že jsou zde různé možnosti zápisu a tedy více hodnot, ale výsledek je jen jeden. Jedno, základní porovnání, by bylo vhodné zvolit pro operátory ==, <.> …)
a mnoho dalších

4.8 Hodnoty je také vhodné ukládat (zobrazovat) a načítat – navrhněte vhodný textový formát pro typ TKomplex.

Možné formáty již byly uvedeny dříve při inicializaci. Je možné volit libovolný formát. Formátů je možné volit i více (na úkor přehlednosti a jednoznačnosti čtení a složitosti obslužných funkcí). Obecně je vhodné, aby byl program schopen načíst řetězec, který vytiskne. Při tvorbě je důležité vhodně zvolit oddělovače (například čárka jako oddělovač může být zaměněna s desetinnou čárkou (při „českém" znakovém prostředí), neoddělené proměnné mohou splývat při ukládání polí – srovnejte orientaci v poli komplexních čísel:
15;32;35.5;123;54.77;1;
<15;32><35.5;123><54.77;1>

4.9 Díky novým možnostem bylo možné realizovat jednotný způsob vstupu a výstupu, Proto je nutné vybrat jeden z možných formátů jako prioritní a napojit ho na tento mechanizmus.

Pro tento mechanizmus jsou použity operátory bitového posuvu << a >>.

4.10 Projekt: Navrhněte (slovně popište) požadavky na vytvoření třídy pro dvourozměrné pole. Požadavky popište slovně (laicky, bez nutnosti znalosti jazyka. Napište, co by se Vám líbilo, aby typ Pole2D uměl). Návrh napište v kategorií: „proměnnou typu Pole2D by bylo potřeba vytvořit: bez parametrů, v případě jednoho parametru typu … bude tento znamenat …, v případě jedné proměnné typu char ukazatel bude v této proměnné řetězec pro tvorbu proměnné ve formátu … … … . Při zániku proměnné ne/chci uložit data. Hodnoty v proměnné chci číst, … Práce s proměnnou budou realizovány funkcemi …, které dělají … Pole bude umět operátory: =, +, +=, … > … , které budou dělat tyto činnosti …

…

Pozn.:

Zadání by nemělo jít příliš "hluboko", řešením vnitřní reprezentace (proměnných). To není na škodu, pokud je uvedeno, ale vede to k tomu, že se v tom motá více pohledů a celkově je pak zadání slabé.

Různé formy zadání by bylo vhodné od sebe oddělit (obecné zadání popisující pouze činnosti; zadání rozšířené o datové typy členských dat a datové typy parametrů funkcí; konkrétní realizace)

Zadání by se mělo napsat ve stylu textu "co po té třídě (času, datumu, zlomku…) budeme chtít, co by se s tím dalo dělat" bez návaznosti na jazyk C/C++.

Napsat tedy "hodnota bude v intervalu od ... do ... a to pouze celá (nebo desetinná, nebo řetězec) čísla" a při další fázi (až se změníte na programátora) teprve uvažovat o řešení.

Dále slovně uvést jak proměnná vzniká (a jak se dodaný údej reprezentuje) - tj. bez údaje, s jedním/2/3/.../100 údaji a jak se z těch údajů vytvoří výsledek, když jsou těmi údaji čísla, znaky ... (například jak se z jedné hodnoty typu int vytvoří čas (jsou to např. sekundy), zlomek (je to celá část), interval (je to vyšší mez, nižší je nula) ...)

Dále nepište to ve stylu "... může ... , ... nebo ..., ". Teď vytváříte zadání, které musí být jednoznačné - jinak se to podle něj nebude dát naprogramovat.

Dále nezapomínat na operátory - například řeknu, že chci hodnoty sčítat (a musím napsat, jak se podle mě součet chová - ale opět slovy - například "je-li součet větší než 100 (nebo 200), upraví se výsledek tak aby byl v intervalu 0-100 (0-200)".

U odečítání například "je-li výsledek záporný změní se na kladný tak, že..."

Zkrátka dáváte programátorovi pokyny, co to má dělat a jak si to představujete - vlastní realizaci pomocí jazyka zatím neřešte.

Vypracujte samostatně. V dalších částech naprogramujte a zhodnoťte, zda vaše představy byly v pořádku. Nedostatky plynoucí z nedostatečného pochopení návrhu v této fázi učení opravte.

5 Realizace třídy – základy

Při psaní (složitějších) programů v jazyce C docházelo k tomu, že funkce měly často velké množství parametrů, které byly navíc pro určitou skupinu funkcí stejné (například pro každé komplexní číslo je nutné předat dvě proměnné). Nutnost zjednodušit a zpřehlednit program logicky vedla k častému používání složeného datového typu, který umožňoval sdružit související proměnné do jedné.

5.1 Jako výchozí byl pro objektový návrh jazyka C++ vybrán složený datový typ existující v jazyce C.
O který datový typ se jedná?
Nadefinujte tímto způsobem datový typ pro SKomplex pro komplexní čísla. (PC)

Jedná se o složený datový typ struktura.
Definice typu
struct SKomplex { // deklarace=uvedení názvu nového datového typu
double iRe, iIm;   // definice proměnných nového datového typu.
};

5.2 Nadefinujte ve funkci main proměnnou struktury z minulého bodu. Co se děje při překladu s kódem z minulého bloku (definice struct) a co se děje při definici proměnné?

int main() {
struct SKomplex aa; // zápis práce se strukturou pomocí jazyka C – typ se jmenuje „struct SKomplex“
SKomplex bb; // při definici v jazyce C++ vznikne typ „SKomplex“ a struct není povinné uvádět.

Při definici struktury se nevytvoří žádný kód, jedná pouze o návod jak (v případě potřeby) vytvořit proměnnou daného typu.
Při definici proměnné se překladač „podívá“ na definici struktury (datového typu) a za její pomoci rezervuje v paměti dostatečné místo na uložení všech proměnných/struktury.

5.3 Která část definicí z minulých bodů patří do hlavičkového a která do zdrojového souboru?

Do hlavičkového souboru patří (a většinou tam bývá) definice datového typu – jedná se pouze o předpis, který netvoří kód a je nutné, aby definice byla pro všechny zdrojové texty jednotná -> umístění v hlavičkovém souboru.

Definice proměnné vytváří fyzicky proměnnou v paměti, což vylučuje její umístění v hlavičkovém souboru (při vícenásobném použití hlavičkového souboru by došlo ke kolizi jmen).

5.4 Rozdělte definice do hlavičkového a zdrojového souboru. (PC)

Po rozdělení je nutné naincludovat hlavičkový soubor do modulu, kde je nový datový typ používán.
Pozn.: nezapomeňte ošetřit hlavičkový soubor proti vícenásobnému načtení.

Objektový přístup v sobě sdružuje nejen práci s daty, ale zavádí i prvky „kultury" programování.

5.5 První kulturní/bezpečnostní mechanizmus již byl uveden (při rozboru). Který to je?

Jedná se o vytvoření/konstrukci a rušení/destrukci prvku. Jelikož je prováděno volání předdefinovaných funkcí automaticky, není při správně napsaném datovém typu možné, aby se pracovalo s neinicializovanou proměnnou, nebo aby proměnná skončila a zůstalo po ní „neuklizeno".

Dalším z těchto „kulturních" prvků je i trend/pravidlo, že „data jsou to nejcennější, co máme". Toto pravidlo vede k úvaze, že s daty by měl manipulovat jen ten, kdo přesně ví, co si může dovolit. A to není „běžný uživatel", ale pouze autor nového datového typu.

Z tohoto důvodu byl uveden nový složený datový typ – Třída. Při definici se s ním pracuje stejně jako se strukturou, pouze klíčové slovo struct je nahrazeno slovem class.

5.6 Do příkladu vedle struktury nadefinujte třídu CKomplex a vytvořte její proměnné.

Přístup je stejný jako u struktury

```
class CKomplex {  // deklarace=uvedení názvu nového datového typu
double iRe, iIm;    // definice proměnných nového datového typu.
};
```

```
int main() {
struct SKomplex aa; // zápis práce se strukturou pomocí jazyka C – typ se jmenuje „struct SKomplex"
SKomplex bb; // při definici v jazyce C++ vznikne typ „SKomplex" a struct není povinné uvádět.
class CKomplex caa; // zápis práce se třídou – class je nadbytečné, protože
CKomplex cbb; // při definici v jazyce C++ vznikne typ „CKomplex" a class není povinné uvádět.
```

5.7 Důvodem zavedení třídy byla ochrana dat (tj. iRe a iIm). Ověřte ve funkci main, že tyto položky ve struktuře jdou změnit, zatímco ve třídě změnit nejdou (překladač nepovolí přístup = nepřeloží).

```
aa.iRe = bb.Im * 5 + 4; // operace s datovými členy struktury jsou povolené
// (zpětná kompatibilita  s C – struktura se chová jako v C)

caa.iRe = 8; // pokus o práci (zde zápis) do proměnných struktury není možný – data jsou chráněna
aa.iRe = caa.iIm; // není možné použít ani na pravé straně (tj. číst hodnoty)
```

V dalším je vhodné si uvědomit, že až na tento rozdíl v přístupu k proměnným jsou třída a struktura rovnocenné (lze s nimi provádět stejné konstrukce)

5.8 Projekt: Navrhněte datové členy pro realizaci dvourozměrného pole. Napište základ struktury/třídy TPole2D odpovídající probraným tématům. Tj. proveďte definici struktury s proměnnými. Proveďte definici proměnné daného typu. Rozdělte do hlavičkového a zdrojového souboru.

Vypracujte/naprogramujte samostatně. Zhodnoťte návaznost na návrh z předchozích bodů. Nedostatky plynoucí z nedostatečného pochopení návrhu, které se ukázaly v této fázi tvorby třídy, zpětně zapracujte do návrhu, který příslušně opravte.

6 Metody, přístupová práva

U třídy není možné přistupovat k proměnným jednoduše jako u struktury. Hlavním důvodem je možná neukázněnost „běžného“ uživatele. Představme si, například datový typ, který může pracovat pouze s kladnými čísly. Pokud použijeme strukturu, může uživatel změnit hodnoty, aniž by o tom struktura věděla. Může se tedy stát, že zapíše hodnoty na záporné (nehledě na to, že může změnit hodnoty v rozpracovaném výpočtu). Potom by bylo nutné před každým výpočtem kontrolovat, zda nejsou hodnoty záporné. Tento přístup je dosti náročný.
Ve třídě jsou data implicitně schována. I když lze tato data zveřejnit jako u struktury, neděláme to, ale využijeme k přístupu k datům funkcí - metod.

Metoda je speciální funkce, která je přizpůsobena práci se třídou/strukturou. Jelikož je definována pro práci se třídou/strukturou, je definována ve stejném místě, kde jsou definovány proměnné třídy/struktury. Přístup k metodě je stejný jako u proměnné, ale navíc má „funkční“ závorky ().

6.1 Nadefinujte ve struktuře SKomplex metodu GetReal, která vrací double a nemá parametry (definice je stejná jako u funkce, pro začátek vraťte hodnotu nula). Ve funkci main zavolejte metodu GetReal.

```
struct SKomplex { …
double GetReal() {return 0;}
// definice metody se provede stejně jako by to byla funkce, vložená do definice struktury
 …};

double pom = aa.GetReal();  // příklad volání metody
```

Metoda GetReal nám ještě nevrací správnou hodnotu. Plný princip jak pracovat s hodnotami se dozvíme později, nyní si pouze řekneme, že překladač hledá nejbližší definici proměnné s daným názvem. Pokud tedy v metodě GetReal uvedeme proměnnou iRe, bude se nejprve hledat proměnná s tímto názvem ve funkci. Nebude-li zde nadefinována, bude se hledat v definici třídy/struktury, nebude-li ani zde, potom se bude hledat globální proměnná. Jelikož máme tuto proměnnou nadefinovánu ve struktuře, můžeme ji použít.

6.2 Upravte metodu GetReal tak, aby vracela proměnnou iRe. Doplňte i metodu GetImag pro získání imaginární hodnoty.

```
struct SKomplex { …
double GetReal() {return iRe;}
double GetImag() {return iIm;}
// definice metody se provede stejně jako by to byla funkce, vložená do definice struktury
 …};
```

Metody pro získávání a nastavení hodnot se někdy nazývají Gettery a Settery. Mohou vracet i hodnoty upravené/vypočtené. Stejně tak mohou nastavit hodnoty na základě výpočtu z dodaných dat.

6.3 Napište Gettery:
GetAmp pro vrácení amplitudy (délky fázoru daného komplexním číslem),
GetPhase pro vrácení fáze (fázoru amplitudy, daného komplexním číslem)

Dále napište Settery :
SetReal a SetImag mající jeden double parametr.
SetAF mající dva parametry, amplitudu a fázi

Ve funkci main nastavte proměnnou aa na hodnotu souřadnic -10,10 a bb na amplitudu 5 a fázi 90 stupňů. Výsledek zkontrolujte.

```
struct SKomplex { …
double GetAmp() {return sqrt(iRe*iRe+iIm*iIm;)}
double GetPhase() {return atan2(iIm,iRe);} // atan2 na rozdíl od atan vrací úhel od 0 do 360 místo 0-180

void SetReal(double aRe) {iRe = aRe;}
void SetImag(double aIm) {iIm = aIm;}

void SetAF(double aA, double aF)  {iRe = aA * cos(aF/180*3.1415); iIm = aA * sin(aF/180*3.1415);}
…};
```

v main
aa.SetReal(-10); aa.SetImag(10); // je vidět, že toto nastavení je složité, chtělo by to ještě aa.Set(-10,10)
x = aa.GetReal(); y=aa.GetImag(); // kontrola zadání a získání složkového tvaru
x = aa.GetAmp(); y=aa.GetPhase(); // kontrola zadání a získání fázového tvaru
bb.SetAF(5,90);
x = bb.GetReal(); y=bb.GetImag(); // kontrola zadání a získání složkového tvaru
x = bb.GetAmp(); y=bb.GetPhase(); // kontrola zadání a získání fázového tvaru

Hodnoty by bylo vhodné vytisknout. V C++ existuje mechanizmus tisku, kdy se všechny proměnné tisknou stejným způsobem, a překladač podle typu proměnné vybere správnou funkci. Stále se jedná o knihovní funkce (a ne klíčová slova jazyka) takže je nutné includovat správnou knihovnu (fstream). Jelikož všechny knihovní funkce jsou „schovány" v prostoru std („ochranný" mechanizmus jazyka C++, který vysvětlíme později), je nutné je zpřístupnit pomocí příkazu using namespace std;

Vlastní tisk se potom provádí pomocí operátoru << (načítání pomocí >>) a proměnné přiřazené konzole cout (pro načítání cin). Operátor/tisk lze použít řetězově. Tisk potom vypadá následovně:

cout << "Slozkovy tvar " << aa.GetReal() << " " << aa.GetImag << endl; // endl provede odřádkování

6.4 Proveďte tisk obou proměnných (aa i bb) v obou tvarech (složkový i fázový)

Správnost poznáte podle toho, jak to funguje.

Jak jsme si řekli, třída a struktura jsou si velice podobné. Již jsme narazili na problém s přístupem k proměnným. Tento mechanizmus souvisí s přístupovými právy. Jedná se opět o ochranný mechanizmus, který říká, ke kterým proměnným a metodám lze přistoupit „z venku" – tj. může je používat uživatel. Třída může používat vše co má „uvnitř" bez omezení.

Přístupová práva je možné nastavit pomocí přepínačů (následovaných dvojtečkou).
Pro přepnutí se používají klíčová slova (blok až do další změny má tato přístupová práva):
public: - přístupné všem, veřejné
private: - přístupné třídě, privátní, chráněné
později ve spojení s děděním se použije protected: (pro jednu třídu bez dědění stejné jako private).

struct SSS { // tady je implicitně public: *.. // toto je přístupné* *private: // přepnutí* *… // toto je nepřístupné* *public: // přepnutí* *… // toto je přístupné* *private: // přepnutí* *… // toto je nepřístupné* *};*	*class CCC { // tady je implicitně private:* *.. // toto neje přístupné* *public: // přepnutí* *… // toto je přístupné* *private: // přepnutí* *… // toto je nepřístupné* *public: // přepnutí* *… // toto je přístupné* *};*

6.5 Pomocí klíčových slov private a public nastavte ve struktuře přístupová práva tak, aby proměnné byly chráně/skryté/privátní a metody veřejné.

```
struct {
private:
// proměnné
public: // metody
};
```

6.6 Naprogramujte třídu CKomlex tak, aby se chovala jako struktura SKomplex. Metody fungují u třídy stejně. Rozdíl je pouze v místě umístění klíčových slov private a protected, která opět nastaví ve třídě přístupová práva tak, aby proměnné byly chráně/skryté/privátní a metody veřejné.
V main vyzkoušejte „volání“ metod stejně jako u struktury. (U struktury „přestane fungovat“ přístup k proměnným a class bude fungovat stejně jako struktura).

Zkopírujte metody do třídy a nastavte přístupová práva.

```
class  { // private: je implicitní, není nutné psát
// proměnné
public: // metody
};
```

Takhle by se měly používat přístupová práva. Proměnné až na zdůvodněné výjimky jsou private. Metody dělíme na (pro uživatele) bezpečné, které jsou složitější tím, že mají naprogramovány testy (ochrany). Metody bez ochran (rychlejší) nebo pomocné metody jsou potom v sekci private.
Privátní metodou pro typ čas by mohla být normalizace na sekundy a minuty aby byly v intervalu nula až šedesát. Tato metoda by se volala po výpočtech – využívala by se, ale uživatel by neměl proč ji volat.

7 Neobjektové vlastnosti jazyka

Neobjektové vlastnosti jazyka jsou vlastnosti, kterými byl jazyk rozšířen, ale netýkají se bezprostředně objektů. Tyto vlastnosti je možné použít i při programování ve stylu jazyka C.
Neobjektové vlastnosti byly přidány především z důvodů snadnější práce. Některé museli být zavedeny, protože bez nich by nešlo realizovat některé vlastnosti objektů (reference, manipulace s pamětí (new,delete), načítání a tisk (byť postaveno na objektech lze využít pro základní typy) …). Jiné mechanizmy pomohly kultuře programování (komentáře, definice (+inicializace) proměnné v libovolném místě, výjimky (byť mohou používat objektů)). Další pomohly k znovupoužití kódu (šablony). Ke zpřesnění psaných kódů (inline na místo #define, const, …)

Původní jazyk C umožňoval definovat proměnné pouze na začátku bloku, před blokem příkazů. Výhodou byla přehlednost proměnných a jejich typů, které byly na jednom místě.
Následně ovšem zvítězila myšlenka, která říká, že by v programu neměly být neinicializované proměnné. A

následně tedy vyplynula úprava povolující definici proměnné v jakémkoli místě, a tedy až v době, kdy přesně víme, na co budeme hodnotu potřebovat, a tedy známe typ i hodnotu.

7.1 Upravte následující kód tak, aby neobsahoval neinicializovanou proměnnou. void * uk;

Nabízí se void *uk = NULL;, což je správně z hlediska jazyka C. Jazyk C++ v rámci sjednocení různých reprezentací NULL zavedl nové (významově a typově přesně definované) klíčové slovo nullptr. Správný zápis tedy je:
void *uk = nullptr;

7.2 Jaké možnosti předávání parametrů do a z funkcí znáte?

Do funkcí se předávají parametry vždy hodnotou. V případě, že je proměnná velká (z hlediska zabrané paměti), je potřebné změnit ve funkci její hodnotu, nebo předáváme pole, potom předáváme proměnnou zprostředkovaně pomocí její adresy (předávanou hodnotou je adresa) = ukazatelem.

Další možností pro předání hodnot z funkce je využití její návratové hodnoty. Tato má nevýhodu, že lze použít pouze pro jednu hodnotu (byť při použití struktury můžeme vrátit v jedné struktuře více hodnot). Potřebujeme-li vrátit více proměnných, musíme použít ukazatele v sekci argumentů funkce.

7.3 Jaké výhody má předávání struktury/třídy pomocí ukazatele? Srovnejte s předáváním hodnotou.

Výhodou předávání pomocí ukazatele je rychlost a zabrání konstantního (malého) místa na zásobníku, protože se předává pouze ukazatel. Je nutné si uvědomit, že pokud se předává struktura hodnotou, je na zásobníku nutné vytvořit kopii proměnné. Jelikož se struktura používá pro složený datový typ, který má více proměnných, většinou zabírá struktura větší paměťový blok. Kopie tedy zabere na zásobníku blok paměti, následně je nutné vytvořit kopii hodnot – tato činnost trvá nějaký čas.

Nevýhodou je, že se nejedná o kopii, a proto nelze použít v případě, kdy potřebujeme proměnnou vně funkce zachovat, ale ve funkci její hodnoty měnit.

Z výše uvedených důvodů zabírání paměti a nutnosti tvořit kopii (čas), se dává přednost předávání parametrů (struktur) pomocí ukazatelů. Pouze ve zdůvodnitelných situacích volíme předání hodnotou.

Ukázalo se však, že předávání pomocí ukazatele není možné ve všech situacích spojených s prací s objekty (například operátory). Proto bylo nutné „vymyslet“ (aplikovat do jazyka) nový způsob předávání parametrů. Tento způsob předávání parametrů se v jazyce C++ nazývá reference a používá se k předávání odkazem. Referenci si lze představit jako přezdívku, alias ke stávající proměnné. Jedná se tedy o nové jméno pro stávající proměnnou. Obě (či více) proměnné se potom dělí o stejný paměťový prostor a tedy manipulace (včetně její změny) s oběma proměnnými vede ke stejným výsledkům.

Samotná reference (odkaz) se nedá měnit, je možné ji nastavit pouze v definici s inicializací. Pro definici se používá znak & (nezaměňovat s AND, nebo s operátorem pro získání adresy. Toto je další možné využití).

```
int ii = 4, j; // „normální“ proměnná
int &ri = ii; // int &ri je definice reference ri. Inicializace se provede pomocí přiřazení.
// ri a ii jsou teď z hlediska uložení v paměti totožné proměnné – zabírají stejné místo
j = ri;  // v j bude hodnota 4, protože ri je totéž co ii, které bylo inicializováno na ii. Stejná činnost jako j = ii
ii = 8;
j= ri; // v j bude hodnota 8
ri = 10;
j = ii; // v j bude hodnota 10
```

7.4 Využijte reference ke zjednodušení zápisu proměnné uvnitř struktury v následujícím příkladu.

```
struct A {double hodnota;}
struct B {struct A Promenna};  // definice struktur s vnořením

struct B val; // definice proměnné struktury B
val.Promenna.hodnota = 12.34; // zápis do proměnné
??? // realizujte zápis z minulého řádku pomocí proměnné pom,
// která bude referencí na použitou proměnnou
```

Řešením je vytvoření reference a její použití

```
double &pom = val.Promenna.hodnota; // typ musí být stejný jako „finální“ proměnná.
pom = 12.34 ; // ekvivalent předchozího zápisu – reference manipuluje přímo s odkazovanou proměnnou
// pom je totéž (je umístěna ve stejné paměti, jako val.Promenna.hodnota
```

7.5 Napište metodu Swap, která bude mít parametr aVal typu SKomplex předaný pomocí reference. Funkce Swap přepíše hodnoty z aVal do aktuálního prvku (který metodu zavolal) tak, že zamění reálné a imaginární složky. zkuste volání bb.Set(10,20); aa.Swap(bb) a aa.Swap(aa).

Do definice struktury dopíšeme

```
void Swap(SKomplex &aVal) // v metodě Swap budou dvě proměnné typu SKomplex.
// Proměnná, která ji vyvolala a proměnná, předaná jako parametr. Hodnota aVal bude ležet ve stejné paměti
// tj. bude pouze jiným jménem (aliasem) pro proměnnou, která bude při volání na místě parametru
{ // iRe = aVal.iIm; iIm = aVal.iRe; nebude fungovat pro a.Swap(a) – proč?
 double pom = aVal.iIm;
 iIm = aVal.iRe; iRe = pom;
}

// volání
aa.Swap(bb) ; // uvnitř metody buda aVal pouze jiným jménem pro bb.
```

```
// Přístup k aVal bude zároveň přístupem k bb
aa.Swap(aa) ; // obě proměnné uvnitř metody Swap budou aa, což by mohlo vést k chybě.
// Při každé tvorbě funkce je nutné ošetřit variantu, že proměnné budou tytéž.
```

Při volání pomocí reference je někdy potřeba zabránit změně proměnné uvnitř metody/funkce. K tomu slouží klíčové slovo const, které říká, že proměnná se nemá měnit a případný pokus o její změnu je odhalen při překladu jako error.

7.6 Upravte metodu Swap tak, aby nešlo změnit předávaný parametr.

```
void Swap(const SKomplex &aVal) // proměnnou aVal nebude možné v metodě změnit.
{ aVal.iRe = 5; // pokus o změnu povede k chybě překladu.
```

Při volání aa.Swap(aa); jsou sice obě proměnné stejné, ale uvnitř metody jsou brány jako obecné proměnné. Proto označení const u parametru (který je vlastně aa stejně jako druhá proměnná) nemá vliv na možnost přiřazení a tedy se bude měnit i aa (ale pomocí přístupu přes jinou cestu než přes aVal!!!).

Jak jsme si řekli, metoda se liší od funkce tím, že ji vyvolá objekt. Již jsme si také ukázali, že s proměnnými daného objektu můžeme v metodě pracovat. Nyní se podíváme na tento mechanizmus detailněji.
Aby se s proměnnou dalo v metodě pracovat, je nutné, aby v ní byla přítomna. O přítomnost proměnné v metodě se postará překladač.

kód	Možná představa realizace pro pochopení

a.Metoda(); // volání metody v kódu
 	
b.Metoda(); // volání téže metody jinou proměnnou
 	
// hlavička metody
int TKomplex::Metoda()
{
this->iRe = this->iIm; // „plný“ přístup k proměnným
// použití this-> je nepovinné. Plyne z kontextu
 	
}
	Metoda(&a); // překladač realizuje jako funkci
	// adresa objektu, který metodu volá,
	// je předána jako parametr
	Metoda(&b);
	
	Int Metoda (TKomplex *const this)
	// objekt, který metodu zavolal je předán
	// ukazatelem, a jmenuje se this
	// v návaznosti na výše uvedené volání je this
	// při prvním volání adresa proměnné a,
	// při druhém volání adresa proměnné b

7.7 Napište metodu Set nastavující parametry proměnné CKomplex, na základě jiné proměnné CKomplex, tak, aby:
- vracela nově nastavenou hodnotu pomocí reference (tj. vrátí objekt, který metodu zavolal s již přepsanými hodnotami)
- v případě, že oba parametry jsou stejné, neprovedla výpočet/kopii (zrychlí se metoda), (testuje se, zda adresa parametru předaného referencí je stejná, jako adresa prvku, který metodu zavolal)

```
CKomplex& Set(Ckomplex &aVal)
{
 if (this == &aVal) // je-li adresa prvku, který metodu volá stejná jako adresa parametru
    return *this; // vrátí se proměnná ihned
… // kopie
 return *this; // vrátí se aktuální (již změněná) hodnota
}
```

Volání x = a.Set(b); y = c.Set(c);

7.8 Napište funkce Min, mající dva parametry typu double, která vrátí minimální z obou parametrů. Uvažujte funkce Min s definicí
double Min1(double a, double b) {} a
double& Min2(double &a, double &b) {} tato vrátí odkaz na předaný parametr s menší hodnotou

Před realizací funkcí zkuste popsat, co se stane při voláních:
double p,q,r = 3, s = 5;
p = Min1(r,s);
q = Min2(r,s);
Min1(r,s) = 8;
Min2(r,s) = 8;
Min2(s,r) = 10;
Zkuste si nakreslit paměťové mapy pro umístění proměnných v paměti.

```
double Min1 (double a, double b)
{ return a < b ? a : b ; }

double& Min2 (double& a, double& b)
{ return a < b ? a : b ; }
```

Zápisy funkcí jsou až na hlavičku stejné. Činnost je však velice odlišná, což se projeví zvláště u použití funkce na levé straně rovná se.

p = Min1(r,s); Zde se zavolá funkce a uvnitř se vytvoří dvě nové proměnná a a b, které budou inicializovanými hodnotami z proměnných r a s (tj a se bude rovnat 3 a b 5). Ve funkci se vypočte minimum a to se vrátí jako hodnota. Tato hodnota je přiřazena do proměnné p. Proměnná p tedy bude obsahovat hodnotu 3, která se sem dostane postupným přepsáním z r do a, z a do návratové hodnoty a z návratové hodnoty do p.

q = Min2(r,s); Zde je situace podstatně jiná. Proměnné ve funkci jsou umístěny do stejného paměťového prostoru jako původní proměnně – jsou pouze novými jmény pro původní proměnné. Proměnná a je tedy pouze nové jméno pro proměnnou r a v paměti leží obě na tomtéž místě. Obdobně pro proměnné s a b. Následně se pracuje s hodnotami (nových i původních proměnných – protože jsou totožné) a najde se minimální hodnota. Jelikož se opět vrací reference, je návratová hodnota opět pouze odkazem (jiným jménem) použité proměnné. Vrátí se tedy odkaz/reference na proměnnou a (což je totéž jako r), nebo b (totéž jako s). To znamená, že na pravé straně je návratová proměnná (která nemá jméno), která je pouze jiným vyjádřením proměnné původní a leží v (odkazuje na) paměť původní proměnné – což je r nebo s.

Min1(r,s) = 8; nepůjde přeložit, protože výsledkem je hodnota double (například 12.34, navíc je to proměnná dočasná), a překladač nebude vědět jak z ní zjistit adresu pro uložení výsledku (jedná se vlastně o zápis 12,34 = 8)

Min2(r,s) = 8; zde je opět situace jiná a tato varianta přeložit půjde. Do funkce jsou předány odkazy a a b, které jsou vlastně pouze jiná jména pro r a s. Na konci funkce je vybrána jedna z proměnných a vrácena pomocí reference. Je tedy pomocí reference vrácená proměnná r nebo s, lépe řečeno dočasná návratová hodnota reprezentuje místo v paměti, které sdílí s proměnnou r, nebo s. Protože toto místo v paměti existuje a je možné do něj zapsat hodnotu, bude do něj hodnota 8 zapsána. Celkově tento řádek říká, že do proměnné s menší (vstupní) hodnotou se zapíše nově hodnota 8.

7.9 Napište makro DELENI(a,b) s funkčním voláním (pomocí #define), které vydělí dva parametry (tj. a/b).
Jaký bude výsledek v proměnné c pro definice proměnných d,e a volání:
a) int d = 6, e = 4; double c; c = DELENI(d,e);
b) double d = 6, e = 4; double c; c = DELENI(d,e);

```
#define DELENI(a,b) ((a)/(b))
```

pro a) bude výsledek c = 1, jelikož oba parametry jsou celá čísla a tedy dělení je celočíselné a výsledek je celé číslo. Teprve následně se toto celé číslo zkonvertuje na double a přiřadí do proměnné c.
pro b) ačkoli je makro stejné, bude výsledek c = 1.5. Jelikož jsou parametry double, počítá se s desetinnými čísly.
Můžeme tedy konstatovat, že za určitých okolností může být u maker nepříjemné to, že typ výsledku se liší podle parametrů.
Další nepříjemností maker je nutnost „ozávorkování“ proměnných.
Výhodou maker je to, že se nevolá funkce, ale dojde k rozvinutí do kódu přímo v místě použití.
Pozn.: Pro jednoduchost kódu není ošetřeno dělení nulou.

Makra s funkčním voláním by měla být postupně nahrazována mechanizmem inline funkcí. Jak plyne z klíčového slova inline, funkce se rozvine v místě použití a opět se nevolá. Na rozdíl od makra je ale v prototypu funkce udán typ argumentů i návratový typ, takže při rozvinutí do kódu překladač vloží navíc i přetypování na správný typ.

7.10 Pro makro z minulého příkladu by byla ekvivalentem (v rámci zmíněných pravidel) funkce:
inline double Deleni(double aa, double bb) {return aa/bb}
Funkce je napsána pro vhodnější typ (vhodnost určí programátor na základě plánovaného použití).
Napište, jak bude do kódu rozvinuto použití funkce při volání:
c = Deleni(5+a, 6 *b);

Výraz by byl nahrazen:
c = (double) (…) výsledek je přetypován na výstupní typ
c = (double) ((double)(5+a) …) argumenty jsou vyčísleny a přetypovány na vstupní typ
c = (double) ((double)(5+a) / (double) (6*b);) pro výpočet (pro realizaci kódu) bude použito tělo funkce

7.11 Napište inline funkci pro výpočet velikosti fázoru (vzdálenost od počátku) komplexního čísla z jeho složek.

```
inline double Delka(double aRe, double aIm) {return sqrt(aRe*aRe+aIm*aIm);}
```

K neobjektovým vlastnostem přiřadíme vlastnosti pro manipulaci s pamětí a pro řešení chyb. Jsou určeny pro práci s objekty, ale lze je využít i pro standardní typy.
Řešení chyb pomocí výjimek umožňuje oddělit řešení chyb od chodu programu. Parametry funkcí jsou určeny k předávání parametrů, chyby se řeší odděleně/paralelně pomocí výjimek.
Nutnost nových mechanizmů pro manipulaci s pamětí s sebou přineslo vytváření a zanikání objektů, které je doprovázeno inicializací (konstruktory) a „úklidem“ (destruktor). Jelikož z důvodu zpětné kompatibility toto není možné doplnit do malloc a free, které se musí chovat stejně jako v C, bylo nutné zavést nové mechanizmy, které by konstruktory a destruktory volaly. Pro získání paměti slouží klíčové slovo (operátor) new (pro pole new [ ]) a pro uvolnění paměti klíčové slovo (operátor) delete (pro pole delete [ ]).

Chybu při řešení programu (například dělení nulou, odmocnění záporného čísla) je možné řešit několika způsoby:
- vytisknout zprávu na obrazovku (obtěžuje a obsluha většinou neví co má dělat)
- upravit hodnotu tak aby s ní šlo pracovat (dále počítáme se špatnými hodnotami)
- pomocí returnů předávat chybovou zprávu až do místa kde ji vyřešíme (složité konstrukce)
Mechanizmus výjimek spočívá v tom, že v místě chyby popíšeme chybu a její vznik (například pomocí struktury, do které umístíme aktuální data a zprávu o chybě) a tento popis necháme putovat až do místa, kde se někdo přihlásí k řešení chyby (tj. oznámí, že struktuře rozumí a dokáže z ní zjistit původ chyby – tj. dělení nulou je chyba, ale mohla vzniknou z různých příčin na základě dodaných dat. Samotná funkce dělící nulou většinou počítá s čísly a neví co znamenají, proto popis chyby může učinit až funkce, která ví o jaká data se jedná.

Mechanizmus výjimek používá tři klíčová slova:
- mechanizmus řešení výjimek se provádí v označených blocích. Blok předchází klíčové slovo try
- za blokem pro řešení výjimek je seznam řešených výjimek. Výjimek může být více a rozlišují se podle datového typu. K zachycení výjimky se využívá klíčové slovo catch následované typem, který se má řešit a názvem proměnné do které se má výjimka uložit.
- klíčové slovo throw slouží k vygenerování („hození“) výjimky. To znamená, že se vytvoří proměnná libovolného zvoleného datového typu a přidá se za throw. Potom se přeruší řešení programu a program se postupně vrací ze zanoření ve volaných funkcí až do místa, kde je odchycena výjimka pomocí catch. Poslaná proměnná se zapíše do proměnné v hlavičce catch.

```
int delka; …
try { // začátek bloku pro řešení pomocí výjimek
if (delka <= 0)
 throw delka; // pokud je delka nulova nebo záporná, vytvoří se výjimka typu int s hodnotou proměnné delka
if (ukazatel == nullptr)
  throw (void*) ukazatel; // není-li ukazatel použitelný, vytvoříme výjimku typu ukazatel
…
} // konec bloku pro řešení výjimek
catch (int pom)
{ .. } // je-li vygenerována výjimka typu int, provede se tento blok s tím, že hodnota výjimky je v pom
catch (void *)
{ … } // je-li vygenerována výjimka typu ukazatel, provede se tento blok s tím, že hodnota mě nezajímá
catch ( … )  // jsou-li uvedeny tři tečky, odchytí se všechny výjimky
{ … }
```

7.12 Napište funkci Tst, která bude mít dva parametry typu int. Bude-li první parametr záporný, vytvořte výjimku typu int s hodnotou parametru. Bude-li druhý parametr záporný, vytvořte výjimku obsahující text “zaporna“, pro nulovou hodnotu druhého parametru potom výjimku obsahující text “nulova“. Ve funkci main zavolejte tuto funkci Tst a vyřešte vrácené výjimky tak, že je vytisknete.

```
void Tst(int aa, int bb)
{
if (aa < 0) throw aa;
if (bb < 0) throw “zaporna“;
if (bb == 0) throw “nulova“;

}
```

```
Int main(){
try {
Tst(10,10); // Tst(-1,-2); Tst(-1,2); Tst(1,-2); Tst(-1,0); Tst(1,0);
}
catch(int pom)
{cout << pom << endl;}
catch (char * txt)
{cout <<txt<< endl;}
```

7.13 Pro získání (alokaci) paměti se využívá klíčové slovo new následované datovým typem, pro který požadujeme paměť. Návratovou hodnotou new je adresa alokovaného bloku. Vrácení (odalokace) paměti se provede pomocí klíčového slova delete následovaného ukazatelem na rušenou paměť (který vrátilo new).
Pokud nebyla paměť přidělena, je vygenerována výjimka std::bad_alloc (z knihovny <new>. Napište ošetřenou část kódu, který naalokuje místo na jeden double

```
#include <new>


{
double *uk;
try {
uk = new double;
}
catch(std::bad_alloc) { // řešení chyby alokace }


delete uk; }
```

7.14 Pro alokaci pole se používá stejný mechanizmus, pouze za požadovaný typ se uvede v hranatých závorkách počet požadovaných prvků. Důležité je, že toto new má svou vlastní „polní“ verzi delete a to následovanou hranatými závorkami a ukazatelem delete[]. Jednotlivé verze nesmí být zaměňovány. Napište obdobný příklad jako předchozí s tím, že naalokujete 100 prvků double.

```
#include <new>


{
double *uk;
try {
uk = new double [100];
}
catch(std::bad_alloc) { // řešení chyby alokace }


delete [] uk; }
```

Přetěžování funkcí

V jazyce C se může v rámci kódu vyskytovat identifikátor (zde jméno funkce) pouze jednou. V jazyce C++ může být funkcí se stejným jménem více. Jedinou podmínkou je, že při použití funkce musí překladač jednoznačně určit, kterou funkci má zavolat. Rozlišení může provést podle počtu nebo typu parametrů.

7.15 Pomocí mechanizmu přetěžování napište funkci Min pro dva parametry typu int a funkci Min pro tři parametry typu int. Zavolejte jednotlivé funkce a ověřte, že je program přeložitelný a že funguje.

```
int Min(int a, int b) {
return a<b? a : b; }

int Min(int a, int b, int c) {
return a<b && a<c? a : (b<c? b:c); }


x = Min(1,2); y = Min(2,1);
x = Min(1,2,3); y = Min(3,2,1); z = Min(3,1,2);
```

7.16 Další možností přetížení je pomocí typu. Napište funkci Min se dvěma parametry pro typ long int. Přidejte k předchozímu programu a zavolejte s typem long int.
Co se stane pokud zavoláte funkci Min s typem char?

```
long int Min(long int a, long int b) {
return a<b? a : b; }
```

```
lx = Min(12l,15l); //  za čísly je znak „L“ značící long
ly = Min('a','b'); // zatímco v minulém příkladu by toto volání bylo bez problémů, nyní překladač zahlásí chybu
// jedná se o to, že překladač dokáže typ char převést na int i long int ale nedokáže rozhodnout, který je lepší

lz = Min(12,12l); // jedno číslo je int a druhé long int –
// překladač opět nedokáže odhadnout, která konverze je lepší

ly = Min((int) 'a',(int)'b'); // programátor musí překladači napovědět, kterou verzi vybrat
lz = Min((long)12,12l);
```

Dalším mechanizmem pro ulehčení zápisů je mechanizmus implicitních parametrů. V hlavičce funkce můžeme napsat implicitní hodnotu proměnné, která se dosadí v případě, že při volání není tato proměnná uvedena. Využití je omezeno na to, že se používá postupně od posledního parametru (jak při definici tak při volání). Funkce je fyzicky pouze jedna s tím, že při volání se dosadí za chybějící parametry implicitní hodnoty. Hodí se pro funkce, které mají stejný kód pro různý počet parametrů.

7.17 Napište funkci Min pro typ double, která bude společná pro dva a tři parametry. Realizujte jako jednu funkci s použitím implicitních parametrů. (Předpokládejte pouze kladná čísla a minimum nula)

```
double Min(double a, double b, double c = 0); // implicitní parametr se uvádí pouze při prvním výskytu
```

```
double Min(double a, double b, double c )  // realizace funkce, zde už se implicitní hodnota nesmí uvést
{  // realizuje se pouze funkce s maximálním počtem parametrů a ta jediná je přeložena ve výsledném kódu
return a<b && a<c? a : (b<c? b:c); }
```

```
dx = Min(1.2,3.4,4.5); // volání se třemi parametry
dx = Min(1.2,3.4);
// Při volání se dvěma parametry je poslední parametr překladačem doplněn implicitní hodnotou
// Takže v kódu je přeloženo a volá se Min(1.2,3.4,0);
```

Dalším mechanizmem je princip template = šablona. Používá se pro naprosto stejný kód, který se používá pro různé datové typy. Definice (funkce, datový typ) se napíše pouze jako předpis, kde se místo konkrétního typu napíše zástupný typ (nejčastěji se volí velké písmeno z konce abecedy).
Mechanizmus si lze **představit** tak, že překladač si šablonu „přečte“ a uloží si ji do „paměti“. Teprve až v kódu narazí na použití (datového typu, funkce), zjistí si z použití datový typ, který reprezentuje zástupné písmeno. Potom do šablony dosadí za písmeno skutečný typ a tento přeloží. Je-li použito pro více datových typů, pak šablonu použije vícekrát.
Jelikož se jedná o předpis a netvoří se (ihned při „překladu“ šablony) kód, uvádí se šablony do hlavičkového souboru.

Použití pro minimum:

```
template <typename T>   // označení, že se jedná o template/šablonu/předpis.
// Typ, který se mění je zastoupen písmenem T, v dalším se používají standardní typy. Typ T se používá
// tam kde je typ, který se mění v závislosti na vloženém parametru
T Min(T &p1, T &p2) // hlavička je již standardní, dá se použít „zástupný“ typ T
{
 T pom =  p1 < p2 ? p1 : p2;   // „zástupný/proměnný“ typ můžeme používat i v těle
 return pom;
}
```

Před funkcí by mělo být uvedeno, co musí splňovat datový typ T. V našem případě musí „umět“ porovnání (tj. mít implementován operátor <). Dále musí být schopen vytvořit kopii proměnné (vracení hodnotou).

Volání:

```
double a=3,b=4,c;
c = Min(a,b); // v tomto místě překladač zjistí, že funkci nezná, ale má šablonu (předpis) jak ji vytvořit
// jelikož parametr a je double, nastaví T na double. Totéž provede pro druhý parametr. Jelikož i zde je
```

parametr double, nedochází k rozporu (v tom případě by vznikla chyba, protože T musí být stejné). Do míst, kde je v šabloně T se dosadí double a provede se překlad funkce. Ta se potom zavolá.

7.18 Zkuste zavolat Min pro proměnné různých typů. Například typu int a double. Co se stane? V čem je problém?

Nejde přeložit. Problém je v tom, že T je jednou int a jednou double. Překladač tedy nemůže rozhodnout pro jaký typ přeložit.

7.19 Definice template může mít v sekci uvedení typů více položek oddělených čárkou. Naprogramujte Min tak, aby mohlo mít parametry různého typu.

```
template <typename T,typename S>   // proměnných typů může být v šabloně více
T Min(T &p1, S &p2) // hlavička je již standardní, dá se použít „zástupný“ typ T i S
{
 T pom =  p1 < p2 ? p1 : p2;   // „zástupný/proměnný“ typ můžeme používat i v těle
 return pom;
}
```

Volání:

```
int a = 3;
double b=4,c,d;
c = Min(b,a);
c = Min(a,b); //je vygenerována jiná funkce než v předchozím – problém je v návratové hodnotě.
// i když je minimum typu double, z definice funkce plyne, že se vrací typ první proměnné
// to znamená, že se nevrátí přesnější typ
```

Tento problém se dá ošetřit pomocí nového klíčového slova auto, kdy si překladač zjistí návratové typy a vybere ten „správný“.

```
template <typename T,typename S>   // proměnných typů může být v šabloně více
auto Min(T &p1, S &p2) // hlavička je již standardní, dá se použít „zástupný“ typ T i S
{
 auto pom =  p1 < p2 ? p1 : p2;   // překladač zjistí správný typ
 return pom; // na základě typu pom určí návratovou hodnotu
}
```

Určení typu pomocí auto je vhodné pro právě uvedené případy (template). Pro obecné použití je jeho nevýhodou, že pokud chceme zjistit jakého typu jej příslušná proměnná, musíme po tom pátrat (což může být složitý proces díky „souvislostem“ za nichž proměnná vznikla.

7.20 Napište template pro funkci Stred, která má tři parametry a vrátí ten s „prostřední“ hodnotou (seřadí prvky podle velikosti, a vrátí ten prostřední). Ukažte, že funguje pro různé datové typy

```
template <typename T>
T stred(T &p1, T &p2, T &p3) {
 T pom =  … ;   // výpočet
 return pom;
}
```

použití

```
double a=3,b=4,c=1,r;
r = Stred(a,b,d); // vytvoření pro typ double
int x = 3,y = 2,z = 6,w;
w = Stred(x,y,z); // pro typ int
```

```
w = Stred(y,x,z);
w = Stred(y,z,x); // a další kombinace
```

7.21 Napište pomocí šablony funkci Alloc, která bude alokovat pole daného typu o určitém počtu prvků. Funkce bude mít jako parametr počet prvků. Naalokované pole vrátí pomocí návratové hodnoty (chyba by se řešila výjimkou – pro jednoduchost není nutné). Napište prototyp funkce. Volání pouze promyslete.

```
Template <typename T>
T * Alloc(unsigned pocet)
{
return new T[pocet];
}
```

Jelikož jazyk C/C++ nerozlišuje podle návratové hodnoty, překladač v tomto případě nemůže určit typ T. Proto je nutné při použití rozšířit název funkce o požadovaný datový typ ve špičatých závorkách.

7.22 Zkuste zavolat funkci Alloc pro typ double a int

```
double *dp;
int *ip;

dp = Alloc<double>(10);
di = Alloc<int>(20);
// odalokování paměti
```

Z výše uvedeného plyne, že funkce vytvořená pomocí šablony má název složený z názvu funkce a typu pro který vznikla. Pro nás: Alloc<double > a Alloc<int>.

7.23 Napište pomocí šablony funkci Alloc1 pro alokaci vektoru realizovaného pomocí struct. Struct bude také realizována pomocí šablony. Struct obsahuje pole dat a délku. Napište i funkci Dealloc1. Ukažte použití pro strukturu s daty int a double.

```
template <typename X> // proměnný datový typ ve struktuře. Písmeno může být libovolné
struct  SVect {
X *iData;
unsigned iPocet;
}

template <typename U> // písmeno může být stejné nebo jiné než u předchozího – nesouvisí spolu
void Alloc1(SVect<U> &aVal,unsigned aPocet) // datový typ U se zjistí z typu, pro který je
// nadefinovaná struktura
{
 if (aPocet <= 0) return;
aVal.iData = new U[aPocet];
aVal.iPocet = aPocet;
}

template <typename V>
void Dealloc(SVect<V> &aVal) // i když to vypadá, že typ nebude potřeba (delete funguje pro všechny typy)
// je nutné typ uvést, protože delete volá destruktory.
{
 delete[] aVal.iData;
aVal.iData = nullptr;
}
```

8 Použití const u (parametrů) třídy

Použití modifikátoru const u parametrů funkcí a metod je důležité. Vede k lepší orientaci v kódu, protože to co je označeno const se nemění (to hlídá řekladač). To co není označeno const se může (ale nemusí) měnit – je ovšem výhodné psát programy s tím, že není-li const, potom se proměnná mění. Při této „domluvě“ je z hlavičky ihned patrné co se s proměnnými děje.

Dalo by se říci, že const označíme na začátku každou proměnnou a pokud si její změnu dokážeme zdůvodnit (potřebujeme ji měnit), potom teprve od ní const smažeme.

Překladač nedovolí změnu const proměnné. Nedovolí ale ani volání funkce/metody, která by ji mohla změnit. Proto u metod, které nemění proměnné parametru this, musíme uvést, že je konstantní – to naznačíme tím, že napíšeme const mezi hlavičku a tělíčko metody. Takto uvedenou metodu potom mohou volat i konstantní proměnné.

Pozn.: uvědomme si, že je-li uveden konstantní parametr, je funkce vhodná jak pro konstantní tak pro nekonstantní proměnné.
Není-li parametr const, je možné funkci volat pouze pro nekonstantní proměnné – a const musíme řešit jinak (!!!).

8.1 vytvořte funkci, která bude mít parametr typu const reference na CKomplex. V této funkci pomocí getterů zkuste vytisknout hodnoty proměnné. Upravte gettery (pomocí const) tak aby program šel přeložit.

```
void funkce(const CKomplex &aParam)
{
x = GetRe();
y = GetIm();
}
```

úpravy metod
double GetRe() const {}
double GetIm() const {}

8.2 Máte metodu Met, ve které pracuje s nekonstantním parametrem this a vrací tento objekt:
CKomplex& Met() {return *this;}
s voláním:
CKomplex a, const b,c;
c = a.Met();
c = b.Met(); // vyřešte tento řádek napsáním nové/další metody Met
Popište rozdíly mezi jednotlivými popsanými metodami.

CKomplex Met() const {return *this;}
Přidali jsme const k parametru, který metodu volá (k this). Proto jsme museli změnit i návratovou hodnotu. Zde jsou možnosti dvě, obě mají omezení – výše popsaná vrací hodnotu, a tedy bude se volat oproti původní metodě navíc konstruktor a destruktor. Další volbou by bylo vrátit konstantní referenci.
const CKomplex& Met() const {return *this;}. Metoda by byla stejně rychlá jako pro nekonstantní proměnnou. V tomto případě bychom ovšem na návratovou hodnotu nesměli použít žádnou metodu pracující s nekonstantní proměnnou, což by mohlo být značné omezení.
Všímněnte si, že metody mohou existovat vedle sebe a překladač je schopen je na základě const rozlišit.

Const atributy třídy
const proměnná třídy

proměnná const a neconst
funkce stejné jméno – jiná pro const a jiná pro neconst - srovnat

9 Vytvoření a zánik prvku – konstruktory a destruktory

Jak jsme si již řekli, proměnnou je možné při definici inicializovat.

9.1 O jaké typy inicializace se jedná v následujícím kódu? int i, j = 5.7, k = j, m = 4; Vysvětlete co se děje v tomto místě programu (jak je řádek přeložen).

První inicializace je bez parametrů – u typu int se nic neděje – jedná se o implicitní inicializaci.
U druhé proměnné se jedná o konverzní inicializaci, protože hodnota typu double je zkonvertována na int a přiřazená proměnné.
Poslední dva případy vytvářejí kopii hodnot – v obou případech se vytváří proměnná typu int ná základě jiné hodnoty typu int.
Při vykonávání této části kódu se v paměti vyhradí místo pro čtyři proměnné typu int (tj. velikost zabrané paměti pro každou proměnnou bude dostatečná pro uložení proměnné int). Po rezervaci paměti pro proměnnou bude v proměnné i hodnota, která byla v „její“ paměti před jejím vytvořením (proměnná se neinicializuje). Do (paměti) proměnné j bude uložena hodnota 5.7 upravená na int. Do (paměti) proměnné k bude uložena hodnota z (paměti) proměnné j. Do (paměti) proměnné m bude uložena hodnota 4 (celočíselná, zabírající velikost int).

V objektovém programování se při vzniku proměnné volá konstruktor.

Statická proměnná na počítání prvků

9.2 V programu je nutné implementovat kód sloužící ke korektnímu ukončení života proměnné. Co je nutné doplnit v následujícím kódu?

```
{
int *pi = malloc(…);
FILE *uf = fopen(…);
… // ošetření vzniku a práce s proměnnými

}
```

Je nutné vrátit systému soubor a alokovanou paměť.
Paměť proměnných vrátí ten, kdo si ji rezervoval – tj. paměť ve které jsou umístěny hodnoty proměnných pi a uf je na konci bloku uvolněna překladačem. Programátor vrací jen to o co (systém) požádal on.

Inicializace a „úklid“ proměnné patří k základním programátorským dovednostem – ke kultuře programování. Proto v jazyce C++ se tyto vlastnosti staly součástí objektového návrhu. Pro vznik proměnné se používá metoda nazývaná konstruktor, při zániku je volána metoda zvaná destruktor. Konstruktorů může být několik, musí se lišit parametry. Konstruktory i destruktor volá překladač automaticky (těla těchto metod ovšem musí být napsána programátorem, aby překladač věděl, co má dělat).

Při zániku se uplatňuje pravidlo, že žádná data nebo zdroje by se neměly ztratit. Proto před ukončením života proměnné, který vyvolává překladač, je překladačem volána metoda destruktoru, která umožní těmto ztrátám zabránit.

9.3 Napište konstruktor bez parametrů (implicitní) a destruktor (nemá parametry) pro třídu CKomplex.
Konstruktor je metoda, která má stejný název jako třída.
Destruktor je metoda, která má stejný název jako třída, kterému předchází znak „~“ (tilda, vlnovka…).
Konstruktor ani destruktor nemají návratovou hodnotu (návratová hodnota se neuvede).
Napište „volání“ konstruktoru a destruktoru a pomocí trasování (nebo tisku na obrazovku v těle metod) zjistěte, zda jsou skutečně volány.
„Volání“ je provedeno automaticky překladačem při definici proměnné a při jejím zániku (na konci bloku, ve kterém byla definována).

Obě metody by měly být v sekci public:

```
~CKomplex (void) {}  // destruktor – nedělá nic (zatím)
nebo:
~CKomplex (void) {clog<< “CKomplex : destruktor“ << endl;}  // destruktor – vypíše info o běhu


CKomplex (void) // implicitní konstruktor
{
 iRe = iIm = 0;
}
```

Pro konstruktory platí pravidlo, že se nejprve provede konstrukce členských proměnných třídy a teprve potom se provede tělo konstruktoru. Konstruktory členských proměnných se uvádí mezi hlavičku a tělo metody. Je to především určeno pro členy typu třída (ve třídě), kdy se dá volat určitý konstruktor. Toto je možné dělat i pro standardní typy.
Modifikovaný konstruktor by vypadal

```
CKomplex (void): iRe(0), iIm(0) // konstruktory členských proměnných (= prvotní inicializace)
{ // vlastní činnost konstruktoru mimo inicializace proměnných, které jdou realizovat pomocí konstruktoru
clog << “CKomplex: Konstruktor implicitni. “ << endl; // například kontrolní tisk
}


Int main()
{
{  // pomocný blok
CKomplex a; // při definici proměnné se volá konstruktor.
// Jelikož není žádná bližší specifikace, zavolá se implicitní. To je bez parametru.
} // konec pomocného bloku. Zde by se měla proměnná rušit, a tedy by se měla projevit činnost destruktoru
return 0;
}
```

9.4 Konstruktor je speciální metoda (funkce) a může tedy mít parametry. Pokud má jeden parametr (jiného datového typu než je sám), nazývá se parametr konverzní, jelikož zadaný typ konvertuje na typ třídy (z parametru jednoho typu vytvoří typ jiný).

Vytvořte konverzní konstruktor z double pro třídu CKomplex. (Konstruktor může dělat cokoli například:) Předaný parametr je reálná složka. Imaginární složku vynulujte.

Ukažte použití („(za)volání“) konverzního konstruktoru

CKomplex(double aParam):iRe(aParam), iIm(0) {}

Konstruktor bude volán/použit při definicích:

CKomplex aa=8.2, bb(4.5), dd(3);
Poslední parametr je int, ale překladač může při volání vyzkoušet/použít standardní konverze, takže int převede na double a zavolá konstruktor s parametrem double.

9.5 Konverzní konstruktor může být z jakéhokoli (známého) typu. Vyzkoušejte konstruktor z řetězce. Použití s formátem řetězce je: CKomplex e(“ 12.3+3i “) ;

```
CKomplex (char *aTxt)
{
// postupné načítání a testy na správnost řetězce a hodnot
}
```

9.6 Speciální konstruktor s jedním parametrem je konstruktor, jehož parametrem je prvek stejné třídy (použití pro předání je **reference**!!!). Jelikož tímto dojde k vytvoření kopie, nazývá se kopykonstruktor nebo kopírovací konstruktor. Napište kopykonstruktor. Ukažte použití.

CKomplex(const CKomplex &aParam) : iRe(aParam.iRe),iIm(aParam.iIm) {}

Pro použití kopykonstruktoru zkuste trasovat následující kód a spočtěte kopykonstrukory. Všimněte si i destruktorů.

```
CKomplex Metoda(CKomplex aParam) {
CKomplex bb(aParam);
return bb;
}
```

S voláním:

```
CKomplex ff(3),gg(4),ee;
ee = ff.Metoda(gg);
```

9.7 Je vhodnější prototyp CKomplex (const CKomplex &aval) nebo CKomplex (CKomplex &aval) ? Popište výhody jednotlivých řešení.

Řešení bez const má výhodu, že se prvek aval může měnit. To by ale měla být výjimečná situace, protože se snažíme, aby se třídy a jejich metody chovaly (pokud možno) stejně/podobně jako základní datové typy. A zde se při inicializační proměnná nemění.

Varianta s const má ještě tu výhodu, že ji lze použít pro proměnné const i neconst. Jak jsme si řekli, datový typ s const je jiný než bez const => int a const int jsou dva (různé) typy, které překladač rozlišuje. Pokud je u parametru metody const, lze tuto použít pro const i neconst proměnnou. Pokud není const v hlavičce funkce uvedeno, lze funkci použít jen pro nekonstantní proměnné (což je omezení). Proto pokud můžeme, dáváme přednost uvádění const u parametrů funkcí.

9.8 Protože některé z předchozích konstruktorů se jeví jako „zbytečné“, protože vznikají dočasné proměnné, které ihned zaniknou, ale zároveň je potřeba čas na jejich vytváření a zánik. Pro tyto případy je vhodné vytvořit tzv. move konstruktor (náš příklad ukáže pouze, kde se volá. Jeho hlavní výhoda je při „přesunu“ dat z proměnné do proměnné, zvláště pak u dynamických dat. Jelikož náš typ má proměnných málo tato vlastnost nemusí být patrná). Zápis move konstruktoru je stejný jako u kopykonstrukotru, pouze znak & je zdvojený. Tento konstruktor překladač použije na dočasné/nepojmenované hodnoty (r-value).

```
CKomplex(CKomplex &&aParam) {
Copy(this, aParam); // přímá kopie obsahu pomocí „čistého“ kopírování
SetNull(aParam); // původní proměnná se „vynuluje“ tak aby byla data „neplatná“
// Toto nastavení by mělo být takové, aby následné volání destruktoru na aParam bylo co nejkratší
// v provádění kódu
}
```

Jelikož se používá pro r-hodnotu (která po použití zaniká), můžeme činnost konstruktoru realizovat tak, že data přesuneme do nového objektu a starý objekt vynulujeme (jinak by došlo při destruktoru ke zničení dat sdílených s novým objektem).

9.9 Konstruktory (jsou metody) a proto mohou mít i různý počet parametrů. Napište konstruktor, který bude mít dva parametry – reálnou a imaginární složku.
Napište konstruktor se třemi parametry pro zadání komplexního čísla ve formátu amplituda, úhel (třetí parametr nebude přímo použit, bude pouze sloužit k odlišení od předchozího dvouparametrového konstruktoru).

```
CKomplex (double aRe,double aIm): iRe(aRe),iIm(aIm){}

CKomplex(double aAmp, double aPha, int pom):
 iRe(aAmp*cos(aPha*180/3.14)), iIm(aAmp*sin(aPha*180/3.14)) {}
```

Použití

```
CKomplex mm(5,5), nn(1.4. 45.0)
```

10 Statické proměnné a metody

Statické proměnné a metody třídy jsou součástí třídy, ale jsou použitelné i v případě, kdy není vytvořen žádný prvek třídy.

Jelikož nejsou vázány na prvek třídy, znamená to, že proměnná nepatří konkrétně žádnému objektu – a patří tedy zároveň všem. Z toho plyne, že proměnná označená v definici třídy jako static bude v programu pouze jediná, společná všem objektům třídy. Zároveň to znamená, že bude na „globálním“ prostoru. A také, že bude nutné ji inicializovat „mimo objekty“. Taková proměnná tedy bude v definici třídy označena jako statik a její definice bude podobná jako definice globální proměnné, s tím, že bude nutné u jejího jména uvést i třídu do které patří (viz. Operátor příslušnosti).

Statické metody jsou na tom podobně. V definici ve třídě se uvede, že jsou static. Definovány jsou ve zdrojovém (cpp) souboru s tím, že je nutné opět uvést, do které třídy patří.

10.1 Nadefinujte ve třídě CKomplex statické proměnné iAktual a iTotal. Fyzicky je nadefinujte v příslušném zdrojovém souboru – použijte definici s inicializací. Proměnné budou sloužit pro počítání celkově vzniklých objektů třídy a pro počítání kolik je právě aktuálně vytvořeno objektů.
Nadefinujte statické metody GetAktual a GetTotal, které vrátí hodnoty příslušných proměnných.
Vyzkoušejte volání těchto metod (přes jméno třídy a operátor příslušnosti) pro tisk zjištěných hodnot.

Ve třídě (h soubor):

```
Class CKomplex {
static int iAktual, iTotal; // definice statických proměnných, nejsou v objektech/proměnných

public:

static int GetAktual() {return iAktual;}; // jednoduchá funkce může být inline (je v těle třídy)
static int GetTotal();

}
```

Ve zdrojovém (cpp) souboru:

```
// static už se neuvádí – je patrno z deklarace v inkludovaném hlavičkovém souboru třídy
int CKomplex::iAktual = 0; // definice s inicializací. Nutno říci, ke které třídě patří.
// jinak by to byla „obyčejná“ globální proměnná bez návaznosti na třídu
int CKomplex::iTotal = 0;

int CKomplex::GetTotal() {return iTotal;} // funkci je nutno „propojit“ se třídou pomocí operátoru příslušnosti
// jelikož funkce patří ke třídě, proměnná iTotal je dostupná. Nejsou ovšem dostupné proměnné iRe a iIm,
// které jsou v objektech a přináší je „this“. Ve statických metodách se this nepředává.
```

Volání:

```
cout << CKomplex::GetTotal() << endl; // je nutné přesně říci, kterou funkci máme na mysli
{
CKomplex aa; // až zde vzniká proměnná
} // již neexistuje žádná proměnná
cout << CKomplex::GetTotal() << endl; // je nutné přesně říci, kterou funkci máme na mysli
```

10.2 Rozšiřte konstruktory a destruktor tak, aby se každý vznikající objekt započítal do total i aktual a každý zanikající prvek se odečetl od aktual.

U konstruktoru přidáme
CKomplex () : iRe(0),iIm(0) {++iTotal;++iAktual;}

U destruktoru přidáme:

~CKomplex() {--iAktual;}

10.3 Při větším množství proměnných jedné třídy bývá dost těžké se orientovat, která proměnná je která. Jednoznačně to lze určit podle adresy, na které leží. Lze pro to ovšem použít mechanizmus, který jsme si předvedli v minulých bodech. Máme zde jednoznačně určeno (v proměnné iTotal), kolik objektů dané třídy vzniklo. Každý objekt tedy vzniká za jiné hodnoty této proměnné. Ve třídě přidejte proměnnou iIndex a do ní v konstruktoru zapište aktuální hodnotu proměnné iTotal. Toto číslo bude pro proměnnou unikátní a bude možné ji podle něj identifikovat. Na rozdíl od ukazatelů, které se mohou při každém překladu měnit, pokud nepřidáme proměnné, identifikace pomocí indexu bude jednoznačná.

U konstruktoru přidáme
CKomplex () : iRe(0),iIm(0) ,iIndex (++iTotal) {++iAktual;}

Do private sekce přidáme iIndex.
Proměnná iIndex by mohla být označena const. Po inicializaci v konstruktoru by ji tak nebylo možné změnit a byla by pro objekt skutečně jedinečná (nebylo by možné ji změnit).

11 Členění třídy do souborů

Při programování třídy jsme zatím převážně využívali pro popis/definici třídy hlavičkový soubor. Je to proto, že tato cesta je možná a je jednodušší. Neuvedli jsme si ovšem co to znamená, že jsou metody v hlavičkovém souboru. Z toho co víme o hlavičkových souborech by to šlo odvodit.

11.1 Co jsou hlavičkové soubory. Jaké části kódu obsahují?

Hlavičkové soubory slouží k „propojení“ zdrojových kódů tím, že jsou v nich uvedeny prototypy/deklarace funkcí a deklarace proměnných. Dále zde mohou být definice konstantních proměnných, #define … Makra, inline funkce, definice (složených) typů, šablony …
Uvedení výše zmíněných prvků následně využije překladač při překladu zdrojového kódu (který již vede ke spustitelnému kódu umístěnému v paměti).
Může zde být tedy jakýkoli zdrojový kód, který netvoří kód „fyzický“ (v paměti).
Při psaní nezapomínejte na ošetření hlavičky proti vícenásobnému načtení.

11.2 Které části programu se píší do hlavičkového a které do zdrojového (cpp) souboru? Pokud toto pravidlo „otočíme“ co z něj odvodíme pro metody uvedené v hlavičkovém souboru? Znáte mechanizmus, kdy jsou funkce uvedené v hlavičkovém souboru (a není to na závadu)? Jaké funkce to jsou?

V hlavičkovém souboru jsou uvedeny části, které netvoří kód (deklarace funkcí, deklarace proměnných, popisy tříd a struktur, inline funkce …). Pokud některé z těchto nechceme „publikovat“ pro širší použití mimo náš modul, můžeme je dát i do zdrojového souboru.
Ve zdrojovém souboru jsou části, které tvoří kód (definicie funkcí, proměnných). Nesmí být v souboru hlavičkovém (který je vícekrát includován – s čímž se počítá).
Při „otočení“ pravidla tedy můžeme říci, že „funkce/metoda uvedená v hlavičkovém souboru nesmí tvořit kód“. Tuto podmínku splňují inline funkce. A proto metody uvedené v hlavičkovém souboru (lépe řečeno uvnitř definice třídy) jsou inline.

Kratší/jednodušší metody proto volíme inline a ty mají tělíčka v hlavičkovém souboru, delší/složitější metody uvádíme do zdrojového souboru. Inline metody se rozvinou přímo do kódu, metody ze zdrojového souboru jsou používány pomocí funkčního volání (call – return).

Jelikož definice třídy popisuje třídu z hlediska možností jejího využití, je požadavek, aby tato část byla přehledná. Tělíčka metod tuto přehlednost narušují. Proto existuje způsob, který umožňuje říci, že je metoda inline, ale její tělo nadefinovat mimo definice třídy.
V definici uvnitř třídy se uvede pouze prototyp metody s klíčovým slovem inline. Po ukončení definice třídy se teprve (v hlavičkovém souboru) uvede tělo metody. Jelikož již jsme mimo třídy, musíme metodu uvést včetně názvu třídy, ke které patří (za použití operátoru příslušnosti).

11.3 Vyberte si ve vytvářené třídě tři metody (jakékoli, včetně konstruktorů..) a každou z nich realizujte jiným způsobem.

```
class CKomplex {

CKomplex () : iRe(0),iIm(0) {++iTotal;++iAktual;} // velice jednoduchá, tělo uvnitř –> inline
```

Inline CKomplex (CKomplex const & a) ; // jednoduché, ale tělo je delší -> inline s tělem mimo definici třídy

CKomplex (char *txt); // nemá inline ani tělo -> tělo ve zdrojích a funkční volání. Delší kód.

};

// vlastní definice těla inline metody je v hlavičkovém souboru. Je již bez inline, ale zato se jménem třídy
CKomplex::CKomplex (CKomplex const & aParam) : : iRe(aParam.iRe),iIm(aParam.iIm)
{ cout << “Nejaky delší text ale kratsi kod “ << endl;}

Zdrojový soubor *.cpp

#include “CKomplex.h“

CKomplex:: CKomplex (char *txt)
{
// kód pro zpracování řetězce je dlouhý a složitý, takže vytvářet tuto funkci jako inline nemá smysl
}

12 Operátory

Ke standardním typům patří i operátory. Jelikož vytvářený datový typ pomocí class je ekvivalentním typem, musí umět totéž co standardní typy, a tedy i operátory.

12.1 Jaké kategorie (typy, množiny, třídy, …) operátorů znáte?

Dělení je možné například :

Matematické operátory (=, +,-,*,/,%,++,--)
Bitové operátory (~, &, |,^ ,<<,>>).
Logické (==,!, !=. <, >, >=,<=)
Přiřazovací (=, +=,-=, &=, <<=, …)
Ostatní (new, delete, sizeof, konverzní operátor, funkční operátor (), operátor indexování[], operátory přístupu k prvkům * ->)

12.2 Jaké operátory znáte v závislosti na počtu operandů?

Unární (jeden parametr) +,-,!,~,++,--
Binární (+,-,&,&&, =, == , <=, ..)
Ternární – nejde předefinovat pro třídu

Z praktických důvodů jsou operátory definovány jako funkce/metody. Pro tuto variantu slouží klíčové slovo operator. Operátory se tedy vyznačují tím, že mají dvě možnosti použití – zjednodušenou (to je ta, kterou znáte ze standardních datových typů) a plnou, funkční (v této formě je použito klíčové slovo operátor následované znaménkem operátoru.
Pro typ int by tedy klasický/zjednodušený zápis byl:
a = -b * c - d;
úplný funkční zápis by byl
operator=(a , operator-(operator*(operator –(b) , c), d))
všimněte si operátoru -, který je zde dvakrát – jako unární s jedním a jako binární se dvěma parametry.
Jelikož jsou operátory na místech parametrů, musí mít návratové hodnoty.

Definice by mohla vypadat například (pro int je ve skutečnosti součástí překladače):

```
Int& operator=(int &cil, int &zdroj)
int    operator-(int &a, int &b)
int    operator-(int &a)
```

Pro metody je situace obdobná, pouze plné volání má první parametr „přes tečku“ a.operator*(b). A při definici metody je nutné uvažovat, že jeden operátor (ten „před tečkou“) se do metody dostává implicitně, tedy metoda má vždy uveden o jeden parametr méně (a druhý je tam jako this)

12.3 Naprogramujte pro třídu CKomplex operátor přiřazení (rovná se). Pro volání/použití operátoru přiřazení zkuste obb

Operátor přiřazení je binární operátor. Proto bude mít jeden parametr + thsi.

```
CKomplex &operator = (CKomplex const &aParam)
// vrací referenci, protože vrací prvek, který je k dispozici. Důvodem je možnost zřetězení a = b = c = d = 0;
// vhodnost const u parametru je rozebráno u kopy konstruktoru
{
if (this == & aParam) // pokud někdo použil přiřazení ve formátu a = a
  return *this; // pak další vynecháme, protože provedení kopie trvá a tak ušetříme
iRe = aParam.iRe; iIm = aParam.iIm;
return *this; // kvůli zřetězení vrací hodnotu – výsledkem je nově vzniklý prvek
}
```

Použití je standardní

```
CKomplex a,b;
a = b;
a.operator=(b);
```

12.4 Jaký je rozdíl mezi operátorem přiřazení a kopykonstruktorem?

Často se dělá chyba, že se nerozlišuje mezi přiřazením a kopy konstruktorem. Rozdíl je v tom, že u kopy konstruktoru je proměnná použitá poprvé a nejsou v ní tedy platná data. V těle je tedy pouze inicializace. U operátoru přiřazení je ovšem nutné počítat s tím, že proměnná již obsahuje data. Před přiřazením je tedy nutné nejprve proměnnou „uklidit“ (a to je rozdíl oproti kopykonstruktoru).

Oba používají znak „=“. Kopykonstruktor při definici s inicializací T aa = bb; kde se znak „=“ čte jako „je inicializován“ tj. „vzniká na základě“ tj. „je konstruován pomocí prvku stejného typu“ a nejedná se tedy o operátor přiřazení. Operátor přiřazení se vyskytuje mimo definici proměnné (v „obyčejném“ kódu) … ; aa = bb ;
Jedná se vlastně o podobnost s „*“ či „&“, které také v definici mají jiný význam než v příkazu.

12.5 Pro CKomplex realizujte unární operátory + a -. Zamyslete se nad návratovou hodnotou (hodnota x reference). Uvědomte si jak tyto operátory fungují pro int a snažte se aby se chovaly (v rámci možností) podobně. Zkuste oba způsoby volání.

int a = 5,b,c;
b = +a; // kolik bude po provedení hodnota v b a kolik a?
c = -a; // kolik bude po provedení hodnota v c a kolik a?

```
// oba operátory jsou unární, takže budou bez parametr + this

// operátor unární + nemění prvek, na který je aplikován a vrací stejnou hodnotu
CKomplex& operator+() {return *this;}

použití b = +a;  // jelikož a je totéž co +a, můžeme vracet referenci, jelikož vracená hodnota je k dispozici
// ve funkci volající i volané


// operátor unární - nemění prvek, na který je aplikován a vrací jinou/novou hodnotu
CKomplex operator-() {return CKomplex(-iRe,-iIm);}
```

použití c = -a; // jelikož -a je různé od a, nemůžeme vracet referenci, jelikož vracená hodnota není k dispozici
// ve funkci volající i volané

b.operator=(a.operator+());
c.operator=(a.operator-());

12.6 I pro operátor přiřazení lze vytvořit tzv. move operátor (obdobně jako move konstruktor.

Tento operátor se využívá pro přiřazení nepojmenovaných proměnných (r-value). Parametr je opět s &&.
Naprogramujte tento operátor a vyzkoušejte, zda je volán v minulém příkladu (unární operátory + a -).
Promyslete, kdy se použije a proč.

Plyne z použití v minulém příkladu. Musíme si opět uvědomit, že se vyplatí zvláště pro složitější proměnné, kde se ušetří výpočty spojené s kopií proměnné.

Tento operátor se dá realizovat tak, že se pouze vymění členská data (původní se přesunou do r-hodnoty, která při svém (brzkém) zániku zajistí jejich korektní ukončení).
(Pro přesun dat bývají přítomny knihovní funkce swap, move…(pokud by ovšem využívaly move operator=, mohlo by dojít k zacyklení při jejich použití v tomto operátoru))

12.7 Pro CKomplex realizujte unární operátory ++x a x++. Jedná se o unární operátory. Operátory se odliší tím, že jeden z nich (který?) má uveden (fiktivní) parametr jako typ int. Jeden je prefixový, druhý postfixový. Jeden proměnnou nejprve zvětší a pak použije (vracená hodnota je stejná jako hodnota objektu), druhý ji nejdříve použije (musí tedy vrátit hodnotu původní) a potom ji změní (je tedy hodnota změněná ale musíme vrátit tu minulou, tudíž jinou než je k dispozici).

Operátor realizujte tak, že zvětší reálnou i imaginární složku o jedničku.
Uvědomte si, jak tyto operátory fungují pro int a snažte se aby se chovaly (v rámci možností) podobně. Zkuste oba způsoby volání.
int x = 5,b,c;
b = x++; // kolik bude po provedení hodnota v b a kolik x?
c = ++x; // kolik bude po provedení hodnota v c a kolik x?

```
// oba operátory jsou unární, takže by měly být bez parametru + this
operator++( ) // takhle bude vypadat první
operator++(int ) // druhý je odlišen parametrem int (který se nepoužije při volání)
// int doplňte ke správnému ++ z metod níže


CKomplex& operator++( ) // jelikož vracím zvětšený prvek, můžu vrátit referenci
{ ++iRe ; ++iIm; // nejprve zvětším
return *this;} // zvětšené vrátím


CKomplex operator++() { // jelikož se prvek změní, musím vrátit hodnotu
CKomplex pom(*this) ; // zapamatuji si původní hodnotu. Vytvořím kopii
++iRe;++iIm; // provedu operace pro zvětšení hodnot proměnné
return pom;} // vrátím původní hodnotu



// aby se odlišilo volání, je zde nutné přidat do ++ parametr – například nulu

b.operator=( a.operator++());
b.operator=( a.operator++(0));
```

12.8 Parametrem může být i jiný datový typ. Napište metodu pro násobení CKomplex proměnnou typu double.

```
CKomplex a(4.2);
double b = 6;
xx = a * b; // jakého typu bude xx?
```

```
// výsledný typ by měl být ten „přesnější“ což je pro nás komplexní číslo

CKomplex operator*(double aParam)
{
CKomplex pom(iRe * aParam, iIm *aParam);
return pom;
}
```

12.9 Někdy se stane, že potřebujeme vytvořit funkci (ne metodu), která pracuje s daným typem (daný typ je pouze jedním z parametrů), ale přístup k proměnným třídy přes gettery a settery by byl zbytečně zdlouhavý a komplikovaný. Pokud se rozhodneme, že této funkci povolíme užívat privátní proměnné, potom její prototyp (hlavičku) přesuneme do třídy a před ní uvedeme klíčové slovo friend.
Napište funkci Clear(), která bude mít parametr typu CKomplex (bez použití getterů a seterů (či jiných metod)) a vynuluje členské proměnné. Vrátí změněnou proměnnou.

Uvnitř definice třídy

friend CKomplex& Clear(CKomplex &a) ; // není metoda, díky friend je to funkce

Ve zdrojovém souboru :

```
CKomplex& Clear(CKomplex &a) // friend se neuvádí – byl by to nesmysl.
// Není ani operátor příslušnosti = není this
{
 a.iRe = a.iIm = 0; // musíme použít „a.“ pro přístup k proměnným (protože není this)
return a; // vracíme změněnou proměnnou
}
```

Tento příklad je použit na uvedení friend funkcí. Jelikož má jediný parametr a to typ této třídy, byla by vhodnější realizace pomocí metody.

12.10 Jelikož na pořadí parametrů u násobení nezáleží, je možné násobit komplexní číslo doublem i opačně:
CKomplex aa(4.2);
double b = 6;
xx = b * aa; // jakého typu bude xx?
Protože volání b.operator*(aa) pro double a CKomplex by musel být již hotový v překladači („vyvolává“ ho první parametr a to je double), nemůžeme použít operátor jako metodu, ale jako funkci. Funkce bude tvořená podobně jako metoda, bude se lišit v tom, že bude mít dva parametry (a nebude mít this). Naprogramujte. Využijte friend.

Uvnitř definice třídy

```
friend CKomplex operator*(double a, CKomplex &b);
```

Ve zdrojovém souboru

```
CKomplex operator*(double a, CKomplex &b)
{
CKomplex pom( a * b.iRe, a * b.iIm);
return pom; // Výsledek je jiný než vstupní parametry a proto vracíme novou proměnnou hodnotou.
}
```

12.11 Napište operátor pro porovnání na rovnost dvou čísel ze třídy CKomplex. Ukažte použití

```
bool operator==(CKomplex &aParam) {
if (iRe != aParam.iRe) return false;
if (iIm != aParam.iIm) return false;
return true;
}
```

12.12 Napište operátor !. (Ten se často používá k určení, zda se dá s proměnnou pracovat). Definujme si, že pro nás se s ním dá pracovat, když nejsou obě složky nulové.

```
bool operator!()
{if (iRe  ||  iIm) return true;
return false;
}
```

12.13 Speciálním operátorem je operátor konverzní. Jeho jméno je názvem typu na který se konvertuje. Neuvádí se u něj typ návratové hodnoty, protože ten je stejný jako název operátoru a byl by tedy nadbytečný. Používá se k přetypování (konverzím, i implicitním kdy je využívá překladač.) Napište konverzní operátor na double – hodnota bude vyjadřovat amplitudu (ale pokud by se hodilo něco jiného, je to možné – je to v rukou návrháře/programátora, vybrat nejvhodnější realizaci pro konverzi). Ukažte jeho použití explicitní i implicitní.

```
operator double() {return sqrt(iRe*iRe+iIm*iIm;}
```

V případě nutnosti je možné zakázat jeho využití překladačem pro implicitní konverze pomocí klíčového slova explicit.
Rozdíl mezi konverzním konstruktorem a konverzním operátorem je ten, že konverzní konstruktor vytváří z „cizího“ prvku proměnnou třídy. Konverzní operátor převádí proměnnou třídy na jiný datový typ (tedy je to proces opačný jako u konstruktoru).

použití:

explicitní

```
CKomplex aa (1,2);

double  x = (double) aa; // explicitní konverze
double y = double (aa); // nově zavedený „funkční“ zápis explicitní konverze
double z = aa; // pomocí trasování zjistěte zda se na tomto řádku zavolá konverzní operátor (implicitně)
// toto volání je možné zakázat pomocí explict. Předchozí explicitní volání zůstanou funkční.
```

12.14 Přetypovat lze i operátor indexování []. Jedná se o binární operátor, takže indexovat je možné pouze jedním indexem (druhý parametr je vlastní proměnná).
Napište pro CKomplex operátor indexování, který vrátí na sudý index reálnou část a na lichý index imaginární část.
Ukažte použití.
Pozn.: jedná se o procvičení operátoru a jeho použití. Jeho praktické využití pro CKomplex by bylo minimální.

```
double operator[](int aIndex) {
…
}
```

```
CKomplex bb(1,2), cc[5];
double pom = bb[0]+bb[1];
double pom = cc[3][0]+bb[3][1]; // jak to funguje?
```

12.16 Přetypovat lze i funkční závorky (). Objekt, který má tento operátor se nazývá funkční. Tento operátor má ovšem nevýhodu v tom, že mate čtenáře. Jeho zápis pro proměnnou se tváří stejně jako funkce a může tedy dojít ke zmatení čtenáře (proto je některými programátory neoblíben). Je to operátor, který může mít libovolný počet parametrů.
Napište a vyzkoušejte tento operátor pro CKomplex. Činnost implementujte stejnou jako pro operátor [].
Pozn.: jedná se o procvičení operátoru a jeho použití. Jeho praktické využití pro CKomplex by bylo minimální.

```
double operator()(int aIndex) {
…
}
```

```
CKomplex dd(3,4);
double pom = dd(1) + dd(2); // vypadá jako že dd je funkce, ale je to objekt
```

13 Streamy

stream ve třídě

formátovaný tisk complex

14 Třída jako template

15 Dědění

16 Dědění – virtuální metody

17

18 y