Škola pre Mimoriadne Nadané Deti a Gymnázium, Teplická 7, 831 02 Bratislava

Anino BELAN

KURZ JAZYKA C++

učebný text pre sextu osemročného gymnázia

BRATISLAVA
2005

Obsah

Úvod....4
Objektové programovanie alebo "Ži a nechaj žiť"....5
Trieda Kruh alebo "Nech žije ping-pong"....8
KDevelop alebo "Predsa len Hello, world!"....13
Príkazy new a delete alebo "Dynamická alokácia po novom"....16
Dedičnosť alebo "Aký otec, taký syn (ledaže by nie)"....25
Zákernosti so smerníkmi alebo "Na čo všetko treba dať pozor"....33
Preťažovanie operátorov alebo "Čo všetko si môžete spraviť po svojom"....39
Výnimky alebo "Robme chyby profesionálne"....42
Šablóny alebo "Hyper makro systém"....47
Úvod do STL alebo "Nevymýšľajte koleso"....50
Ďalšie kontajnery alebo "Nie je smetiak, ako smetiak"....55

Úvod

Tento kurz jazyka C++ vznikol ako učebný text pre Školu pre Mimoriadne Nadané Deti a Gymnázium v Bratislave. Nadväzuje na kurz jazyka C a kurz grafiky v jazyku C, takže pri jeho štúdiu je nutné poznať aspoň základy syntaxe jazyka C a práce s knižnicou Allegro.

Podobne, ako v predošlých kurzoch sa nejedná o úplný popis jazyka, ani o referenčnú príručku. Išlo o to, aby boli na príkladoch ilustrované základné črty jazyka C++, metódy tvorby tried a niektoré techniky, ktoré sa pri používaní jazyka C++ používajú. Aby tí, čo sa cez kurz budú predierať aspoň približne začali tušiť, aké sú výhody použitia objektov a objektového prístupu k programovaniu.

Súčasne bolo cieľom pokryť všetky základné konštrukcie, ktoré sa v C++ používajú, aby ten, kto kurz absolvuje, vedel čítať cudzie programy a mohol sa pustiť do štúdia niektorej z rozsiahlych knižníc na tvorbu používateľského rozhrania (či už Microsoft Foudation Class Library, alebo Qt a KDE knižníc používaných pod OS Linux).

Všetkým, ktorí sa jazykom C++ chcú zaoberať viac do hĺbky, odporúčame hľadať si ďalšiu literatúru. Zvlášť kurz, ktorý na univerzite v holandskom Groningene zostavil a prednáša pán Frank B. Brokken. Kurz je k dispozícii na adrese *http://www.icce.rug.nl/documents/*.

Všetkým, kto tento kurz budú čítať a učiť sa z neho jazyk C++ prajem veľa radosti z programovania.

Anino Belan

1. lekcia
Objektové programovanie alebo "Ži a nechaj žiť"

O AT&T Bellových laboratóriách sme už hovorili. Áno, sú to presne tie tie, v ktorých páni Kernighan a Ritchie vymysleli jazyk C a v ktorých vznikol UNIX. Toto všetko sa udialo začiatkom sedemdesiatych rokov dvadsiateho storočia.

Jazyk C je klasický procedurálny jazyk – keď v ňom programujete nejakú veľkú úlohu, rozdelíte si ju na menšie, každú z nich na ešte menšie, pre každú z nich si spravíte funkciu, zlepíte to dohromady a ak tam niekde nemáte chybu[1], tak to bude fungovať. Týmto spôsobom sa písali programy veľmi dlho a menšie úlohy sa tak často riešia dodnes.

Lenže... Predstavte si, že programujete nejaký väčší projekt – napríklad hru. Po obrazovke vám má pobiehať zodvadsať avatarov[2] asi piatich rôznych typov a vy píšete jednu veľkú procedúru, ktorá vám má zaručiť, že sa to všetko bude pohybovať správne, že sa nejako vyriešia kolízie medzi zúčastnenými postavami a že zastrelené potvory či už diskrétne, alebo s veľkým randálom zmiznú. Mnohý programátor, ktorý sa ocitol v tejto situácii (a v mnohých podobných) zatúžil po tom, aby túto veľkú činnosť nemusel rozbíjať do množstva malých procedúr, ale aby si mohol každý typ avatara naprogramovať pekne samostatne a z hlavnej procedúry už len posielať jednotlivým postavám správu „pohni sa“ a ony sa už o seba postarajú. A to je presne ten moment, kedy programátor začína chcieť opustiť staré zaužívané cestičky procedurálneho programovania a vytvárať objekty.

A tak začali vznikať jazyky, ktoré boli priamo navrhnuté tak, aby sa v nich objekty vytvárali dobre. Mnohé boli krásne a prepracované (spomeňme napríklad Smalltalk od firmy Xerox). Ale... Mnohým programátorom boli objektové črty v týchto jazykoch sympatické a radi by ich používali. Ale mali radi jazyk C a nechcelo sa im zriekať rýchlosti a efektívnosti kódu, ktorú s ním mohli dosiahnuť. A tak Bjarne Stroustrup – opäť z AT&T Bell Laboratories – začiatkom osemdesiatych rokov dvadsiateho storočia vyvinul C++. Nechal sa inšpirovať jazykom Simula 67[3]. Spravil prekladač, ktorý kód napísaný v C++ preložil do jazyka C a tento sa ďalej kompiloval ako obyčajný Cčkový program. V roku 1983 sa C++ rozšíril aj mimo AT&T a v roku 1988 vyšla z AT&T druhá verzia C++, ktorá sa stala základom ANSI štandardu jazyka.

Jazyk C++ je hybridný jazyk. Pretože je rozšírením jazyka C, zachoval si aj jeho procedurálnu stránku. Takže ak niekto chce, môže používať niektoré jeho výhody a programovať klasicky procedurálne. Jazyk C++ ale umožňuje aj vytvárať objekty, a celý návrh vášho projektu postaviť objektovo. Aké to má výhody, uvidíte neskôr.

Kým sa pustíme do programovania (a to sa, žiaľ, v tejto lekcii ešte nepustíme) treba povedať, čo znamenajú niektoré slová, ktoré budeme používať. Asi by sa patrilo začať slovom **objekt**. Objekt

1 Zaručene máte...

2 Avatar je akákoľvek hýbúca sa postavička či predmet v hre.

3 Simula 67 bol historicky prvý objektovo orientovaný jazyk.

je voľačo, čo má svoju vlastnú identitu (vieme to rozlíšiť od ostatných vecí), svoj vlastný stav a svoje vlastnosti (vie sa to nejako správať ku svojmu okoliu). Premenná `i` typu `int` je z tohto hľadiska tiež objekt. Vieme ju rozoznať od ostatných premenných (napríklad od premennej `j` typu `int`). Jej stav je to, čo je v nej práve uložené (napríklad 517). A aj reakcia s okolím aká-taká je (vieme do nej niečo priradiť a zas z nej vybrať).

Premenná typu `int` je samozrejme veľmi jednoduchý objekt. Objekty, ktoré budeme vytvárať, budú často oveľa komplikovanejšie. Budú sa skladať z mnohých vecí, ktoré budú mať vnútornú logiku a budú poprepájané rôznymi spôsobmi. Preto by sme nechceli, aby sa stav objektu menil nejak náhodne a chaoticky. Preto je dobré dodržať túto zásadu: Urobíme si funkcie, s pomocou ktorých môže okolitý svet objektu dať vedieť, čo od neho chce.[4] Všetko ostatné sa rieši výhradne vo vnútri objektu. **Objekt sa správa ako čierna skrinka**. Iba on si môže meniť stav svojich premenných. Okolitý svet ich môže čítať alebo meniť iba s pomocou funkcií.

Ak si všimnete, tento prístup bol použitý aj pri knižniciach, ktoré sme používali v Cčku. Napríklad k takému súboru sme pristupovali iba s pomocou funkcií (`fopen`, `fclose`, `fprintf`, …) Nemali sme ani šajn, ako to vyzerá „vo vnútri“ súboru, nevedeli a ani nepotrebovali vedieť, ako štruktúra `FILE` vlastne vyzerá, aké premenné si musí nastaviť, aby sa program dostal na správne miesto na disku a prečítal, či zapísal, čo treba. A keby sme sa tie premenné pokúsili meniť bez toho, že by sme tušili, čo robíme, mohlo by to s diskom (alebo aspoň s našim súborom) urobiť rôzne podivné veci, podaktoré končiace katastrofou. Skrátka – keď robíte objekt, držte sa hesla „nepovolaným vstup zakázaný“.

Posledná veta vlastne nie je celkom presná. Pri vytváraní premenných, vnútorných funkcií a funkcií rozhrania nevytvárame objekt, ale **triedu objektov**. Trieda objektov je „to, čo majú všetky jej objekty spoločné“. Ak si spomenieme na náš prvý príklad objektu – premennú `i` typu `int`, tak trieda objektov je v tomto prípade „typ `int`“. Jednotlivé objekty tejto triedy sa môžu svojimi hodnotami prudko líšiť, ale majú spoločné to, že všetko sú to `int`y. Podobne ak naprogramujete triedu `okno`, tak vaše okná môžu mať mnoho roztodivných tvarov a podôb, ale všetky si vedia nejak poradiť s tým, že sa majú vedieť vykresliť a že majú nejak zareagovať na to, že na ne kliknete myšou.

Medzi triedami môžu byť rôzne vzťahy. Základné tri druhy vzťahov sú tieto:

- Jedna trieda obsahuje inú ako svoju súčasť. Napríklad trieda `bod2d` bude pravdepodobne obsahovať medzi svojimi premennými dva objekty typu `int` (učene povedané „dve inštancie triedy `int`“) a trieda `MojeDialogoveOkno` bude s najväčšou pravdepodobnosťou obsahovať nejaké premenné, ktoré budú inštanciami triedy `Button`[5] alebo nejakej podobnej. Tento typ vzťahu, v ktorom jedna trieda obsahuje inú, ako svoju súčasť, sa nazýva **agregácia** (po slovensky „obsiahnutie“).
- Jedna trieda vie o existencii inej a prispôsobuje podľa nej svoje správanie. Napríklad trieda `StrasnyNetvorZPlanetyX` by mala obsahovať smerník alebo iný kontakt na zoznam inštancií triedy `StrelaZMojejRakety`, aby vedela, kedy má veľkolepo zahynúť. Trieda `StrasnyNetvorZPlanetyX` nemá na tie strely žiadny monopol, ani nie sú jej súčasťou. Inak

4 Táto množina správ, ktoré je objekt ochotný prijať, sa nazýva rozhranie (anglicky interface)

5 Po slovensky „gombík“ alebo „tlačidlo“.

by k nim nemohli pristupovať iní strašní netvori z našej hry. Aj ich súradnice striel zisťuje s pomocou rozhrania triedy `StrelaZMojejRakety`. Ale `StrasnyNetvorZPlanetyX` musí mať v sebe citlivosť na strely naprogramovanú, inak by bol nezničiteľný a nikoho by nebavilo hrať to. Typ vzťahu, v ktorom jedna trieda vie o inej a prispôsobuje podľa nej svoje chovanie, sa nazýva **asociácia**.

- Jedna trieda je vylepšením druhej. Napríklad spravíme triedu `okno`, ktorá obsahuje základnú funkčnosť, ktorú požadujeme od všetkých okien (nejaké vykresľovanie a reakcie na klávesnicu a myš) a potom si môžeme vyrobiť triedu, ktorá bude, v podstate trieda `okno`, ale ešte nejako špecializovaná. Napríklad si môžeme vyrobiť triedu `tlacidlo` alebo `EditBox`. Obe tieto triedy sa budú navzájom líšiť. Tlačidlo bude reagovať na kliknutie myšou a prekresľovať sa podľa toho, či je práve stlačené, alebo nie. Editovacie pole sa bude prekresľovať podľa toho, čo do neho práve budete písať. Ale obe zdedia všetky vlastnosti triedy `okno`. Tento vzťah sa nazýva **dedičnosť** (anglicky inheritance). Niekedy sa používa aj názov **špecializácia**.

Takže toľko šedivá teória. V ďalšej lekcii už začneme programovať a ukážeme si, ako spraviť nejaké veci, o ktorých sme tu teoretizovali.

2. lekcia

Trieda Kruh
alebo "Nech žije ping-pong"

V predošlej lekcii sme si pekne zateoretizovali. Teraz ideme konečne nejaký objekt vytvoriť. A ako už názov lekcie napovedá, bude to kruh. Chceli by sme, aby nám po obrazovke pobiehal vyplnený krúžok, aby sa pohyboval vždy rovnomerne priamočiaro, ale aby sa od hranice obrazovky odrážal ako od mantinelu. V tejto lekcii si ukážeme, ako si taký objekt vytvoriť.

Keď vyrábame nový objekt, treba sa v prvom rade zamyslieť, ako tento objekt opísať. Aké dátové štruktúry (ej, aleže sme použili odborný výraz :) charakterizujú všetky vlastnosti objektu.[6] V prípade nášho kolieska to budú súradnice stredu a polomer (premenné `x`, `y` a `polomer`), ďalej jeho rýchlosť (tú si budeme pamätať v premenných `dx` a `dy` – budú nám hovoriť o koľko sa súradnice zmenia v jednom kroku) a farba (prekvapivo premenná `farba`). Okrem toho koliesku ešte zriadime premennú `bitmap`, ktorá bude obsahovať smerník na bitmapu, do ktorej sa má kresliť (keby sme čírou náhodou potrebovali kresliť niekam inam, než na obrazovku).

Tieto premenné nám naše koliesko charakterizujú dostatočne dobre. Ďalší krok bude vymyslieť rozhranie – rozmyslieť si, aké funkcie bude koliesko obsahovať a ktoré z nich budú prístupné verejnosti. Budú to jednak funkcie `Nakresli()` a `Zmaz()` z ktorých prvá nakreslí koliesko jeho farbou a druhá ho nakreslí čiernou, takže zmizne. Funkcia `Nakresli` bude verejne prístupná, funkcia `Zmaz` nie. Neverejné funkcie môžu používať iné funkcie z danej triedy, ale nemôže ich volať nikto iný.[7] Potom to bude funkcia `Pohni()`, ktorá koliesku vyráta nové súradnice. Všetky tri spomenuté funkcie bude v sebe obsahovať funkcia `Krok()`. Jej úlohou bude zmazať koliesko, posunúť ho a znova nakresliť. A nakoniec to bude funkcia `NastavBitmapu(BITMAP* bmp)`, ktorá koliesku prestaví bitmapu, do ktorej sa má kresliť, ak sa nám zachce kresliť niekde inde.

Pri programovaní v C++ vytvárame väčšinou pre každú triedu dva súbory. Prvý bude hlavičkový (bude mať koncovku `.h`) a bude obsahovať rozhranie triedy. To je čosi ako deklarácia funkcie v jazyku C. Každý kód, ktorý bude chcieť našu triedu používať si tento súbor includuje a bude vedieť, aké metódy, či premenné naša trieda obsahuje a ktoré z nich smie použiť. Druhý súbor bude mať väčšinou koncovku `.cpp` a bude obsahovať definície jednotlivých metód (t.j. funkcii) triedy. Takéto rozdelenie ale nie je povinné a pri jednoduchých príkladoch sa niekedy všetko napchá do jedného súboru.

Keď sme si teda rozmysleli, čo od našej triedy budeme chcieť, môžeme napísať interface (rozhranie). Bude v súbore `kruh.h` a bude vyzerať takto:

```
#ifndef KRUH_H
#define KRUH_H

#include <allegro.h>
```

6 Po slovensky – aké premenné do toho napchať.

7 Nie je žiaden vážny dôvod spraviť funkciu `Zmaz` privátnou. Je to tu len preto, aby ste videli, ako sa to robí.

```
class Kruh{
private:
    int x, y;
    int polomer;
    int dx, dy;
    int farba;
    BITMAP* bitmap;
    void Zmaz();
public:
    Kruh();
    void Nakresli();
    void Pohni();
    void Krok();
    void NastavBitmapu(BITMAP* bmp);
};

#endif
```

Ako ste si mohli všimnúť, celý kód je uzavretý medzi direktívami kompilátora `#ifndef KRUH_H #define KRUH_H` a `#endif`. To je finta, ktorá má zabezpečiť, aby sa hlavičkový súbor počas kompilácie načítal iba raz. Ak totiž vyrábate zložitú štruktúru navzájom previazaných objektov, celkom bežne sa stáva, že jeden hlavičkový súbor sa načíta z viacerých miest. Pre kompilátor je to ale veľmi mätúce. Ak sa hlavičkový súbor načíta druhýkrát, z pohľadu kompilátora to vyzerá, akoby sa deklarovala ďalšia trieda s rovnakým názvom a preto vyhlási chybu. Keď tam ale dáte to `#ifndef`, celý ďalší kód až po `#endif` sa bude kompilovať len vtedy, ak nie je definované makro `KRUH_H`. A keďže hneď v druhom riadku makro `KRUH_H` zadefinujeme, ak sa hlavičkový súbor načíta ešte odinakiaľ, druhýkrát už kompilovaný nebude.

Pred samotným vytváraním triedy ešte načítame `allegro.h`, pretože inak by kompilátor nemal odkiaľ vedieť, čo je to `BITMAP*`.

A konečne prichádzame k samotnej triede. Vytvára sa kľúčovým slovíčkom `class` po ktorom nasleduje meno triedy, kučeravé zátvorky a bodkočiarka.[8] Vo vnútri kučeravých zátvoriek sú dva odseky. Jeden uvedený kľúčovým slovom `private` a druhý začínajúci `public`. V privátnom odseku sú veci, ktoré sú vnútornými záležitosťami objektov danej triedy a nikto zvonka do nich siahať nemôže. Sú tam všetky atribúty objektu (To by tak malo byť vždy!!! Ak hodláte toto pravidlo porušiť, musíte na to mať véééééľmi dobrý dôvod.) a funkcia `Zmaz()`o ktorej sme rozhodli, že bude privátna. Pripomeňme, že slovíčko `void` znamená, že funkcia nevracia žiadnu hodnotu. V odseku `public` sú verejné funkcie, ktoré môže používať okolitý svet. Okrem funkcií spomenutých vyššie je tam ešte jedna. Má rovnaké meno, ako samotná trieda. Takáto funkcia sa volá **konštruktor** a volá sa vždy, keď vzniká objekt danej triedy. Konštruktor nesmie vracať žiadnu hodnotu a dokonca sa mu tam nepíše ani to `void`.

Dobre. Rozhranie by sme mali vytvorené. Hlavičkový súbor uložíme. Ako sme už povedali, každý, kto chce vo svojom programe používať objekty našej triedy ho bude musieť includovať. Vytváranie hlavičkového súboru ale stále viac-menej patrí do oblasti plánovania. Čo budú jednotlivé funkcie robiť, povieme inde – konkrétne v súbore s menom `kruh.cpp` ktorý si za tým účelom vytvoríme.

Súbor `kruh.cpp` bude na začiatku obsahovať direktívu `#include "kruh.h"`, pretože

8 Situácia sa môže ešte v detailoch meniť, ale o tom neskôr...

ak chceme implementovať (ďalšie pekné cudzie slovíčko, znamená to jednoducho naprogramovať) jednotlivé funkcie, musíme poznať ich deklaráciu. To, že sú okolo mena súboru úvodzovky a nie zobáky spôsobí to, že sa daný súbor bude hľadať medzi systémovými knižnicami, ale v tom istom adresári, ako je súbor `kruh.cpp`.[9]

A ideme na jednotlivé funkcie. Implementácia funkcie začína vždy menom triedy po ktorom nasledujú dve dvojbodky a meno funkcie s parametrami. Takže najprv ku konštruktoru. Je to funkcia, ktorá sa zavolá vždy, keď objekt vzniká. Preto je rozumné inicializovať v nej jednotlivé premenné triedy. Takže konštruktor bude vyzerať takto:

```
Kruh::Kruh()
{
    x = random() % SCREEN_W;
    y = random() % SCREEN_H;
    polomer = 10 + random() % 50;
    dx = random() % 15;
    dy = random() % 15;
    bitmap = screen;
    farba = makecol(random() % 256, random() % 256, random() % 256);
}
```

Keď vytvoríme nový kruh, jeho pozícia, polomer, rýchlosť a farba sa nastavia na náhodné hodnoty z patričného rozsahu a premenná `bitmap` sa nastaví na `screen`, aby sa kreslilo na obrazovku.

Ďalej nasledujú funkcie `Zmaz` a `Nakresli`:

```
void Kruh::Zmaz()
{
    circlefill(bitmap,x,y,polomer,makecol(0,0,0));
}

void Kruh::Nakresli()
{
    circlefill(bitmap,x,y,polomer,farba);
}
```

Funkcia `Pohni` upraví súradnice podľa rýchlosti. Ak by hrozilo, že koliesko ujde z monitora, zmení rýchlosť v danom smere na opačnú. Bude to vyzerať nasledovne (pripomeňme, že `||` znamená „alebo“):

```
void Kruh::Pohni()
{
    x += dx;
    y += dy;
    if ((x > SCREEN_W) || (x < 0)) dx = -dx;
    if ((y > SCREEN_H) || (y < 0)) dy = -dy;
}
```

Funkcia `Krok` zavolá všetky predošlé:

9 Napriek tomu, že v súbore `kruh.cpp` budeme hojne používať funkcie z knižnice Allegro, nepotrebujeme includovať súbor `allegro.h`. Viete prečo? (Mimochodom – ak ho includnete, nie je to chyba.)

```
void Kruh::Krok()
{
    Zmaz();
    Pohni();
    Nakresli();
}
```

A nakoniec funkcia `NastavBitmapu` nastaví cieľovú bitmapu, ak by sme si zmysleli kresliť kolieska niekam inam:

```
void Kruh::NastavBitmapu(BITMAP* bmp)
{
    bitmap = bmp;
}
```

Pekné, jasné, prehľadné. Súbor `kruh.cpp` môžeme uložiť. Naša prvá trieda je hotová.

Triedu sme si urobili, teraz by ju bolo dobré použiť. Urobíme si tretí súbor. Nazveme ho `pokus.cpp` a vložíme do neho toto:

```
#include <allegro.h>
#include "kruh.h"

int main()
{
  allegro_init();
  install_keyboard();
  set_color_depth(24);
  if (set_gfx_mode(GFX_AUTODETECT_FULLSCREEN, 640, 480, 0, 0) != 0)
  {
    set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
    allegro_message("Nemozem nastartovat grafiku\n%s\n",
                                                  allegro_error);
    return 1;
  }

  srandom(time(NULL));

  Kruh k;
  while (!keypressed())
  {
     vsync();
     k.Krok();
  }

  return EXIT_SUCCESS;
}
END_OF_MAIN()
```

Najprv includujeme `allegro.h` a `kruh.h` (áno, ja viem, jedno z toho sa dá vynechať). Potom si urobíme funkciu `main`. V nej naštartujeme grafiku a podľa aktuálneho systémového času nastavíme generátor náhodných čísel. A teraz to príde. Teraz sa zúročí celá naša drina. Deklarujeme si premennú (objekt) `k` typu (triedy) `Kruh`. A v cykle, ktorý bude bežať dovtedy, kým niekto niečo na klávesnici nestlačí budeme stále dokola volať funkciu (metódu) `Krok` objektu `k`. [10]

10 Pred každým pohybom kruhu ešte zavoláme funkciu `vsync()`. Táto funkcia čaká, kým sa nezačne prekresľovať obrazovka. S jej použitím sa zníži (aj keď úplne neodstráni) blikanie spôsobené tým, že kruh najprv zmažeme a potom kreslíme inde.)

Takže naprogramované by bolo, ide sa kompilovať. Najprv si skompilujeme veci okolo kruhu príkazom

```
g++ -c kruh.cpp
```

Kompilátor vytvorí súbor `kruh.o`. Potom rovnako skompilujeme súbor `pokus.cpp`:

```
g++ -c pokus.cpp
```

Vytvorí sa súbor `pokus.o`. A teraz spojíme tie súbory spolu s knižnicou allegro do spustiteľného súboru `pokus`:

```
g++ -o pokus pokus.o kruh.o -lalleg -lalleg_unsharable
```[11]

Hotovo. Spustíte program `pokus` (napríklad príkazom `./pokus`) a môžete sa tešiť.

Úloha č.1: Celé to dajte dokopy, napíšte do súborov a rozbehajte.
Úloha č.2: Zmeňte súbor `pokus.cpp` tak, že vytvoríte pole 10 kruhov a tie necháte poskakovať po obrazovke.
Úloha č.3: Zmeňte správanie sa kruhu tak, aby sa od okraja neodrážal, ale objavil sa na opačnej strane monitora.
Úloha č.4: Doplňte kruhu funkciu na nastavovanie farby.
Úloha č.5: Skúste z funkcie `main()` zavolať príkaz `k.Zmaz()`. Akú chybovú hlášku vygeneruje kompilátor?
Úloha č.6 (pre machrov): Odstráňte úplne to blikanie. Spravte to tak, aby to nebolo na úkor rýchlosti.

11 Ak pracujete pod inou platformou, ako Linux, pravdepodobne parameter `-lalleg_unsharable` nebudete používať.

3. lekcia

KDevelop
alebo "Predsa len Hello, world!"

Každý, kto skúsil spraviť čo i len trochu väčší projekt vie, že preplietať sa zákutiami vlastných zdrojových súborov, hľadať funkciu, ktorú robil kedysi pred mesiacom a spomínať, čo kde bolo uložené je robota pre masového samovraha. Keď máte funkčnosť pekne rozdelenú do objektov, je to o niečo jednoduchšie. Väčšinou si aspoň približne pamätáte, čo ktorý objekt robí a funkciu, ktorá zabezpečuje vykresľovanie postavičky v hre nebudete hľadať v objekte, ktorý vám sprístupňuje dátový súbor. Pri veľkom projekte sa ale môže stať, že kým nájdete niektorú konkrétnu metódu konkrétneho objektu, zabudnete, čo ste v nej vlastne chceli robiť. Toto je jeden z dôvodov, prečo si programátori robia programy, ktoré uľahčujú orientáciu v projekte, udržiavajú v ňom aspoň čiastočne poriadok a prinášajú celkom príjemný komfort.

Snáď najznámejšie nástroje z tejto série pochádzajú od firmy Microsoft (.NET či Visual C++) a od firmy Borland. My sa v tejto lekcii budeme zaoberať nástrojom Kdevelop[12], ktorý je súčasťou väčšiny distribúcii Linuxu a je šírený pod licenciou GPL, takže sú k dispozícii zdrojové kódy a za jeho používanie netreba platiť. V prípade, že ho použiť nemôžete (napríklad preto, že si nechcete nainštalovať Linux), túto lekciu pokojne preskočte. V ďalších lekciách budeme uvádzať iba zdrojové texty jednotlivých programov a tie by mali byť vzhľadom na použité knižnice nezávislé na platforme, operačnom systéme či vývojovom prostredí.

Takže ideme spustiť Kdevelop. Poobzerajte sa, či niekde na ploche nevidíte ikonu podobnú tej vpravo na obrázku. Ak takú nenájdete, malo by postačiť napísať na konzole príkaz `kdevelop`. Ak ani to nezabralo, tak to asi nemáte Kdevelop nainštalovaný.

Ak sa vám úspešne podarilo Kdevelop prvýkrát spustiť, pravdepodobne vás na chvíľu prepadne panika, pretože je na prvý pohľad zrejmé, že sa jedná o rozsiahly software, ale na druhú stranu to nič nerobí a človek hneď nevie, čo s tým. Takže pocit paniky teraz na chvíľu prekonajte, ideme vytvárať projekt.

Prekvapivo treba kliknúť v menu na položku *Projekt* a vvybrať možnosť *Nový projekt...* Kdevelop na vás vychrlí okienko, v ktorom treba vybrať zo spleti jazykov *C++* (áno, tušíte správne, C++ nie je jediný jazyk, ktorý Kdevelop podporuje) a z možností uvedených pri C++ vybrať *Simple Hello world program*. Okrem toho treba ešte vyplniť meno aplikácie (ak robíte projekt z minulej lekcie, nazvite si to trebárs kolieska), skontrolovať, či vám vyhovuje, kde Kdevelop projekt vytvorí a ostatné nastavenia a môžete stlačiť tlačidlo *Dozadu>*, ktoré má napodiv význam „Ďalej“.

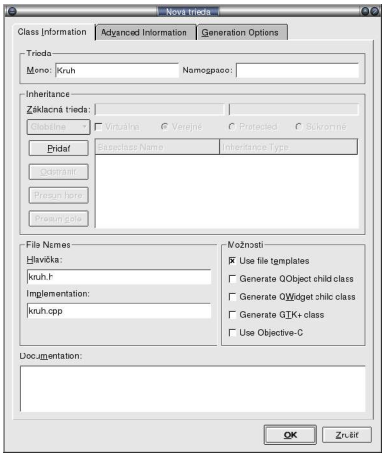
Druhá obrazovka slúži na nastavenie systému kontroly verzií (používa sa, ak na projekte pracuje viacero ľudí alebo ak je dôležité zachovať jednotlivé verzie počas vývoja projektu). Zatiaľ tam nechajte *None* a stlačte *Dozadu>*.

12 Táto lekcia bola písaná podľa verzie Kdevelop 3.0.1

Na tretej obrazovke vám v závislosti na licencii, ktorú ste zvolili navrhne Kdevelop nadpis, ktorý sa pri vytvorení každého hlavičkového súboru (to sú tie, ktorých meno končí na .h) automaticky pridá na začiatok. Zmeňte si to k obrazu svojmu (ale dbajte na to, aby to boli samé komentáre) alebo to úplne vymažte. Na štvrtej obrazovke vás čaká to isté pre súbory .cpp. Okrem toho sa tam objaví tlačidlo *Dokončiť*, ktoré môžete stlačiť. Vytvorí sa projekt, ktorý spraví náš obľúbený program vypisujúci „Hello, world!“.

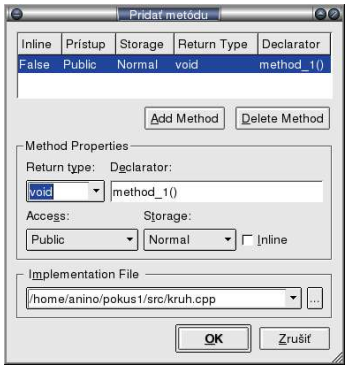
Tým, čo vlastne to do toho programu Kdevelop napchal, sa teraz zaoberať nebudeme. Ideme to skompilovať a budeme dúfať, že tam nie sú nezmysly. Na kompiláciu slúži tlačidlo . Keď ho v projekte stlačíte prvýkrát, Kdevelop vyhlási, že si ešte nejaké detaily musí zariadiť. Vy mu odklepnete, že áno, on tam pospúšťa nejaké podivnosti a potom vám program skompiluje. Keď si program chcete vyskúšať spustiť, môžete stlačiť tlačidlo .

Poďme teraz do nášho projektu pridať novú triedu. (Ona nebude až tak veľmi nová, bude to naša stará známa trieda Kruh.) Aj táto činnosť je samozrejme automatizovaná. Keď stlačíme tlačidlo určené na výrobu novej triedy , objaví sa dialógové okno. Niektoré veci, ktoré v ňom vidíte, oceníte až neskôr. Teraz spomenieme len dve dôležité kolónky – prvá z nich je samozrejme meno triedy. Keď do nej napíšeme „Kruh“, automaticky sa vyplnia aj kolónky pre meno hlavičkového súboru a súboru s implementáciou. Druhá dôležitá vec je – napodiv – komentár. Do kolónky s nadpisom *Documentation* je treba napísať, čo je daná trieda zač. Každý riadok komentára má pri návrate k starému projektu na ktorom ste asi mesiac nerobili cenu zlata.



Keď vyplníte všetko, čo máte, môžete stlačiť *OK*. Kdevelop sa pre istotu ešte raz spýta, že či má uvedené súbory naozaj vytvoriť. Znovu mu povedzte *OK*.

Nová trieda je na svete, len ju nikde nie je vidieť. A teraz prichádza k slovu mocný navigačný nástroj, ktorý sa skrýva pod nenápadnou ikonou v ľavej nástrojovej lište. Keď na ňu kliknete, objaví sa vám stromová štruktúra, v ktorej máte prehľadne uložené všetky objekty vášho projektu, ich metódy a atribúty aj všetky globálne funkcie. Stačí kliknúť na funkciu na ktorú chcete a v okamihu sa dostanete na jej implementáciu.



Nástroj na prezeranie tried je užitočný aj z iných dôvodov. Ak chceme triede pridať novú funkciu (metódu), stačí kliknúť pravým tlačidlom na ikonu danej triedy a vybrať *Pridať metódu...* a znovu na nás vyskočí dialógové okno, ktoré je tentokrát určené na pridávanie funkcie. Vyberiete, akú hodnotu má funkcia vrátiť (ak žiadnu, necháte `void`), zmeníte názov z `method_1()` na nejaký rozumnejší (napríklad `Pohni()`), pridáte parametre, nastavíte, či je *Public* alebo *Private* a stlačíte *OK*. Nová funkcia sa pridá do deklarácie v hlavičkovom súbore a súčasne sa vytvorí jej (zatiaľ prázdna) implementácia. Rovnako jednoducho (pravým kliknutím na ikonu triedy) sa pridá aj nový atribút (premenná).

Už by to bolo všetko skoro úplne dokonalé, ale je tu ešte jeden problém – predstavte si, že robíte program s použitím knižnice Allegro. Kdevelop vám všetko pekne skompiluje, ale... nezlinkuje. Nikde ste mu totiž nepovedali, aké knižnice má prilinkovať. Aby ste to nastavili, nájdite

si v pravej nástrojovej lište ikonu (píše sa pri nej niečo ako „správca automake“) a stlačte ju. Objaví sa vám nástroj rozdelený na dve tabuľky. Obe obsahujú ikonku . Stlačte ju na dolnej tabuľke, ktorá je venovaná konkrétnemu podprojektu vášho projektu. Vyskočí na vás dialógové okno. V ňom si vyberte roletku *Knižnice*. Tá je zas rozdelená na dve polovice a opäť nás bude zaujímať tá dolná. Stlačíte tlačidlo *Pridať...* a pridáte knižnicu `-lalleg`. Stlačíte *Pridať...* znovu a pridáte `-lalleg_unsharable`. Ešte je treba dať pozor na to, aby sa knižnice linkovali v správnom poradí. Keďže `alleg` potrebuje nejaké funkcie z `alleg_unsharable`, je nutné, aby `alleg` bola vyššie. To ľahko zariadite s pomocou tlačidiel *Presun hore* a *Presun dole*.

Úloha č.1: Urobte tie poletujúce kolieska v Kdevelope. Samozrejme to celé nepreklepávajte. Vytváranie novej triedy, atribútov a metód si vyskúšajte tu opísaným systémom, ale implementáciu funkcií si môžete skopírovať z predošlej kapitoly.

Ďalšou zaujímavou vecou, ktorá sa dá v Kdevelope podnikať je ladenie. Programy často robia naprosté hlúposti a jediný spôsob, ako zistiť, čo sme spravili zle je pekne krok po kroku sledovať, čo to vlastne robí. Predpokladajme, že tie kolieska sa vám podarilo rozbehať, ale nejak sa vám nepozdáva metóda `Pohni` triedy `Kruh` a chceli by ste sa podrobnejšie pozrieť, čo sa v ňom vlastne deje. Keďže chcete mať naraz pred očami aj to, čo program robí, aj jeho zdrojový kód, nezabudnite si nastaviť režim grafiky na `GFX_AUTODETECT_WINDOWED`.

Prejdite na funkciu `Pohni` triedy `Kruh` a tam kliknite na prvý riadok pravým tlačidlom myši. A z menu, ktoré na vás vybehne, vyberte možnosť *Prepnúť zarážku*. Celý riadok sa prefarbí na ružovo. Znamená to, že ste na ten riadok nastavili breakpoint (to je normálny názov pre zarážku). Ak teraz spustíte program v ladiacom režime (tlačidlom), rozbehne sa, ale na breakpointe sa zastaví a čaká na to, čo spravíte. Teraz sa môžete pozrieť, čo práve máte uložené v premenných. Stlačíte tlačidlo {x} na ľavej lište, do dolného riadku napíšete meno premennej, ktorú chcete zobraziť a stlačíte tlačidlo *Pridať*. Premenná sa zaradí do zoznamu zobrazovaných premenných a môžete sa pokochať jej hodnotou. Program zas môžete pustiť ďalej po najbližší breakpoint (tlačidlom) alebo ho môžete posunúť o jeden príkaz (tlačidlom). Ak je najbližší príkaz volanie funkcie, vykoná sa celá. Keby sme chceli do tej funkcie vliezť a pozrieť sa, čo sa deje v jej vnútri, treba použiť tlačidlo . Ak chceme z nejakej funkcie vyjsť a pozrieť sa, ako to vyzerá „o úroveň vyššie“ tesne po jej dobehnutí, použijeme tlačidlo . (Medzitým samozrejme zastavíme na všetkých riadkoch, ktoré sme nastavili ako breakpointy.) Pri tom môžeme stále sledovať jednak to, čo program robí, jednak to, čo je uložené v jednotlivých premenných. Ak sa nám breakpoint na niektorom riadku zunuje, vypneme ho rovnako, ako sme ho zapli (teda pravým tlačidlom myši).

Takže asi tak. Kdevelop má množstvo funkcií, z ktorých sme sa letmo dotkli iba niekoľkých. Vôbec sme nehovorili o dokumentácií, nespomínali sme žiaden z nástrojov na dolnej lište, nevraveli sme, ako automaticky formátovať zdrojový kód a mnoho iných vecí. Ale aspoň budete mať čo objavovať. Hlavne sa nezľaknite, ak niečo pokazíte a neprestaňte kvôli tomu experimentovať. A ak sa vám nechce objavovať, prečítajte si manuál.

4. lekcia

Príkazy new a delete
alebo "Dynamická alokácia po novom"

Možno si ešte pamätáte, ako to fungovalo, keď ste v Cčku potrebovali prideliť kus pamäte. Použili ste funkciu `malloc`, ktorej ste povedali, koľko tej pamäte potrebujete a ona vám ju vyhradila. Keď ste tú pamäť už nepotrebovali, tak ste ju uvoľnili príkazom `free`.

Ak použijete patričnú knižnicu[13], bude vám to fungovať aj v C++, lenže to má zádrhele. Ak napríklad vo funkcii `main()` z druhej kapitoly napíšete príkaz

```
Kruh *pk = (Kruh*) malloc(sizeof(Kruh));
```

kompilátor žiadnu chybu nenájde a patričná časť pamäte sa vám vyhradí. Problém je ale v tom, že ak by ste chceli objekt, na ktorý ukazuje smerník `pk` používať, zistíte, že v jeho atribútoch sú naprosté blbosti. Skrátka – nespustil sa konštruktor, ktorý premenné nastavuje a tak je v nich to, čo sa práve v pamäti nachádzalo.

C++ má samozrejme spôsob, ako pamäť pre objekt alokovať tak, aby sa všetko udialo tak, ako sa má. A má to spravené krajšie, ako Cčko. Slúži na to príkaz **new**. Ak by sme teda chceli alokovať pamäť pre jeden objekt triedy `Kruh` tak, aby sa spustil aj konštruktor, spravíme to príkazom

```
Kruh *pk = new Kruh();
```

Za príkaz `new` sme jednoducho napísali konštruktor daného objektu. Okrem toho, že príkaz vyhradil pamäť pre daný objekt, zavolal aj konštruktor, ktorý sme uviedli.

Teraz je vhodný čas, aby ste sa dozvedeli ďalšiu novinu: Podobne ako objekt vzniká a pritom sa volá jeho konštruktor, objekt často aj zaniká a – áno, hádate správne – pritom sa volá jeho **deštruktor**. Trieda môže mať konštruktorov viacero (o tom niekedy nabudúce), ale deštruktor je vždy iba jeden. Je to funkcia bez parametrov vracajúca `void` (čiže nič), ktorej meno je rovnaké, ako meno triedy, akurát, že sa pred ním (tesne pred ním bez medzery) nachádza `~`. Deštruktor triedy `Kruh` by bol teda deklarovaný ako `void ~Kruh();` Zavolá sa automaticky, keď objekt triedy `Kruh` zaniká.

Úlohou deštruktora je po objekte poupratovať. Ak objekt sám pre seba dynamicky alokoval nejakú pamäť, táto sa často uvoľňuje práve v deštruktore. Ukážeme si to ešte v tejto lekcii.

Pri uvoľňovaní použitej pamäte ale stojíme pred podobným problémom, ako pri jej alokovaní. Cčková funkcia `free` síce uvoľní pamäť po objekte, ale ak sa v uvoľnenej pamäti nachádzali smerníky na ďalšie dynamicky alokované veci, tie už neuvoľní nikto až do konca behu programu. Preto to treba zariadiť tak, aby sa zavolal deštruktor a v ňom pamäť pouvoľňovať predtým, než objekt definitívne zanikne.

Na takéto korektné mazanie slúži funkcia **delete**. Príkaz

```
delete pk;
```

13 Teda ak includujete `stdlib.h`

uvoľní pamäť po dynamicky alokovanom kruhu, na ktorý ukazuje smerník `pk` tak, že ak by mala trieda `Kruh` definovaný deštruktor, tak sa zavolá ešte pred zlikvidovaním samotného objektu.

Predtým, než sa po teoretickej časti zase začneme zaoberať programovaním, ešte drobná teoretická poznámka – objekt dynamicky alokovať vieme. Ako dynamicky alokovať pole? Jednoducho. Ak chceme pole celých čísel, napíšeme

```
int *pole = new int[1000];
```

ak chceme pole kruhov, napíšeme

```
Kruh *pole_kruhov = new Kruh[10]();
```

Potom s nimi môžeme normálne pracovať. Môžeme napríklad napísať `pole[412] = 33;` alebo `pole_kruhov[3].Pohni();` Musíme ale dať pozor, keď budeme polia uvoľňovať z pamäte. Totiž napríklad smerník `pole_kruhov` ukazuje na prvý kruh v poli a keby sme napísali iba

```
delete pole_kruhov;
```

zmazal by sa iba on. Príkazu `delete` skrátka treba dať vedieť, že ideme likvidovať pole. Správne sa pole zmaže príkazom

```
delete [] pole_kruhov;
```

Takto sa jednak zlikviduje celé pole a jednak sa pre každý jeho prvok predtým zavolá deštruktor, takže sa to pozatvára správne.[14]

Dobre. Poďme sa teraz pozrieť na projekt, ktorý sme zanechali v druhej lekcii. Po obrazovke nám tam pobiehalo desať kruhov a odrážalo sa od stien (prípadne cez ne prechádzalo). Teraz by sme chceli zmeniť dve veci – jednak by sme chceli, aby sa dali kruhy pridávať počas behu. A dvak by sme chceli, aby sa kruhy okrem toho, že sa odrážajú od stien, odrážali aj vzájomne od seba. A naviac sa pokúsime spraviť prekresľovanie metódou špinavých obdĺžnikov. (To je taká finta, ktorá má zabezpečiť, aby tie kruhy neblikali.)

Predtým, než začnete programovať si veci z druhej lekcie niekam skopírujte. Budeme ich totiž používať a prerábať a bola by škoda, keby sa vám nezachoval funkčný program. Dávajte pozor a robte všetky zmeny zároveň s tým, ako o nich čítate. (Alebo si to najprv prečítajte a pochopte, potom sa sem vráťte a urobte ich až pri druhom čítaní.)

Vyrobíme si novú triedu `SpravcaTelies.` Táto bude obsahovať vlastnú pomocnú pamäťovú bitmapu, do ktorej s bude všetko kresliť a na obrazovku sa to z nej už bude iba kopírovať. Ďalej sa bude starať o pohyb a prekresľovanie jednotlivých kruhov a bude dbať o to, aby boli správne ošetrené zrážky medzi kruhmi. Okrem toho bude vedieť pridať nový kruh.

Bolo by vhodné pripomenúť prvú lekciu a vzťahy medzi triedami. Keď sme vyrobili triedu `Kruh`, obsahovala niekoľko celých čísel a jeden smerník. Tieto veci boli v triede **agregované**. Teraz ideme vyrábať triedu, ktorá bude mať dosah na všetky kruhy a bude s nimi môcť istým spôsobom manipulovať. Ale nemôže ich obsahovať ako svoje premenné (čiastočne aj preto, lebo nie je jasné, koľko ich vlastne bude a ich počet sa bude meniť). Trieda `SpravcaTelies` bude teda s triedou `Kruh` **asociovaná**.

14 Tie hranaté zátvorky ostanú prázdne. Sú tam naozaj iba na to, aby bolo jasné, že ideme mazať pole.

Predtým, ako začneme pracovať na novej triede, ale potrebujeme spraviť nejaké zmeny na samotnej triede `Kruh`. V prvom rade potrebujeme mať prístup k niektorým jeho atribútom, aby správca vedel zistiť, že sa niektoré kruhy zrazili. Priamo do hlavičkového súboru pridáme funkcie

```
int GetX()  {return x;}
int GetY()  {return y;}
int GetR()  {return polomer;}
int GetDX() {return dx;}
int GetDY() {return dy;}
```

Ako si môžete všimnúť, tieto funkcie sú tam nie iba deklarované, ale aj rovno definované. Funkcia, ktorá je v hlavičkovom súbore aj definovaná je **inline** funkcia. Takáto funkcia má tu výhodu (alebo nevýhodu), že sa v už skompilovanom programe netvári ako funkcia, ale jej kód sa proste vloží na každé miesto v programe, kde sa vyskytuje jej volanie.[15] Tým sa ušetrí niekoľko inštrukcií, ktoré procesor potrebuje pri volaní každej funkcie vykonať, ale program sa tým zväčší, lebo kód každej takejto funkcie sa v ňom bude nachádzať na rôznych miestach. Takže ako `inline` je vhodné robiť iba funkcie, ktoré sú buď veľmi krátke, alebo funkcie, ktoré sa volajú veľmi často a je nutné, aby sa vykonali rýchlo. Inak by bol skompilovaný program neúmerne veľký.

Okrem toho potrebujeme po zrážke nastaviť novú rýchlosť. Na to poslúžia nasledujúce dve funkcie, ktoré tiež vložte do sekcie `public`

```
int SetDX(int i) {dx = i;}
int SetDY(int i) {dy = i;}
```

Ďalšie zmeny sa budú týkať toho, že každý kruh by mal byť schopný povedať, akú časť obrazovky treba prekresliť, aby sa zmazala jeho stará a nakreslila jeho nová pozícia (teda „určiť svoj špinavý obdĺžnik“). Na to si ale musí pamätať aj svoju predošlú pozíciu. Takže pridáme dva atribúty (do sekcie `private`)

```
int oldx, oldy;
```

Na začiatok ich nastavíme na tie isté hodnoty, ako `x` a `y`, takže prvé dva riadky konštruktora môžeme zmeniť na

```
oldx = x = random() % SCREEN_W;
oldy = y = random() % SCREEN_H;
```

a na začiatku metódy `Krok`, pred tým, než vyrátame nové hodnoty pre `x` a `y` pridáme

```
oldx = x;
oldy = y;
```

aby sme si zapamätali staré hodnoty.

Kým budeme robiť funkciu, ktorá by nám vypočítala špinavý obdĺžnik, načim sa nám zamyslieť, ako tá funkcia bude odovzdávať svoj výsledok. Optimálne by bolo, keby sme mali nejakú triedu `Rect`, ktorá by reprezentovala obdĺžnik. V mnohých knižniciach sa niečo také nachádza, ale žiaľ, momentálne máme smolu a musíme si také niečo urobiť sami. Takže si vytvoríme triedu `Rect`. Bude jednoduchá, všetko, čo potrebujeme, napcháme do hlavičkového súboru `rect.h` a ten bude

15 Aj funkciu, ktorej definícia sa nenachádza v hlavičkovom súbore môžeme definovať ako `inline`. Stačí, ak pred jej deklaráciu (ktorá v hlavičkovom súbore je) napíšeme ono slovko `inline`.

vyzerať takto:

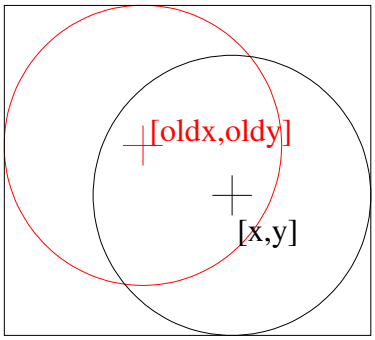
```
#ifndef RECT_H
#define RECT_H

class Rect{
private:
     int left,top,width,height;
public:
     void SetLeft(int i)    {left = i;}
     void SetTop(int i)     {top = i;}
     void SetWidth(int i)   {width = i;}
     void SetHeight(int i)  {height = i;}
     int  GetLeft()         {return left;}
     int  GetRight()        {return left + width - 1;}
     int  GetTop()          {return top;}
     int  GetBottom()       {return top + height - 1;}
     int  GetWidth()        {return width;}
     int  GetHeight()       {return height;}
};

#endif
```

Trieda je jednoduchá, pamätá si ľavý horný roh a rozmery nejakého obdĺžnika. Všetko sa v nej dá nastaviť aj o nej zistiť s pomocou funkcii.

Takže ideme počítať špinavý obdĺžnik. Najprv sa pozrite na obrázok.



Červený kruh znázorňuje starú polohu nášho kruhu, čierny novú. Ľavú stranu špinavého obdĺžnika vypočítame tak, že zistíme, ktorý zo stredov je viacej vľavo (na obrázku je to stred starej kružnice) a od neho sa posunieme ešte o polomer vľavo (a pre istotu ešte o 1). Vrch zistíme podobne. Šírka obdĺžnika bude rozdiel x-ových súradníc na každú stranu zväčšený o polomer kružnice (a tiež pre istotu ešte pridáme 3). Výška bude to isté s y-ovými súradnicami. Takže triede `Kruh` pridáme do hlavičkového súboru `#include "rect.h"` a do časti `public` funkciu

```
Rect SpinavyObdlznik();
```

ktorej implementácia (v súbore `kruh.cpp`) bude vyzerať takto:

```
Rect Kruh::SpinavyObdlznik()
{
     Rect obdlznik;
     obdlznik.SetLeft(min(x,oldx) - polomer - 1);
     obdlznik.SetTop(min(y,oldy) - polomer - 1);
     obdlznik.SetWidth(2 * polomer + abs(x – oldx) + 3 );
     obdlznik.SetHeight(2 * polomer + abs(y - oldy) + 3 );
     return obdlznik;
}
```

Drobný problém je, že C++ nepozná funkciu `min`. Môžeme si ju nadefinovať ako makro (vložte ho kdesi ku začiatku súboru `kruh.cpp`)[16]

```
#define min(x,y) (((x) < (y)) ? (x) : (y))
```

Posledná zmena, ktorú na triede `Kruh` musíme spraviť je, že musíme zariadiť, aby sa jednotlivé kruhy dali zoradiť za sebou do zoznamu. (Áno, bude to fungovať úplne rovnako, ako zreťazené zoznamy – vláčiky – o ktorých sme písali v kurze grafiky.) Každý kruh teda bude obsahovať smerník na kruh nasledujúci v zozname po ňom. Pridáme teda atribút

```
Kruh* dalsi;
```

do konštruktora prihodíme riadok

```
dalsi = NULL;
```

(pri vytvorení za kruhom nič nie je) a do sekcie `public` pridáme dve inline funkcie na prácu týmto smerníkom:

```
Kruh* Dalsi() {return dalsi;}
void SetDalsi(Kruh* k) {dalsi = k;}
```

Slúžia na to, aby sme sa mohli opýtať, aký kruh po našom kruhu nasleduje a aby sme za náš kruh mohli ďalší zaradiť.

Uf. To by snáď bolo, čo sa našej nebohej triedy `Kruh` týka všetko. Môžeme začať pracovať na triede `SpravcaTelies`.

Hlavičkový súbor triedy `SpravcaTelies` bude vyzerať takto:

```
#ifndef SPRAVCATELIES_H
#define SPRAVCATELIES_H

#include "kruh.h"
#include "rect.h"

class SpravcaTelies{
public:
     SpravcaTelies();
     ~SpravcaTelies();
     void Pridaj();
     void Krok();
     void Nakresli();
private:
     bool Zrazka(Kruh* k1, Kruh* k2);
     BITMAP* buffer;
     Kruh* prvy;
     Kruh* posledny;
};

#endif
```

Atribúty sú `buffer`, pomocná bitmapa do ktorej budeme všetko kresliť a z ktorej budeme iba špinavé obdĺžniky kopírovať na obrazovku a smerníky `prvy` a `posledny`, ktoré budú ukazovať na začiatok a na koniec zoznamu kruhov. Verejné funkcie sú konštruktor a deštruktor, funkcia `Pridaj`, ktorá pridá nový kruh, funkcia `Krok`, ktorá všetkým pohne a funkcia `Nakresli`, ktorá

16 Pamätáte sa ešte na podmienené výrazy?

všetko nakreslí na obrazovku. Funkcia `Zrazka` bude súkromná a bude vedieť zistiť, či sa kruhy, ktoré dostane na vstupe zrazili.

Poďme sa pozrieť na implementáciu. Takže najprv konštruktor:

```
SpravcaTelies::SpravcaTelies()
{
     buffer = create_bitmap(SCREEN_W,SCREEN_H);
     prvy = posledny = NULL;
}
```

Nedeje sa nič zvláštne, pomocná bitmapa sa nastaví na rovnakú veľkosť, ako má obrazovka, smerníky sa vynulujú. Funkcia `Pridaj` bude zaujímavejšia:

```
void SpravcaTelies::Pridaj()
{
     Kruh* pk = new Kruh();
     pk -> NastavBitmapu(buffer);
     if (prvy == NULL)
     {
          prvy = pk;
          posledny = pk;
     }
     else
     {
          posledny -> SetDalsi(pk);
          posledny = pk;
     }
}
```

Dynamicky si vytvoríme nový kruh a nastavíme mu, že sa má kresliť do buffra. Ak zatiaľ nemáme žiadne kruhy, smerníky `prvy` aj `posledny` nastavíme na novovytvorený kruh. Ak už tam nejaký zoznam je, tak poslednému kruhu nastavíme, že po ňom nasleduje novovytvorený a smerník `posledny` posunieme na neho.

Funkcia `Zrazka` vracia logickú hodnotu (bool). Vyráta vzdialenosť stredov zadaných kružníc (z Pytagorovej vety) a ak je menšia ako súčet polomerov, kruhy sa zrazili. Aby sme nemuseli odmocňovať, porovnávame druhé mocniny uvedených hodnôt, teda namiesto nerovnosti $\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}<r_1+r_2$ budeme overovať nerovnosť $(x_1-x_2)^2+(y_1-y_2)^2<(r_1+r_2)^2$

```
bool SpravcaTelies::Zrazka(Kruh* k1, Kruh* k2)
{
    return (k1->GetX() - k2->GetX()) * (k1->GetX() - k2->GetX()) +
           (k1->GetY() - k2->GetY()) * (k1->GetY() - k2->GetY()) <
             (k1->GetR() + k2->GetR()) * (k1->GetR() + k2->GetR()) ;
}
```

Funkcia `Krok` vymaže buffer a pohne všetkými kruhmi. Potom skontroluje, či sa nejaké kruhy nezrazili, a ak áno, vymení im rýchlosti. (V prípade zrážky dvoch rovnako ťažkých telies sa udeje presne to.)

```
void SpravcaTelies::Krok()
{
     clear_to_color(buffer,makecol(0,0,0));

     Kruh* pk;
     for(pk = prvy; pk != NULL; pk = pk->Dalsi() )
          pk->Krok();

     Kruh* pk2;
     for(pk = prvy; pk != NULL; pk = pk -> Dalsi())
          for(pk2 = prvy; pk2 != pk; pk2 = pk2->Dalsi() )
          {
               if (Zrazka(pk,pk2))
               {
                    int dx = pk->GetDX();
                    int dy = pk->GetDY();
                    pk->SetDX(pk2->GetDX());
                    pk->SetDY(pk2->GetDY());
                    pk2->SetDX(dx);
                    pk2->SetDY(dy);
               }
          }
}
```

Zaujímavé je, ako je spravený ten prvý cyklus `for`. Riadiacou premennou je smerník `pk`, ktorý sa na začiatku nastaví na prvý kruh, po každom prebehnutí cyklu sa nastaví na ďalší a cyklus sa opakuje až kým `pk` nie je `NULL`, takže až kým sa neprebehne celý zoznam. (`NULL` je „`dalsi`“ kruh posledného kruhu v zozname. Bol tak nastavený ešte v konštruktore a nikdy nebol zmenený.)

Funkcia `Nakresli` prejde všetky kruhy a ich špinavé obdĺžniky prekreslí na obrazovku

```
void SpravcaTelies::Nakresli()
{
     Kruh* pk;
     vsync();
     for(pk = prvy; pk != NULL; pk = pk->Dalsi())
     {
          Rect obdl = pk->SpinavyObdlznik();
          blit( buffer, screen, obdl.GetLeft(), obdl.GetTop(),
               obdl.GetLeft(), obdl.GetTop(),
               obdl.GetWidth(), obdl.GetHeight() );
     }
}
```

Cyklus `for` je tu rovnaký, ako v metóde `Krok`.

A na záver deštruktor. Jeho úlohou je pomazať pri skončení práce správcu buffer a všetky kruhy z pamäte.

```
SpravcaTelies::~SpravcaTelies()
{
     destroy_bitmap(buffer);

     Kruh *pk, *pk2;
     pk = prvy;
     while (pk != NULL)
     {
          pk2 = pk->Dalsi();
          delete pk;
          pk = pk2;
     }
}
```

V hlavnom cykle sú až dva smerníky. Ak by sa totiž zmazal kruh na ktorý ukazuje smerník `pk` predtým, než by sa zistilo, kde je ďalší kruh, už by sme sa k nemu nedostali. A tak si jeho pozíciu v pamäti uložíme do smerníka `pk2`.

Hotovo. Už len trochu upraviť funkciu `main`. Po inicializácii generátora náhodných čísel tam dajte niečo takéto:

```
SpravcaTelies spravca;

while(!key[KEY_ESC])
{
     if (key[KEY_P])
          spravca.Pridaj();
     spravca.Krok();
     spravca.Nakresli();
}

return EXIT_SUCCESS;
```

a môžete kompilovať.

Na záver tri poznámky. Prvá sa týka špinavých obdĺžnikov. Ich použitie je vhodné, ak je na ploche relatívne málo pohybujúcich sa objektov. Ak je ich veľa, tak oproti dvojitému bufferingu nič neušetríme, naopak, niektoré miesta budeme prekresľovať zbytočne veľakrát. Vtedy je výhodnejšie kopírovať celý buffer na obrazovku naraz. (Môžete to spraviť tak, že si správca bude pamätať počet kruhov a ak je ich viac, než 50, skopíruje celý buffer na obrazovku naraz.)

Druhá sa týka zisťovania havárií. Keďže treba pri zisťovaní havárií preveriť každú dvojicu kruhov, pri zvyšovaní ich počtu, počet dvojíc rastie oveľa rýchlejšie (pri troch kruhoch tri dvojice, pri desiatich štyridsaťpäť dvojíc, pri tridsiatich štyristo tridsaťpäť dvojíc). To je tiež jedna z vecí, ktoré môžu váš program spomaľovať.

Tretia poznámka sa týka podivného správania niektorých kruhov – zaseknú sa na okraji a nechcú sa odtiaľ pohnúť. Postupne sú ostatnými kruhmi vyrazené za okraj. Toto správanie sa dá zmeniť nasledujúcou úpravou metódy `Pohni` triedy `Kruh`:

```
void Kruh::Pohni()
{
     oldx = x;
     oldy = y;
     x += dx;
     y += dy;
     if (x > SCREEN_W)
          dx = -abs(dx);

     if (x < 0)
          dx = abs(dx);
     if (y > SCREEN_H)
          dy = -abs(dy);
     if (y < 0)
          dy = abs(dy);
}
```

Uf. Táto lekcia bola ťažká preťažká, ba priam zabitá, napriek tomu ešte nejaké úlohy:

Úloha č.1: Klasická. Vyskúšajte, pochopte.

Úloha č.2: Prečo sa dialo to mrznutie kruhov pri okraji? Prečo tá nová varianta funkcie `Pohni` funguje?

Úloha č.3: Dorobte do správcu funkciu `Zmaz`, ktorá zruší kruh zo zoznamu. (Najľahšie sa ruší prvý.) Dajte pozor na prípad, že v zozname už nezostane nič. Skúste vyriešiť prekresľovanie.

Úloha č.4: (Ťažká!) Skúste do toho dorobiť fyziku. Hmotnosť kruhov nech je približne druhá mocnina polomeru. Vyrátajte rýchlosti po zrážke tak, aby boli fyzikálne správne.

5. lekcia

Dedičnosť
alebo "Aký otec, taký syn (ledaže by nie)"

Vieme už, čo to znamená, keď jedna trieda obsahuje druhú ako atribút (tento vzťah sa volá agregácia). Vieme, ako to vyzerá, keď jedna trieda hovorí druhej do života (v minulej lekcii objekt triedy `SpravcaTelies` menil rýchlosť objektom triedy `Kruh`). Tento vzťah sa volá asociácia. Ostáva nám posledný zo vzťahov, ktorý medzi triedami môže nastať – **dedičnosť** alebo **špecializácia**.

V našom projekte momentálne žijú objekty triedy `Kruh`, ktoré môžeme pridávať ako sa nám páči. Kvôli jednoduchosti teraz nebudeme vzájomné odrážanie brať do úvahy.[17] Všetko krásne funguje, ale v tomto momente príde požiadavka z ministerstva, že kruhy nie sú jediné existujúce geometrické útvary a že by radi aj štvorčeky. A že v najbližšom čase dodajú zoznam ďalších geometrických útvarov, ktoré bude treba do projektu pridať.

Prvá vec, ktorá človeka v takej situácii napadne je, že treba vyrobiť novú triedu pre štvorce a upraviť správcu telies tak, aby mal jeden zreťazený zoznam pre kruhy a ďalší pre štvorce.[18] A tak začne vyrábať triedu štvorec. Lenže čoskoro zistí, že v podstate opisuje väčšinu vecí z triedy `Kruh`. Nová trieda `Stvorec` sa totiž správa až na detaily úplne rovnako. Tiež má nejaký „hlavný“ bod, ktorý sa pohybuje a okolo ktorého sa ten štvorec kreslí. Funkcie na nastavovanie pozície tohto bodu, zmena jeho polohy v závislosti na rýchlosti, zmena rýchlosti na hranici obrazovky – toto všetko ostáva úplne nezmenené. Všetky funkcie s výnimkou funkcií `void Nakresli()` a `Rect SpinavyObdlznik()` sú naprosto rovnaké.

A práve v momente, kedy si toto programátor uvedomí, zistí, že potrebuje takú vec, ako je dedičnosť. Uvedomí si totiž, že by bolo oveľa výhodnejšie, keby si mohol napísať nejakú základnú triedu (nech sa volá `Teleso`), ktorá by sa starala o pohyb a všetky základné veci by sa vyriešili v nej a potom by spravil dve triedy, ktoré by boli potomkami triedy `Teleso` – všetky funkcie by po nej podedili, ale pridali by si vlastné, prípadne by si niektoré funkcie svojho predka prispôsobili k obrazu svojmu. Teda triedy `Kruh` a `Stvorec` by boli len akýmsi vylepšením – špecializáciou – triedy `Teleso`. A keby sa ukázala nutnosť pridať ďalší geometrický útvar, tak by stačilo pridať ďalšieho potomka triedy `Teleso`. Väčšina roboty by už bola hotová.

Takže ako na to? Hlavičkový súbor `Teleso.h` bude vyzerať takto:

```
#ifndef TELESO_H
#define TELESO_H

#include <allegro.h>
#include "rect.h"
```

17 Áno, je nám zrejmé, že aj vynechanie funkčnosti vyžaduje istú námahu. Ale ako uvidíte ďalej, triedu `Kruh` budeme znovu prerábať pomerne zásadne, takže je to jedno.

18 Prvá vec, ktorá človeka pri podobnej požiadavke napadne je samozrejme často úplne iná. Tu hovoríme o prvej veci, ktorá sa týka toho, ako to naprogramovať.

```
class Teleso{
public:
    Teleso();

    /* Pristupove funkcie */
    void NastavBitmapu(BITMAP* bmp) {bitmap = bmp;}
    int GetDX() {return dx;}
    int GetDY() {return dy;}
    int GetX()  {return x;}
    int GetY()  {return y;}
    int GetOldX()  {return oldx;}
    int GetOldY()  {return oldy;}
    int SetDX(int i) {dx = i;}
    int SetDY(int i) {dy = i;}
    int Farba() {return farba;}
    int SetFarbu(int i) {farba = i;}
    Teleso* Dalsie() {return dalsie;}
    void SetDalsie(Teleso* t) {dalsie = t;}

    /* Funkcie objektu */
    void Pohni();

    /* Virtualne funkcie */
    virtual void Nakresli();
    virtual Rect SpinavyObdlznik();

private:
    int x,y;
    int oldx, oldy;
    int dx, dy;
    int farba;
    Teleso* dalsie;
protected:
    BITMAP* bitmap;
};

#endif
```

V tomto súbore sa až na dve novinky nedeje nič nezvyčajné. Prvá novinka je kľúčové slovo `protected`. Všetky atribúty sú definované ako `private`. To znamená, že k ním nemá prístup nikto iný. Atribút `bitmap`, ktorý označuje, do ktorej bitmapy sa má teleso nakresliť, sme dali do sekcie `protected`. Znamená to, že bude prístupný nie iba pre danú triedu, ale aj pre všetkých jej potomkov. Spravili sme to preto, lebo na jednej strane je dôležité, aby potomkovia mali prístup k vlastnej bitmape – bez toho by sa nemohli nakresliť – a na druhú stranu by bolo zbytočné písať prístupovú funkciu, pretože nie je dôvod, aby sa niekto zvonku dožadoval informácie, do ktorej bitmapy sa daný objekt triedy `Teleso` mieni nakresliť.[19]

Metódy triedy `Teleso` sú ekvivalentné podobným metódam triedy `Kruh` z predošlej lekcie. Na začiatku je konštruktor, potom funkcie na prístup k jednotlivým atribútom (všetky sú inline, pretože sú definované priamo v hlavičkovom súbore). Potom nasleduje funkcia `Pohni()` ktoré zmení pozíciu telesa.

A dostávame sa k druhej novinke – ku kľúčovému slovu `virtual`, ktoré sa nachádza pred

19 Čo sa používania kľúčových slov `private` a `protected` týka, sú rôzne stratégie ich používania. V tutoriáloch od Microsoftu sa skryté atribúty deklarujú väčšinou ako `protected` a všetci potomkovia k nim tým pádom majú priamy prístup. Ušetrí to síce niektoré problémy, ale spreneveruje sa to zásade objektového programovania, ktorá hovorí, že každý objekt má púšťať okolitý svet iba k tomu, čo je naozaj nutné a účelné. Vždy je dobré zamyslieť sa a zvoliť vhodný kompromis medzi bezpečnosťou a pohodlím.

deklaráciou funkcií `Nakresli` a `SpinavyObdlznik`. Toto slovíčko má na svedomí zaujímavé dôsledky. Povieme si o nich ale až potom, keď si vyrobíme potomkov danej triedy.

Súbor `Teleso.cpp` bude vyzerať takto:

```
#include "teleso.h"

#define min(x,y) ((x) < (y) ? (x) : (y))

Teleso::Teleso()
{
    oldx = x = random() % SCREEN_W;
    oldy = y = random() % SCREEN_H;
    dx = random() % 15;
    dy = random() % 15;
    bitmap = screen;
    farba = makecol(random() % 256,
                    random() % 256,
                    random() % 256 );
    dalsie = NULL;
}

void Teleso::Pohni()
{
    oldx = x;
    oldy = y;
    x += dx;
    y += dy;
    if (x > SCREEN_W)
        dx = -abs(dx);
    if (x < 0)
        dx = abs(dx);
    if (y > SCREEN_H)
        dy = -abs(dy);
    if (y < 0)
        dy = abs(dy);
}

void Teleso::Nakresli()
{
    putpixel(bitmap,x,y,farba);
}

Rect Teleso::SpinavyObdlznik()
{
    Rect r;

    r.SetLeft(min(x,oldx));
    r.SetTop(min(y,oldy));
    r.SetWidth(abs( x - oldx ) + 1);
    r.SetHeight(abs( y - oldy ) + 1);

    return r;
}
```

V konštruktore sa atribúty nastavia na náhodné hodnoty (okrem atribútu `bitmap`, ktorý sa nastaví na `screen` a smerníka na ďalšie teleso `dalsie`, ktorý sa nastaví na `NULL`). Metóda `Pohni` ostáva nezmenená. Aby teleso aspoň voľačo kreslilo, implementovali sme aj metódy `Kresli` (nakreslí bodku) a `SpinavyObdlznik`. (Táto funkcia predpokladá, že máme k dispozícii triedu `Rect` z predošlej lekcie.)

Poďme teraz vytvoriť nanovo triedu `Kruh`. Súbor `kruh.h` bude takýto:

```
#ifndef KRUH_H
#define KRUH_H

#include "teleso.h"

class Kruh : public Teleso
{
public:
    Kruh();
    int GetR() {return polomer;}
    void SetR(int r) {polomer = r;}
    virtual void Nakresli();
    virtual Rect SpinavyObdlznik();
private:
    int polomer;
};

#endif
```

Základom úspechu je riadok `class Kruh : public Teleso`. V ňom sa hovorí, že nová trieda `Kruh` je potomkom triedy `Teleso`. To znamená, že všetky veci z triedy `Teleso` preberá aj trieda `Kruh`. Samozrejme trieda `Kruh` bude mať aj nejaké vlastné veci. K tým veciam patrí atribút `polomer` (ktorý pôvodné `Teleso` nemalo) a jeho prístupové funkcie, konštruktor (v ktorom sa nastaví polomer) a funkcie `Nakresli` a `SpinavyObdlznik`, ktoré v triede `Kruh` budú fungovať inak.

Súbor `kruh.cpp` bude takýto:

```
#include <allegro.h>
#include "kruh.h"

#define min(x,y) ((x) < (y) ? (x) : (y))

Kruh::Kruh()
 : Teleso()
{
    polomer = 10 + random() % 50;
}


void Kruh::Nakresli()
{
    circlefill(bitmap,GetX(),GetY(),polomer,Farba());
}


Rect Kruh::SpinavyObdlznik()
{
    Rect obdlznik;
    obdlznik.SetLeft(min(GetX(),GetOldX()) - polomer - 1);
    obdlznik.SetTop(min(GetY(),GetOldY()) - polomer - 1);
    obdlznik.SetWidth(2 * polomer + 3 + abs(GetX() - GetOldX()));
    obdlznik.SetHeight(2 * polomer + 3 + abs(GetY() - GetOldY()));
    return obdlznik;
}
```

Funkcie `Nakresli` a `SpinavyObdlznik` sú rovnaké, ako boli pri kruhu (aj keď miesto atribútov `x`, `y`, … sme použili prístupové funkcie `GetX()`, `GetY()`, …) Zaujímavá je ale definícia konštruktora. Začína `Kruh::Kruh():Teleso()`. Znamená to, že predtým, než sa spustí samotný konštruktor triedy `Kruh`, sa spustí ešte konštruktor rodičovskej triedy `Teleso`, ktorý

ponastavuje jeho atribúty.

Triedu `Stvorec` spravíme podobne. Súbor `stvorec.h` bude vyzerať takto:

```
#ifndef STVOREC_H
#define STVOREC_H

#include <teleso.h>

class Stvorec : public Teleso
{
public:
    Stvorec();
    int GetA() {return strana;}
    int SetA(int a) {strana = a;}
    virtual void Nakresli();
    virtual Rect SpinavyObdlznik();
private:
    int strana;
};

#endif
```

Atribút `strana` má názov na zmätenie nepriateľa. V skutočnosti sa v ňom nachádza iba polovica dĺžky strany (a ani to nie je celkom pravda). Však si pozrite, ako je urobené v súbore `stvorec.cpp`:

```
#include "stvorec.h"

#define min(x,y) ((x) < (y) ? (x) : (y))

Stvorec::Stvorec()
 : Teleso()
{
    strana = 5 + random() % 25;
}


void Stvorec::Nakresli()
{
    rectfill(bitmap, GetX() - strana, GetY() - strana,
             GetX() + strana, GetY() + strana, Farba());
}


Rect Stvorec::SpinavyObdlznik()
{
    Rect obdlznik;
    obdlznik.SetLeft(min(GetX(),GetOldX()) - strana);
    obdlznik.SetTop(min(GetY(),GetOldY()) - strana);
    obdlznik.SetWidth(2 * strana + 1 + abs(GetX() - GetOldX()));
    obdlznik.SetHeight(2 * strana + 1 + abs(GetY() - GetOldY()));
    return obdlznik;
}
```

Teraz sa môžeme vrátiť k sľúbenému vysvetleniu kľúčového slova `virtual`. Vytvorte si súbor `dedicnost.cpp`, includnite na začiatku súboru súbory `teleso.h`, `kruh.h` a `stvorec.h`, vyrobte funkciu `main()` a inicializujte Allegro. Potom zo súboru `teleso.h` dočasne vymažte spred funkcie slovíčko `virtual`. Teraz do funkcie `main()` napíšte tieto príkazy:

```
Kruh* pk = new Kruh();
Teleso* pt = new Kruh();
pk -> Nakresli();
pt -> Nakresli();
```

Prvé priradenie je jednoduché. `pk` je smerník na kruh. Vyrobíme nový kruh a smerník `pk` na neho bude ukazovať. Druhé priradenie je podozrivejšie. Tam chceme, aby na novovyrobený kruh ukazoval smerník na `Teleso`. Ale vzhľadom na to, že `Kruh` je potomkom triedy `Teleso`, aj sám je `Teleso`[20], takže uvedené priradenie bude fungovať. Čo sa ale bude diať, keď zavoláte jednotlivé funkcie `Nakresli`? Smerník `pk` je smerník na `Kruh`, takže sa zavolá funkcia `Nakresli` z triedy `Kruh` a nakreslí sa kruh. Smerník `pt` je smerník na `Teleso`, takže sa zavolá funkcia `Nakresli` z triedy `Teleso` a nakreslí sa bodka.

Toto správanie je logické. Trieda `Kruh` v sebe schováva obe varianty funkcie `Nakresli`. Keby sme chceli zavolať funkciu `Nakresli` z triedy `Kruh` a nie z triedy `Teleso`, môžeme smerník pretypovať – zavolať to spôsobom `((Kruh*) pt) -> Nakresli();`

Problém nastane, keď začneme chcieť napísať nového správcu telies. Vyzerá to tak, že sa nevyhneme tomu, aby sme mali zvlášť zoznam pre štvorce a zvlášť zoznam pre kruhy, pretože tak či tak musíme programu povedať, ktorú z funkcií má použiť. Ale zachráni nás práve slovíčko `virtual`. Znovu ho vráťte do súboru `teleso.h` pred funkciu `Nakresli`, skoompilujte a spustite horné štyri príkazy. Na prekvapenie sa teraz objavia dva kruhy. Ak je totiž funkcia deklarovaná ako `virtual`[21], znamená to, že počas kompilácie ešte nie je jasné, čo sa bude volať. Smerník `pt` totiž môže ukazovať na `Teleso`, na `Kruh` aj na `Stvorec` a počas behu programu sa to môže meniť. A keď sa funkcia nakoniec zavolá, tak sa zavolá tá, ktorá patrí k triede určenej pri alokácii (keď teda alokujeme príkazom `new Kruh()`, zavolá sa funkcia `Nakresli` ktorá patrí triede `Kruh`).[22] Aj keď teda voláme funkciu `Nakresli` cez smerník ukazujúci na predka daného objektu, bude sa kresliť s pomocou tej funkcie, ktorá je danému potomkovi vlastné.

Takže je to v suchu. Môžeme napísať správcu telies, ktorý si bude pamätať objekty triedy `Teleso` a bez ohľadu na to, či tam budeme ukladať štvorce alebo kruhy, budú sa vykresľovať dobre a dobre sa bude počítať aj špinavý obdĺžnik.

Hlavičkový súbor `spravcatelies.h` bude takýto:

```
#ifndef SPRAVCATELIES_H
#define SPRAVCATELIES_H

#include "teleso.h"

class SpravcaTelies{
public:
    SpravcaTelies();
```

20 Každý `Kruh` je `Teleso`. Ale pozor! Nie každé `Teleso` je `Kruh`.

21 Ako `virtual` stačí označiť metódu v predkovi. Ale je slušné písať to aj v potomkoch. Nič sa tým nepokazí a programátor má informáciu viac poruke.

22 Situácia je trošku komplikovanejšia. Hierarchia predkov a potomkov môže byť zložitá a rozsiahla (niektoré knižnice napr. MFC od Microsoftu, alebo Qt od TrollTechu majú rádovo stovky tried). V prípade, že je funkcia virtuálna, zavolá sa tá jej verzia, ktorá je „najbližšie“ k tej triede, do ktorej objekt patrí.

```
    ~SpravcaTelies();
    void Pridaj(int co);
    void Krok();
    void Nakresli();
private:
    BITMAP* buffer;
    Teleso* prve;
    Teleso* posledne;
};

#endif
```

Je to rovnaký správca telies, ako v predošlej lekcii. Jediná drobná zmena je tá, že metóda `Pridaj` má parameter, ktorý jej povie, čo má vlastne pridať. Ak je parameter 0, pridá sa objekt triedy `Teleso`, ak je to 1, pridá sa `Kruh`, ak je to 2, pridá sa `Stvorec`. Implementácia v súbore `spravcatelies.cpp` bude vyzerať takto:

```
#include <allegro.h>
#include "spravcatelies.h"
#include "teleso.h"
#include "kruh.h"
#include "stvorec.h"

SpravcaTelies::SpravcaTelies()
{
    buffer = create_bitmap(SCREEN_W,SCREEN_H);
    posledne = prve = NULL;
}

SpravcaTelies::~SpravcaTelies()
{
    destroy_bitmap(buffer);
    while (prve != NULL)
    {
        Teleso* pt = prve->Dalsie();
        delete prve;
        prve = pt;
    }
}

void SpravcaTelies::Pridaj(int co)
{
    Teleso* pt;

    switch (co)
    {
    case 0:
        pt = new Teleso();
        break;
    case 1:
        pt = new Kruh();
        break;
    case 2:
        pt = new Stvorec();
        break;
    }

    pt -> NastavBitmapu(buffer);
    if (prve == NULL)
        prve = posledne = pt;
    else
    {
        posledne->SetDalsie(pt);
```

```
            posledne = pt;
        }
    }

    void SpravcaTelies::Krok()
    {
        clear_to_color(buffer,makecol(0,0,0));

        Teleso* pt;
        for(pt = prve; pt != NULL; pt = pt->Dalsie())
        {
            pt->Pohni();
            pt->Nakresli();
        }
    }

    void SpravcaTelies::Nakresli()
    {
        Teleso* pt;
        for(pt = prve; pt != NULL; pt = pt->Dalsie())
        {
         Rect obdl = pt->SpinavyObdlznik();
         blit( buffer, screen,
               obdl.GetLeft(), obdl.GetTop(),
               obdl.GetLeft(), obdl.GetTop(),
               obdl.GetWidth(), obdl.GetHeight() );
        }
    }
```

Oproti minulej lekcii je okrem metódy `Pridaj` zmenená aj metóda `Krok`. Jednak sme z nej vyhodili zisťovanie kolízií, jednak sme celú akciu rozpísali ako `Pohni` a `Nakresli`, keďže sme teraz do triedy `Teleso` nedali metódu `Krok`.

Takže už to je skoro celé hotové. Ešte do funkcie `main()` treba hodiť nejaké slušné ovládanie, napríklad takéto:

```
SpravcaTelies spravca;

while(!key[KEY_ESC])
{
    if (key[KEY_B])
        spravca.Pridaj(0);
    if (key[KEY_K])
        spravca.Pridaj(1);
    if (key[KEY_S])
        spravca.Pridaj(2);
    spravca.Krok();
    spravca.Nakresli();
}
```

Úloha č.1: Pozorne prečítajte, pochopte a vyskúšajte. (Aj s tými vecami okolo smerníkov, pridávania a vynechávania slovíčka `virtual` !!! Je to dôležité!!!)

Úloha č.2: Pridajte trojuholníky.

Úloha č.3: Spomeňte si na svoju obľúbenú strategickú hru a pokúste sa predstaviť si, ako asi by mohla vyzerať hierarchia tried objektov, ktoré sa v nej nachádzajú. Skúste rozdeliť aspoň 10 rôznych objektov z tej hry do logického stromu podľa spoločných a rozdielnych vlastností.

6. lekcia

Zákernosti so smerníkmi alebo "Na čo všetko treba dať pozor"

V tejto lekcii nič veľkolepého nevytvoríme ani nebudeme používať knižnicu Allegro. Ukážeme si nejaké finty, ktoré nám zjednodušia život a naznačíme, čoho je vhodné sa vyvarovať a čo treba ošetriť, ak vo vnútri objektu alokujeme nejakú pamäť dynamicky. Všetky príklady ale budú klasické konzolovité.

Na začiatok jedna šikovná novinka, ktorá sa volá **referencia**. Je to o tom, že nejakej premennej môžete vytvoriť iné meno. Pozrite sa na nasledujúci kúsok kódu:

```
int i;
int &referencia = i;
referencia = 5;
```

Najprv sme si vytvorili klasickú premennú `i` typu `int`. Potom sme vytvorili referenciu na premennú typu `int`, ktorú sme nazvali `referencia` a priradili jej premennú `i`. To, že sme vytvorili referenciu, sme dali vedieť tým, že sme napísali `int &`. Referenciu je treba inicializovať hneď, ako sa vytvorí (aj sme to urobili). Od tohto okamihu je `i` aj `referencia` tá istá premenná, takže ak priradím premennej `referencia` hodnotu 5, v premennej `i` bude 5.[23]

Na čo je takáto vec dobrá? Skúste si spomenúť, ako sme kedysi dávno v základnom kurze jazyka C v kapitole o smerníkoch písali funkciu `swap`, ktorá mala vymeniť obsahy dvoch premenných. Bez smerníkov sme to spraviť nevedeli. Teraz máme možnosť spraviť to aj s referenciami:

```
void swap(int &i, int &j)
{
     int k = i;
     i = j;
     j = k;
}
```

Parametre funkcie `swap` tentokrát nie sú lokálne premenné danej funkcie, ale sú to referencie. Takže ak mám v nejakej inej funkcii napríklad premenné `hugo` a `zuza` typu `int` a zavolám funkciu `swap(hugo,zuza);` tak `i` bude referencia na `hugo`, `j` bude referencia na `zuza` a keď v `i` a `j` vymením hodnoty, vymenia sa hodnoty aj v premenných `hugo` a `zuza`.

Ďalšie použiteľné a zaujímavé novinky uvidíte v nasledujúcom príklade. Vyrobíme si jednoduchú triedu `BOD`. Pozrite si pozorne zdrojový kód a skúste si najprv uvedomiť, čo je na ňom nové alebo zvláštne. Podrobný popis noviniek nasleduje za príkladom.

```
#include <iostream>
#include <math.h>

using namespace std;
```

23 Podobné veci by sme vedeli urobiť aj cez smerníky, ale takto aspoň nemusíme vypisovať hviezdičky.

```
class BOD
{
public:
    BOD() {x = 0; y = 0;}
    BOD(float x1, float y1) {Set(x1,y1);}
    void Set(float x1, float y1) {x = x1; y = y1;}
    void SetX(float x1) {x = x1;}
    void SetY(float y1) {y = y1;}
    float X() {return x;}
    float Y() {return y;}
    float Vzdialenost(float x1, float y1)
        { return(sqrt((x-x1)*(x-x1)+(y-y1)*(y-y1))); }
    float Vzdialenost(BOD b) {return Vzdialenost(b.X(),b.Y());}
private:
    float x,y;
};

main()
{
     BOD a(3.2,2.7);
     BOD *pb = new BOD();
     BOD c(a);
     BOD d;

     cout << "Suradnice bodu a su "
          << a.X() << " a " << a.Y() << endl;
     cout << "Suradnice bodu c su "
          << c.X() << " a " << c.Y() << endl;
     d = c;
     cout << "Vzdialenost bodov c a d je "
          << c.Vzdialenost(d) << endl;
     delete pb;
}
```

Hneď na začiatku môžeme vidieť prvú novinku. Includuje sa súbor `iostream`. Dokonca mu chýba aj obligátna koncovka `.h`. Súbor `iostream` je ekvivalent súboru `stdio.h` v Cčku. Objekty v ňom majú na starosti vstup a výstup, či už na konzolu, alebo do súborov. Jeho použitie (konkrétne použitie výstupného prúdu na konzolu `cout`[24]) si skomentujeme pri funkcii `main`.

Ďalší zaujímavý riadok je `using namespace std;` Hovorí sa v ňom, že v celom zvyšku súboru sa bude používať menný priestor `std`. Totiž – programátori veľkých projektov prišli na to, že sa im mená objektov či funkcií z jednotlivých knižníc prekrývajú. A tak si vymysleli menné priestory. Všetky veci zo súboru `iostream` patria do menného priestoru `std`. Mohli by sme teda všade písať `std::cout`, prípadne `std::endl`. Ale ak veci z niektorého menného priestoru budeme používať často a sme si istí, že jeho objekty sa nevolajú rovnako, ako naše, môžeme napísať `using namespace menný_priestor;` a veci z uvedeného menného priestoru sa budú používať automaticky. Stačí potom teda písať `cout` či `endl`.

Úvod súboru sme hravo prekonali, poďme sa pozrieť na triedu `BOD`. Prvé, čo je na nej hneď podozrivé je, že má dva konštruktory. Podľa čoho má program vedieť, ktorý z nich má zavolať? Odpoveď je jasná – oba konštruktory sa líšia parametrami. Jeden nemá žiadne, druhý má dva typu `float`. Ak teda zavoláte konštruktor s dvoma parametrami typu `float` (napríklad v prvom riadku procedúry `main` – tam, kde sme použili statickú deklaráciu), zavolá sa ten, ktorý rovno nastaví hodnoty. V druhom riadku pri dynamickej alokácii sme použili konštruktor bez parametrov. Nový bod na ktorý ukazuje `pb` bude mať teda súradnice `[0,0]`.

24 Áno, hádate správne, je to skratka z „console output“.

To, že môžete mať viacero funkcií s rovnakými menami, ktoré sa líšia iba vstupnými parametrami, je výhoda C++ oproti Cčku.[25] Treba ju ale používať rozumne. Nie je celkom múdre, ak dve funkcie, ktoré sa volajú rovnako, robia úplne odlišné veci. Ale ak chcete napríklad napísať funkciu, ktorá vám vyráta vzdialenosť od iného bodu, je príjemné mať z nej dve varianty, prvú, ktorá dostane súradnice druhého bodu, druhú, ktorá dostane na vstupe priamo objekt triedy `BOD`, ako sme to spravili v našej triede. Pokojne môže (ale nemusí) jedna z tých funkcií volať druhú. Ak by ale jedna funkcia `Vzdialenost` počítala vzdialenosť k nejakému bodu a druhá funkcia `Vzdialenost` vypisovala hviezdičky na terminál, bolo by to strašne mätúce.

Opusťme našu triedu `BOD` a poďme sa pozrieť, čo sa deje vo funkcii `main`. Predvádzame v nej statickú aj dynamickú alokáciu objektov. Na začiatku sme (staticky) deklarovali objekt `a` (bod `[3.2,2.7]`) Potom sme si vyrobili smerník `bp` a rovno sme ho nasmerovali na dynamicky alokovaný bod. (Ktorý bude mať súradnice`[0,0]` – volal sa konštruktor bez parametrov.) V ďalšom riadku sme spravili následnú rafinovanú statickú deklaráciu: `BOD c(a);` Tu sme použili vec, ktorá sa volá kopírovací konštruktor. Objekt `c` si skrátka skopíroval všetky atribúty objektu `a`. Však to aj vidieť na výpise. Podobne sa správa aj priradenie. Keď tam kdesi ďalej napíšeme `d = c;`, všetky atribúty objektu `c` sa skopírujú do `d`.

Úloha č.1: Vyskúšajte a pochopte. (Malo by to byť v pohode.)

Dobre. Toľko príjemný svet objektov, ktorých atribúty sú pekne staticky alokované. Poďme sa teraz pozrieť, ako to vyzerá, keď je nejaký atribút alokovaný dynamicky. Ide to tiež celkom dobre, ale ak si nedáte pozor, môžete vyrobiť zákerné chyby, ktorých hľadaním môžete stráviť dlhé hodiny. Celý zvyšok tejto lekcie bude teda hlavne o chybách. Pozrite si nasledujúci kus kódu:

```
#include <iostream>
#include <string.h>

using namespace std;

char *strdupnew(char *str)
{
    return(strcpy(new char[strlen(str) + 1], str));
}

class Retazec
{
private:
    char * pointer;
public:
    Retazec() {pointer = NULL;}
    Retazec(char *str) {pointer = strdupnew(str);}
    ~Retazec() {delete [] pointer;}
    char * Obsah() {return pointer;}
};

main()
{
     Retazec *pr1 = new Retazec("Ahoj!");
     Retazec r2;
```

25 Táto vlastnosť sa volá po anglicky **overloading**. Do slovenčiny sa to prekladá nie príliš šťastne ako **preťažovanie funkcií**.

```
        r2 = *pr1;
        cout << pr1 -> Obsah() << endl;
        cout << r2.Obsah() << endl;
        delete pr1;
        cout << r2.Obsah() << endl;
}
```

Na začiatku máme pomocnú funkciu `strdupnew`, ktorá funguje rovnako, ako Cčková funkcia `strdup` – kopíruje reťazce. Akurát, že na vytvorenie nového reťazca používa `new` miesto `malloc`, takže sa s tým pracuje pohodlnejšie a bezpečnejšie.

Ďalej si urobíme triedu `Retazec`, ktorá by mala aspoň trochu automatizovať ohavnú a nepríjemnú prácu s reťazcami z jazyka C.[26] Dobre si pozrite, ako funguje! Je to dôležité!!! Potom vo funkcii `main` vyrobíme dva reťazce. Jeden dynamicky a druhý staticky. Potom do reťazca `r2` hodíme reťazec ktorý sme alokovali dynamicky a obsah oboch reťazcov vypíšeme. (Pekne vidieť dva možné prístupy – dynamickú a statickú alokáciu – a ich použitie.) Potom zmažeme dynamicky alokovaný reťazec a obsah `r2` vypíšeme ešte raz. Dokopy by sa nám teda malo trikrát vypísať `Ahoj!`.

Úloha č.2: Vyskúšajte.

Zrada. `Ahoj!` sa vypísalo iba dvakrát.

Úloha č.3: Napriek tomu, že to hneď v ďalšom odstavci vysvetlíme, skúste prísť sami na to, kde to viazne.

Takže riešenie hádanky je takéto: Keď sme alokovali prvý reťazec, zavolal sa konštruktor so vstupným parametrom. Ten vytvoril s pomocou funkcie `strdupnew` kópiu vstupného reťazca, vyhradil pre ňu pamäť a atribút `pointer` na neho nasmeroval. Všetko v pohode. Keď sme deklarovali reťazec `r2`, neuvádzali sme žiadne parametre, takže sa zavolal konštruktor bez parametrov. Nič sa nealokovalo, nič sa nedialo. Potom sme spravili priradenie `r2 = *r1;` Priradenie skopíruje všetky atribúty a trieda reťazec má ten atribút jediný – smerník `pointer`. Od tejto chvíle ukazoval atribút `pointer` objektu `r2` na to miesto v pamäti, na ktorom si miesto pre svoju kópiu reťazca vytvoril prvý objekt, ktorý sme vytvárali dynamicky. Potiaľto bolo všetko v poriadku. (Aj keď v poriadku ako v poriadku. Keby sme zmenili čokoľvek na prvom reťazci, automaticky by sa zmenil aj druhý.) Keď sme nechali vypisovať obsah oboch reťazcov, vypísal sa ten istý kus pamäte. Ale potom sa stala zásadná vec – zrušili sme prvý objekt. A pri tej príležitosti sa zavolal jeho deštruktor, ktorý uvoľnil pamäť, v ktorej bolo ono slovko „`Ahoj!`“ uložené a premazal ju. A tak sa stalo, že `pointer` objektu `r2` zrazu ukazoval na naprosto nezmyselný kus pamäte. A keď sme od `r2` chceli, aby vypísal svoj obsah, žiaden div, že nevypísal nič.

No dobre. Príčinu by sme poznali. Ale ako z tejto šlamastiky von. Prísť o možnosť priradiť jeden objekt druhému by bola škoda. Ako to spraviť tak, aby to fungovalo? Riešenie je opäť

26 Jazyk C++ má prácu s reťazcami urobenú pomerne dobre. Tento príklad je len cvičný. Potrebujeme skrátka nejakú triedu, ktorá si kus textu (alebo čokoľvek iné) alokuje dynamicky, aby sme sa s tým mohli hrať.

v preťažení (priradení iného významu). Zaujímavé na tom je to, že teraz budeme pridávať nový význam znamienku =. V jazyku C++ sa to robí takto:

Do hlavičky triedy `Retazec` do sekcie `public` pridáme riadok

```
Retazec& operator = (Retazec& r);
```

Tým povieme, že ideme preťažiť operátor =, že vpravo bude na vstupe referencia na `Retazec` a že výsledkom operácie bude zase referencia na reťazec (aby sme mohli za sebou napísať viacero priradení). A teraz (kdesi mimo hlavičku) napíšeme, čo sa má pri takomto priradení vykonať:

```
Retazec& Retazec::operator = (Retazec &r)
{
     if (pointer != NULL)
          delete [] pointer;
        pointer = strdupnew(r.Obsah());
        return r;
}
```

Ak teda ideme reťazcu niečo priradiť, najprv skontrolujeme, či náhodou nemáme nejakú alokovanú pamäť (a ak áno, tak ju uvoľníme). Potom si s pomocou funkcie `strdupnew` skopírujeme obsah druhého reťazca ku sebe. Nakoniec vrátime referenciu na druhý reťazec, keby sme náhodou chceli urobiť viacnásobné priradenie typu `a = b = c;` .

Ak chcem v hlavnom programe túto funkciu zavolať, môžem napísať

```
r2.operator = (*pr1);
```

alebo klasicky

```
r2 = *pr1;
```

Oba tieto spôsoby sú úplne rovnocenné, oba zavolajú naše upravené priradenie a ten druhý má tú výhodu, že už je vo funkcii `main` napísaný a netreba ho meniť.[27]

Úloha č.4: Vyskúšajte.

Mohlo by sa zdať, že je všetko v poriadku. Lenže nie je. Skúste do funkcie `main` pod to prvé priradenie dopísať nevinne vyzerajúci riadok `r2 = r2;`

Úloha č.5: Pridajte a skompilujte. Prečo to robí to, čo to robí? (Odpoveď je v ďalšom odstavci, ale pozrite si ju len ak zlyhá aj debugger a vy budete prepadať depresiám, že na to neviete prísť.)

Skúste sa pozrieť na našu variantu priradenia a predstavte si, že reťazec na vstupe funkcie je ten istý, ako reťazec, do ktorého sa priradzuje. Najprv si skontrolujeme, či naša hodnota `pointer` na niečo ukazuje. Zistíme, že áno a tak to vymažeme. Lenže... keď to zničíme, zničíme to aj reťazcu, ktorého hodnotu ideme priradzovať. No a potom už nie je čo kopírovať.

Ako tomu predísť? Potrebujeme nejakým spôsobom zistiť, či reťazec na vstupe nie je ten istý, ako reťazec, ktorému priradzujeme hodnotu. A teraz prichádza na scénu ďalšie kľúčové slovo jazyka C++, slovíčko **this**. Smerník `this` je smerník na aktuálny objekt. Ak teda na začiatok našej

27 Podobne ako = sa dajú preťažiť aj iné bežné operátory, napríklad +. Bude o tom samostatná lekcia.

priradzovacej funkcie dodáme

```
if (&r == this)
     return r;
```

nášmu trápeniu je koniec. Skrátka si hneď na začiatku skontrolujeme, či smerník na ten druhý reťazec nie je náhodou smerník na mňa samého a ak je, tak nič nepriradzujeme a iba vrátime, čo máme vrátiť.

Úloha č.6: Vyskúšajte to.

Naša trieda `Retazec` sa už správa skoro tak, ako má. Ale... áno, ešte jedno ale zostáva. Skúste zrušiť (alebo odkomentovať) celý obsah funkcie `main` a dať tam miesto neho toto:

```
Retazec r1("Ahoj!");
Retazec *pr2 = new Retazec(r1);
delete pr2;
cout << r1.Obsah() << endl;
```

Nedeje sa tu dohromady nič svetoborné. Urobíme si reťazec obsahujúci `Ahoj!` (voláme konštruktor s parametrom) potom dynamicky vytvoríme druhý (použijeme kopírovací konštruktor) a vzápätí ho zmažeme. Potom by sme radi vypísať hodnotu prvého reťazca a ono – nič.

Pointa je skrytá v kopírovacom konštruktore. Ten sa správa rovnako ako priradenie – skopíruje atribúty a tým to pre neho hasne. A stane sa nám presne to, čo na začiatku pri priradení. Atribút `pointer` z oboch reťazcov ukazuje na ten istý kus pamäte. A pri zavolaní deštruktora jedného z nich (toho dynamicky alokovaného) sa vymaže a prestáva existovať aj pre ten druhý.

Liekom na tento neduh je vyrobiť si vlastný kopírovací konštruktor, ktorý novému reťazcu vytvorí jeho vlastnú kópiu dát. Takže do sekcie `public` pridajte riadok

```
Retazec(Retazec &r) {pointer = strdupnew(r.Obsah());}
```

a bude to fungovať tak ako má.

Úloha č.7: Vyskúšajte to.

Takže zhrnutie toho celého: Ak máte v triede nejaké smerníky a pamäť alokovanú dynamicky, treba zabezpečiť, aby mal každý objekt svoju vlastnú kópiu dát. Treba preťažiť priradenie (a dať pozor, aby sa pri priradzovaní objektu samému sebe dáta nezničili) a spraviť kopírovací konštruktor.

Úloha č.8: (Ťažká, pre machrov.) Trieda `SpravcaTelies` z predošlej lekcie obsahuje dynamicky alokovanú bitmapu `buffer` a dynamicky alokované dáta (zreťazený zoznam všetkých možných telies) reprezentované smerníkmi `prve` a `posledne`. Upravte ju tak, aby ste mohli pridávať a odoberať viacerých správcov a fungoval vám kopírovací konštruktor a priradenie.[28]

28 Možno sa vám bude hodiť pre telesá virtuálnu funkciu `Skopiruj`, ktorá dynamicky vytvorí kópiu daného telesa a vráti na ňu smerník.

7. lekcia

Preťažovanie operátorov alebo "Čo všetko si môžete spraviť po svojom"

V predošlej lekcii sme videli, že ak potrebujeme, môžeme klasickému operátoru priradenia (=) dať iný význam, než má obyčajne. Tiež sme sa stretli s tým, že nejaká funkcia mala viacero variant, ktoré sa líšili iba parametrami. (Napríklad konštruktory s parametrami a bez nich.) V tejto lekcii si ukážeme, čo všetko si môžeme dovoliť v C++ preťažiť. Ukáže sa, že skoro všetko.

Aby sme sa mali na čom zabávať, vytvoríme si triedu `Vektor`, ktorá bude mať dva atribúty. A to konkrétne `x` a `y`. Na začiatku bude mať táto trieda iba dva konštruktory. Jeden bez parametrov, ktorý vytvorí nulový vektor a druhý s parametrami, ktorý nastaví atribúty na konkrétne hodnoty. Na začiatku to teda bude vyzerať takto:

```
#include <iostream>

using namespace std;

class Vektor
{
    public:
        Vektor():x(0),y(0) {}
        Vektor(float x1, float y1):x(x1),y(y1) {}
    private:
        float x,y;
};

main()
{
    Vektor v(1,4);
}
```

Deje sa tu zatiaľ iba jedna novinka – atribútom v oboch konštruktoroch sa hodnota nepriradzuje, ale sa volajú ich konštruktory. Áno, aj obyčajné dátové typy, na ktoré sme zvyknutí z jazyka C majú aspoň kopírovací konštruktor a tu ho veselo využívame.[29]

Prvá vec, ktorú by sme radi spravili je, že by sme chceli mať prístup k jednotlivým atribútom. A to tak, aby sa vektor správal ako dvojprvkové pole. Skrátka chceme, aby `v[0]` bol atribút `x` vektora `v` a `v[1]` bol atribút `y` tohto vektora. (Áno, spreneverujeme sa tak zapuzdreniu dát, ale niekedy – ako napríklad v tejto triede – je to pohodlné.) A tak si preťažíme operátor `[]`. Do sekcie `public` triedy `Vektor` si pridajte `float & operator[] (int index);` – operátor vracia referenciu na premennú a má vstupný parameter – `index`. Telo funkcie bude vyzerať takto:

```
float & Vektor::operator[] (int index)
{
```

29 Samozrejme by sa vôbec nič nestalo, keby konštruktor vyzeral napríklad takto: `Vektor() {x = y = 0;}` Pri takých malých dátových typoch je to jedno. Ale ak máte triedu, ktorá v sebe obsahuje nejaké väčšie objekty, je lepšie ich inicializovať týmto spôsobom, pretože inak sa najprv zavolá konštruktor bez parametrov (ktorý ich nejako nastaví) a potom ich budete nastavovať nanovo v tele procedúry, čo zbytočne predlžuje beh programu.

```
    if (index == 0)
        return x;
    return y;
}
```

Jednoduché. Ak je `index` nula, vrátime `x`, inak vrátime `y`. Ak bude funkcia `main()` vyzerať ako v uvedenom príklade, malo by to na výstupe vypísať `6 4`.

```
main()
{
    Vektor v(1,4);
    v[0] = 6;
    cout << v[0] << " " << v[1] << endl;
}
```

Ako vidno, to, že funkcia môže vrátiť referenciu, má svoje príjemné stránky. S `v[0]` a `v[1]` môžeme vďaka tomu narábať ako s normálnymi premennými, priradzovať do nich hodnoty a vypisovať ich.

Ďalej by sme chceli vektory sčítať. Tak preťažíme operátor `+`. Výsledok operátora bude `Vektor` a na vstupe bude tiež `Vektor`. Takže deklarácia v sekcii `public` bude vyzerať takto:

```
Vektor operator+ (Vektor v1);
```

A definícia môže byť takáto:

```
Vektor Vektor::operator+ (Vektor v1)
{
    Vektor ret;
    ret[0] = x + v1[0];
    ret[1] = y + v1[1];
    return ret;
}
```

Vektor, ku ktorému pripočítavame iný, samozrejme meniť nemôžeme, pretože ide iba o to vypočítať výsledok, takže hodnoty `x` a `y` necháme tak, ako sú. Vyrobíme si pomocnú premennú `ret` typu `Vektor`, do nej súčet vektorov zrátame a vrátime ju ako výsledok. A teraz si môžeme dovoliť nasledujúce veci:

```
Vektor u(2,-1), v, w;
v = Vektor(1,4);
w = u + v;
cout << w[0] << " " << w[1] << endl;
```

Sčítanie vektorov funguje tak, ako má. Mimochodom – všimnite si priradenie v druhom riadku. Použili sme fintu, pri ktorej sme zavolali konštruktor triedy `Vektor`, ten nám vyrobil objekt a ten objekt sme potom skopírovali do objektu `v`. Je to obľúbený trik, ale ak by mal konštruktor a priradenie zaberať veľa času, treba to používať opatrne a s rozvahou.

Operátor sa vytvára ako členská metóda danej triedy, ak objekt, ktorý ideme s danou triedou používať je v operácii napravo. Ak si chceme spraviť násobenie vektora (napríklad `v`) číslom (napriklad `c`), a budeme to používať vždy v poradí `v*c`, tak triede vektor pridáme `operator *` s jedným parametrom `c`. Ak by sme ale chceli počítať aj súčin `c*v`, nastane problém. Vektor je vpravo a k triede `int` prístup nemáme, takže jej operáciu `*` preťažiť nemôžeme. Vtedy sa to rieši tak, že vytvoríme funkciu, ktorá nie je členskou funkciou žiadnej triedy, je globálna a na vstupe má dva parametre. Bude to vyzerať takto:

```
Vektor operator*(float c, Vektor v)
{
    Vektor ret;
    ret[0] = c * v[0];
    ret[1] = c * v[1];
    return ret;
}
```

a potom ju použiť napríklad takto:

```
Vektor v(1,2), w;
w = 2 * v;
cout << w[0] << " " << w[1] << endl;
```

Ono sa dá preťažiť aj klasické pretypovanie, s akým ste sa okrajovo stretli v jazyku C. Predstavte si, že z nejakých dôvodov chceme, aby sa vektor dal pretypovať na `float` a aby vtedy vyzeral ako svoja dĺžka. Stačí do sekcie `public` pridať

```
operator float() {return sqrt(x*x+y*y);}
```

a môžete veselo pretypovávať:

```
Vektor v(1,2);
float f;
f = v;
cout << f << endl;
```

Keďže `f` a `v` sú rôzneho typu, kompilátor skúsi, či môže jeden objekt nejako rozumne previesť na druhý. A keď nájde pretypovaciu funkciu, automaticky ju použije.

Takže asi tak. Preťažiť sa dá kadečo. Dávajte ale pozor, aby ste to robili zmysluplne. Ak si vytvoríte objekt `Kruh`, nebude mať veľký zmysel definovať preň sčítanie. Dbajte na to, aby to, čo naprogramujete, bolo ľahko čitateľné a malo to logiku.

Úloha č.1: Dorobte odčítanie vektorov, zabezpečte aby fungovalo násobenie číslom aj z druhej strany (`w = w * 4;`) a urobte skalárny súčin. Skúste na sčítanie a odčítanie vektorov dorobiť skratky `+=` a `-=` a na násobenie reálnym číslom dorobiť skratku `*=`.

8. lekcia

Výnimky
alebo "Robme chyby profesionálne"

Celá nasledujúca lekcia by mala byť o chybách. Ako vraví Murphy, „Mýliť sa je ľudské, ale niečo naozaj zašmodrchať, na to treba počítač.“ Ale skôr, než sa pustíme do tejto závažnej problematiky, jedna dobrá správa. Pamätáte sa ešte, ako ohavne mal jazyk C vyriešené reťazce?[30] Dobrá správa – v C++ je situácia kúsok ružovejšia. Na ukážku si pozrite nasledujúci program:

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

using namespace std;

main()
{
    string k, s = "Pod mostom";
    s[1] = 'r';
    s[4] = 'h';
    s += " dame";
    k = s + " " + s;
    cout << k << endl;
    cout << "Retazec ma dlzku " << k.length() << endl;
}
```

Skrátka – dobrí programátori na nás mysleli a vyrobili pre nás triedu `string`, s ktorou sa dá pracovať relatívne pohodlne.

Takže teraz k tým chybám. Predstavte si, že ideme implementovať[31] triedu `Zlomok`. Prvá verzia by mohla vyzerať približne takto:

```
#include <iostream>

using namespace std;

class Zlomok
{
public:
    Zlomok():citatel(0),menovatel(1) {}
    Zlomok(int a, int b):citatel(a),menovatel(b) {}
    operator float() {return (float)citatel / (float)menovatel;}
    void SetCitatel(int c) {citatel = c;}
    void SetMenovatel(int m) {menovatel = m;}
    void Vykrat();
private:
    int citatel, menovatel;
};
```

30 Ono ťažko od nízkoúrovňového jazyka takmer na úrovni assembleru akým C je čakať viac, ale aj tak to bolo hrozné.

31 Implementovať je skvelé slovo. Keby náhodou prišla inšpekcia, vy by ste práve sedeli za mašinou a niečo programovali a oni by sa pýtali, že čo robíte, vhodná odpoveď je „implementujem“.

```
main()
{
    Zlomok z(2,3);
    cout << "* " << z << " *" << endl;
}
```

Zatiaľ by bolo všetko v poriadku. Program vypisuje všetko, čo má. Skúste ale na chvíľu prepísať hlavný program takto:

```
main()
{
    Zlomok z(2,0);
    cout << "* " << z << " *" << endl;
}
```

Úloha č.1: Vyskúšajte, čo to spraví.

Áno, päť z piatich učiteľov matematiky vám povie, že deliť nulou je nemravnosť. A my by sme chceli nejakým spôsobom zabezpečiť, aby naša trieda zlomok vyhlásila nejakým spôsobom, že sa deje chyba. A práve na vyhlásenie chyby slúži príkaz `throw`. Pri vykonaní vyhlási príkaz `throw` výnimočný stav. Program prestane vykonávať činnosť, ktorú práve robí a ak sa výnimočná situácia nejako neošetrí, skončí úplne.[32] Výnimočnú situáciu budeme vyhlasovať, ak sa niekto pokúsi zlomok z nulovým menovateľom pretypovať na reálne číslo. (Vhodnejšie by bolo spustiť paniku už pri výrobe zlomku s nulovým menovateľom. Uvedený príklad je len ilustrácia mechanizmu.)

```
operator float() throw ()
    {
        if (menovatel == 0) throw;
        return (float)citatel / (float)menovatel;
    }
```

Kľúčové slovo `throw` nám tam pribudlo hneď dvakrát. Prvýkrát pribudlo do deklarácie funkcie. Je totiž potrebné, aby kompilátor vedel, z ktorých funkcií môže byť vyvolaný výnimočný stav. Zátvorky za ním určujú, akú výnimku je možné vyvolať. (Zatiaľ zostali prázdne.) Druhýkrát sa slovo `throw` nachádza priamo v kóde funkcie.

Príkaz `throw` sme použili v jeho najzákladnejšej verzii – bez parametra. Takto sa však používa málokedy. Často je totiž užitočné vedieť, k akej chybe vlastne došlo. A okrem toho nie každá chyba je dôvodom k okamžitému ukončeniu programu. Niekedy je dobré výnimku spracovať a program nechať bežať ďalej.

Na to, ako identifikovať chybu, ktorá nastala, slúži jednoduchý mechanizmus. Príkaz `throw` môže mať parameter. Týmto parametrom môže byť ľubovoľný objekt. Môže to byť napr. číslo alebo reťazec, v niektorých knižniciach sa nachádzajú už hotové triedy pre objekty vhodné ako výnimky. My si na ukážku nejaké také triedy vyrobíme. Z cvičných dôvodov budeme okrem nuly v menovateli protestovať aj vtedy, keď je hodnota zlomku 1. Základná trieda bude `Vynimka`, jej potomkovia budú triedy `DelenieNulou` a `SkaredaJednicka`. Bude to vyzerať takto:

32 Možno vás už niektorý program bežiaci pod MS Windows obšťastnil hláškou `Unhandled exception, program Aborted`. Tá vzniká práve tak, že sa vyhlásila výnimka a potom sa neošetrila.

```
class Vynimka
{
private:
    string text;
public:
    Vynimka(string s):text(s) {}
    string GetText() { return text; }
};

class DelenieNulou : public Vynimka
{
private:
    int coSmeDelili;
public:
    DelenieNulou(string s, int i):Vynimka(s),coSmeDelili(i) {}
    int GetCoSmeDelili() { return coSmeDelili; }
};

class SkaredaJednicka : public Vynimka
{
public:
    SkaredaJednicka(string s):Vynimka(s) {}
    string HanlivyVyrok() {return string("Nechceme jednicku!!!");}
};
```

Každá výnimka v sebe obsahuje text, do ktorého sa dá uložiť napríklad informácia o mieste, odkiaľ bola výnimka vyvolaná. Výnimka `DelenieNulou` má v sebe okrem toho informáciu, čo sme vlastne delili a výnimka `SkaredaJednicka` je schopná vygenerovať hanlivý výrok na adresu čísla 1.

Operátoru pretypovania teraz treba povedať, že bude generovať ako výnimky smerníky na triedu `Vynimka`. Vzhľadom na to, že aj `DelenieNulou` aj `SkaredaJednicka` sú potomkovia triedy `Vynimka`, smerníky na ne sú súčasne aj smerníky na objekt triedy `Vynimka`. Generovanie výnimiek sa udeje nasledujúcim spôsobom:

```
operator float() throw (Vynimka*)
{
  if (menovatel == 0)
    throw new
      DelenieNulou("Funkcia operator float() trieda Zlomok", citatel);
  if (citatel == menovatel)
    throw new
      SkaredaJednicka("Funkcia operator float() trieda Zlomok");
  return (float)citatel / (float)menovatel;
}
```

Ako ste si isto všimli, objekty výnimiek boli alokované dynamicky. Keby sme ich spravili statické, prestali by totiž existovať súčasne s odchodom z funkcie a nebolo by čo spracovávať.

Teraz nastáva čas prezradiť, ako výnimku zachytiť a spracovať. Slúžia na to kľúčové slová `try` a `catch`. Po slove `try` nasledujú kučeravé zátvorky, v ktorých sa nachádza kritický kus kódu (to je ten, v ktorom sa môže generovať výnimka). Po tomto bloku nasleduje viacero blokov začínajúcich slovom `catch` po ktorom v zátvorkách nasleduje, aký objekt sa má chytiť a potom v kučeravých zátvorkách kus kódu, ktorý hovorí, čo sa s tým chyteným objektom má urobiť. Prvý blok, ktorý v chytaní uspeje, sa vykoná, ostatné sa preskočia. Ak vygenerovaný objekt nechytí žiadna sekcia `catch`, program sa ukončí.

Poďme si ukázať, ako to bude fungovať v našom príklade. Nejakému zlomku budeme meniť

menovatele a budeme sa pozerať, ako sa mení jeho hodnota. Pri jej počítaní sa ale môžu vyvolať výnimky, ktoré budeme odchytávať. Kód bude vyzerať takto:

```
main()
{
    Zlomok z(2,0);

    for (int i = -2; i <= 3; i++)
    {
      try
     {
         z.SetMenovatel(i);
         cout << z << endl;
     }

     catch (DelenieNulou* v)
     {
         cout << "Delili sme nulou: " << v->GetText() << endl;
         cout << "pozadovane delenie bolo " << v->GetCoSmeDelili()
               << "/0" << endl;
         delete v;
     }

     catch (SkaredaJednicka* v)
     {
         cout << "Ten zlomok je jedna!!! " << v->GetText() << endl;
         cout << v->HanlivyVyrok() << endl;
         delete v;
     }

     catch (Vynimka* v)
     {
         cout << "Chyba, ktora nemoze nastat "
              << v->GetText() << endl;
         delete v;
     }
   }
}
```

Samotný kritický kód obsahuje iba dva riadky (nastavenie menovateľa a výpis). Ak sa počas jeho výkonu vygeneruje delenie nulou, zachytí ju prvý blok, ktorý vypíše, kde sa chyba udiala a čo sa tou nulou delilo. Ak bude zlomok rovný 1, výnimku zachytí druhý blok, ktorý tiež vypíše, kde sa to udialo a pridá aj hanlivý výrok. Ak by sa niekde generovala výnimka typu `Vynimka*`, zachytil by ju tretí blok.

Úloha č.2: Vyskúšajte. Skúste prehodiť poradie `catch` blokov tak, aby sa hneď po `try` bloku nachádzal `catch` blok zachytávajúci výnimku `Vynimka*`. Čo sa zmenilo? Prečo?

Niekedy je vhodné vyrobiť si `catch` blok, ktorý zachytí všetky vygenerované výnimky. To sa spraví tak, že namiesto typu zachytávanej výnimky sa do zátvoriek napíšu tri bodky. Blok teda bude začínať `catch (...)`

Na záver ešte ukážka „knižničnej“ výnimky. Je to výnimka `bad_alloc`, ktorá sa vyvolá, keď sa nepodarí alokovať pamäť (napríklad preto, lebo sme všetku vyčerpali).

```
#include <iostream>

using namespace std;

main()
{
    int i = 0;
    try
    {
        while (1)
       {
            new int[1024 * 1024];
           i++;
       }
    }
    catch(bad_alloc &b)
    {
        cout << "Bolo alokovaných " << i << " MB" << endl;
    }
}
```

Úloha č.3: Urobte nejakú poriadnu triedu na zlomky, ktorá bude mať sčítanie odčítanie, násobenie, delenie, krátenie, prevod na `float`, priradzovanie, … V prípade nekalostí generujte výnimky.

Pre expertov: Okrem delenia nulou skúste kontrolovať pretečenie.

9. lekcia

Šablóny
alebo "Hyper makro systém"

S makrami sme sa v našich príkladoch niekoľkokrát akoby mimochodom stretli. Makrá sú príkazy pre preprocesor. Znamená to, že ak do svojho zdrojového kódu napíšete príkaz `#define PI 3.1415926`, tak ešte pred tým, než sa spustí samotná kompilácia, prebehne program zvaný preprocesor váš zdrojový kód a všetky reťazce `PI`, ktoré po tej definícií nájde nahradí reťazcami `3.1415926`. (Reťazcu `PI` sa potom hovorí „makro“.) Príkazy pre preprocesor sú aj klasické `#include <subor>`, ktoré na určené miesto vloží daný súbor alebo dvojica `#ifdef MAKRO #endif`, ktorá spôsobí, že ak nie je makro `MAKRO` definované, celý úsek medzi nimi kompilátor ignoruje, prípadne dvojica `#ifndef MAKRO #endif`, ktorá spôsobí, že sa ňou ohraničený úšek skompiluje iba ak makro `MAKRO` definované nie je.

Makrá môžu mať aj parametre. Stretli sme sa už s definíciou

```
#define max(a,b) ((a)>(b) ? (a) : (b))
```

ktorá nám vedela nájsť väčšie z dvoch čísel. Pretože išlo o makro a nie o funkciu, fungovalo to aj pre hodnoty typu `int`, aj pre hodnoty typu `float` a ak by bolo veľmi treba, tak aj pre hodnoty typu `char`. Výraz `max(4.1,dlzka)` sa totiž iba nahratil reťazcom

```
((4.1)>(dlzka) ? (4.1) : (dlzka))
```

a typ tohto výrazu záleží od typu hodnôt `4.1` a `dlzka`.

Ľudia, ktorí navrhovali jazyk C++ sa rozhodli, že systém makier dovedú do dokonalosti a tak zaviedli šablóny (po anglicky templates). Začnime jednoduchším príkladom. Predpokladajme, že chceme nájsť maximálnu hodnotu v nejakom poli. V prípade, že by sme vedeli, že sa jedná o pole napr. celých čísel, nie je problém napísať si funkciu. Dá sa to napríklad takto:

```
int max(int pole[], int dlzka)
{
     int i,maximum = pole[0];

         for(i = 0; i < dlzka; i++)
           if (pole[i] > maximum)
                     maximum = pole[i];

         return maximum;
}
```

Čo ale robiť, ak by sme chceli spraviť funkciu, ktorá by bola podobne univerzálna, ako makro, ktoré nám tak pekne fungovalo pre dva prvky? Treba siahnuť po šablónach. Šablónová verzia funkcie bude vyzerať takto:

```
template <class T>
T max(T pole[], int dlzka)
{
     int i;
         T maximum = pole[0];
```

```
        for(i = 0; i < dlzka; i++)
          if (pole[i] > maximum)
                    maximum = pole[i];

        return maximum;
}
```

Definícia šablóny funkcie začína riadkom `template <class T>`. Programu sa tým povie, že nasledujúca funkcia bude vlastne iba šablóna. A že všade, kde sa v nej bude vyskytovať `T`, môže byť dosadená ľubovoľná trieda.[33] Keď potom kompilátor nájde niekde ďalej napríklad takýto kus kódu:

```
int p[30];
/* nejake naplnanie pola */
cout << max(p,30);
```

tak z definície poľa `p` pochopí, že trieda `T` má byť v tomto špeciálnom prípade `int`, sám pre seba si vyrobí špeciálnu variantu funkcie pre `int` a tú použije. Ak by niekde ďalej našiel kód

```
float p[30];
/* nejake naplnanie pola */
cout << max(p,30);
```

tak si vyrobí variantu pre `float`. O toto sa ale už programátor starať nemusí a ani o tom nemusí vedieť. Funkcia mu bude fungovať pre pole objektov akejkoľvek triedy, ktorá má definované porovnávanie a priradenie.

Šablóna na funkciu je príjemná vec. Čo takto spraviť si šablónu na triedu? Možno by ale bolo treba najprv povedať, na čo môže byť taká šablóna na triedu dobrá. Predstavte si, že ste už sedemkrát programovali zreťazený zoznam. Raz mal v sebe skladovať geometrické útvary, potom zase smerníky na ohavných pavúkov vo vašej skvelej hre a posledne to bol zreťazený zoznam, v ktorom boli uložené merania zo sondy vyslanej na Pluto. A to je presne tá chvíľa, kedy to programátora prestane baviť a povie si „nešlo by to naprogramovať raz navždy tak, aby sa to dalo použiť s hocijakými objektami?“ A presne vtedy prídu ku slovu šablóny na triedy.

Na ukážku si skúsime spraviť jednu takú univerzálnu triedu. Keďže ide o cvičný príklad, bude to niečo úplne jednoduché. Bude to trieda, ktorá bude reprezentovať pole desiatich prvkov zadaného typu a okrem kontroly pretečenia nebude robiť nič. Urobené to môže byť napríklad takto:

```
#include <iostream>

using namespace std;

template<class Typ>
class pole
{
public:
     pole();
     Typ& operator[](int i);
protected:
     Typ p[10];
};

template<class Typ>
pole<Typ>::pole()
{
     cout << "Dakujeme, ze pouzivate nasu skvelu triedu" << endl;
}
```

33 Treba samozrejme dať pozor na to, aby daná trieda mala dobre definované alebo preťažené operátory = a <. Oba sa totiž vo funkcii používajú. Ak funkciu použijeme pre ľubovoľný základný typ, je to samozrejme v poriadku.

```
template<class Typ>
Typ& pole<Typ>::operator[] (int i)
{
     if (i < 0 || i >= 10)
          throw new string("Pretecenie indexu v triede pole");
     return p[i];
}
```

Keďže šablónou je nielen deklarácia triedy, ale aj definície funkcií, musíme pred každou definíciou napísať `template<class Typ>`. Okrem toho ako triedu pri funkciách nestačí napísať `pole`, ale musíme uviesť triedu `pole<Typ>`, pretože výsledná trieda, ktorá sa zo šablóny vygeneruje, bude závislá od toho, aký typ sa pre ňu použije. Inak naša šablóna iba vypíše vlezlú hlášku počas behu konštruktora a vracia referencie na jednotlivé prvky poľa. V prípade, že pretečie index poľa, vygeneruje výnimku.

Šablónu môžeme použiť napríklad nasledujúcim spôsobom:

```
main()
{
     pole<int> psenicne;
     pole<float> makove;

     try
     {
          psenicne[3] = 5;
          makove[2] = 1.23;
          cout << psenicne[3] << " " << makove[2] << endl;
          cout << makove[10];
     }
     catch (string *s)
     {
          cout << "Chyba: " << *s << endl;
          delete s;
     }
}
```

Deklarovali sme si dve polia, jedno typu `int`, jedno typu `float`. Môžeme ich používať rovnako ako obyčajné polia, ale v prípade, že použijeme index za hranicou poľa, ohlási sa chyba.

Mnoho užitočných šablón je už samozrejme hotových. Štandardná zbierka takýchto šablón sa nachádza v knižnici **Standard Template Library**, ktorá zvykne byť súčasťou všetkých väčších kompilátorov C++. (A keďže ide o šablóny, v prípade, že kompilátor C++ už máte, nie je problém stiahnuť si túto knižnicu z internetu.) Povieme si o nich niečo viac v nasledujúcich lekciách.

Úloha č.1: Pochopte a vyskúšajte uvedené príklady.

Úloha č.2: Urobte šablónu funkcie, ktorá dostane na vstupe pole a jeho dĺžku (podobne ako funkcia `max` z nášho príkladu) a zistí súčet prvkov v poli. (Samozrejme to má fungovať aj pre polia typu `int`, aj pre polia typu `float`.)

Úloha č.3: Predpokladajme, že v poli z nášho príkladu sú uložené čísla. Dorobte poľu metódu, ktorá vráti ako výsledok súčet prvkov v poli.

10. lekcia

Úvod do STL alebo "Nevymýšľajte koleso"

Ako už bolo niekoľkokrát spomenuté, programátori sú jednak ľudia pomerne inteligentní a učenliví a jednak pomerne leniví. A keď už nejakú podobnú vec programujú dvanásty raz, väčšinou sa zamyslia nad tým, či si nejako neuľahčiť robotu.

Výsledkom jedného takého zamyslenia je aj knižnica Standard Template Library (Štandardná knižnica šablón) ktorú vyprodukovali vo firme Silicon Graphic Inc. (Táto firma vyprodukovala viacero zaujímavých vecí, medzi iným aj dnes už klasickú knižnicu na prácu s grafikou OpenGL.) Podľa manuálu je STL „C++ knižnicou, ktorá obsahuje triedy kontajnerov, algoritmy a iterátory a poskytuje základné algoritmy a dátové štruktúry počítačovej vedy“. V tejto náramne vznešenej úvodnej vete sa vyskytujú možno na prvý pohľad prehliadnuteľné, ale napriek tomu zaujímavé slovíčka. Prvé z nich je **kontajner**.

Kontajner je trieda, ktorá je určená na to, aby sa v nej skladovali iné triedy. Áno, tušíte dobre, napríklad pole alebo zreťazený zoznam je kontajner. A zreťazené zoznamy nebude treba odteraz krvopotne programovať, stačí použiť našu úžasnú triedu STL. A ak vám nevyhovuje zoznam, môžete sa poobzerať po nejakom inom kontajneri, STL ich obsahuje viacero. Porozprávame o nich v ďalšej lekcii.

Ďalšie zaujímavé slovíčko v úvodnej vete z manuálu je **iterátor**. Iterátor je niečo ako smerník a používa sa na to, aby ukazoval na nejaký objekt uložený v kontajneri. Iterátory sa používajú rovnako, ako smerníky, ale nemusia vedieť všetko, čo smerníky. Napríklad ak je `i` iterátor určený na čítanie nejakého zoznamu, mal by vedieť spraviť `i++`, aby sme sa dostali k ďalšiemu prvku zoznamu, ale nemusí nutne vedieť zistiť, kam ukazuje `i + 4`, čo by obyčajný smerník vedel. Detaily uvidíte pri konkrétnom prípade.

A dnešným konkrétnym príkladom je – hádate správne – zoznam. Ak ho chcete použiť, bude treba na začiatku napísať `#include <list>` Trieda `list` je ako správna súčasť STL šablóna. Pri deklarácii konkrétneho zoznamu treba teda povedať, objekty akého typu sa v ňom vlastne budú skladovať. Keby sme napríklad chceli urobiť zoznam `merania`, do ktorého by sme chceli ukladať nejaké merania teploty, pravdepodobne by deklarácia vyzerala `list<float> merania;`

Začnime teda s programovaním. Na začiatok umiestníme obligátne

```
#include <allegro.h>
#include <list>
using namespace std;
```

V dnešnom príklade si urobíme triedu `ciara`, ktorá si bude pamätať čiaru nakreslenú myšou a bude ju vedieť posúvať. Ešte predtým, ako sa pustíme do tejto triedy, spravíme si jednoduchú triedu `bod`, od ktorej budeme chcieť iba to, aby si vedela zapamätať dve súradnice a aby vedela porovnať dva body, či sú rovnaké, alebo rôzne (preťažíme operátory `==` a `!=`):[34]

34 Všetky nasledujúce veci môžete dávať do jedného súboru. Ide o ukážku a žiadnu z uvedených tried nebudeme používať nikde inde.

```
class bod
{
public:
     bod(int a, int b): x(a), y(b) {}
     bool operator== (bod& iny)
          {return x == iny.x && y == iny.y;}
     bool operator!= (bod& iny)
          {return x != iny.x || y != iny.y;}
     int x;
     int y;
};
```

Je nám jasné, že sa v tejto triede spreneverujeme dobrej zásade zapúzdrenia dát a atribúty triedy deklarujeme ako `public`. Robíme to z čírej pohodlnosti a lenivosti.

Teraz sa už môžeme venovať tvorbe našej triedy `ciara`. Táto trieda bude obsahovať jediný atribút – zoznam bodov, ktoré do čiary patria. Okrem konštruktora bude trieda obsahovať metódu `void zaznamenaj()`, ktorá bude slúžiť na načítanie čiary, metódu `void nakresli()`, ktorá čiaru nakreslí, metódu `void posun(int dx, int dy)`, ktorá každý bod posunie o určený vektor a metódu `void vymaz()`, ktorá body zo zoznamu zmaže. Deklarácia bude teda vyzerať takto:

```
class ciara
{
public:
     ciara() { vymaz(); }
     void zaznamenaj();
     void nakresli();
     void posun(int dx, int dy);
     void vymaz() { body.clear(); }
private:
     list<bod> body;
};
```

Metóda `zaznamenaj` si najprv overí, či je stlačené tlačidlo na myši a ak nie je, rovno sa ukončí. Potom vymaže prípadné staré body a uloží bod na ktorom sa práve nachádza myš do premennej `b` a s pomocou funkcie `push_back` ho pridá aj do zoznamu bodov. V hlavnom cykle, ktorý beží dovtedy, kým nebohá obeť nepustí myš, sa vždy skontroluje, či sa pozícia myši od posledného razu zmenila. Ak áno, nakreslí sa čiara od predošlého bodu do nového, nový bod sa pridá do zoznamu (opäť funkcia `push_back`). Potom sa nový bod stane starým a opäť sa čaká, kým sa myš nepohne niekam inam. Naprogramované to vyzerá takto:

```
void ciara::zaznamenaj()
{
     if (!(mouse_b & 1))
          return;

     vymaz();

     bod b(mouse_x,mouse_y);
     body.push_back(b);

     while (mouse_b & 1)
     {
          bod b1(mouse_x,mouse_y);
          if (b != b1)
          {
               show_mouse(NULL);
```

```
                line(screen,b.x,b.y,b1.x,b1.y,makecol(255,255,255));
                show_mouse(screen);
                body.push_back(b1);
                b = b1;
            }
        }
    }
```

Pri vykresľovaní čiary konečne príde k slovu iterátor. Každý kontajner má svoj vlastný typ iterátoru, takže ak chceme deklarovať iterátor, ktorý patrí k zoznamu bodov, urobíme to takto: `list<bod>::iterator smernicek;` Iterátor `smernicek` je špeciálny typ iterátora určený iba pre zoznamy bodov.

Kde zoznam bodov začína, zistíme s pomocou funkcie `body.begin()` – funkcia vracia iterátor na začiatok zoznamu. Podobne funkcia `body.end()` vracia iterátor ukazujúci za koniec zoznamu. Pozor! Tento iterátor (podobne ako smerník na NULL na konci našich klasických zoznamov) už na žiadny prvok neukazuje. Používa sa iba na to, aby ste vedeli, že už ste na konci.

V našej funkcii používame ešte funkciu `body.front()`, ktorá vráti prvý prvok zoznamu (v tomto prípade prvý `bod`). Po napísaní to vyzerá takto:

```
void ciara::nakresli()
{
    bod b = body.front();
    list<bod>::iterator smernicek = body.begin();
    smernicek++;
    show_mouse(NULL);

    while (smernicek != body.end())
    {
        bod b1 = *smernicek;
        line(screen,b.x,b.y,b1.x,b1.y,makecol(255,255,255));
        b = b1;
        smernicek++;
    }

    show_mouse(screen);
}
```

Teraz už bude jednoduché napísať funkciu, ktorá súradnice každého bodu zmení o zadaný vektor. Samozrejme s použitím iterátorov.

```
void ciara::posun(int dx, int dy)
{
    list<bod>::iterator smernicek;

    for(smernicek = body.begin();
        smernicek != body.end();
        smernicek++)
    {
        (*smernicek).x += dx;
        (*smernicek).y += dy;
    }
}
```

Vo funkcii `main` budeme vždy pracovať iba s jednou čiarou (premenná `ciarocka`). Hlavný cyklus funkcie `main` bude pracovať v dvoch režimoch. V režime „kresli“, do ktorého sa dá prepnúť tlačidlom D (ako „draw“) a v režime „ťahaj“, do ktorého sa prepína s pomocou tlačidla G (ako

„grab“). V prípade, že ste v kresliacom režime a stlačíte tlačidlo na myši, zmaže sa obrazovka a zavolá sa načítavanie novej čiary (a táto funkcia pobeží, až kým tlačidlo na myši znova nepustíte). V prípade, že ste v posúvacom režime a máte stlačené ľavé tlačidlo na myši, sú dve možnosti. Buď ste s posúvaním práve začali, alebo už posúvate. Ak ste práve začali, v premenných `oldx` a `oldy` sa uloží aktuálna pozícia myši a nastaví sa, že už posúvame. Ak už posúvame, vypočíta sa, o koľko sme sa od starej pozície posunuli. V prípade, že sme sa neposunuli, nespraví sa nič (`switch` sa zruší príkazom `break`). Ak sme sa posunuli, zavolá sa metóda `posun`, ktorá vypočíta nové súradnice všetkých bodov, celá obrazovka sa zmaže a čiara sa prekreslí. (Udržiavať špinavý obdĺžnik pre triedu `ciara` je pomerne jednoduché, tu to nerobíme kvôli stručnosti.) Funkciu `main` si poriadne naštudujte a pochopte do detailov, ako to vlastne celé funguje.

```
int main()
{
     allegro_init();
     install_keyboard();
     install_timer();
     install_mouse();

     set_color_depth(32);
     if (set_gfx_mode(GFX_AUTODETECT, 640, 480, 0, 0) != 0) {
          set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
          allegro_message("Nebeha grafika\n%s\n", allegro_error);
          return 1;
     }

     clear_to_color(screen, makecol(0,0,0));

     ciara ciarocka;

     int rezim = 0;
     bool posuvame = false;
     int oldx, oldy;
     show_mouse(screen);
     while (!key[KEY_ESC])
     {
          if (key[KEY_D])
          {
               clear_to_color(screen, makecol(0,0,0));
               rezim = 0;
          }
          if (key[KEY_G]) rezim = 1;

          switch (rezim)
          {
          case 0:
               if (mouse_b & 1)
               {
                    show_mouse(NULL);
                    clear_to_color(screen, makecol(0,0,0));
                    show_mouse(screen);
                    ciarocka.zaznamenaj();
               }
               break;
          case 1:
               if (mouse_b & 1)
               {
                    if (posuvame)
                    {
                         int dx = mouse_x - oldx;
```

```
                    int dy = mouse_y - oldy;
                    oldx += dx;
                    oldy += dy;
                    if (dx == 0 && dy == 0)
                        break;
                    ciarocka.posun(dx,dy);
                    show_mouse(NULL);
                    clear_to_color(screen,makecol(0,0,0));
                    ciarocka.nakresli();
                    show_mouse(screen);
                }
                else


                {
                    oldx = mouse_x;
                    oldy = mouse_y;
                    posuvame = true;
                }
            }
            else
                posuvame = false;
        }
    }
    return 0;
}
END_OF_MAIN()
```

Ešte pre vaše potešenie krátko zmienime ďalšie metódy, ktorými trieda `list` disponuje. Podobne, ako `front()` vráti prvý prvok zoznamu, `back()` vráti posledný. Metóda `pop_front()` zruší prvý prvok zoznamu, funkcia `pop_back()` posledný, tak ako `push_back(prvok)` pridá `prvok` na koniec zoznamu, `push_front(prvok)` pridá `prvok` jeho na začiatok. Metóda `insert(iterator,prvok)` vsunie `prvok` do zoznamu pred `iterator`. Metóda s nečakaným názvom `erase(iterator)` vymaže prvok, na ktorý ukazuje `iterator`. Metóda `remove(hodnota)` vyhodí zo zoznamu všetky prvky, ktoré majú hodnotu `hodnota`, metóda `unique()` zabezpečí, aby sa v utriedenom zozname každý jeho prvok vyskytoval vždy iba raz, metóda `reverse()` zoznam otočí a metóda `sort()` ho utriedi.[35] Ďalšie metódy a varianty uvedených metód si pozrite priamo v dokumentácií ku STL.

Úloha č.1: Pochopte a vyskúšajte.

Úloha č.2: Spravte zoznam objektov typu `string` a uložte do neho mená spolužiakov. Seba tam uložte trikrát, niekoho neobľúbeného vymažte, vymažte človeka, ktorý je v zozname na piatom mieste, zoznam utrieďte, ujednoťte (vyhádžte z neho duplicitné hodnoty), otočte a vypíšte.

Úloha č.3: Definujte v triede `bod` z tejto lekcie operátor `<` podľa toho, ako je ktorý bod vzdialený od počiatku súradnicovej sústavy. (Deklarácia bude `bool operator< (bod& iny)`) Nakreslite čiaru, potom zoznam s jej bodmi utrieďte a nakreslite ju.

35 Na to je nutné, aby bol na objektoch ukladaných do zoznamu definovaný operátor <

11. lekcia
Ďalšie kontajnery
alebo "Nie je smetiak, ako smetiak"

V predošlej lekcii sme naznačili existenciu knižnice zvanej STL, povedali sme čo-to o iterátoroch a predviedli sme jeden kontajner – konkrétne triedu `list`. Táto lekcia bude taký malý zverinec. Ukážeme si nejaké ďalšie triedy z knižnice STL (zďaleka nie všetky) a povieme niečo o niektorých ich metódach (zďaleka nie o všetkých). Prípadných záujemcov o detaily odkazujeme na manuál priamo od Silicon Graphics Inc.

Prvý, úplne jednoduchý kontajner je trieda `pair<T1,T2>`. Môžeme v nej uskladniť dvojicu čohokoľvek, `T1` a `T2` sú triedy toho čohokoľvek. K jednotlivým položkám sa dostaneme pomocou atribútov `first` a `second`. Aby sme ho mohli použiť, treba includovať súbor `utility`. Väčšinu detailov sa dozviete z nasledujúcej ukážky:

```
#include <utility>
#include <iostream>

using namespace std;

typedef pair<bool,int> cisloSoZamkom;

void vypisCisloSoZamkom(cisloSoZamkom c)
{
     if (c.first)
          cout << c.second << endl;
     else
          cout << "Je mi luto, ale toto cislo je zamknute" << endl;
}

main()
{
     cisloSoZamkom c007(true,1);
     vypisCisloSoZamkom(c007);
     c007.first = false;
     vypisCisloSoZamkom(c007);
}
```

Ďalším kontajnerom, ktorý je našťastie trošku zložitejší, je kontajner `vector`. Je to v podstate pole, ibaže má pár vylepšení. Môže meniť svoju veľkosť, dajú sa do neho pridávať a odoberať prvky. Pri deklarácii objektu tejto triedy netreba zabudnúť na to, že veľkosť poľa je parameter pre konštruktor a že sa miesto hranatých zátvoriek použijú okrúhle. Pri prístupe k jednotlivým členom sa už používajú štandardné hranaté zátvorky.

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

using namespace std;

main()
{
     vector<string> pole(7);
     pole[0] = string("Miso");
     pole[1] = string("Kristina");
```

```
    pole[2] = string("Erik");
    pole[3] = string("Zuza");
    pole[5] = string("Marek");
    pole[6] = string("Oliver");

    for (int i = 0; i < pole.size(); i++)
        cout << pole[i] << endl;

    cout << "**************************" << endl;
    pole.insert(pole.begin() + 3, string("Laco"));

    for (int i = 0; i < pole.size(); i++)
        cout << pole[i] << endl;

    cout << "**************************" << endl;
    pole.resize(6);
    pole.erase(pole.begin() + 1, pole.begin() + 4);

    for (int i = 0; i < pole.size(); i++)
        cout << pole[i] << endl;
}
```

Funkcia `size` vráti veľkosť poľa, iterátor `pole.begin() + 3` znamená „štvrtý prvok poľa“, funkcia `insert` vsunie na zadanú pozíciu nový prvok (všimnite si, že veľkosť poľa v našom príklade tým stúpne na 8), funkcia `resize` zmení veľkosť poľa (podľa potreby buď pridá nové prvky, alebo koniec poľa usekne) a funkcia `erase` zmaže prvky v zadanom intervale (tento interval je zľava uzavretý a sprava otvorený – dobre si všimnite podľa výsledku, aké prvky to vlastne zmazalo). Ďalšie zaujímavé funkcie sú napríklad `push_back`, ktorá rovnako ako pri šablóne `list` pridá jeden prvok na koniec, alebo funkcia `clear`, ktorá celé pole vymaže (počet prvkov bude 0).

Ďalším (už tretím) kontajnerom v poradí je množina (po anglicky `set`)[36]. Množina je kontajner, ktorý každý svoj prvok môže obsahovať maximálne raz. Ak teda do nej niečo vložíte niekoľkokrát, stále sa tam bude nachádzať iba jeden exemplár daného objektu. Toto skúsime využiť pri riešení klasického problému – koľkými spôsobmi možno na lístku s deviatimi číslami cvaknúť štyri dierky. Riešenie bude vyzerať takto:

```
#include <set>
#include <iostream>

using namespace std;

typedef set<int> ciselka;

main()
{
    set<ciselka> cviknutia;
    for(int i = 1; i <=9; i++)
        for(int j = 1; j <=9; j++)
            for(int k = 1; k <=9; k++)
                for(int m = 1; m <=9; m++)
                {
                    ciselka vyber;
                    vyber.insert(i);
                    vyber.insert(j);
                    vyber.insert(k);
                    vyber.insert(m);
```

36 Tenisti by sa divili.

```
                        if (vyber.size() == 4)
                            cviknutia.insert(vyber);
                    }
        cout << cviknutia.size() << endl;
}
```

Vytvorili sme si typ `ciselka` do ktorého môžeme uložiť ľubovoľnú množinu prirodzených čísel. Potom do seba vnoríme štyri cykly, v ktorých necháme štyri premenné `i`, `j`, `k` a `m` nadobúdať všetky hodnoty od 1 do 9. Každú takúto štvoricu vložíme do množiny `vyber`. Ak výsledná množina bude mať štyri prvky (čo napríklad pri variante `i=1`, `j=7`, `k=1`, `m=2` mať nebude) vložíme ju do množiny `cviknutia`. Keďže možnosti `i=3`, `j=7`, `k=1`, `m=6` a `i=7`, `j=1`, `k=6`, `m=3` nám vygenerujú tú istú množinu, bude táto množina nakoniec v cviknutiach zaradená iba raz. Na záver už iba vypíšeme, koľko rôznych cviknutí sme našli.[37]

Prvky množiny vypisujeme rovnako ako prvky zoznamu s pomocou iterátorov, metóda `begin` vráti iterátor ukazujúci na prvý prvok množiny, metóda `end` vráti iterátor ukazujúci za posledný prvok.[38] Na množiny môžeme používať klasické množinové operácie prienik (funkcia `set_intersection`), zjednotenie (`set_union`), množinový rozdiel (`set_difference`) a symetrickú diferenciu (`set_symmetric_difference`). Každá z týchto funkcií má 5 parametrov – všetko samé iterátory. Parametre sú začiatok a koniec prvej množiny, začiatok a koniec druhej množiny a piaty je tzv. výstupný operátor množiny, do ktorej sa bude ukladať výsledok. Ako si taký operátor zriadiť, uvidíte v ukážke:

```
#include <set>
#include <iostream>

using namespace std;

main()
{
     set<string> korheli, fajciari, mrtvi;
     insert_iterator< set<string> > ins(mrtvi,mrtvi.begin());

     korheli.insert(string("Jozo"));
     korheli.insert(string("Duro"));
     korheli.insert(string("Fero"));
     fajciari.insert(string("Fero"));
     fajciari.insert(string("Jozo"));
     fajciari.insert(string("Boris"));
     set_intersection( korheli.begin(),
                  korheli.end(),
                  fajciari.begin(),
                  fajciari.end(),
                  ins );
     cout << "Mrtvi:" << endl;
     for(set<string>::iterator i = mrtvi.begin(); i != mrtvi.end();
i++)
          cout << (*i) << endl;
}
```

Posledný kontajner, ktorý v tejto lekcii spomenieme, sa nazýva `map`. Po slovensky sa niekedy

37 Tento algoritmus z hľadiska efektívnosti nie je šťastný. Ide to urobiť lepšie, ale išlo nám o to, aby sme ukázali silu dátového kontajnera `set`.

38 Úplne rovnako sa dajú vypísať aj prvky kontajnera `vector`.

nazýva **asociatívne pole**, niekedy zobrazenie.[39] Je to kontajner určený na ukladanie dvojíc, pričom podľa prvej hodnoty sa v ňom dá jednoducho vyhľadávať. Klasickým príkladom na asociatívne pole je telefónny zoznam:

```
#include <map>
#include <iostream>

using namespace std;

main()
{
     map<string,int> zoznam;
     zoznam[string("Policajti")] = 158;
     zoznam[string("Hasici")] = 150;
     zoznam.insert(make_pair(string("Zachranka"), 155));

     cout << "Zachranka ma cislo "
          << zoznam[string("Zachranka")] << endl;

     string profi("Potapaci");
     if (zoznam.count(profi))
          cout << "Potapaci maju cislo" << zoznam[profi] << endl;
     else
          cout << "Potapacov v zozname nemame" << endl;

     cout << "Potapaci:" << zoznam[profi] << endl;

     if (zoznam.count(profi))
          cout << "Potapaci pribudli" << endl;

     zoznam.erase(profi);

     map<string,int>::iterator i = zoznam.find(string("Hasici"));
     cout << (*i).first << " " << (*i).second << endl;
     i = zoznam.find(profi);
     if (i == zoznam.end())
          cout << "Potapacov uz zase nemame" << endl;
}
```

Metóda `count` zistí, či sa daný kľúč v poli nachádza. Všimnite si podivné správanie operátoru `[]`. Ak sa v asociatívnom poli dožadujeme prvku s neexistujúcim kľúčom, tak tam prvok s takým kľúčom pridá, aj keď predtým neexistoval. Ak sa tomu chcete vyhnúť, tak pred prístupom testujte s pomocou `count`, alebo použite metódu `find`. Tá chce na vstupe kľúč a vráti iterátor ukazujúci na patričný prvok asociatívneho poľa. (Pozor! Tento prvok je objekt triedy `pair` a k jeho prvkom treba pristupovať jednotlivo.) Ak sa kľúč v poli nenachádza, metóda `find` vráti iterátor ukazujúci na koniec asociatívneho poľa. Metódou `erase` sa dá prvok zo zoznamu vymazať (ako parameter sa dá použiť kľúč aj iterátor).

Takže tak. Knižnica STL vám v mnohých prípadoch môže uľahčiť život a ušetriť kopu času.

Úloha č.1: Všetko to pochopte a vyskúšajte.

Úloha č.2: Príklad počítajúci všetky možnosti cvaknutia lístka prerobte tak, aby tie možnosti aj vypísal.

39 Matematici tomu hovoria aj funkcia. V tomto prípade na konečnom definičnom obore.

Úloha č.3: Spravte program, v ktorom použijete aj ďalšie množinové operácie (zjednotenie, rozdiel a symetrickú diferenciu). Urobte to tak, aby ten program aj vypisoval nejaké rozumné výsledky.

Úloha č.4: Spravte asociatívne pole, v ktorom bude pre názvy jednotlivých mesiacov uložené, koľko má daný mesiac dní. Nechajte si vypísať počet dní pre mesiac jaguár (to nie je tlačová chyba!) a potom pole opäť uveďte do pôvodného stavu.