Programování v C++
První kroky

Karel Mozdřeň

29. října 2009

Obsah

1 Úvod

Tento text vznikl na přání čtenářů seriálu "cpp", který je k dispozici na webových stránkách "http://homel.vsb.cz/~moz017/cpp". Chtěl bych jim tímto poděkovat za impuls k jeho vytvoření.

Text je stále rozšiřován a upravován.

1.1 Pro koho je kniha určena

Kniha je vhodná jak pro začátečníky, tak pro mírně pokročilé programátory. Pokud se chcete něco naučit, nebo si jen něco připomenout, může Vám být tento text k užitku. Hlavním cílem je rychle, stručně a jasně osvětlit nejdůležitější aspekty jazyka C++.

1.2 Proč první kroky?

Tak jako se dítě učí chodit a mluvit, aby mohlo v životě obstát, tak i programátor začíná svojí cestu prvními kroky, kdy se učí programovací jazyk. Někteří se chodit a mluvit nenaučí pořádně nikdy. Přeji Vám proto, aby vaše chůze byla svižná a hlavně aby Vás bavila.

2 Začínáme

C++ je velice oblíbený programovací jazyk mezi programátory po celém světě. Může se sice na první pohled zdát, že již není konkurencí pro novější programovací jazyky, ale opak je pravdou. Pokud například chcete psát program, jenž má být bez větších problémů přenositelný mezi různými platformami, je volba tohoto jazyka vcelku logická. Dalším důvodem může být například to že jsou C i C++ normované jazyky a většina dalších jazyků je z nich odvozena. Dost bylo motivace a jde se na programování.

2.1 Hello, World!

Snad bych si ani nemohl odpustit vynechat kus kódu, který se vyskytuje již stereotypně v každém tutoriálu.

```
#include <iostream>

// takto se pisou jednoduche komentare na 1 radek

using namespace std;

/* takto se pisou komentare na
vice radku */

int main(){
  cout << "Hello, world!" << endl;
  return 0;
}
```

2.2 Rozbor kódu

Teď si probereme řádek po řádku. A dozvíme se co dělá co a proč.

```
#include <iostream>
```

Na tomto řádku se definují hlavičkové soubory, které budou použity při překladu. Tyto soubory obsahují funkce a konstanty, které v kódu můžeme dále použít. Tento hlavičkový soubor definuje funkce související se standardním vstupem a výstupem.

```
// takto se pisou jednoduche komentare na 1 radek

/* takto se pisou komentare na
vice radku */
```

Tímto způsobem jsou psány komentáře. To je taková oblast kódu, která není překládaná. Do kódu se zařazuje z důvodu přehlednosti. Měla by obsahovat informace, které pomáhají se rychle orientovat v kódu.

```
using namespace std;
```

Takto se definuje použití názvosloví. Názvosloví "std"se používá ke standardním objektům. Názvosloví si popíšeme později v samostatné sekci.

```
int main(){
  cout << "Hello, world!" << endl;
  return 0;
}
```

Funkce main() se spouští na začátku programu. Je to část kódu do které se píše hlavní část programu. Funkce si probereme taktéž později v samostatné sekci.

cout slouží jako tok dat na obrazovku. Znaky << určují směr přesměrování toku dat. V našem případě se přesměruje tok řetězce "Hello, World!"na obrazovku (Zobrazí se na obrazovce). Slovo "endl"slouží k ukončování řádku. Je to to samé jako stisknout v textovém editoru ENTER.

Return u funkcí vrací návratovou hodnotu. Každá funkce musí vracet nějakou hodnotu. Main by měl v případě že proběhl v pořádku vrátit 0.

2.3 Úkoly

1. Napište program, který by vypsal na obrazovku řetězec "Ahoj světe".
2. Upravte váš program tak, aby se slova "Ahoj"a "světe"zobrazily každé na jiném řádku.
3. Přepište program tak, aby fungoval jako vaše vizitka. Na každém řádku zobrazte jméno, příjmení, povolání, adresu a činnost, kterou se zabýváte.

3 Proměnné a jejich typy

Při programování potřebujeme v průběhu programu ukládat svá data, ať už pro další zpracování, nebo pro zobrazení uživateli. K tomu slouží proměnné. Při běhu programu si pojmenujeme proměnnou a ta se stane naším úložným prostorem. V jazyce C++ se musí proměnné nejen pojmenovat ale musí se i definovat jakého typu budou. Určit zda jde o číslo, řetězec nebo znak. K dispozici máme mnoho datových typů. Ukážeme si jen několik základních.

3.1 Zdrojový kód

```
#include <iostream>
#include <string>

using namespace std;

// globalni promenne...

int celecislo;
float desetinnecislo;
double presnejsidesetinnecislo;
char znak;
string retezec;

// hlavni funkce main

int main(){
  int lokalnicelecislo;
  lokalnicelecislo = 23;
  celecislo = 12;
  desetinnecislo = 0.2f;
  presnejsidesetinnecislo = 4.2;
  znak = 'A';
  retezec = "Ahoj svete";
  cout
    << lokalnicelecislo << " " << celecislo << endl
    << desetinnecislo << " " << retezec << endl;
  return 0;
}
```

3.2 Rozbor kódu

```
#include <iostream>
#include <string>
```

V části načítání hlavičkových souborů jsme přidali string, abychom mohli používat datový typ string, do kterého lze ukládat řetězce.

```
// globalni promenne...

int celecislo;
float desetinnecislo;
double presnejsidesetinnecislo;
char znak;
string retezec;
```

Tímto způsobem jsou vytvořeny proměnné, které lze následně jednoduše použít. Pokud definujeme proměnné na tomto místě budou globální. To znamená, že platí v celém kódu. Následující tabulka ukazuje k čemu se používá jaký typ.

int	celá čísla
float	desetinná čísla
double	desetinná čísla s větší přesností
char	1 znak
string	textový řetězec
bool	2 hodnoty true/false (platny/neplatny) pro podmínky

```
int main(){
  int lokalnicelecislo;
  lokalnicelecislo = 23;
  celecislo = 12;
  desetinnecislo = 0.2f;
  presnejsidesetinnecislo = 4.2;
  znak = 'A';
  retezec = "Ahoj svete";
  cout
    << lokalnicelecislo << " " << celecislo << endl
    << desetinnecislo << " " << retezec << endl;
  return 0;
}
```

Na začátku funkce main vidíme další proměnnou "lokalnicelecislo", která je uzavřena v bloku funkce main. Je tedy lokální a její platnost je pouze v tomto bloku.

Pokud chceme přiřadit proměnné hodnotu, použijeme přiřazovací operátor "=". Vždy je proměnná nalevo od operátoru a hodnota napravo. Naopak to nefunguje.

Proměnnou vypíšeme jednoduše na obrazovku tak, že ji přesměrujeme na standardní výstup pomocí "cout".

3.3 Úkoly

1. Napište program, který vypíše na obrazovku celé číslo 345, které si předem uložíte do proměnné.

2. Přidejte do programu proměnnou, která bude obsahovat řetězec "jablek"a vypište jej na obrazovku hned za číslem v předchozím úkolu.

3. Přepište program vizitka z předchozí lekce tak, aby všechny hodnoty byly první uložené v proměnné.

4 Pole a struktury

Většinou si při programování nevystačíme pouze s jednoduchými proměnnými. Často potřebujeme pracovat s množinou dat nebo strukturovanými daty. V této lekci si ukážeme jednoduché použití polí a struktur, které tyto problémy řeší.

4.1 Zdrojový kód

```
#include <iostream>
#include <string>

using namespace std;

// jednoducha struktura dat cloveka

typedef struct{
  string jmeno;
  string prijmeni;
  int vek;
} TClovek;

int main(){

  TClovek clovek;
  clovek.jmeno = "Karel";
  clovek.prijmeni = "Mozdren";
  clovek.vek = 22;

  cout << clovek.jmeno << " "
       << clovek.prijmeni << " "
       << clovek.vek << endl;

  // deklarujeme pole celych cisel o velikosti 3 prvku
  int pole[3];

  pole[0] = 1; // naplnime pole hodnotami
  pole[1] = 2;
  pole[2] = 3;

  cout << pole[0] << " "
       << pole[1] << " "
       << pole[2] << endl;

  return 0;

}
```

4.2 Rozbor kódu

```
typedef struct{
  string jmeno;
  string prijmeni;
  int vek;
} TClovek;
```

Toto je základní způsob definování struktury. Struktura není nic jiného než soubor proměnných, které mají nějakou logickou souvislost a lze je jako celek nějakým způsobem použít.

V tomto případě jsme vytvořili strukturu TClovek která popisuje, jaké základní vlastnosti může člověk mít. V našem případě jde o jméno, příjmení a věk.

```
TClovek clovek;
clovek.jmeno = "Karel";
clovek.prijmeni = "Mozdren";
clovek.vek = 22;
```

Toto je způsob jakým můžeme struktuře přiřadit data. K jednotlivým prvkům struktury přistupujeme pomocí ".".

```
int pole[3];
```

Tímto způsobem se vytváří pole. Do hranatých závorek se zapisuje, jak velké pole má být (kolik pojme prvků).

```
pole[0] = 1; // naplnime pole hodnotami
pole[1] = 2;
pole[2] = 3;
```

Takto můžeme pole naplnit hodnotami. Je důležité uvědomit si, že se prvky pole číslují od 0. Číslu v hranaté závorce se říká index. Při deklaraci určoval velikost pole. Pokud přistupujeme k jednotlivým prvkům slouží jako číselný odkaz.

4.3 Úkoly

1. Vytvořte pole celých čísel o velikosti 5 a naplňte jej hodnotami 5,4,3,2,1.
2. Přepište program vizitka z předchozích lekcí tak, aby data byla uložena ve struktuře TVizitka.
3. Vytvořte pole vizitek o velikosti 3 a naplňte je vámi vybranými daty.

5 Větvení a cykly

S tím co zatím umíme je sice možné udělat nějaké základní prográmky, ale není možné udělat složitější aplikace, které by již mohly být užitečné. Je třeba, aby se program dokázal řídit podle toho, jak se mění vlastnosti toho, s čím pracuje. S tím nám pomůžou jednoduché nástroje, které jsou v C++ dobře implementovány, jsou to podmínky a cykly. Dost řečí, přejdeme na příklad.

5.1 Zdrojový kód

```
#include <iostream>

using namespace std;

// bool slouží k určování stavů true/false
// (pravda/nepravda)

bool pravda = true;
bool nepravda = false;

int main(){

  // slovo "if" muzeme prekladat jako "pokud"

  if (pravda)
     cout << "tato podminka je splnena" << endl;
  if (nepravda)
     cout << "tato podminka neni splnena" << endl;

  if (1<2)
     cout << "tato podminka je splnena" << endl;
  if (1>2)
     cout << "tato podminka neni splnena" << endl;

  // slovo "for" budeme pokladat za "pocitadlo"

  for (int i=0; i<10; i++) cout << i << " ";
  cout << endl;

  // "do" cyklus s podminkou na zacatku

  int d = 9;

  do{
    d--;
    cout << d << " ";
```

```
  }while (d >= 0);
  cout << endl;

  // "while" cyklus s podminkou na konci

  int w = 0;

  while (w <= 10) {
    w++;
    cout << w << " ";
  }
  cout << endl;

  return 0;

}
```

5.2 Rozbor kódu

V této lekci poněkud netradičně nebudeme popisovat kód, který je napsán, ale popíšeme si syntaxi příkazů, které byly v příkladu použity. Předpokládám, že každý bude schopný po naučení se syntaxe přeložit co je napsáno sám.

5.2.1 Podmínky

Podmínku si můžeme představit jako dotaz na který je možné odpovědět pomocí slov "ano"a "ne"(logický výraz). Abychom mohli tvořit podmínky, bylo by dobré trochu se vyznat v množinových operacích a mít "selský rozum". K vytváření podmínek budeme potřebovat různé druhy operátorů. Důležitý přehled si uvedeme v tabulce.

$<, <=$	menší, menší nebo rovno
$>, >=$	větší, větší nebo rovno
$==, !=$	rovná se, nerovná se
!	logické NOT (!true == false)
&&	a zároveň (množinová operace průnik)
||	nebo (množinová operace sjednocení)

5.2.2 Větvení (if)

Větvením rozumíme směrování běhu programu na základě podmínek. K tomu slouží "if".

```
if (podmínka) {
      příkazy ...
} else if (podmínka) {
      příkazy ...
} else {
      příkazy ...
```

}

Syntaxe if začíná jednoduchým "if". pokud je podmínka splněna provedou se příkazy, které jsou v bloku kódu hned za ní. Blok kódu se ohraničuje složenými závorkami blok kódu . Pokud za if nenásleduje blok kódu, provede se pouze 1 příkaz hned za podmínkou. Tato část syntaxe je povinná. Následuje část nepovinná. Pokud chceme kontrolovat další podmínky, které se vylučují s podmínkou první můžeme navázat "else if (podmínka)". To je v překladu "v jiném případě, pokud ...". Nakonec přichází blok "else", ten se provede vždy, pokud všechny předešlé podmínky byly označené za neplatné.

Pokud deklarujete nějakou proměnnou uvnitř bloku , bude její platnost pouze v tomto bloku a v blocích do ní vnořených. Na konci bloku se proměnná strácí (dealokuje se).

5.2.3 Větvení (switch)

Větvení můžeme provádět i pomocí switch. Tady pomůže při popisu jenom příklad. popisování by mohlo být zdlouhavé a zavádějící.

```
int d=1;
switch (d) {
    case 0: cout << "0"; break;
    case 1: cout << "1"; break;
    case 2: cout << "2"; break;
}
```

Switch je v překladu přepínač. Postupně kontroluje pro jaké hodnoty (case "případ") se rovná a pokud najde tak spustí kód následující za ním. Je důležité si uvědomit, že se zde nepracuje s bloky. Pracuje se přímo se sadou příkazů následujících po case. Pokud tedy chceme provést jen nějakou část, můžeme předčasně ukončit blok pomocí "break"(zlomit, rozbít).

5.2.4 Cyklus s podmínkou na začátku (počítadlo for)

Počítadlo for se chová jako cyklus s podmínkou na začátku. To znamená, že se první ověří platnost podmínky a teprve poté se provedou příkazy. Pokud podmínka neplatí cyklus se ukončí. Poznámka: Pokud použijeme místo podmínky slovo true, vytvoříme nekonečný cyklus.

```
// for (deklarace pocitaci promenne;podminka;pocitani)
for (int i=0; i<10; i++) cout << i << " ";
```

Do počítadla se zadává počítací proměnná, které se nestaví počáteční hodnota. Za prvním středníkem se zadává podmínka do kdy má čítat. A za posledním středníkem operace která se má s čítací proměnou provádět v každém cyklu. Příkaz "i++"dělá to samé jako příkaz "i=i+1". Přičítá tedy k proměnné jedničku. Příkaz "i–"pracuje na stejném principu, akorát odečítá.

Kód k "for"zobrazí čísla 0 - 9 oddělené mezerami.

5.2.5 Cyklus s podmínkou na začátku (while)

Toto už není počítadlo. Je to strohý cyklus, který se řídí podmínkou, kterou kontroluje vždy na začátku cyklu. While je v překladu "dokud".

```
while ( podmínka ) {
      příkazy ...
}
```

5.2.6 Cyklus s podmínkou na konci (do while)

Do while je cyklus s podmínkou na konci. To znamená že se první provedou příkazy a až na konci cyklu se zkontroluje zda podmínka platí. To znamená, že se cyklus vždy provede alespoň jednou.

```
do {
      příkazy ...
} while ( podmínka );
```

5.3 Úkoly

1. Vypište na obrazovku 1000x "Ahoj".
2. Vytvořte pole celých čísel o velikosti 100 a naplňte jej hodnotami 0 - 99.
3. Ke každému číslu v poli přičtěte číslo 7 a výsledek zobrazte.
4. Pokuste se přepsat algoritmus tak, aby jste použily jiný cyklus než jste zatím používali.

6 Procedury a funkce

Proč psát pořád dokola ten samý kód, když můžu napsat kód na jedno místo a pak se na něj odvolávat? To nám umožňují procedury a funkce. Je to oblast kódu, kterou si napíšeme bokem a pojmenujeme si ji. Můžeme se poté na kód odkazovat pomocí tohoto jména.

```
#include <iostream>

using namespace std;

// funkce neco vraci

int secti(int a, int b){
   return a+b;
}

// procedura nic nevraci

void pozdrav(){
   cout << "ahoj" << endl;
}

// rekurzivni funkce vola sebe sama

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

int main(){

   pozdrav();
   cout << secti(1,1) << endl;
   cout << faktorial(5) << endl;
   return 0;

}
```

6.1 Operátory

Teď už je nevyhnutelné si prohlédnout další důležité operátory, se kterými budeme pracovat.

$+, -, *, /$	plus, mínus, krát, děleno
$+=, -=, *=, /=$	"$a+=b$"je to samé jako "$a=a+b$"
$<<, >>$	přesměrování toku vlevo, vpravo
&	logický operátor AND
|	logický operátor OR
~	logický operátor negace
$++, --$	přičte, odečte od proměnné 1

6.2 Funkce

Funkce je operace, u které podle daného vstupu požadujeme určitý výstup. Vstupem rozumíme parametry a výstupem návratovou hodnotu.

```
int secti(int a, int b){
    return a+b;
}
```

"int"na začátku udává typ hodnoty kterou funkce vrací. "secti"je název funkce, tímto budeme funkci volat. Za názvem funkce je závorka s parametry, které jsou vstupem pro funkci. Parametry jsou odděleny čárkou a každý má svůj vlastní typ. Tělo funkce je uzavřeno do bloku kódu. Funkce je ukončena klíčovým slovem "return", za kterou je návratová hodnota, kterou funkce vrací. Slovo return může být na jakémkoliv místě v těle funkce. Můžeme například vracet rozdílné hodnoty, podle větvení pomocí if.

Pokud deklarujete nějakou proměnnou uvnitř funkce, bude její platnost pouze v této funkci. Na konci funkce se proměnná ztrácí (dealokuje se).

6.3 Procedura

Procedura má stejnou konstrukci jako funkce, s tím rozdílem, že nevrací žádnou hodnotu (nemá return). Místo návratového typu má "void". Myslím že není třeba dále vysvětlovat.

```
void pozdrav(){
    cout << "ahoj" << endl;
}
```

6.4 Rekurzivní funkce

Rekurzivní funkce je taková funkce, která volá sama sebe. Důležitým prvkem takové funkce je ukončovací podmínka, která vrací nějakou hodnotu. Příkladem takové funkce je například faktoriál.

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

V této funkci předpokládáme že faktoriál čísla 1 je 1 a to bude také naše ukončovací podmínka. Faktorial je vlastně funkce jejíž výsledek je vynásobení vstupního čísla všemi čísly menšími než než je ono. Postupujeme tedy od čísla nejvyššího a násobíme ho vždy číslem o 1 menším, dokud se nedostaneme k číslu 1. To nám zařídí rekurzivní volání "return faktorial(cislo-1)*cislo;". Doslova v překladu vezmi aktuální číslo a vynásob jej faktoriálem čísla o 1 menším. Tak se bude postupně docházet až k číslu 1 a výsledná hodnota se vrátí z funkce.

6.5 Úkoly

1. Napište proceduru která na obrazovku vypíše 50x ahoj.
2. Napište funkci, která bude dělit. Ošetřete případ kdy se dělí nulou. V takovém případě vraťte hodnotu -1 a vypište upozornění: "Nulou nelze dělit".
3. Použijte strukturu vizitka z předchozích lekcí a napište proceduru, která bude mít v parametru tuto strukturu a bude vypisovat hodnoty vizitky.
4. Napište rekurzivní funkci, která bude číslo dělit dvěma dokud nebude číslo menší než 1.

7 Standardní vstup a výstup

Vstup a výstup jsou nedílnou součástí komunikace programu s uživatelem. Vstupy a výstupy, se kterými se budeme dnes setkávat by pro vás neměly být vyloženě nic nového. Přesto je třeba tuto část bedlivě prostudovat, neboť nás bude provázet po celou dobu.

Každý programátor si nejprve potřebuje otestovat funkčnost svého programu, než jej spojí s uživatelským prostředím. K tomu mu poslouží standardní vstupy a výstupy. Dnes nebudeme rozebírat příklad jeden, ale dva.

7.1 Příklad 1: konzole

Nejčastěji se komunikuje pomocí konzole. Klávesnice bude náš vstup a obrazovka náš výstup. Budou to 2 základní proudy cout(obrazovka) a cin(klávesnice).

```
#include <iostream>
#include <string>

using namespace std;

int main(){

    cout << "Zadej prosim sve jmeno: ";
    string jmeno;
    cin >> jmeno;
    cout << "Vase jmeno je " << jmeno << endl;
    return 0;

}
```

7.2 Příklad 2: soubory

Někdy je třeba, aby data byla uložena mimo zdrojový kód. C++ umožňuje pracovat se soubory velice jednoduše. K tomu nám složí datový proud "ifstream".

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main(){

    ofstream soubor1; // vystupni proud do souboru (zapis)
    soubor1.open("test.txt", ios::out);
    soubor1 << "Karel" << endl << "Mozdren" << endl << 22;
    soubor1.close();
```

```
    string jmeno, prijmeni;
    int vek;

    ifstream soubor2; // vstupni proud ze souboru (cteni)
    soubor2.open("test.txt", ios::in);
    soubor2 >> jmeno >> prijmeni >> vek;
    cout << jmeno << " " << prijmeni << " " << vek << endl;
    soubor2.close();

    return 0;

}
```

7.3 Popis programu

7.3.1 Konzole

```
    cout << "Zadej prosim sve jmeno: ";
    string jmeno;
    cin >> jmeno;
    cout << "Vase jmeno je " << jmeno << endl;
    return 0;
```

"cout"je proud standardního výstupu na obrazovku. přesměrování dat se provádí pomocí operátoru <<. Tento operátor si můžeme představit jako šipku, která ukazuje směr toku dat. U cout ukazuje na cout, data jdou tedy ven na obrazovku. U cin ukazuje šipka směrem do proměnné (>>), data jdou tedy z klávesnice do proměnné.

7.3.2 Soubory

```
#include <fstream>
```

"fstream"nám umožňuje používat datové proudy propojené se soubory.

```
    ofstream soubor1; // vystupni proud do souboru (zapis)
    soubor1.open("test.txt", ios::out);
    soubor1 << "Karel" << endl << "Mozdren" << endl << 22;
    soubor1.close();
```

"ofstream"je proud, který umožňuje výstup do souboru. Soubor se otevře pomocí metody open která má parametry jméno souboru a přepínač. V našem případě jsme použili přepínač ios::out, který říká, že chceme zapisovat do souboru. K souborovému proudu se chováme jako k cout, s tím rozdílem, že se data nezobrazí na obrazovce, ale zapíší se do souboru. Každý soubor je třeba po ukončení práce uzavřít to uděláme pomocí close.

```
string jmeno, prijmeni;
int vek;

ifstream soubor2; // vstupni proud ze souboru (cteni)
soubor2.open("test.txt", ios::in);
soubor2 >> jmeno >> prijmeni >> vek;
cout << jmeno << " " << prijmeni << " " << vek << endl;
soubor2.close();
```

"ifstream"Slouží jako proud ke čteni dat ze souboru. Funguje stejně jako cin, s tím rozdílem, že se data nenačítají z klávesnice, ale ze souboru. Pravidla jsou stejná: open otvírá (ios::in == cteni dat), close zavírá soubor.

7.4 Úkoly

1. Napište proceduru, která zapíše do souboru text "Ahoj".
2. Napište proceduru, která přečte ze souboru vytvořeného předchozím příkladem "Ahoj".
3. Napište program, který se zeptá na jméno, příjmení a věk a údaje uloží do souboru. Použijte při tvorbě tohoto programu struktury a procedury.
4. Rozšiřte tento program, tak aby se na začátku zeptal kolik lidí chce uživatel zadat. Podle počtu se bude dotazovat a bude postupně ukládat. Na konci programu se soubor přečte a data se vypíší na obrazovku. Testovat konec souboru lze pomocí eof() (eof == end of file). Například: if(soubor2.eof())

8 Paměť a ukazatelé

Do teď jsme nemuseli tušit, že vše v počítači systém reprezentuje jako oblast paměti, kterou disponuje. Každé zařízení je v systému reprezentováno jako rozsah z celé paměti a pokud k tomu rozsahu chceme přistupovat, musíme systém požádat o přidělení přístupu. Pokud nám je rozsah paměti přidělen systémem, můžeme s ním nakládat podle libosti. Jestli přistupujeme k paměti nám nepřidělené, pak systém ohlásí chybu a program skončí chybou. Dnes si budeme povídat o tom, jak si zažádat o paměť a jak k ní můžeme přistupovat, aby k chybám nedocházelo.

8.1 Pojmy

Nejprve si vysvětlíme základní pojmy.

- Adresa je číslo, které jednoznačně určuje pozici v celé paměti počítače. Značíme pomocí "&".
- Ukazatel můžeme chápat jako prst, který ukazuje na nějaké místo v paměti. Značíme pomocí "*".

8.2 Příklad

```
#include <iostream>
#include <string>

using namespace std;

int main(){

    int promenna;
    // staticke vytvoreni promenne cele cislo
    int *ukazatel;
    // vytvoreni ukazatele na cele cislo
    ukazatel = &promenna;
    // prirazeni ukazateli adresu na kterou ma ukazovat
    *ukazatel = 10;
    // nastaveni hodnoty na adrese na kterou ukazatel ukazuje

    cout << "Promenna s hodnotou " << promenna
         << " lezi na adrese " << &promenna << endl;
    cout << "Ukazatel s hodnotou " << *ukazatel
         << " ukazuje na adresu " << ukazatel << endl;

    ukazatel = new int;
    // prideleni (vytvoreni) noveho celeho cisla
```

```
    cout << "Ukazatel s hodnotou " << *ukazatel
         << " ukazuje na novou adresu " << ukazatel
         << endl;
    delete(ukazatel); // uvolneni z pameti

    return 0;

}
```

8.3 Statické přidělování paměti

```
    int promenna;
    // staticke vytvoreni promenne cele cislo
    int *ukazatel;
    // vytvoreni ukazatele na cele cislo
    ukazatel = &promenna;
    // prirazeni ukazateli adresu na kterou ma ukazovat
    *ukazatel = 10;
    // nastaveni hodnoty na adrese na kterou ukazatel ukazuje
```

Se statickým přidělováním paměti jsme se doposud setkávali pořád. Stačilo si zadat typ proměnné a dostali jsme ji. Nevýhodou je, že pokud si ji jednou vyžádáme, nemůžeme již měnit adresu, ale pouze hodnotu takové proměnné.

Je důležité si uvědomit přístupy s jakými můžeme přistupovat k proměnným a ukazatelům. Pokud chceme získat adresu statické proměnné, dáme před takovou proměnnou operátor &, který v tomto případě vystupuje jako operátor paměti. Pokud chceme získat hodnotu proměnné ponecháme ji bez operátoru. Hodnotou ukazatele je adresa na kterou ukazuje. Proto pokud nepoužijeme žádný operátor, získáme adresu na kterou ukazuje. Pokud použijeme "*"což je operátor ukazatele, pak se vrátí hodnota, která je uložena na adrese v paměti, na kterou ukazuje ukazatel. Hvězdičku můžeme chápat jako prst, který ukazuje na místo v paměti.

Ukazatel musí být stejného typu, jako proměnná v paměti na kterou ukazuje. Vyjímkou je obecný ukazatel *void, který může ukazovat na jakoukoli proměnnou. Pro získání hodnoty proměnné na kterou ukazuje void je pak ale třeba použít přetypování v podobě typu v závorce před ukazatelem.

```
    int *cele_cislo;
    void *obecny_ukazatel;
    cele_cislo = (int *)obecny_ukazatel;
```

8.4 Dynamické přidělování paměti

```
    ukazatel = new int;
    // prideleni (vytvoreni) noveho celeho cisla
    cout << "Ukazatel s hodnotou " << *ukazatel
         << " ukazuje na novou adresu " << ukazatel
         << endl;
    delete(ukazatel); // uvolneni z pameti
```

Dynamické přidělování paměti je pro nás nové, ale velice důležité. Přidělená dynamická paměť zůstává uložena v počítači po celou dobu jeho běhu. Zmizí až se počítač vypne. Je proto důležité si při práci s takovou pamětí dávat pozor, ať ji uvolníme, když už ji nepotřebujeme. Mohlo by dojít jednoduše k zaplnění paměti.

S dynamickou pamětí pracujeme pouze s ukazateli. Příkaz "new"přiděluje paměť a vrací ukazatel na paměť přidělenou. Velikost paměti se přiděluje podle požadovaného typu za slovem "new". Pokud paměť odstraňujeme(mažeme) použijeme proceduru delete(), jejímž parametrem je ukazatel.

8.5 Úkoly

1. Napište program, který dynamicky přidělí paměť celému číslu. Vložte do paměti hodnotu 124. Vypište adresu a hodnotu této proměnné.

2. Napište funkci, která bude dynamicky přidělovat paměť celému číslu a bude vracet jeho ukazatel.

3. Napište program, který dynamicky přidělí paměť strktůře vizitka z předchozích lekcí a naplňte ji hodnotami a hodnoty vypište na obrazovku. V dynamicky vytvořené struktuře se nepoužívá operátor "."ale operátor "->".

8.6 Nápověda

```
TClovek *clovek = new TClovek;
clovek->vek = 10;
cout << "Vek: " << clovek->vek << endl;
```

9 Hlavičkové soubory

Každý programátor se jedou dostane do stavu, kdy se jeho zdrojové kódy rozrostou do nekontrolovatelné velikosti a stanou se tak velice nepřehledné. Existují dvě možné řešení: Přestat programovat(smích), nebo rozdělit zdrojový kód do několika souborů. Zdrojový kód rozdělujeme do několika logických částí podle toho jaký mají k sobě vztah. Takovému přístupu můžeme říkat vrstvení kódu. Například můžeme rozdělit kód na části prezentační, aplikační a datové. Do skupiny prezentační zařadíme vše co pracuje se v stupem a výstupem(komunikaci s uživatelem). Do skupiny aplikační zařadíme programovou logiku. Poslední částí bude datová vrstva, která bude pracovat s daty. Každá z těchto částí bude dále rozdělena podle chrakterů funkcí a procedur. Často se také píšou třídy každá zvlášť do vlastního souboru. V takto rozděleném kódu se programátor bude dobře orientovat. Není nad to si ukázat příklad.

9.0.1 hlavicka.h

```
#ifndef _HLAVICKA_H
#define _HLAVICKA_H

#include <iostream>

using namespace std;

void Pozdrav(){
    cout << "ahoj" << endl;
}

#endif
```

9.0.2 main.cpp

```
#include <iostream>
#include "hlavicka.h"

using namespace std;

int main(){
    pozdrav();
    return 0;
}
```

9.1 Popis programu

9.1.1 Hlavičkový soubor

```
#ifndef _HLAVICKA_H
```

```
#define _HLAVICKA_H

    // tady bude kod hlavickoveho souboru

#endif
```

Tato část kódu čte překladač. Zajišťuje to načtení souboru pouze 1x. Stejně lze použít na začátku souboru *#pragmaonce*. Vice načtení by mohlo způsobit problémy. _HLAVICKA_H je název který si programátor většinou zapisuje podle názvu hlavičkového souboru. Předpokládám, že je jasné jakým způsobem byl zapsán.

Mezi tyto značky píšeme náš kód. Hlavičkový soubor by neměl obsahovat funkci main.

9.1.2 main.cpp

```
#include "hlavicka.h"
```

Pro načtení hlavičkového souboru zapíšete na začátku kódu další include, ale protože váš hlavičkový soubor nepatří do standardních, musíte jej uvodit uvozovkami. Pak vám budou k dispozici všechny funkce.

9.2 Úkoly

1. Napište funkci součin, která vynásobí 2 čísla z parametru. Funkce bude v odděleném souboru.
2. Vymyslete další 2 funkce které přidáte do hlavičkového souboru.
3. Vymyslete si 2 třídy, které budou v oddělených souborech (každá zvlášť) a napište program který je bude používat.

10 Třídy v C++

Třídy jsou hodně podobné strukturám které byly hlavním stavebním prvkem v jazyce C. C++ však nabízí práci s objekty. Takovýto druh programování se nazývá "Objektově orientované programování"(OOP). Dnes již nikdo neprogramuje programy, které by se neskládaly z tříd. Třídy popisují objekty, které se z nich dají vytvořit. Každá třída obsahuje Procedury, funkce a proměnné. Funkce popisují co lze s daným objektem dělat a proměnné slouží k popisu vlastnosti objektu. Procedurám a funkcím se v třídách říká metody.

```
#include <iostream>
#include <string>

using namespace std;

class Televize{

  public:
    void zapnout();
    void vypnout();
    void prepnout(int k);
    int getKanal();
    bool jeZapnuta();

  private:
    int kanal;
    bool zapnuta;
};

void Televize::zapnout(){
  kanal = 0;
  zapnuta = true;
}

void Televize::vypnout(){
  kanal = 0;
  zapnuta = false;
}

void Televize::prepnout(int k){
  kanal = k;
}

int Televize::getKanal(){
  return kanal;
}
```

```
bool Televize::jeZapnuta(){
  return zapnuta;
}

int main(int argc, char *argv[])
{

  Televize *telka = new Televize();

  telka->zapnout();
  telka->prepnout(10);

  if (telka->jeZapnuta()) cout << "Televize je zapnuta" << endl;
  cout << "Kanal: " << telka->getKanal() << endl;

  telka->vypnout();
  delete(telka);

  return 0;

}
```

10.1 Rozbor kódu

10.1.1 Tělo třídy

```
class Televize{

  public:
    void zapnout();
    void vypnout();
    void prepnout(int k);
    int getKanal();
    bool jeZapnuta();

  private:
    int kanal;
    bool zapnuta;

};
```

Třídu začínáme definovat pomoci klíčového slova class. Většinou se v třídě vypisují jen názvy a parametry metod. Implementace funkcí se dělá mimo třídu. To však neznamená, že by nešlo metody psát přímo do těla třídy.

Určitě jste si všimli slov private(osobní) a public(veřejný). Taková slova slouží k určení, jak se třída chová ke svému okolí. Private(osobní) určuje že zvenčí nebude daná část třídy viditelná. Public(veřejný) zase naopak zpřístupňuje vše.

10.1.2 Metody třídy

```
void Televize::prepnout(int k){
  kanal = k;
}
```

Metody ve třídách se od procedur a funkcí téměř neliší. Jediným rozdílem je názvosloví, které zapíšeme před jméno, aby se dalo poznat že daná metoda je metodou dané třídy. Pokud Píšeme metodu přímo do těla třídy, není třeba názvosloví zapisovat, neboť je již jasné že metoda do dané třídy patří.

10.1.3 Úkoly

1. Napište třídu Vizitka, která bude obsahovat metody pro zápis, čtení a výpis všech hodnot, které jsou pro danou třídu vhodné.
2. Zkuste napsat metody třídy, tak aby všechny její metody byly uvnitř těla třídy.
3. Napište třídu, která by popisovala Okno internetového prohlížeče.

10.2 Konstruktory a destruktory

Při programování tříd je vhodné provádět nějakou akci při vytvoření třídy a nějakou akci při jejím rušení. K tomu nám velice dobře poslouží konstruktory a destruktory. Představme si tedy, že chceme vytvořit objekt který se má vykreslovat na obrazovku. V takovém případě chceme aby měl souřadnice [x,y]. Pojmenujeme tedy tento objekt „Kresba“ .

10.2.1 Implementace

```
class Kresba{
        private:
                int x,y;
        public:
                Kresba();
                ~Kresba();
                void setX(int xn);
                void setY(int yn);
                int getX();
                int getY();
};
```

Zajímavým konstrukčním prvkem jsou metody které nevracejí žádné hodnoty(nemají žádný návratový typ ani void). Takto jednoduše označujeme konstruktory a destruktory. Kód konstruktoru(Kresba()) se provádí při vytvoření objektu z třídy a destruktor(~Kresba) se provádí před jejím zrušením. Do konstruktoru vkládáme takzvaný inicializační kód třídy, který nám nastaví základní vlastnosti které má třída mít(například počáteční pozice). U destruktoru se většinou jedná o uvolňování paměti(například volání delete() pro objekty použité v třídě).

Jako příklad můžeme implementovat konstruktor který nám zaručí, že počátek kresleného objektu bude ve středu obrazovky s rozlišením 800x600.

```
void Kresba::setX(int xn){
        x = xn;
}

void Kresba::setY(int yn){
        x = yn;
}

Kresba::Kresba(){
        setX(400);
        setY(300);
}
```

10.2.2 Úkoly

1 Vytvořte třídu zvíře, s vlastností počet nohou a k tomu příslušný konstruktor, který ji nastaví na hodnotu 4.

2 Upravte předchozí třídu Televize tak, aby po vytvoření objektu z ní byl nastaven kanál číslo 1.

10.3 Polymorfizmus

Stejně jako na kamarády můžeme volat Jardo, tak i na metody voláme jejich jménem. Ale v případě, že máme mezi kamarády více Jardů, tak musíme upřesnit kterého voláme, aby se neotočili všichni. Potřebujeme například Jardu elektrikáře a né Jardu vodoinstalatéra, protože se nám rozbila televize. Jak toho využijeme v c++? Jednoduše! Ponecháme stejný název metody a zadáme rozdílné parametry.

V předchozím příkladě jsme implementovali konstruktor, který při inicializaci objektu nastavil jeho pozici do středu obrazovky. My bychom však ještě chtěli mít možnost nastavit si při inicializaci pozici sami. Přidáme tedy konstruktor s parametry xn a yn. Po jeho zavolání se objekt posune na námi určené souřadnice.

10.3.1 Implementace třídy s polymorfizmem

```
class Kresba{
        private:
                int x,y;
        public:
                Kresba();
                Kresba(int xn, int yn);
                ~Kresba();
                void setX(int xn);
                void setY(int yn);
                int getX();
                int getY();
};

void Kresba::setX(int xn){
        x = xn;
}

void Kresba::setY(int yn){
        x = yn;
}

Kresba::Kresba(){
        setX(400);
        setY(300);
}

Kresba::Kresba(int xn, int yn){
        setX(xn);
        setY(yn);
}

int Kresba::getX(){
        return x;
}

int Kresba::getY(){
        return y;
}
```

10.3.2 Úkoly

1 Do třídy televize dodejte kromě konstruktoru, který po vytvoření objektu nastaví kanál 1 i konstruktor, který nastaví kanál z parametru.

10.4 Dědičnost

Tak jako se dědí vlastnosti z generace na generaci, tak se dědí i vlastnosti a metody ze třídy na třídu. Nejlépe se chápe dědičnost na rodinných vztazích: dědeček-otec-syn. Představme si tedy rodinu, kde se dědí řemeslo z otce na syna. Když dědeček pracoval jako truhlář, tak pro jeho syna(otce) je truhlařina také jasná volba. Stejně tak i vnuk bude pracovat v truhlařině. Pro všechny znamená slovo práce to samé.

Pokud si tedy vezmeme třídu Člověk mají všichni svoje jméno, věk a pohlaví a také všichni ví co znamená jít spát, jíst, jít do práce atd. . Pro každého člověka však znamená jít do práce něco jiného. Rodina truhlářů dědí základní vlastnosti člověka a sloveso jít do práce si přepisuje na jít dělat truhlařinu. Tak jsme si na jednoduchém příkladě ukázali jak vlastně ve skutečnosti vypadá dědičnost. Ještě se jí musíme naučit napsat v c++.

Již dříve jsme začali pracovat s třídou Kresba. Nyní si ji trochu rozšíříme a budeme z ní dědit další třídy. Úprava je v následujícím kódu.

10.4.1 Kresba.h

```
#ifndef _KRESBA_H_
#define _KRESBA_H_

class Kresba{
        private:
                int x,y;
        public:
                // konstruktory a destruktory
                Kresba();
                Kresba(int xn, int yn);
                ~Kresba();
                // setry a getry
                void setX(int xn);
                void setY(int yn);
                int getX();
                int getY();
                // dalsi metody
                void draw();
};

// nastavi souradnici X
void Kresba::setX(int xn){
        x = xn;
}

// nastavi souradnici Y
void Kresba::setY(int yn){
        x = yn;
}

// konstruktor (na stred)
Kresba::Kresba(){
        setX(400);
        setY(300);
}

// konstruktor (souradnice)
Kresba::Kresba(int xn, int yn){
        setX(xn);
        setY(yn);
}

// vrati souradnici X
int Kresba::getX(){
```

```
        return x;
}

// vrati souradnici Y
int Kresba::getY(){
        return y;
}
// vykresli objekt na akt. souradnicich
void Draw(){
}
#endif
```

Z této třídy bude dědit třída kružnice, která bude mít na rozdíl od předchozí třídy ještě vlastnost poloměr. Musíme ještě poznamenat, že dědičnosti jsou 3 druhy. (private, protected, public) Ty určují jakým způsobem budou přístupné metody třídy ze které dědíme. V tomto i většině jiných případů si vystačíme s public.

10.4.2 Kruznice.h

```
#ifndef _KRUZNICE_H_
#define _KRUZNICE_H_

class Kruznice: public Kresba{
        private:
                int r;
        public:
                Kruznice();
                Kruznice(int nx, int ny, int nr);
                ~Kruznice();
                void setR(int rn);
                int getR();
};

Kruznice::Kruznice(){
}

void Kruznice::setR(int rn){
        r = rn;
}

Kruznice::Kruznice(int nx, int ny, int nr){
        setX(nx);
        setY(ny);
        setR(nr);
}
```

```
int Kruznice::getR(){
        return r;
}

#endif
```

Jistě jste si všimli, že jsme už implementovali pouze části které jsou pro nás nové. Můžeme si takto ušetřit hodně psaní například v případech, kdy máme vytvořit pro několik objektů třídy, které mají hodně společných vlastností. Zapíšeme tak společné vlastnosti do jedné třídy a z té pak dědíme.

10.5 Virtuální metody(třídy) a abstrakce

Dalším krokem je abstrakce. To je způsob jakým popíšeme jak má skupina tříd vypadat a jaké metody má implementovat. Jde vlastně jenom o třídu, která nemá implementované své metody a má určený jenom jejich tvar(název a parametry). Z takové třídy nemůžeme přímo vytvořit objekty, ale můžeme z ní dědit. Takto zděděné třídy musí tedy mít tyto metody implementované a je jedno jakým způsobem. Příkladem toho může být zase naše třída Kresba, kde máme metodu Draw(). Pokud tuto metodu označíme slovem virtual bude možné z ní dědit a každý potomek třídy si bude moci kreslit podle svého algoritmu.

Přepíšeme si tedy naše třídy tak, jak mají správně být napsány. Často se virtuálním třídám(třídy se samými virtuálními metodami) říká interface(rozhraní). Proto také názvy těchto tříd uvozuji písmenem I.

10.5.1 main.cpp

```
#include<iostream>
using namespace std;

class IKresba{
      public:
             virtual void setX(int xn) = 0 ;
             virtual void setY(int yn) = 0 ;
             virtual int getX() = 0 ;
             virtual int getY() = 0 ;
             virtual void Draw() = 0 ;
};

class Kresba : public IKresba{
      private :
              int x, y;
      public :
              void setX(int xn );
              void setY(int yn );
              int getX();
              int getY();
};

void Kresba::setX(int xn){
      x = xn;
}

void Kresba::setY(int yn){
      y = yn;
}

int Kresba::getX(){
    return x;
}

int Kresba::getY(){
     return y;
}

class Kruznice : public Kresba{
       private :
               int r;
       public :
               Kruznice(int xn, int yn, int rn);
               void setR(int rn);
```

```
                int getR();
                void Draw();
};

void Kruznice::setR(int rn){
      r = rn;
}

int Kruznice::getR(){
     return r;
}

Kruznice::Kruznice(int xn, int yn, int rn){
     setX(xn);
     setY(yn);
     setR(rn);
}

void Kruznice::Draw(){
     cout << "kreslim kruznici r=" << getR ( )
          << " S=[" << getX ( ) << "," << getY ( )
          << "] " << endl ;
}

int main ( ) {
    IKresba *kruznice = new Kruznice( 10 , 10 , 5 ) ;
    kruznice->setX(20);
    kruznice->Draw();
    system("PAUSE");
    return 0 ;
}
```

Tento kód byl poněkud delší a může se tak stát pro někoho méně pochopitelný. Přitom je zcela jednoduché jej pochopit. Nejprve jsme si vytvořili rozhraní IKresba která definuje 4 metody pro čtení a zápis souřadnic a jednu pro vykreslení objektu. Nemá nic implementovat, pouze říká jak se dá přistupovat k třídám které z ní dědí. Třída která dědí z IKresba je Kresba. Ta implementuje základní metody čtení a zápisu souřadnic. Poslední je Kružnice která dědí z Kresba a doplňuje se o poloměr r.

V mainu můžeme sledovat vytvoření Kružnice, pro kterou je nastaveno rozhraní IKresba. Můžeme tak použít všechny metody které IKresba definuje, ale v paměti bude stále uložena jako kružnice.

Výhodou takového zápisu tříd je například ukládání do pole. Představíme si že máme obrázek, který se skládá z jednoduchých objektů jako jsou čtverce, kružnice, přímky, oblouky a podobně. Všechny implementují rozhraní IKresba. Vytvoříme si tedy pole ukazatelů na IKresba a nazveme ho obrázek. Potom když

budeme chtít celý obrázek vykreslit stačí nám v poli zavolat v cyklu na všechny objekty metodu Draw().

```
int main ( ) {
    IKresba *pole[10];
    for (int i=0; i<10; i++){
        pole[i] = new Kruznice(i,i,i);
    }
    for (int i=0; i<10; i++){
        pole[i]->Draw();
    }
    return 0 ;
}
```

Nelekněte se prosim konstrukce IKresba *pole[10];. jde pouze o vytvoření pole ukazatelů o velikosti 10 typu IKresba.

10.5.2 Úkoly

1 Doplňte kód o třídu Čtverec s parametry x, y, a, kde a je strana čtverce.

2 Doplňte kód o třídu Obdélník s parametry x, y, a, b, kde a, b jsou strany obdélníku.

3 V main vytvořte pole ukazatelů o velikosti 3, kde vložíte Kružnici, čtverec a obdélník. Všechny je pak „vykreslete“ do konzole.

11 Přetěžování operátorů

11.1 Motivace

Taky někdy přemýšlíte, že by bylo fajn, kdyby váš počítač řešil některé matematické příklady za vás? Ano můžete si koupit profesionální matematicky software, ale taky si můžete napsat vlastní. Ne teď vážně. Někdy se jako programátoři dostanete do situací kdy se může hodit pro nějaké třídy použít operátory. Například programujete složitý 3D software který potom bude využívat nějaká strojírenská firma. Za takovým programem je schováno mnoho matematiky. Hned na začátku víte, že budete potřebovat třídy pro výpočet s vektory a maticemi. Jistě již máte alespoň malou představu jak by jste vytvořili třídu Matice a Vektor (pokud nevíte co je vektor nebo matice tak hledejte na internetu). Realizovat operace mezi těmito třídami můžeme buď tak, že vytvoříme v každé metodu například vynasobVektorem(Vektor v), nebo přetížíme operátory násobení sčítání a odčítaní pro dané třídy.

11.2 Jak přetížit operátor

Bohužel nejdou přetížit všechny operátory, ale většina, kterou zpravidla je potřeba přetížit, jde. Uvedu seznam těch, které přetížit nelze:

..* ::? :

Operace přetížení není nijak náročná. Ukážeme si jí na příkladě třídy Vektor3D.

11.2.1 Vektor3D.h

Abychom zde nezapisovali zbytečně dlouhý kód dám zde pouze definici třídy a vysvětlím na ní přetížení operátorů.

```
class Vektor3D{
      private:
        float x,y,z;
      public:
        Vektor3D();
        Vektor3D(float xn, float yn, float zn);
        void setVektor3D(float xn, float yn, float zn);
        void normalize();
        void setX(float xn);
        void setY(float yn);
        void setZ(float zn);
        float getX();
        float getY();
        float getZ();
        float getLength();
        // pretizene operatory
        Vektor3D operator +(Vektor3D v);
        Vektor3D operator -(Vektor3D v);
        Vektor3D operator *(Vektor3D v);
        float operator &(Vektor3D v);
};
```

Jak jste si mohli všimnout, obsahuje třída mnoho metod, u kterých sice neukážu implementaci, ale přesto by jste měli být schopní je dopsat sami. Pro nás jsou teď hlavní poslední 4 metody operátor jejich tvar je celkem zřejmý.

```
[NavratovyTyp] operator [operator](Parametr p);
```

Návratový typ je určením výstupu z dané operace(součet dvou čísel je zase číslo). Klíčové slovo operátor udává že chceme přetížit nějaký operátor za čímž přímo zapisujeme co chceme přetížit. V parametru se udává typ operandu na pravé straně. Operand na levé straně operátoru je přímo třída, ve které se přetěžovací metoda nachází.

```
Vektor3D Vektor3D::operator +(Vektor3D v){
	Vektor3D ret;
	ret.setX(this->getX() + v.getX());
	ret.setY(this->getY() + v.getY());
	ret.setZ(this->getZ() + v.getZ());
	return ret;
}

Vektor3D Vektor3D::operator -(Vektor3D v){
	Vektor3D ret;
	ret.setX(this->getX() - v.getX());
	ret.setY(this->getY() - v.getY());
	ret.setZ(this->getZ() - v.getZ());
	return ret;
}

Vektor3D Vektor3D::operator *(Vektor3D v){
	Vektor3D ret;
	ret.setX(this->getY()*v.getZ() - this->getZ()*v.getY());
	ret.setY(this->getZ()*v.getX() - this->getX()*v.getZ());
	ret.setZ(this->getX()*v.getY() - this->getY()*v.getX());
	return ret;
}

float Vektor3D::operator &(Vektor3D v){
	return
	this->getX()*v.getX() +
	this->getY()*v.getY() +
	this->getZ()*v.getZ();
}

Vektor3D operator *(Vektor3D v, float f){
	Vektor3D ret;
	ret.setX(v.getX()*f);
	ret.setY(v.getY()*f);
	ret.setZ(v.getZ()*f);
	return ret;
}

Vektor3D operator *(float f, Vektor3D v){
	Vektor3D ret;
	ret.setX(v.getX()*f);
	ret.setY(v.getY()*f);
	ret.setZ(v.getZ()*f);
	return ret;
}
```

```
ostream & operator <<(ostream &out, Vektor3D v){
        out << "["
        << v.getX() << ","
        << v.getY() << ","
        << v.getZ()
        << "]";
        return out;
}
```

V implementaci přetěžování jste si mohli všimnout, že se zpravidla nemění data uvnitř třídy. Nejčastěji se vytváří pomocná proměnná, do které si výsledek výpočtu uložíme a vrátíme jej. Další zvláštností budou 3 metody, které přímo nepatří do třídy a přesto s ní souvisí. První dvě jsou skalární násobení číslem z desetinnou čárkou zleva i zprava a poslední je definování posílání objektu do proudu(výpis na obrazovku nebo do souboru). Protože se zde nepracuje přímo s třídou, je třeba definovat v parametru typy levého i pravého operandu operátoru. Na pořadí parametrů záleží(levý, pravý operand).

Přetěžování výstupního proudu je vyjímkou oproti ostatním. Zde se na výstupní proud získá reference, zapíše se co je potřeba a reference se posílá returnem dále.

11.3 Úkoly

1. Zkuste doplnit všechny metody třídy. Může se Vám někdy hodit.
2. Pokuste se vytvořit podobnou třídu pro matice.

12 Generické programování

Každý ví co je nákupní taška. Když se jde nakupovat, tak se u větších nákupů bere více tašek a každá se používá na jiný druh zboží. Do jedné si dáte zeleninu do druhé mražené potraviny a tak podobně. Představte si, že máte naprogramovat třídu taška do které můžete vkládat potraviny. Budete v ní mít metody jako vlož do tašky, vyber z tašky a jiné užitečné metody. A však pro každý druh nákupů budete muset implementovat zvláštní třídu.

Teď abychom to trochu přiblížili, řekněme že máme třídu, která reprezentuje nějakou datovou strukturu do které se ukládají data. Musíme pro různý druh dat vytvářet zvlášť třídy. Například třídu, která pracuje s *int* a jinou třídu, která pracuje s *float*. Mnohem jednodušší by bylo vytvořit šablonu kde bychom pouze zaměňovali datové typy. Přesně to nám umožňuje generické programování.

12.1 vector

Datová struktura vector je přímo součástí jazyka c++ a umožňuje použití genericity. Vector je vlastně takové dynamické pole do kterého můžete data vkládat a vybírat několika různými způsoby. Na příkladu si ukážeme jak lze vytvořit instanci vectoru pro různé datové typy.

```
#include <cstdlib>
#include <iostream>
#include <vector>

using namespace std;

int main(int argc, char *argv[])
{
    vector<int> v1;
    vector<float> v2;
    for (int i=0; i<10; i++){
        v1.insert(v1.end(), i);
        v2.insert(v2.end(), (float)i/10.0f);
    }
    cout << "v1 \t v2" << endl;
    for (int i=0; i<10; i++){
        cout << v1[i] << " \t " << v2[i] << endl;
    }
    v1.clear();
    v2.clear();
    return 0;
}
```

První ze dvou vektorů je inicializován pro práci s datovým typem int a druhý s float. Jakmile jednou určíte datový typ nelze jej v již dále v kódu změnit.

```
vector<[Datový typ]> [Název];
```

Nemusíte mu zadávat pouze základní datové typy. Můžete mu přiřadit jakýkoliv, třeba i třídu. V jedné z předchozích kapitol jsme vytvářeli virtuální třídu IKresba a dědili z ní několik dalších tříd abychom je mohli uložit všechny do jednoho pole. Pokud však stejný příklad vytvoříme s vectorem může kresba obsahovat „neomezený“ počet objektů (omezení pamětí počítače).

```
vector<IKresba> obrazek;
```

12.1.1 Vkládání do a odebírání dat z vektoru

Pro vkládání dat používáme 2 možnosti. Buďto pomocí metody push_back(Typ data); nebo insert(iterator, Typ data); . První z metod se používá pokud chceme k vektoru přistupovat jako k datovému typů zásobník(FILO) a druhý pokud k němu chceme přistupovat jinak. Oba způsoby lze libovolně kombinovat. Pro vybírání a vkládání dat používáme iterátor kde se odkazujeme metodou begin() nebo end(), které ukazují na začátek a na konec vektoru. Pro výběr dat můžeme taktéž použít přístup jako k normálnímu poli jak je vidět v předchozím kódu.

```
#include <cstdlib>
#include <iostream>
#include <vector>

using namespace std;

int main(int argc, char *argv[])
{
    vector<int> v1;
    vector<float> v2;
    for (int i=0; i<10; i++){
        v1.push_back(i);
        v2.push_back((float)i/10.0f);
    }
    cout << "v1 \t v2" << endl;
    for (int i=0; i<10; i++){
        cout << v1.back() << " \t " << v2.back() << endl;
        v1.pop_back();
        v2.pop_back();
    }
    system("PAUSE");
    return 0;
}
```

12.2 Generické metody

V c++ se genericita vytváří pomoci takzvaných šablon(templates). Obecně lze říct, že označíme místa v kódu, která se mají nahradit naším daným datovým typem. Jistě si dokážete představit metodu, která zaměňuje dvě čísla. Standardně bychom museli napsat pro každý datový typ jednu metodu, ale my můžeme použít šablonu a ušetřit si psaní.

```
template <class T>
void swapt(T &a, T &b){
        T pomocny = a;
        a = b;
        b = pomocny;
}
```

Všimněte si klíčového slova template(šablona), takto vytvoříme obecný typ T, který můžeme dále použít v metodě jako standardní typ. Pak pouze metodu voláme a ta se automaticky přetypuje na potřebný typ.

Ampersanty(&) v parametrech metody nám zaručují, že pracujeme s proměnnými které jsou předány. Tedy jestliže provedeme změnu hodnoty parametru v metodě, projeví se to i v proměnné, kterou jsme jako parametr předali.

```
int main(){
        int a=1, b=2;
        double c=3, d=4;
        swapt(a,b);
        swapt(c,d);
        cout << "␣a=" << a << ",␣b=" << b << endl;
        cout << "␣c=" << a << ",␣d=" << b << endl;
}
```

12.3 Generické třídy

Stejně jako generické metody můžeme vytvářet celé generické třídy. Můžeme si představit, že máme vyřešenou třídu pro výpočty s vektory. Třídu vektor máme však vyřešenu pouze pro data typu float. Pokud použijeme šablony, můžeme zapsat třídu natolik obecnou, že jednoduše vytvoříme například vektor z komplexních čísel.

Pro ukázku napíšu malou třídu kterou posléze přepíšu na třídu generickou pomocí šablon.

```
class Test{
        public:
                int a;

                Test(){
                }

                Test(int an){
                        this->a = an;
                }
};
```

Teď třídu přepíšeme pomocí šablon.

```
template <class T>
class Test{
        public:
                T a;

                Test(){
                }

                Test(T an){
                        this->a = an;
                }
};
```

Potom můžeme používat třídu takto:

```
Test<int> a(10);
Test<double> b(0.5);
cout << a.a << endl;
cout << b.a << endl;
```

Za název třídy zapisujeme datový typ, kterým chceme v šabloně nahrazovat.

13 Preprocesor

Preprocesor je část překladače, která se provádí jako první před vlastním překladem. Umožňuje nám programátorům zjednodušit práci při tvorbě programu. Stará se o spojování souborů, řeší podmíněný překlad, makra a další užitečné nástroje a konstrukce o kterých si něco povíme.

13.1 Direktivy

Rovnou si ukážeme jaké direktivy máme k dispozici a jak se používají.

13.1.1 #define

Define je velice užitečná direktiva. Umožňuje nám definovat vlastní výrazy. Například si můžeme nadefinovat Pi

```
#define PI 3.14159265
```

To funguje tak, že preprocesor nahrazuje všechny výskyty PI v kódu definovaným textem. Až poté vše zkompiluje. Můžeme také define použít k definování makra. Makro definujeme stejně jako definici, ale navíc používáme parametry které se do výsledného přepisu kódu zahrnou.

```
#define MOCNINA2(x) ((x)*(x))
```

V tomto příkladu překladač postupuje stejně. Nahrazuje výskyty textu MOCNINA2 výrazem s tím, že do něj doplní parametry. ve výsledku pak můžeme pro druhou mocninu zapsat v kódu takto:

```
int x = MOCNINA2(2);
```

Což preprocesor přepíše do tvaru:

```
int x = ((2)*(2));
```

Pro víceřádková makra, vložíme na konec řádku zpětné lomítko. Například chceme-li makro na vypsání určitého počtu hvězdiček můžeme postupovat stejně jako v následujícím příkladu.

```
#define HVEZDICKY(pocet) \
for (int h_poc = 0; h_poc < pocet; h_poc++){ \
        cout << "*"; \
}
```

Pokud by snad mela být nějaká definice zrušena, stačí nám použít direktivu #undef.

13.1.2 #ifdef, #ifndef a #endif

Direktivy #ifdef, #ifndef a #endif se používají na podmíněný překlad. Toho můžeme například využít pokud programujeme nějakou knihovnu pro více operačních systémů. Například pokud budeme kompilovat pro linux, tak budeme kontrolovat zda je definován OS_LINUX a pokud chceme zase kompilovat pro OS_WINDOWS.

```
#define OS_LINUX
//#define OS_WINDOWS

#ifdef OS_LINUX
        #include "funkce_linux.h"
#endif

#ifdef OS_WINDOWS
        #include "funkce_windows.h"
#endif
```

#ifndef je jenom logickým opakem #ifdef, tedy pokud není definováno.

13.2 Operátory preprocesoru

Preprocesor má definované i některé operátory, které se nám mohou hodit například při definování maker.

13.2.1 Operátor

Tento operátor uzavírá definici do uvozovek, viz příklad:

```
#define VYPIS(x) cout << #x
VYPIS(ahoj)
```

Preprocesor přepíše vyvolané makro na:

```
cout << "ahoj";
```

13.2.2 Operátor #@

Tento operátor uzavírá definici do jednoduchých uvozovek, viz příklad:

```
#define VYPIS(x) cout << #@x
VYPIS(a)
```

Preprocesor přepíše vyvolané makro na:

```
cout << 'a';
```

13.3 Pragmata

Pragmata jsou rozšíření preprocesorů, které umožňují různé funkcionality a nastavení preprocesoru a kompilátoru. Jsou povětšinou systémově a hardwareově závyslá.

Pro nás je nejdůležitější #pragma once, které dáváme na začátek hlavičkového souboru. Což zaručí, že se hlavičkový soubor nahraje pouze jednou. Nahrazujeme tak zápis, který jsme používali dříve.

```
#ifndef _SOUBOR_H_
#define _SOUBOR_H_

// kod hlavickoveho souboru

#endif
```

To jednoduše nahradíme zápisem

```
#pragma once

// kod hlavickoveho souboru
```

14 Nejčastější chyby

Pokud programujete může se vám stát, že narazíte na chyby, které se nehlásí rozumnou chybovou hláškou. Napíšu zde ty, na které si teď vzpomenu. Pokud přijdete na další takové zvláštní chyby, tak mi je prosím pošlete a já je do textu zařadím.

- V podmínkách nezapomínejte, že musíte při porovnávání použít operátor == místo =. Pokud použijete pouze jedno =, bude se to brát jako přiřazení a výsledkem bude vždy true pokud bude výsledek různý od nuly nebo nebude NULL.
- Při psaní tříd nesmíte zapomenout dát na konec definice středník. Chyba se může hlásit zcela na jiném místě.
- Pokud pracujete v nějakém IDE, může se stát, že při spuštění konzolové aplikace přímo z tohoto IDE, neuvidíte výsledek, protože než okno ve kterém proběhl program se již zavřelo. V takovém případě na konec programu můžete vložit instrukci která bude požadovat vstup uživatele.

15 Závěrem

Doufám, že vám text poslouží dobře jako startovní můstek v programování v C++. Pokud vám něco chybí, nebo chtěli upravit. Ozvěte se mi na email "mozdren@seznam.cz".

S pozdravem Karel Mozdřeň