1. Úvod

Přednášky
– 2 hodiny – středa
– cca v devátém týdnu půlsemestrálka – opakování C
– převážně C++ . C jen opakování
– C je prerekvizita – probráno v BPC1 a BPC2
– předpokládaný obsah přednášek a cvičení je na síti: dzin_app: everyone / vyuka / richter / bppc nebo ftp přístup. ftp dzin.feec.vutbr.cz
– nejlépe předem pročíst téma hodiny,
– Přednášející: Richter richter@feec.vutbr.cz

Cvičení
- rozdělení podle rozvrhu – dvě hodiny týdně
- lab. 240 – ve skupinách po dvou lidech, projekty po třech
- pracujeme v Microsoft Visual C++, je možné jakýkoli překladač C++ respektující aktuálně platnou normu, před odevzdáním v MVC++
- vytváření programů na úrovni konzolových aplikací
- hlavní náplní bude tvorba tříd a jejich rozhraní
- pro každé cvičení bude na síti připraveno zadání
- vystaveno starší cvičení – s větší váhou na C

Cvičící: Richter, Petyovský petyovsk@feec.vutbr.cz

Domácí práce
- příprava na přednášky, cvičení
- přítomnost na přednáškách, cvičeních
- zkusit si naprogramovat příklady ze cvičení
- vypracovat alespoň vzorové třídy a projekt
- nutná motivace – vyberte si zajímavý úkol

Konference

- loňská a předloňská konference na www. pandora. cz. Praktické programování v C++. Někdy nežije – problémy se školními maily. Je dobré si ji přečíst
- Pokusíme se buď oživit pandoru, nebo založit konferenci na školním serveru
- Určeno především pro „technické“ diskuze o jazyku C a C++.

Konzultační hodiny

- dotazy na přednáškách
- dotazy na cvičeních
- konference
- email
- Richter 10:00- 10:45 středa
- individuální dohoda

Hodnocení - Bodování

způsob je kompromisem
Půlsemestrální písemný test – klíčová slova, makra, jednoduchý příklad z jazyka C. 30 bodů
Projekt – tvorba třídy a její použití, prezentace na cvičeních 20 bodů
Písemná zkouška – příklady jazyka C++, 50 bodů. Jeden termín řádný, dva opravné.
Testy jsou písemné
Zápočet je ze cvičení – odevzdán projekt a dosaženo alespoň 5 bodů.
Zápočet nutný k přístupu na zkoušku

Literatura

– Skripta – kapitola o C a základy C++ jsou v pořádku, dědění je zatím informativně. Na konci kapitol skript je shrnutí – nutno znát a vědět proč. Poznámky v textu – rozšíření látky
– Knihy – každý měsíc vychází nová, je možno si vybrat tu nejvhodnější

– Odkazy na síti – většina se ukázala jako špatná (nepřesnosti, neúplnosti)
– Norma

2. Motivace C++

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

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

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

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

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

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

Návrh metod
- metoda – funkce ve třídě pro práci s daty třídy
- metody vzniku a zániku
- metody pro práci s daty
- metody pro práci s objektem
- operátory
- vstupy a výstupy
- metody vzniklé implicitně (ošetřit dynamická data)
- zde se (hlavně) zužitkuje C - algoritmy

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

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

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

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

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

Objektové vlastnosti
- objekty

- dědění
- výjimky

3.1 Objektové programování, základní definice

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

Základní pojmy

- ***třída (class)*** - datový celek (datová abstrakce), data + operace, přístupová práva
- ***instance*** - proměnná
- ***objekt*** - instance nějaké třídy
- ***metoda*** - funkce definovaná ve třídě pro práci s daty
- ***zapouzdření (encapsulation)*** –
- ***konstruktor a destruktor*** - metody pro inicializaci a likvidaci objektu
- ***rozhraní (interface)*** - co třída nabízí ven
- ***implementace*** - jak to dělá třída uvnitř
- ***dědičnost (inheritance)*** - použití kódu pro podobnou třídu
- ***polymorfismus*** - třídy se stejným rozhraním, a různou implementací, jednotný přístup k instancím

3.2 komentáře (no)

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

```
int k;  // komentář
```

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

3.3 pojem třídy a struktury v C++ (o)

- složený datový typ – jako struct
- objektové vlastnosti rozšířeny i pro struct a union
- klíčové slovo class
- deklarace třídy - pouze popis třídy – nevyhrazuje paměť

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

- deklarace jména třídy – lze použít pouze adresu
- vlastní popis třídy později

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

- definice proměnné

```
jméno_třídy a, b, *pc;
```

3.4 deklarace a definice proměnných (no)

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

```
for (int i=0;i<10;i++)
 { /* zde je i známo */
   …
   …
   double dd;
   …
```

```
 }  // zde končí i a dd
```

3.5 data, metody - práce s nimi (o)

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

```
class Jmeno_třídy {
  int data1;
  float data2;
  Jmeno *j;
  char string[100];

  int metoda1() {…return 2;}
  void metoda2(int a,float b) {…}
  float mmetoda3( int  a1,  Jmeno *a2);
 };

Jmeno_třídy aa;
int b = aa.metoda3(34,"34.54");
```

3.6 přístupová práva (o)

- klíčová slova – private, public, protected
- přepínače
- možno vkládat libovolně
- rozdíl mezi class a struct – implicitní přístupové právo

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

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

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

Komplex a,b;
a.Re = 1; // je možné
b.Uhel(3.14); // je možné
a.pom = 3; // není možné
b.Velikost(); // není možné


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

3.6 reference (no)

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

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

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

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

- „splývá“ předávání hodnotou a referencí (až na prototyp stejné)
- práce s referencí = práce s původní odkazovanou proměnnou

- nelze reference na referenci, na bitová pole,
- nelze pole referencí, ukazatel na referenci

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

používá se:
- přístup k datům a metodám aktuálního prvku (this je možné vynechat) this->data = 5, b = this->metoda(a)
- objekt vrací sám sebe – return *this;
- kontrola parametru s aktuálním prvkem if (this==&param)

- metoda pro vrácení maxima

```
Komplex& Max(Komplex &param) // & reference
{
if (this == &param) // & adresa
    return *this;// oba parametry totožné
if ( this->Re < param.Re ) return param;
else                       return *thís;
}
```

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

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

3.8 operátor příslušnosti :: (no)

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

```
Prostor::JménoProměnné
Prostor::JménoFunkce
```

```
int Stav;
fce() {
  int Stav;
  Stav = 5;
  ::Stav = 6;
}
```

```
int Komplex::metoda(int, int) {}
```

3.9 statický datový člen třídy (o)

- vytváří se pouze jeden na třídu, společný všem objektům třídy
- např. počítání aktivních objektů třídy,
- v deklaraci třídy označen jako static

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

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

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

3.10 přetěžování funkcí (no)

- v C jediná funkce s daným jménem
- v C++ více stejnojmenných funkcí – přetěžování
- (přetěžovování není překrytí ale přidání)
- funkce odlišené počtem nebo typem parametrů, prostorem,

- typ návratové hodnoty nerozlišuje
- při volání vybírá překladač na základě kontextu
- přednost má „nejbližší“ jinak uvést celé - Prostor::Jméno
- problém s konverzemi

```
int f(int);
float f(int); // nelze rozlišit
float f(float);

float ff = f(3); // volání f(int)
f(3.14); // chyba – double lze na int i float
         // – překladač neví
f( (float) 3.14); // v pořádku
```

3.11 implicitní parametry (no)

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

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

```
f(char); // nejednoznačnost při volání
s parametrem int – konverze na char i float
```

3.12 přetypování (no)

- explicitní (vynucená) změna typu proměnné
- v C se provádí **(float) i**
- v C++ se zavádí „funkční“ přetypování **float(i)** - operátor
- lze nadefinovat tuto konverzi-přetypování u vlastních typů

3.13 const, const parametry (no)

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

U použití ve více modulech (deklarace-definice v h souboru)

- u C je **const char a=´b´;** evivalentní **extern const char a=´b´;**
- pro lokální viditelnost - **static const char a=´b´;**
- u C++ je **const char a=´b´;** evivalentní **static const char a=´b´;**
- pro globální viditelnost - **extern const char a=´b´;**

- potlačení možnosti změn u parametrů předávaných funkcím ukazatelem a odkazem
- volání const parametrem na místě neconst parametru (fce) - nelze

```
int fce(const int *i) …
```

shrnutí:

`T`	`je proměnná daného typu`
`T *`	`je ukazatel na daný typ`
`T &`	`reference na T`

`const T` `T const`	`deklaruje konstantní T` `(const char a=´b´;)`
`T const *` `const T*`	`deklaruje ukazatel na konstantní T`
`T const &` `const T&`	`deklaruje referenci na konstantní T`
`T * const`	`deklaruje konstantntí ukazatel na T`
`T const * const` `const T* const`	`deklaruje konstatntní ukazatel na konstantní T`

3.14 alokace paměti (no)

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

jedna proměnná

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

char*  pch = (char*) new char;
delete pch;
*kk = new Komplex(10,20); // s inicializací
```

pole

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

Komplex *pck = (char*) new Komplex [5*i];
 // implicitní konstruktor pro každý prvek
```

```
delete[]  pck; // 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)
 {
  if (změna) Save("xxx");
  if (ptr!=NULL) ::delete ptr; …}
```

3.15 enum (no)

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

3.16 konstruktory a destruktory (o)

```
{
 int a;
//definice proměnné bez konkrétní inicializace
 int b = 5;
// definice s inicializací
//vytvoření,konstrukce, na základě int hodnoty
 int c = b;
// konstrukce (vytvoření) proměnné na základě
// proměnné stejného typu
…
}  // konec platnosti proměnných – zrušení
```

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

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

```
class Trida {
public:
Trida(void) {…}
Trida(int i) {…}
 }
```

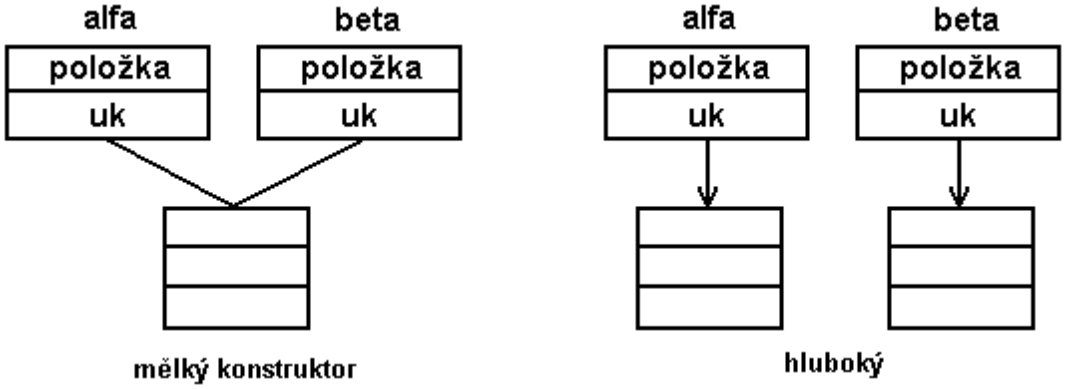
- několik konstruktorů – přetěžování
- implicitní (bez parametrů) – volá se i při vytváření prvků polí
- konverzní – s jedním parametrem
- kopy konstruktor – vytvoření kopie objektu stejné třídy (předávání hodnotou, návratová hodnota …)

```
Trida(void)
Trida(int i)
Trida(char *c)
Trida(const Trida &t)
Trida(float i, float j)
Trida(double i, Trida &t1)

Trida a, b(5), c(b), d=b, e("101001");
Trida f(3.12, 8), g(8.34, b), h = 5;
```

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

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

int i;
 a.Metoda ( i ); // nelze
 b.Metoda ( Trida(i)); // lze
```

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



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

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

```
~Trida (void)
```

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

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

```
class Trida {
int i;
Trida1 a;
Trida2 b;
public:
Trida(int i1,int i2, int i3):b(i3),a(i2),i(i1)
{…}
}
```

3.17 inline funkce (no)
- obdoba maker v C
- předpis pro rozvoj do kódu, není funkční volání
- v hlavičkovém souboru
- označení klíčovým slovem inline

- pouze pro jednoduché funkce (jednoduchý kód)

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

3.18 Hlavičkové soubory a třída (o)

- hlavičkový soubor (.h), inline soubor (.inl, nebo .h), zdrojový soubor (.cpp)
- hlavička – deklarace třídy s definicí proměnných a metod, a přístupových práv („těla“ inline metod – lépe mimo) - předpis
- inline soubor – „těla“ inline metod -předpis
- zdrojový soubor – „těla“ metod – „skutečný“ kód
- soubory kde je třída používána

hlavička:

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

```
těla metod přímo nebo #include inl soubor
```

```
inl soubor:
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
```

3.19 inline metody (o)

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

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

3.20 shrnutí deklarací a definicí tříd a objektů (o)

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

3.21 operátory přístupu ke členům (o)

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

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

float T::*p2;
```

```
      // definice operátoru pro přístup k
      // proměnné
p2=&T::f2;
      // inicializace přístupu na konkrétní
      // proměnnou zatím nemáme objekt ani
      // ukazatel na něj - pouze definici třídy

T tt,*ut=&tt;
ut->*p2=3.14;
(ut->*p1)();//volání fce -závorky pro prioritu
tt.*p2 = 4;
tt.*p1( )
```

3.22 deklarace třídy uvnitř jiné třídy (o)

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

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

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

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

3.23 const a metody (o)

- const u parametrů – nepředávat hodnotou, ochrana proti nechtěnému přepisu hodnot
- const parametry by neměly být předávány na místě neconst

- na const parametry nelze volat metody, které je změní – kontrola překladač
- metody, které nemění objekt je nutno označit jako const a takto označené metody lze volat na const objekty

```
float f1(void) const {   …  }
```

3.24 prototypy funkcí (no)
- v C nepovinné uvádět deklaraci (ale nebezpečné) – implicitní definice
- v C++ musí být prototyp přesně uveden (parametry, návrat)

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

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

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

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

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

```
int fce (int a, int b, …);
```

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

3.27 typ bool (no)

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

3.28 přetížení operátorů (no)

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

--

3.29 operátory (o)

- operátory lze v C++ přetížit stejně jako metody
- správný operátor je vybrán podle seznamu parametrů (a dostupných konverzí), rozliší překladač podle kontextu
- operátory unární mají jeden parametr – proměnnou se kterou pracují, nebo „this“
- operátory binární mají dva parametry – dvě proměnné, se kterými pracují nebo jednu proměnnou a „this“

- unární operátory: +, -, ~, !
- binární +,-,*,/,%,=,^,&,&&,|,||,>,<,>=,==, +=,*=,<<,>>,<<=, …
- ostatní operátory [ ], ( ), new, delete
- operátory matematické a logické
- nelze přetížit operátory: sizeof, ? :, ::, ., .*
- nelze změnit počet operandů a pravidla pro asociativitu a prioritu
- nelze použít implicitních parametrů
- slouží ke zpřehlednění programu
- snažíme se aby se přetížené operátory chovaly podobně jako původní (např. nemění hodnoty operandů, + sčítá nebo spojuje…)
- klíčové slovo operátor
- operátor má plné (funkční) a zkrácené volání

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

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

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

Unární operátory
- mají jeden parametr (u tříd this)
- například + a –

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

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

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

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

```
++( void)   s voláním   ++x
++( int )   s voláním   x++ argument int se
nevyužívá
```

Binární operátory
- mají dva parametry (u třídy je jedním z nich this)
- například +

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

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

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

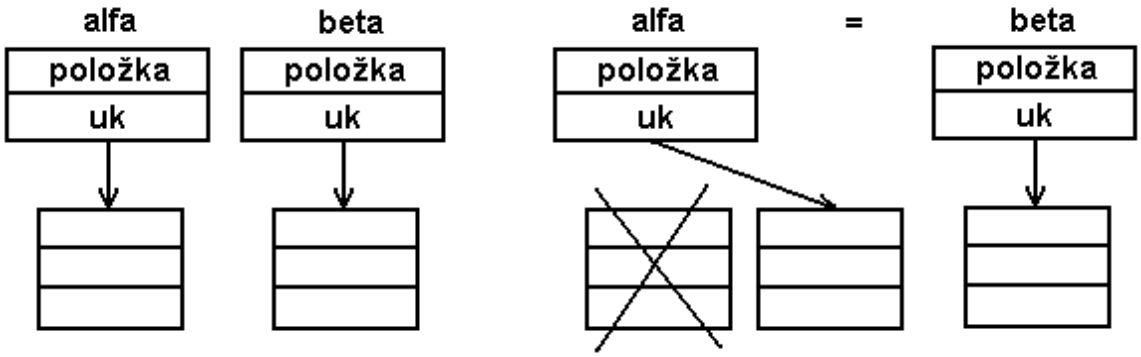
Operátor =
- měl by (díky kompatibilitě) vracet hodnotu
- obzvláště zde je nutné ošetřit případ a = a

- pro činnost s dynamickými daty nutno ošetřit mělké a hluboké kopie
- vytváří se implicitně (mělká kopie – přesná kopie 1:1)
- nadefinování zamezí vytvoření implicitního =
- je-li v sekci private, pak to znamená, že nelze použít (externě)

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

musí umožňovat

```
a = b = c = d = …;
```



Způsoby přiřazení

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

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

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

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

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

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

Konverzní operátory

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

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

volán implicitně nebo

```
T aaa;
int (aaa) ;
(int) aaa; // starý typ – nepoužívat
```

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

```
operator ()(parametry )
double& T::operator()(int i,int j) { }
T aaa;
double d = aaa(4,5);
d  = aaa.operator()(5,5);
aaa(4,4) = d;
```

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

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

přetížení přístupu k prvkům třídy
- je možné přetížit " -> "
- musí vracet ukazatel na objekt třídy pro kterou je operátor -> definován protože:

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

3.30 statické metody (o)
- pouze jedna na třídu

- nemá this
- ve třídě označená static
- vlastní tělo ve zdrojové části
- nesmí být virtuální
- (jako friend funkce, může k private členům)
- externí volání se jménem třídy bez objektu **`Třída::fce( )`**

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

3.31 mutable (o)

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

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

class Y { public: X x; }

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

3.32 prostory jmen (no)

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

prostor::identifikátor
prostor::podprostor::identifikátor

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

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

definice
namespace {
proměnné
funkce
třídy
}

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

3.33 znakové konstanty, dlouhé literály (no)
- typ pro ukládání znakových proměnných větších než char (UNICODE) – tj. „dlouhé“ znakové konstanty
- znaky (char) již proto nejsou konvertovány na int

- v C je sizeof(´a´) = sizeof(int) v C++ je = sizeof(char)
- w_char, wchar_t
- wchar_t b = L´a´;
- wchar_t c[20] = L“abcd“;
- existují pro něj nové funkce a vlastnosti například wprintf( ), wcin, wcout, MessageBoxW( )…
- zatím brán jako „problematický typ“

3.34 typ long long (no)
- nový celočíselný typ s danou minimální přesností 64 bitů

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

3.35 restrict (no)
- nové klíčové slovo v jazyce C, modifikátor (jako const…) u ukazatele
- z důvodů optimalizací – rychlejší kód – (možnost umístit do cache či registru)
- říká, že daný odkaz (ukazatel) je jediným, který je v daném okamžiku namířen na data
- to je - data nejsou v daném okamžiku přístupna přes jiný ukazatel – data
- to je - data se nemění
- const řeší pouze přístup přes daný ukazatel, ne přes jiné (lze const i neconst ukazatel ne jednu proměnnou)

--

pulsem

3.36 Vstupy a výstupy v jazyce C++
- 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;

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

- načítá se proměnný počet znaků – gcount zjistí kolik
- vypouštění bílých znaků – přepínače ws, noskipws, skipws (vynechá bílé znaky, nepřeskakuje, přeskakuje BZ na začátku)

- načtení celého řádku getline(kam,maxkolik) – čte celý řádek, get(kam, maxkolik) – čte řádek bez odřádkování
- načtení znaku get – čte všechny znaky, zastaví se na bílém znaku
- put – uložení znaku (Pouze jeden znak bez vlivu formátování)
- vrácení znaku – putback
- „vyčtení“ znaku tak aby zůstal v zařízení – peek
-

```
int i,j;
 cout << „ zadejte dvě celá čísla /n“;
 cin>> i>> j;

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

formátování základních typů

- je možné pomocí modifikátorů, manipulátorů nebo nastavením formátovacích bitů
- ovlivnění tvaru, přesnosti a formátu výstupu
- může být v „iomanip.h“
- přetížení operátorů << a >> pro parametr typu manip, nebo pomocí ukazatelů na funkce
- přesnost výsledku má přednost před nastavením
- manipulátory - funkce pracující s typem stream – mají stream & jako parametr i jako návratovou hodnotu
- slouží buď pro nastavení nové, nebo zjištění stávající hodnoty
- některé působí na jeden (následující) výstup, jiné trvale
- bity umístěny ve třídě ios (staré streamy), nově v ios_base – kde jsou společné vlastnosti pro input i output, které nezávisí na templatové interpretaci (ios)
- nastavení bitů – pomocí setf s jedním parametrem (vrátí současné)
- setf se dvěma parametry – nastavení bitu + nulování ostatních bitů ve skupině (označení bitu, označení společné skupiny bitů)

- nulování bitů – unsetf
- někdy (dříve) setioflags, resetioflags, flags
- pro uchování nastavení bitů je předdefinován typ fmtflags
- i = os.width(j) – šířka výpisu, pro jeden znak, default 0
- os << setw(j) << i;
- i = os.fill(j) – výplňový znak, pro jeden výstup, default mezera
- os<<setfill(j) << i;
- ios_base::left, ios_base::right, left, right - zarovnání vlevo vpravo – fmtlags orig = os.setf(ios_base::left,ios_base::adjustfield) – manipulátory bity nastaví i nulují
- ios_base::internal, internal – znaménko zarovnáno vlevo, číslo vpravo
- bity left, right, internal patří do skupiny ios_base::adjustfield
- ios_base::showpos, manipulátor showpos, noshowpos – zobrazí vždy znaménko (+, -)
- ios_base::uppercase, uppercase, nouppercase – zobrazení velkých či malých písmen v hexa a u exponentu
- ios_base:: dec,ios_base::hex, ios_base::oct, dec, oct, hex – přepínání formátů tisku (bity patří do skupiny – ios_base::basefield)
- setbase – nastavení soustavy
- ios_base::showbase, showbase, noshowbase – tisk 0x u hexa
- ios_base::boolalpha, boolalpha, noboolalpha – tisk „true“, „false“
- os.precision(j) – nastavení přesnosti, významné číslice, default 6
- os<<setprecision(j) << i
- ios_base::showpoint, showpoint, noshowpoint – nastavení tisku desetinné tečky
- ios_base::fixed, fixed – desetinná tečka bez exponentu
- ios_base::scientific, scientific – exponenciální tvar
- bity fixed, scientific patří do ios_base::floatfield
- eatwhite – přeskočení mezer, writes – tisk řetězce …

práce se soubory

- podobné mechanizmy jako vstup a výstup pro konzolu

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

```
oftream os(„nazev souboru“);
os << „vystup“;
os.close( );
os.open(„jiny soubor.txt“);
if (!os.is_open()) …
```

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

```
ofstream os(„soub.dat“,
ios::out||ios::ate||ios::binary||ios::nocreate
);
istream is(„soub.txt“,
                  ios::in || ios::nocreate);
fstream iostr(„soub.txt“,
                  ios::in || ios::out);
```

- zjištění konce souboru – metoda eof – ohlásí až po načtení prvního za koncem souboru
- zjišťování stavu – bity stavu – ios::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::badbid||ios::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 – obecné definice, většina enum konstant (přesunuty do ios_base), bázová třída nezávislá na typu – ios.h
- 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.h
- iostream = istream + ostream (obousměrný)
- ifstream, ofstream – pro práci s diskovými soubory, automaticky buffer, fstream.h

- istrstream, ostrstream, strstream – pro práci s řetězci, paměť pro práci může být parametrem konstruktoru – strstream.h
- třídy xxx_withassign – rozšíření (istream, ostream) , přidává schopnost přesměrování (např. do souboru,) – cin, cout
- constream – třída pro práci s obrazovkou, clrscr pro mazání, window pro nastavení aktuálního výřezu obrazovky …
- nedoporučuje se dělat kopie, nebo přiřazovat streamy
- stav streamu je možné kontrolovat i pomocí if (!sout), kdy se používá přetížení operátoru !, které je ekvivalentní sout.fail(), nebo lze použít if (sout), které používá přetížení operátoru () typu void *operator (), a který vrací !sout.fail(). (tedy nevrací good).

3.37 Shrnutí tříd

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

#include "komplex2214.h"

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


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

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

e = a += c = d;
a = +b;
c = -d;
d = a++;
e = ++a;
if (a == c)    a = 5;
else           a = 4;

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

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

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

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

float m = e; // pozor - pouľije jedinou moľnou konverzi a to přes int

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

Komplex n(2,7),o(2,7),p(2,7);
```

```
n*=o;
p*=p; // pro první realizaci n*=n je výsledek n a p různy i kdyť
// vstupy jsou stejné

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


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

#ifndef KOMPLEX_H
#define KOMPLEX_H

#include <math.h>

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

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

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

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

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

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

double Amplituda(void)const
                 {   return sqrt(Re*Re + Im *Im);}

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

double Amp(void) const;

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

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

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

Komplex & operator++(void)
                {Re++;Im++;return *this;}
// nejdřív přičte a pak vrátí, takľe můľe vrátit sám sebe
```

```
// (pro komplex patrně nesmysl)

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

//vrací původní prvek, takľe 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 prirazeni,
// implementováno i zřetězení

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

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

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

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

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

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

Komplex& operator! ()
                {Im*=-1;return *this;}; // a tady je ten operátor
```

```
// co mění prvek. Problém je, ľe je to nestandardní pro tento operátor
// a zároveň se mohou plést. Takľe bezpečnějąí je nechat jen ten první
// bool operator&&(Komplex &p) {}

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

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

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

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

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

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

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

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

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

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

#endif


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

#include "komplex2214.h"

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

Komplex::Komplex(const char *txt)
```

```
{
/* vlastni alg */;
Re=Im=0;
Index = Poradi;
Poradi++;
Aktivnich++;
}


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

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

---- ----- ---
určení datumů zkoušek
---- ---- ----

4.1 dědění

- „znovupoužití“ kódu (s drobnými změnami)
- odvození tříd z již existujících
- převzetí a rozšíření vlastností, sdílení kódu
- dodání nových proměnných a metod
- „překrytí“ původních proměnných a metod (zůstávají)
- původní třída – bázová, nová – odvozená
- při dědění se mění přístupová práva proměnných a metod bázové třídy v závislosti na způsobu dědění
- class C: public A – A je bázová třída, public značí způsob dědění a C je název nové třídy

- způsob dědění u třídy je implicitně private (a nemusí se uvádět), u struktury je to public (a nemusí se uvádět) class C: D
- tabulka ukazuje jak se při různém způsobu dědění mění přístupová práva bázové třídy (A) ve třídě zděděné

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

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

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

class Derived : Base {
 public:
 int a ;
  Derived ( int i ) : a ( i*10 ) : Base (a)
{ }
// volání lokálních konstruktoru umožní
// konstrukci podle požadavků, ale nezmění
```

```
// pořadí konstruktorů
 Base::y;  // je možné takto vytáhnout
// proměnnou (zděděnou zde do sekce private)
// na jiná přístupová práva (zde public)
}

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

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

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

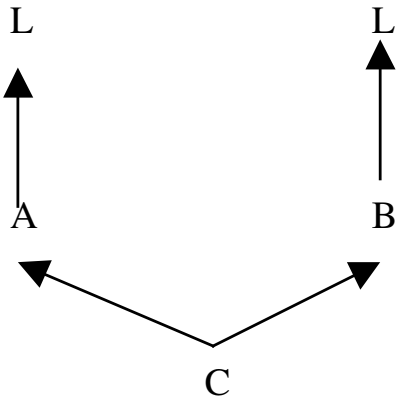
volá se base::base()
complex::complex(void) - pro x
complex::complex(2,1) – pro y
f() … - vlastní tělo konstruktoru

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

4.2 vícenásobné dědění

- lze dědit i z více objektů najednou
- problémy se stejnými názvy – nutno rozlišit
- problémy s vícenásobným děděním stejných tříd - virtual
- nelze dědit dvakrát ze stejné třídy na stejné úrovni C:B,B

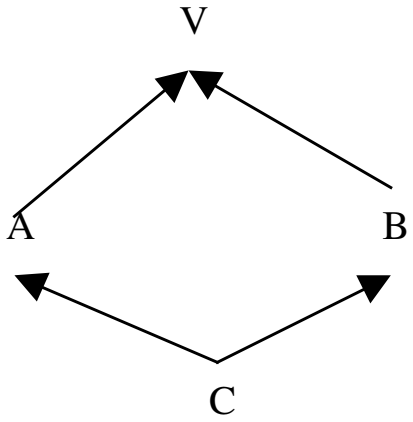
A: public L
B: public L
C: public A, Public B



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

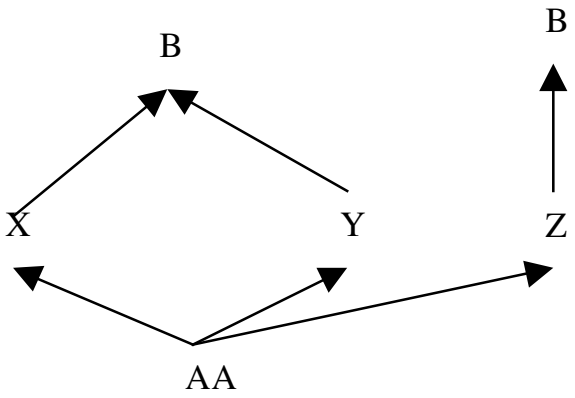
- - - - -

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



- - - - - - - -

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



4.3 virtuální metody

Virtuální metody

- zajišťují tzv. pozdní vazbu, tj. zjištění adresy metody až za běhu programu pomocí tabulky virtuálních metod,
- tvm - se vytváří voláním konstruktoru.
- V "klasickém" programování je volaná metoda vybrána již při překladu překladačem na základě typu proměnné, funkce či metody, která se volání účastní.
- U virtuálních metod není důležité čemu je proměnná přiřazena, ale jakým způsobem vznikla – při vzniku je jí dána tabulka metod, které se mají volat. Tato tabulka je součástí prvku.
- jsou-li v bázové třídě definovány metody jako virtual, musí být v potomcích identické
- ve zděděných třídách není nutné uvádět virtual
- metoda se stejným názvem, ale jinými parametry se stává nevirtuální, tedy statickou
- pokud je virtual v odvozené třídě a parametry se liší, pak se virtual ignoruje
- virtuální metody fungují nad třídou, proto nesmí být ani static ani friend
- i když se destruktor nedědí, může být virtuální
- Využívá se v situaci kdy máme dosti příbuzné objekty, potom je možné s nimi jednat jako s jedním (Např. výkres, kresba – objekty mají parametry, metody jako posun, rotace, data … Kromě toho i metodu kresli na vykreslení objektu)

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

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

- Virtual je povinné u deklarace a u inline u definice.
- ve zděděných třídách není nutn 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 virualní f-ce fungují nad třídou, nesmí být virtual friend, ani static
- i když se destruktor nedědí, destruktor v děděné třídě přetíží (zruší) virtuální
- virtuální funkce se mohou lišit v návratové hodnotě, pokud tyto jsou vůči sobě v dědické relaci

Čisté virtuální metody

- pokud není vitruální 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 viruální metodě, musí být deklarována.
- chceme využít jednu třídu jako Bázovou, ale chceme zamezit tomu, aby se s ní pracovalo. Můžeme v konstruktoru vypsat hlášku a exitovat. Čistější je ovšem, když na to přijde překladač – tj. použít čistě virtuální metody
- deklarace vypadá: virtual void f (int)=0 (nebo =NULL)
- tato metoda se dědí jako čistě virtuální, dkud není definována
- starší překladače vyžadují v odvozené třídě novou deklaraci a nebo definici

- obsahuje-li objetk č.v.m. nelze vytvořit jeho instanci, může být ale ukazatel

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

B b; C c;D d;
b.f(); c.f(); d.f(); // vola normalni metody tridy / tisk b c d – protože promenne jsou typu B C D / prekladac
b.vf1();c.vf1();d.vf1(); // vola virtualni metody tridy / tisk b c d – protože promenne vznikly jako B C D/ runtime
B* bp = &c; // ukazatel na bazovou tridu (prirazeni muze byt i v ifu, a pak se nevi co je dal)
Bp->f(); // vola normalni metodu tridy B / tisk b, protože promenna je typu B / prekladac
bp -> vf1(); // vola virtualni metodu / tisk c – protože promenna vznikla jako typ c / runtime

4.4 abstraktní bázové třídy

- 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 t();
}

Z c; uz jde
c.h() z x
c.f() z y
c.g() z Z
```

4.5 šablony (template)

- umožňují psát kód pro obecný typ
- vytvoří se tak návod (předpis, šablona) na základě které se konkrétní kód vytvoří až v případě potřeby pro typ, se kterým se má použít
- umisťuje se do hlavičkového souboru (předpis, netvoří kód)
-

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

```
}

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

- template – klíčové slovo říkající, že se jedná o předpis
- <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 vytvoření na základě template 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> …

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

T T<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) { ..... }
}
```

4.6 výjimky (exceptions)

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

```
try
    {
```

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

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

4.6 C v C++

- 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);