Pokročilé programování
v jazyce C pro chemiky
(C3220)

Ukazatele v C++, virtuální metody a polymorfismus

Dynamická alokace paměti

- Jazyky C a C++ poskytují programu možnost vyžádat si část volné operační paměti pro umístění proměnné, pracovat s ní a pak ji zase vrátit
- V jazyce C se paměť přidělí voláním funkce `malloc()`, v C++ je tato funkce nahrazena operátorem **new**
- Operátor **new** vrací ukazatel na přidělenou paměť (tj. adresu v paměti, kde začíná přidělená oblast paměti), při chybě vrací nulový ukazatel
- V C++ používáme místo NULL známého z C přednostně klíčové slovo **nullptr**, ve starších verzích (před C++11) nulu (**0**)

```
int main()
{
  // Ukazatele definujeme tak, ze pred jmenem promenne uvedeme *
  // Ukazatele inicializujeme nulovou hodnotou
  int *a = nullptr;

  a = new int;                    // Prideleni pameti v C++
  if (a == nullptr) return 0;  // Otestujeme uspesnost alokace
    // Tady bude dalsi kod
}
```

Operátor new

- Operátor **new** lze použít pro přidělení paměti proměnným základních typů (`int`, `double`, ...) i objektovým proměnnným
- Použijeme-li operátor **new** pro vytvoření objektové proměnné, je ihned po alokaci paměti zavolán konstruktor
- Pokud chceme zavolat konstruktor, který přijímá parametry, můžeme mu příslušné hodnoty předat – uvedeme je v závorkách za jménem třídy

```
// Na zacatku je deklarovana trida Circle
int main()
{
  Circle *circle1 = nullptr, *circle2 = nullptr, *circle3 = nullptr;
  circle1 = new Circle;   // Bude se volat konstruktor bez parametru
  circle2 = new Circle(); // Take se bude volat konstruktor bez parametru
  circle3 = new Circle(150, 200, 50, 3); // Vola se konstruktor s parametry
}
```

Ukazatele a operátor ->

- Pro přístup ke členům třídy u ukazatelů používáme operátor ->

```
  // Tady je nekde deklarovana trida Circle
int main()
{
  Circle *circle = nullptr;

  circle = new Circle;

  circle->setValues(150, 200, 50, 3); // U ukazatelu pristupujeme ke clenum
                                      // tridy pomoci operatoru ->

  circle->x = 300;  // Operator -> muzeme take pouzit pro pristup k datovym
                    // clenum tridy. Clenska promenna x by vsak musela byt
                    // v tomto pripade definovana v sekci public
}
```

Operátor delete

- Operátor **delete** používáme pro uvolnění paměti, která byla alokována operátorem **new** (v jazyce C se používá funkce **free()**)
- Při použití operátoru **delete** je nejdříve zavolán destruktor objektu a poté je paměť uvolněna
- Po uvolnění paměti je vhodné ukazatel vynulovat, aby neukazoval na neplatnou adresu

```
int main()
{
  Circle *circle = 0;

  circle = new Circle;

  // Tady pracujeme s promennou circle

  // Kdyz uz promennou dale nepotrebujeme, uvolnime pridelenou pamet
  // operatorem delete
  delete circle;
  circle = nullptr;   // Odstranime neplatnou adresu z ukazatele
}
```

Chytré ukazatele

- Použití holých ukazatelů a new/delete je nebezpečné a často vede k těžko odhalitelným chybám (použití neplatného ukazatele, přístup ke smazanému objektu, memory leak kvůli chybějícímu delete, atd.)
- Operátory **`new/delete`** se tedy v moderním C++ používají výjimečně
- Pro správu dynamicky alokované paměti jazyk C++ nabízí tzv. **chytré ukazatele**, které samy zajistí korektní uvolnění paměti ve chvíli, kdy přestane být potřeba
- Pro přístup k chytrým ukazatelům přidejte **`#include <memory>`**
- **Holé ukazatele** v C++ používáme prakticky jen tehdy, když potřebujeme odkazovat na nějaký objekt, aniž by to ale zároveň vyjadřovalo vlastnictví, například v situacích jako tato:
  - Třída **`Drawing`** obsahuje objekty třídy **`Circle`**
  - Třída `Circle` ale zároveň potřebuje vědět, v jakém objektu `Drawing` je uložena
  - Do třídy `Circle` tedy přidáme ukazatel **`Drawing *parent`**, který nastavíme v konstruktoru `Circle(p, …) : parent(p), …`
  - Potom např. **`Circle::set()`** může volat **`drawing->draw()`** k překreslení obrázku při změně parametrů kružnice

Chytrý ukazatel unique_ptr

- Ukazatel **unique_ptr<T>** (kde T je typ odkazované proměnné) slouží k uchování odkazu na dynamicky alokovanou paměť
- V jednu chvíli může existovat nejvýše jeden unique_ptr odkazující na daný blok paměti
- Odkazovaná paměť je automaticky uvolněna při zrušení chytrého ukazatele
- K vytvoření ukazatele unique_ptr můžeme použít **new T** nebo **make_unique<T>()**
- Při použití new už ale nevoláme delete (unique_ptr si alokovanou paměť přivlastní)

```
#include <memory>
int main()
{
  unique_ptr<Circle> circle1(new Circle);
  unique_ptr<Circle> circle2; // Prazdny chytry ukazatel

  circle2 = make_unique<Circle>(); // Vytvorime novy objekt, v zavorkach
                                   // mohou byt argumenty konstruktoru

  // Tady pracujeme s promennymi circle1 a circle2,
  // jako by to byly hole ukazatele (tedy circle1->set() apod.)

  circle1 = nullptr; // Vynuti smazani odkazovaneho objektu
  // Objekt odkazovany circle2 bude smazan na konci funkce
}
```

Přiřazování unique_ptr

- Ukazatele **unique_ptr** nelze kopírovat (nebyly by pak už unikátní)
- Můžeme je ale přesouvat pomocí **cil = move(zdroj)**
- Pomocí **move()** bude zdrojový objekt přesunut do cílového ukazatele, takže zdrojový ukazatel zůstane prázdný
- Stejným způsobem lze přesouvat i mnohé další objekty ze standardní knihovny (např. přesunout celý `vector` nebo dlouhý `string` bez kopírování)

```
int main()
{
  unique_ptr<Circle> circle1(new Circle);
  unique_ptr<Circle> circle2; // Prazdny chytry ukazatel

  circle2 = circle1; // CHYBA! Toto se nezkompiluje,
                     // unique_ptr nelze kopirovat!

  circle2 = move(circle1);
  if (!circle1) {
    cout << "OK, circle1 je nyni prazdny" << endl;
  }
}
```

Chytrý ukazatel shared_ptr

- Ukazatel **shared_ptr** se používá podobně, jako unique_ptr, ale lze jej přiřazovat
- Při přiřazení shared_ptr oba výsledné ukazatele ukazují na tentýž objekt v paměti
- **shared_ptr** si interně sleduje, kolik odkazů na daný dynamicky alokovaný objekt právě existuje. Při zrušení posledního shared_ptr je odkazovaný objekt automaticky smazán

```
#include <memory>
int main()
{
  shared_ptr<Circle> circle1(new Circle);
  shared_ptr<Circle> circle2; // Prazdny chytry ukazatel

  circle2 = circle1; // OK, oba ukazatele ted ukazuji na totez

  circle1 = nullptr; // Zrusime prvni odkaz

  // Stale muzeme pracovat s circle2
}
```

Překrytí metod předka

- Pokud definujeme v odvozené třídě metodu stejného jména jako v základní třídě, dojde k překrytí této metody
- Toto překrytí se však projeví pouze v metodách potomka, v metodách předka je stále volána původní metoda předka

```
class Shape
{
  public:
    void printClass() const { cout << "Shape\n"; };
    void test()  { printClass(); };
};

class Circle : public Shape
{
  public:
    void printClass() const { cout << "Circle\n"; };
};

int main()
{
  Circle circle;
  circle.printValues();  // Vola se metoda Circle::printClass(), ktera
                         // prekryla metodu Shape::printClass()
  circle.test();   // Vola se metoda Shape::test(), kterou trida Circle
                   // zdedila od Shape.
                   // Tato metoda vsak zavola Shape::printClass()
}
```

Virtuální metody

- Deklarujeme-li překrytou metodu s klíčovým slovem **virtual**, bude i v metodách předka volána příslušná virtuální metoda potomka
- Metodu musíme deklarovat jako virtuální v předkovi i v potomkovi

```
class Shape
{
  public:
    virtual void printClass() const { cout<< "Shape"; };
    void test() { printClass(); };
};

class Circle : public Shape
{
  public:
    virtual void printClass() const { cout << "Circle" };
};

int main()
{
  Circle circle;
  circle.printValues();  // Vola se metoda Circle::printClass(), ktera
                         // prekryla metodu Shape::printClass()
  circle.test();   // Vola se metoda Shape::test(), kterou trida Circle
                   // zdedila od Shape. Tato metoda vsak zavola
                   // Circle::printClass(), protoze ta je virtualni.
}
```

Ukazatele a nevirtuální metody

- Všechny ukazatele obsahují adresu v paměti bez ohledu na typ ukazatele; to umožňuje přiřadit jednomu ukazateli hodnotu druhého ukazatele
- Typ ukazatele poskytuje překladači informaci o datových položkách a metodách objektu, na který ukazatel ukazuje
- Pokud bychom ukazateli přiřadili ukazatel na proměnnou odlišného typu (např. do `int*` přiřadíme `Circle*`), došlo by k nesprávnému chování programu, proto překladač toto nedovolí
- Je však možné přiřadit ukazateli základní třídy hodnotu ukazatele odvozené třídy, v takovém případě však při práci s ukazatelem na základní třídu bude přistupováno pouze ke členům základní třídy

```
// Zde budou definice trid Shape a Circle, kazda z nich bude mit
// definovanou nevirtualni metodu printClass() (viz dve stranky zpet)
int main()
{
  Shape *shape = 0;
  Circle *circle = new Circle;
  shape = circle;

  shape->printClass();      // Zavola se metoda Shape::printClass()
  circle->printClass();     // Zavola se metoda Circle::printClass()
}
```

Ukazatele a virtuální metody

- Použijeme-li virtuální metody, obsahuje každý objekt informace o virtuálních metodách objektu (interní tabulku virtuálních metod)
- Tabulka virtuálních metod je k objektu přiřazena v okamžiku přidělení paměti (při použití dynamické alokace to je ihned po použití **new**)
- V okamžiku volání metody se z tabulky vybere příslušná metoda odpovídající původnímu objektu a to i v případě, že jsme ukazatel přiřadili do ukazatele základní třídy

```
// Zde budou definice trid Shape a Circle, kazda z nich bude mit
// definovanou virtualni metodu printClass() (viz. dve stranky zpet)
int main()
{
  Shape *shape = 0;
  Circle *circle = new Circle;  // Zde se objektu priradi interni tabulka
                                // virtualnich metod tridy Circle
  shape = circle;

  shape->printValues();  // V tabulce virtualnich metod objektu
                         // se vyhleda ta virtualni metoda printClass(),
                         // ktera odpovida tride pouzite pro vytvoreni
                         // objektu, takze se zavola Circle::printClass()
}
```

Využití virtuálních metod

- Typickým příkladem využití virtuálních metod je situace, kdy potřebujeme pracovat se směsí různých objektů (např. `Circle`, `Rectangle`, `FilledCircle`), které mají společného předka (např. Shape)
- V takovém případě vytvoříme pole ukazatelů na předka a dynamicky alokujeme jednotlivé objekty, které přidáváme do pole
- Pomocí virtuálních metod můžeme docílit odlišného chování jednotlivých prvků v poli
- Mluvíme o **polymorfismu**: ukazatele stejného deklarovaného typu (Shape *) se chovají různě podle toho, jaký je skutečný typ odkazovaného objektu (`Circle`, `Rectangle`, ...)
- Třída potomka může tedy v jakékoli situaci zastoupit třídu předka
- Stejně funguje polymorfismus u referencí (Shape & může být svázána s objektem typu `Circle`)

Zpracování virtuálních metod překladačem

- U nevirtuálních metod je již v době překladu rozhodnuto, která metoda bude volána – mluvíme o časné vazbě
- U virtuálních metod je teprve v okamžiku volání metody vyhodnoceno která metoda se bude volat – mluvíme o pozdní vazbě
- Virtuální metody jsou v jazyce C++ široce používány v souvislosti s knihovnami tříd, kdy odvodíme vlastní třídu jako potomka knihovní třídy a v něm definujeme ty virtuální metody, jejichž chování chceme pozměnit

Polymorfismus a destruktory

- Využíváme-li polymorfismus (třída má virtuální metody), musí mít všechny třídy v dané hierarchii dědictví virtuální destruktor (i v případě, že má destruktor prázdné tělo)
- Pokud jsou destruktory prázdné, stačí definovat jeden v základní třídě (např. Shape), kompilátor sám vyrobí destruktory odvozených tříd

```
class Shape
{
  public:
    virtual void printClass() const { cout<< "Shape"; };
    virtual ~Shape() {}; // Bez destruktoru by měl program nedefinované chování
};

class Circle : public Shape
{
  public:
    virtual void printClass() const { cout << "Circle" };
    // virtual ~Circle(); {} není třeba psát, bude vygenerován automaticky
};

int main()
{
  Shape *obj = new Circle;
  // …
  delete obj; // Pokud by destruktor nebyl virtuální, došlo by zde
              // k nesprávnému zavolání ~Shape() na objektu typu Circle
}
```

Cvičení

1. Vytvořte program vycházející z úlohy 3 ze cvičení 4 (načítání grafických objektů ze souboru `shapes2.dat` a jejich ukládání do vektorů). Program upravte následujícím způsobem:
   - Do třídy Shape a všech tříd od ní odvozených přidejte metodu `printClass()` (ta vypíše jméno třídy, v níž je definována)
   - Ve třídě Shape vytvořte metodu `test()`, která jen volá `printClass()`
   - Předtím než zavoláte metodu `draw()`, volejte vždy metodu `test()`, postupně pro všechny grafické objekty (kružnice, obdélníky, vyplněné kružnice) a sledujte výpis
   - Všechny metody `printClass()` změňte na virtuální a porovnejte výpis s předchozím
   - Program dále modifikujte tak, že grafické objekty budou alokovány dynamicky pomocí chytrých ukazatelů a ukládány do polí chytrých ukazatelů (`vector<unique_ptr<Circle>>`, ...). (*Při vkládání do vektoru budete potřebovat `move()`.)* **2 body**
2. Předchozí program modifikujte následovně:
   - Všechny grafické objekty budou alokovány dynamicky a ukládány do jednoho společného pole `vector<unique_ptr<Shape>>`
   - Zároveň metody `readFile()`, `draw()` a případně `printValues()` (pokud ji používáte) všech grafických objektů budou implementovány jako virtuální. *(Bude potřeba přidat prázdnou `Shape::draw()`.)* **2 body**