➢ **Deklarace proměnných**

- Implicitní – proměnnou nemusíme před jejím použitím deklarovat.
- Explicitní – proměnná musí být před použitím deklarována.

Některé jazyky umožňují implicitní i explicitní deklarace proměnných – Visual Basic. Implicitní deklarace jsou mnohdy zdrojem nepříjemných chyb. Je-li to možné, implicitní deklarace zakážeme (lze to v jazyce Visual Basic) nebo je aspoň nepoužíváme.

- Inicializace proměnných.

  Mnohdy potřebujeme, aby proměnná měla stanovenu počáteční hodnotu. Zde je účelné tuto hodnotu ji přiřadit přímo v deklaraci, pokud to daný jazyk dovoluje.

```
double soucet = 0;
```

  Pokud jazyk nedovoluje inicializaci v deklaraci anebo tuto možnost nevyužijeme, je vhodné inicializace udělat bezprostředně před prvním použitím proměnné.

```
double soucet;
// ...
soucet = 0;
// výpočet součtu ...
```

  Pokud to jazyk dovoluje, nejlépe je mít obojí, deklaraci i inicializaci bezprostředně před místem, kde je proměnná použita.

```
double soucet = 0;
// výpočet součtu ...
```

- Rozsah platnosti proměnné.

  Je část programu, kde je proměnná deklarována a může být použita.

- Oblast života proměnné.

  Je část programu od místa, kde je proměnné přiřazena hodnota, až po poslední použití přiřazené hodnoty (je reference na proměnnou).

```
{
    .....
  i = m/2;
    .....          // zde není reference na proměnnou i
  j = i-1;
    .....          // zde není reference na proměnnou i
  k = i+j;
    .....          // zde není reference na proměnnou i
}
```

- Oblasti života proměnných se snažíme, aby byly co nejkratší.

```
void SouhrnDat(...) {
  ...
nactiStaraData(staraData, &pocetStarychDat);
nactiNovaData(novaData, &pocetNovychDat);
soucetStarychData = Soucet(staraData, pocetStarychData);
```

```
soucetNovychDat = Soucet(novaData, pocetNovychDat);
tiskniStaraData(staraData, soucetStarychDat, pocetStarychDat);
tiskniNovaData(novaData, soucetNovychDat, pocetNovychDat);
ulozStaraData(soucetStarychDat, pocetStarychDat);
ulozNovaData(soucetNovychDat, pocetNovychDat);
...
}
```

V předchozí funkci se střídá zpracování *starých* a *nových* dat. Účelné je udělat nejprve celé zpracování *starých* dat a následně celé zpracování *nových* dat.

```
void SouhrnDat(...) {
  ...
nactiStaraData(staraData, &pocetStarychDat);
celkemStarychData = Soucet(staraData, pocetStarychData);
tiskniStaraData(staraData, soucetStarychDat, pocetStarychDat);
ulozStaraData(soucetStarychDat, pocetStarychDat);
 ...
nactiNovaData(novaData, &pocetNovychDat);
soucetNovychDat = Soucet(novaData, pocetNovychDat);
tiskniNovaData(novaData, soucetNovychDat, pocetNovychDat);
ulozNovaData(soucetNovychDat, pocetNovychDat);
 ...
}
```

- Rozsah platnosti proměnné volíme tak, aby byl co nejmenší vzhledem k oblasti jejího života. Což zejména znamená, že proměnné deklarujeme na lokální úrovni, pokud to jde.
- Každá proměnná by se měla použít jen pro jeden účel.

```
// Výpočet kořenů kvadratické rovnice.
// (Předpokládáme, že diskriminant b*b-4*a*c není záporný.)
  double prac = sqrt(b*b - 4*a*c);
  koren1 = (-b + prac) / (2*a);
  koren2 = (-b - prac) / (2*a);
  ...
  // rozdíl mezi kořeny
  prac = abs(koren1 - koren2);
```

Stejná proměnná zde byla použita dvakrát pro různý účel. Místo toho použije dvě samostatné proměnné.

```
// Výpočet kořenů kvadratické rovnice.
  double diskriminant = sqrt(b*b - 4*a*c);
  koren1 = (-b + diskriminant) / (2*a);
  koren2 = (-b - diskriminant) / (2*a);
  ...
// rozdíl mezi kořeny
  double rozdil = abs(koren1 - koren2);
```

- Každá deklarovaná proměnná by měla být použita.
- Je zbytečné zavádět dočasné proměnné, pokud to nepotřebujeme a nejde ani o zjednodušení komplikovaných výrazů.

```
int c = a[i];
b[i] += 2*c;
```

zde stačí

```
b[i] += 2*a[i];
```

➢ **Jména proměnných**

```
x = x - xx;
xxx = castka + danZObratu(castka);
x = x + xxx - poplatek;
x = x + urok(x);

zustatek = zustatek - posledniPlatby;
mesicniPrijem = noveProdeje + danZObratu(noveProdeje);
zustatek = zustatek + mesicniPrijem - poplatek;
zustatek = zustatek + urok(zustatek);
```

Jména proměnných by měla výstižně popisovat, co hodnoty uložené v proměnných reprezentují. Například *urokovaMira*, *pocetRadkuNaStrance*.

- Délka jména.

  Statistické výzkumy ukazují, že nejpřehlednější jsou programy, ve kterých průměrná délka jmen je mezi 10 až 16 znaky.

- Závislost délky jména proměnné na rozsahu její platnosti.

  I krátké jméno proměnné, jako je *i*, je vyhovující, jde-li o proměnnou s oblastí života, která má jen několik řádků, například proměnná cyklu. Dlouhá jména jsou vhodnější pro globální proměnné, naproti tomu krátká jména proměnných jsou lepší pro lokální proměnné nebo proměnné cyklu. V následujícím kódu, vidíme, jaké hodnoty obsahují proměnné ***r*** a ***s***.

```
regex r("[[:xdigit:]]{4}");
string s;
cin >> s;
if (regex_match(s,r)) { … }
```

  Pokud bychom měli jen samostatně příkaz

```
if (regex_match(s,r)) { … }
```

  pak už není zřejmé, jakou hodnotu a s čím srovnává.

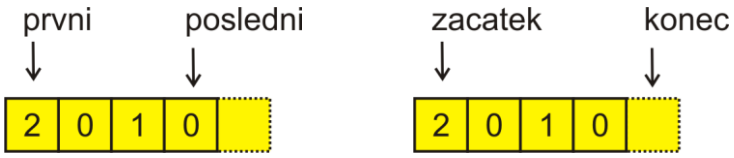
- Proměnné obsahující vypočítané hodnoty.

  O jaký druh vypočtené hodnoty jde (průměr, celkem, maximum, součet, …), uvedeme na konci *trzbaCelkem*, *vydajePrumerne*. Výhoda je jednotný formát a hlavně, že důležitější část jména je na začátku.

- Vhodné jméno proměnné nám může pomoct zjistit chybu v programu

```
navratovaHodnota += a[i];    // zde žádná chyba není viditelná
soucetOdmocnin += a[i];      // nechybí zde odmocnina ?
```

- Obsahuje-li proměnná hodnotu, která má nějakou jednotku, můžeme tuto jednotku přidat ke jménu proměnné. Například místo *vahaOvoce* použijeme *vahaOvoceKg*.
- Ke jménu můžeme i přidat jiný atribut. Například obsahuje-li proměnná heslo, které teprve bude zašifrováno, nazveme proměnnou *hesloNesifrovane*.
- Vhodné pojmenování rozsahu *první-poslední* nebo *začátek-konec*.



První-poslední lze použít v situaci, kdy rozsah hodnot nemůže být prázdný. *Začátek-konec* se naopak používá zejména v případech, kdy rozsah hodnot může být prázdný, což se dá zapsat *začátek=konec*.

➢ **Základní datové typy**

- Používat pojmenované konstanty místo přímých čísel.

```
for (int i=0; i<100; ++i) …

const int POCETPRVKU=100;

for (int i=0; i<POCETPRVKU; ++i) …
```

- Někdy je přehlednější explicitně uvést konverzi.

```
int i,j;
char c;

j = i + c;

j = i + (int)c;
```

- Nepoužívat smíšená srovnání mezi celočíselným typem a typem v pohyblivé řádové čárce.

```
int i; double x;

if (i==x) …
```

- Přehlednost programu lze zvýšit použitím logických proměnných místo prosté podmínky.

```
if (prvekIndex<0 || MAXPRVKU<=prvekIndex ||
                    prvekIndex==posledniPrvekIndex) {
  ...
}

bool mimoRozsah = prvekIndex<0 || MAXPRVKU<=prvekIndex;
bool ukonceno = prvekIndex==posledniPrvekIndex;
if (mimoRozsah || ukonceno) {
  ...
}
```

- Nemá-li programovací jazyk logický datový typ, vytvořte si ho.

  C – jazyk:

```
#define bool char       // nebo typedef char bool;
#define false 0
#define true  1
```

- Odvozený datový typ celkově zpřehlední kód.

```
struct Bod { float x,y; };

typedef float souradnice;
struct Bod { souradnice x,y; };
```

➢ **Podmínkový příkaz**

- Při srovnání dáváme jako první (levou) srovnávanou hodnotu a jako druhou (pravou) hodnotu, s kterou je srovnání.

```
if (index >= 10) …
```

- V příkazu if (..) .. else … jako první by měl být kód, který očekáváme, že se provede. Druhý (po *else*) pak kód, který se provede ve výjimečném případě.

```
float *pole = new float [10000000];
if (pole == NULL) {
  …                          // ošetření chyby
}
else {
  …
}
```

Místo toho ošetření chyby dáme do části *else*.

```
if (pole != NULL) {
  …
}
else {
  …                          // ošetření chyby
}
```

- Zejména ve složitějším případě to zpřehlední kód.

```
ifstream soubor("jmeno");
if (soubor.is_open()) {
  soubor.get(znak);
  if (soubor.eof()) {
      …                      // nejsou další data
  }
  else { if (soubor.good()) {
        …                    // výpočet
    }
```

```
    else {
      ...                          // chyba při čtení
    }
  }
else {
      ...                          // soubor se neotevřel
}
```

Dáme na začátek všechny očekávané případy.

```
if (soubor.is_open()){
  soubor.get(znak);
  if (!soubor.eof()) {
    if (soubor.good()) {
      ...                                // výpočet
    }
    else {
      ...                          // chyba při čtení
    }
  }
  else {
      ...                          // data nenalezena
  }
}
else {
      ...                          // soubor se neotevřel
}
```

- Dále v příkazu *if (..) .. else ...* jako první by měl být kód, jehož provedení je nejobvyklejší (častější). Druhý (po *else*) pak kód, jehož provedení je méně obvyklé.

```
if (isalpha(vstupniZnak)) { typZnaku = PISMENO; }
else { bool jeOddelovac = vstupniZnak==' ' ||
                  vstupniZnak==',' || vstupniZnak=='\n';
  if (jeOddelovac) { typZnaku = ODDELOVAC; }
  else { bool jeInterpunkce= vstupniZnak=='.' ||
                  vstupniZnak=='?' || vstupniZnak=='!';
    if (jeInterpunkce) { typZnaku = INTERPUNKCE; }
    else { typZnaku = JINYZNAK; }
  }
}
```

- I když část po *else* nepotřebujeme, můžeme v ní uvést prázdný příkaz s komentářem, co nastává, když se podmínkový příkaz neprovede.

```
    ifstream("cesta") soubor;
    if (soubor.is_open()) {
```

```
                    ...
    }
    else {
                        // nedostupná data jsou vynechána
    }
```

- Používání ternárního operátoru je ve složitějších případech nepřehledné. Je lepší použít místo něho příkaz *if*, i když zápis je delší. Místo

```
return exponent>=0 ? mantisa*(1<<exponent): mantisa/(1<<-exponent);
```

je lepší použít příkaz *if*

```
if (exponent>=0) {
    return mantisa*(1<<exponent);
} else {
    return mantisa/(1<<-exponent);
}
```

➢ Přepínač

- Uspořádání jednotlivých případů může být
  - Abecedně nebo dle číselných hodnot.
  - Nejdříve obvyklé případy.
  - Seřazení podle toho, jak často se jednotlivé případy vyskytují.
- U rozsáhlejšího přepínače je velmi efektivní, když číselné hodnoty jdou za sebou 0,1,2,3…

```
switch (vyraz) {
  case 0: …
  case 1: …
  case 2: …
  case 3: …
  case 4: …
  case 5: …
  ……
}
```

➢ Cykly

- Ukončením uvnitř těla cyklu lze odstranit opakování kódu.

```
ifstream soubor("cesta");
suma=0;
soubor >> i;
while (!soubor.eof()) {
  suma+=i;
  soubor >> i;
}
```

Čtení proměnné *i* je zde dvakrát. Odstraníme to.

```
ifstream soubor("cesta");
suma=0;
while (true) {
  soubor >> i;
  if (soubor.eof()) break;
  suma+=i;
}
```

- Ukončením uvnitř cyklus lze eliminovat proměnnou pro řízení cyklu.

```
bool hotovo=false;
while ( /* podmínka */ && !hotovo)
      ...
  if (…) { hotovo=true;
           continue;
  }
      ...
}
```

Místo proměnné *hotovo* použijeme ukončení uvnitř cyklu.

```
while ( /* podmínka */ )
    ...
  if (…) break;
    ...
}
```

- V příkazu *for* by neměly v jeho hlavičce být části, které nesouvisí s podmínkou cyklu.

```
int cisla[]={ 5,8,4,0 };
for (i=soucet=0; cisla[i]>0; soucet+=cisla[i],++i);
```

Výpočet součtu dáme do těla cyklu.

```
soucet=0;
for (i=0; cisla[i]>0; ++i) soucet+=cisla[i];
```

- Tělo cyklu je přehlednější vždy psát jako blok (závorkami { }), i když obsahuje jen jeden příkaz.

Přepíšeme tělo cyklu cyklus z předchozího příkladu jako blok.

```
for (i=0; cisla[i]>0; ++i) { suma+=cisla[i]; }
```

Čitelnost následujících vnořených cyklů

```
for (int i=0; i<pocetRadku; ++i)
for (int j=0; j<pocetSloupcu; ++j)
maticeC[i][j] = maticeA[i][j] + maticeB[i][j];
```

zvýší jejich zápis uzavřením těl cyklů do bloků:

```
for (int i=0; i<pocetRadku; ++i) {
```

```
    for (int j=0; j<pocetSloupcu; ++j) {
      maticeC[i][j] = maticeA[i][j] + maticeB[i][j];
    }
  }
```

- Je přehlednější nepoužívat cyklus s prázdným tělem.

```
while ((vstupniZnak = cin.get()) != '\n') { }
```

```
do {
  vstupniZnak = cin.get();
} while (vstupniZnak != '\n');
```

➢ **Vnoření bloků**

Čím více je vzájemně vnořených bloků, tím je obtížnější sledovat strukturu programu. Pokud to jde, snažíme se změnou uspořádání kódu vnoření bloků snížit. Ve funkci lze k tomu někdy využít návrat z funkce *return*, v cyklu lze někdy využít příkaz *continue*.

```
for (int i=0; i<delka; ++i) {
  if (isxdigit(s[i])) {
    c *= 16;
    if (isdigit(s[i])) {
      c += s[i]-'0';
    } else {
      c += toupper(s[i])-('A'-10);
    }
  }
}
```

Použitím příkazu *continue* cyklus upravíme a snížíme hloubku vnoření.

```
for (int i=0; i<delka; ++i) {
  if (!isxdigit(s[i])) continue;
  c *= 16;
  if (isdigit(s[i])) {
    c += s[i]-'0';
    continue;
  }
  c += toupper(s[i])-('A'-10);
}
```

➢ **Specifické případy řízení výpočtu**

- Je nepřehledné používat rekurze, které zahrnují více než jednu funkci.

```
void funA { funB(); }
void funB { funC(); }
void funC { funA(); }
```

- Rekurze by se měla používat jen v případech, kdy je to účelné.
  Typickým příkladem nevhodného a neefektivního použití rekurze je výpočet faktoriálu.

```
int faktorial(int cislo) {
  if (cislo == 1) {
      return 1;
  }
  else {
    return cislo * faktorial(cislo-1);
  }
}
```

Výpočet faktoriálu udělat prostým cyklem bez rekurze.

```
int faktorial(int cislo) {
  int fakt = 1;
  for (int i=2; i<=cislo; ++i) {
    fakt *= i;
  }
  return fakt;
}
```

- Použití příkazu skoku *goto* je často výhodné při opuštění vnořených cyklů.

```
for (…) {
  for (…) {
    for (…) {
      goto Konec;
    }
  }
}
Konec: …
```

- Jiný případ výhodného použití příkazu *goto* je skok při chybě na místo, kde je ošetření chyby.

```
… kód
if (…) goto Chyba;
```

➢ **Logické výrazy**

- Nepoužívat srovnání s logickými hodnotami *true* a *false*.

```
if (hotovo == false) …
if ((a<b) == true) …
```

Lze přepsat bez logických hodnot.

```
if (!hotovo) …
if (a<b) …
```

- Komplikované logické výrazy jsou nepřehledné.

```
if (!konecDat && !chybaVstupu && MINRADKU<=citacRadku &&
    citacRadku<=MAXRADKU && !ChybnaData)
```

  Část podmínky napíšeme samostatně.

```
bool prectenaData = !konecDat && !chybaVstupu,
     spravnyPocetRadku = MINRADKU<=citacRadku &&
                                 citacRadku<=MAXRADKU;
if (prectenaData && spravnyPocetRadku && !ChybnaData)
```

- Úprava logických výrazů s použitím DeMorganových zákonů.

```
if (!zarizeniPripojeno || !dataNactena || !dataVPoradku)
if (!(zarizeniPripojeno && dataNactena && dataVPoradku))
```

  Jiný příklad:

```
if (!(uzel==NULL || uzel->dalsi==NULL)) uzel=uzel->dalsi;
if (uzel!=NULL && uzel->dalsi!=NULL) uzel=uzel->dalsi;
```

- Použitím závorek lze zvýšit čitelnost výrazu.

```
if (a < b && c == d) …
if ((a < b) && (c == d)) …
```

  To platí i pro jiné typy výrazů, než jsou logické výrazy.

```
a << b + c
a << (b + c)
```

- Je důležité vědět, jak se logické výrazy v daném jazyce vyhodnocují.

```
if (a && (--b > 0))
```

  V jazycích C, C++, C# se nejdříve vyhodnotí levý operand logické spojky `&&` nebo `||`. Pokud má hodnotu *false* u spojky `&&` nebo hodnotu *true* u spojky `||`, pravý operand se už nevyhodnotí. Zde v tomto případě nedojde k dekrementaci proměnné *b*.

- Srovnání hodnot s intervalem je přehlednější zapisovat tak, aby menší hodnota byla vlevo od operátoru a větší hodnota vpravo.

```
i>=MINPOCET && i<=MAXPOCET
i<MINPOCET || i>MAXPOCET
...
MINPOCET<=i && i<=MAXPOCET
i<MINPOCET || MAXPOCET<i
```

- Srovnání, zda hodnota je nenulová, je sice delší, ale přehlednější.

```
if (delitel) …
if (delitel != 0) …
```

- Obdobně srovnání v jazycích C, C++, zda nejsme na konci řetězce.

```
while (*ukazRet) …
while (*ukazRet != 0) …
while (*ukazRet != '\0') …
```

➢ Poznámky, komentáře

Čím je kód lépe napsaný, tím méně potřebuje komentáře. Komentáře mnohdy jsou jen snahou kompenzovat situaci, kdy je kód napsán nesrozumitelně. Dobře napsaný kód nepotřebuje výraznější komentování. Navíc při vývoji a změně kódu se často zapomíná na příslušnou změnu komentářů a komentáře se tím často stávají spíše matoucí.

Komentáře špatný kód nespraví. Pokud potřebujeme do kódu často psát komentáře, je lépe se zamyslet nad tím, že bychom měli kód přepsat.

```
// test, zda kořeny kvadratické rovnice jsou reálné
if (diskriminant()>=0) { ... }
```

Po úpravě kódu komentář není zapotřebí:

```
inline bool korenyJsouRealne() { return diskriminant()>=0; }
if (korenyJsouRealne()) { ... }
```

➢ Dobré poznámky, komentáře

- Na začátku souboru se zdrojovým kódem údaje o autorských právech nebo kdo je autorem.

```
// Copyright © 2007 Free Software Foundation
```

- Informativní poznámky.
  Například následující poznámka ukazuje, v jakém formátu je funkcí, která čte čas a datum, tento čas a datum rozpoznáván.

```
// Rozpoznávaný formát: hh:mm:ss dd.mm.rrrr
```

- Vysvětlení volby hodnoty konstanty.

```
// Při této hodnotě většinou není zřetelné zhoršení
// kvality obrazu. Pokud ano, je třeba hodnotu zvýšit.
kvalitaObrazu = 0.75;
```

- Vysvětlení významu komplikovanějšího kódu.

```
// c(s) = 127*s[0] + 31*s[1] + s[delka-1] + delka
c(s) = (s[0]<<7) - s[0] + (s[1]<<5) - s[1] + s[delka-1] + delka
```

- Někdy je uvedení příkladu, jaký je výsledek výpočtu, výstižnější než popis.

```
// Změní pořadí prvků v poli v tak, že všechny
// prvky < pivot budou před pivotem. Vrátí index posledního
// z prvků, které jsou před pivotem (nebo -1, není-li
// žádný).
template<class T>
int rozdeleni(T v[], int n, T pivot) { }
```

Místo toho uvedeme příklad.

```
// Příklad: výsledek rozdělení ([8 5 9 8 2],5,8) je
// [5 2 | 8 9 8] a funkce vrátí index 1
template<class T>
int rozdeleni(T v[], int n, T pivot) { }
```

- Obecné vysvětlení záměru.
  Autor kódu může například vysvětlit, proč použil právě uvedené řešení nebo způsob.

```
// Metoda třídění je přímým vkládáním,
// protože počet prvků je malý.
```

- Obecné vysvětlení významu.
  Při použití knihovních funkcí nebo kódu, který nemůžeme ovlivnit, může být účelné vysvětlit význam parametrů a návratových hodnot funkcí tam, kde je to nejasné.
- Varování pro jiné programátory před následky.

```
// Použití reference na objekt není bezpečné.
// Je nutné udělat kopii objektu.
objektKopie = objekt;
```

- Co bude zapotřebí udělat (TODO, FIXME).
  Například při psaní určité části kódu si uvědomíme, že budeme muset udělat přitom zásah kódu v jiném místě. Abychom na to později nezapomněli, umístíme na příslušné místo poznámku ve specificky označenou, kterou lze editorem snadno vyhledat, například:

```
// UDĚLAT ......
// OPRAVIT ......
```

- Zdůraznění něčeho, co by se mohlo zdát bezvýznamné.

```
// Odstranění případných mezer před a za jménem je nutné.
// Pokud by tam zůstaly, nebude osoba nalezena v seznamu.
... kód odstraňující mezery ...
```

- Komentář umožňující lépe pochopit důvody zvoleného řešení nebo proč by jiné řešení nebylo přínosem.

```
// Použitá heuristická metoda nenajde některá řešení.
// Metoda, která najde všechna řešení, by byla časově
// velmi náročná.
```

- Poznámky vysvětlující části kódu, u kterých lze očekávat, že čtenář se bude podivovat, proč zrovna je použit takový příkaz nebo jiný kód.

```
void zrusit() {
vector<float>().swap(data);
}
```

V předchozí funkci není zřejmé, proč pro vymazání vektoru nebyla použita funkce *clear*, která zruší data.

```
// Použitím funkce swap se nejen zruší data, ale i
```

```
// uvolní i nevyužitá paměť přidělená vektoru.
vector<float>().swap(data);
```

- Upozornění na nedostatky programu.

```
// Použitý algoritmus je pomalý.
// Je zapotřebí nahradit ho efektivnějším algoritmem.
```

➢ **Špatné poznámky a komentáře**

- Zbytečné poznámky.
  Jsou poznámky, které vysvětlují něco, co je zřejmé z kódu.

```
class A {
  A() { } // konstruktor
}
```

- Povinné poznámky.
  Je nesmyslné požadovat, aby před každou funkcí byl komentář, co funkce dělá, jaké jsou její parametry apod.
  Příkladem může být program *javadoc* pro automatické generování dokumentace k programu, který vyžaduje, aby ve zdrojovém kódu byly speciálně upravené komentáře. Tyto komentáře snižují přehlednost kódu.

```
/**
 *
 * @param nazev           Název CD
 * @param autor           Autor CD
 * @param početStop       Počet stop CD
 * @param delkaVMinutach Délka CD v minutách
 */
public void addCD(String nazev, String autor,
                          int pocetStop, int delkaVMinutach)
{
  CD cd = new CD();
  cd.nazev = nazev;
  cd.autor = autor;
  cd.stop = pocetStop;
  cd.delka = delkaVMinutach;
  cdSeznam.pridat(cd);
}
```

- Poznámky za uzavírajícími blokovými závorkami.
  Někteří programátoři umísťují na koncích bloku poznámku, k čemu se blok vztahuje.

```
try {
      while (...) {
        ....
      } // while
} // try
catch (...) {
    ...
```

```
} // catch
```

Tyto poznámky mají smysl jen v případě dlouhých bloků. Lepší je ale kód psát tak, aby v něm dlouhé bloky nebyly.

- Poznámky vysvětlující špatně zvolená jména funkcí nebo proměnných. Místo toho je účelnější tato jména změnit tak, aby jejich význam nebylo zapotřebí vysvětlovat v poznámce.

```
// Funkce zruší obsah tabulky,
// samotná tabulka ale zůstane zachována
zrusitTabulku()
```

Vhodnější název:

```
zrusitObsahTabulky()
```

- Udělat komentář z vyřazeného kódu.
  Není dobré při nahrazení nějaké části kódu novým z původního kódu učinit komentář.

```
// souradnice.x *= 2;
// souradnice.y *= 2;
```

Takové poznámky jednak snižují přehlednost kódu a navíc ten, kdo takový kód později čte, se může domnívat, že kód v komentáři je z nějakého důvodu důležitý a bude mít obavu komentář odstranit.