

**Pokročilé programování  
v jazyce C pro chemiky  
(C3220)**

**Operátory new a delete,  
virtuální metody**

# 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 pro přidělení paměti používá operátor **new** (v C se paměť přidělí voláním funkce `malloc()`)
- Operátor **new** **vrací ukazatel** na přidělenou paměť (tj. adresu v paměti kde začíná přidělená oblast paměti)
- Není-li k dispozici dostatek paměti, vrátí operátor **new** hodnotu 0
- Adresu přidělené paměti zpravidla přiřadíme do ukazatelové proměnné

```
int main()
{
    // Ukazatele definujeme tak ze pred jmenem promenne uvedeme *
    // Ukazatele inicializujeme nulovou hodnotou,
    // v C++ obvykle pouzivame 0 misto NULL pouzivaneho v C
    int *a = 0;

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

# Operátor new

- Operátor **new** lze použít pro přidělení paměti proměnným základních typů (`int`, `double`, ...) a také objektovým proměnným
- 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 začátku je deklarována třída Circle
int main()
{
    Circle *circle1 = 0, *circle2 = 0, *circle3 = 0;
    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 = 0;

    circle = new Circle;

    circle->setValues(150, 200, 50, 3); // U ukazatelu pristupujeme ke clenum
                                        // tidy 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 místo toho používá funkce **free()**)
- Při použití operátoru **delete** je nejdříve zavolán destruktork objektu a poté je paměť uvolněna
- Po uvolnění paměti je vhodné přiřadit ukazateli hodnotu 0 aby neukazoval na neplatnou adresu

```
// Tady je nekde deklarovana trida Circle

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 = 0;    // Do circle priradime 0 aby neobsahovala neplatnou adresu
}
```

# Pole ukazatelů

- V praxi často potřebujeme pracovat s polem ukazatelů
- Pole ukazatelů často používáme v situaci, kdy načítáme objekty (např. ze souboru) a postupně dynamicky alokujeme paměť pro jednotlivé objekty

```
// Tady je nekde deklarovana trida Circle
const int MAX_GRAPHIC = 1000;

int main()
{
    int circlesCount = 0;    // Tady bude pocet nactenych kruznic
    Circle *circles[MAX_GRAPHIC]; // Definujeme pole ukazatelu

    while(/*nejaka podminka cyklu*/)
    {
        circles[circlesCount] = new Circle;
        circlesCount++;
    }

    // Zde by se provadely nejake operace s polem circles[]

    // Nakonec musime pridelenou pamet uvolnit
    for (int i = 0; i < circlesCount; i++)
        {delete circles[i]; circles[i] = 0;}
    circlesCount = 0;
}
```

# Kontejner ukazatelů

- Místo polí je výhodnější použít kontejner `vector`, který dokáže zvětšit svoji velikost

```
#include <vector>

// Tady je nekde deklarovana trida Circle

int main()
{
    Circle *circle = 0;    // Definujeme pomocny ukazatel
    vector<Circle*> circlesContainer; // Definujeme kontejner ukazatelu

    while(/*nejaka podminka cyklu*/)
    {
        circle = new Circle;
        circlesContainer.push_back(circle);
    }

    for (int i = 0; i < circlesContainer.size(); i++)
        circlesContainer[i]->printValues(); // Vypiseme informace o kruznici

    // Tady musime pridelenou pamet uvolnit
    for (int i = 0; i < circlesContainer.size(); i++)
        delete circlesContainer[i];
    circlesContainer.clear(); // Metoda clear() nastavi velikost
                             // kontejneru na 0
}
```

# 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 Graphic
{
    public:
        void printValues() { cout << "Metoda Graphic::printValues\n"; };
        void test() { printValues(); };
};

class Circle : public Graphic
{
    public:
        void printValues() { cout << "Metoda Circle::printValues\n"; };
};

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



# Virtuální metody

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

```
class Graphic
{
    public:
        virtual void printValues() { cout<< "Metoda Graphic::printValues"; };
        void test() { printValues(); };
};

class Circle : public Graphic
{
    public:
        virtual void printValues() { cout << "Metoda Circle::printValues" };
};

int main()
{
    Circle circle;
    circle.printValues(); // Vola se metoda Circle::printValues(), ktera
                        // prekryla metodu Graphic::printValues()
    circle.test();      // Vola se metoda Graphic::test(), kterou trida Circle
                        // zdedila od Graphic. Tato metoda vsak zavola
                        // Circle::printValues(), 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 objektu a metodách na které 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 přístupováno pouze ke členům základní třídy

```
// Zde budou definice tríd Graphic a Circle, kazda z nich bude mit
// definovanou nevirtualni metodu printValues() (viz. dve stranky zpet)
int main()
{
    Graphic *graphic = 0;
    Circle *circle = new Circle;
    graphic = circle;

    graphic->printValues();    // Zavola se metoda Graphic::printValues()
    circle->printValues();    // Zavola se metoda Circle::printValues()
}
```

# 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 tríd Graphic a Circle, kazda z nich bude mit
// definovanou virtualni metodu printValues() (viz. dve stranky zpet)
int main()
{
    Graphic *graphic = 0;
    Circle *circle = new Circle; // Zde se vytvori interni tabulka
                                   // virtualnich metod

    graphic = circle;

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

# Využití virtuálních metod

- Typickým příkladem využití virtuálních metod je situace kdy potřebujeme pracovat se souborem různých objektů (např. `Circle`, `Rectangle`, `CircleFilled`) které mají společného předka (např. `Graphic`)
- 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

# Využití virtuálních metod

```
// Zde bude definována třída Graphic a z ní odvozené Circle a Rectangle
const int MAX_GRAPHIC = 1000;

int main()
{
    int graphicsCount = 0;
    Graphic *graphicsContainer[MAX_GRAPHIC];

    while(/*nejaka podminka cyklu*/)
    {
        if (/*nejaka podminka*/) {
            Circle *circle = new Circle;
            graphicsContainer[graphicsCount] = circle;
            graphicsCount++;
        } else if (/*nejaka podminka*/) {
            Rectangle *rectangle = new Rectangle;
            graphicsContainer[graphicsCount] = rectangle;
            graphicsCount++;
        }
    }

    for (int i = 0; i < graphicsCount; i++)
        graphicsContainer[i]->draw(dev);    // draw() je virtualni metoda
}
```

# Zpracování virtuálních metod překladačem

- U **nevirtuálních** metod je již v době překladače 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 knihovnamí tříd, kdy odvodíme vlastní třídu jako potomka knihovnické třídy a v něm definujeme ty virtuální metody, jejichž chování chceme pozměnit

# Dodržujte následující pravidla

- Všechny ukazatelové proměnné inicializujte hodnotou 0
- Nezapomeňte uvolnit paměť alokovanou operátorem `new` poté co již alokované proměnné nebudete potřebovat. K uvolnění paměti použijte operátor `delete` a ukazatelovým proměnným poté přiřadte hodnotu 0 aby bylo jasné, že ukazují na neplatnou adresu.

# Cvičení

1. Vytvořte program vycházející z úlohy 1 ze cvičení 6 (načítání grafických objektů ze souboru a jejich ukládání do obyčejných polí), pro vykreslování však nepoužívejte šablonu. Program upravte následujícím způsobem:
  - Ve třídě `Graphic` vytvořte metodu `test()` která volá metodu `printValues()` (ta vypíše jméno třídy a metody, např. `Circle::printValues`).
  - Předtím než zavoláte metodu `draw()` volejte vždy metodu `test()`, postupně pro všechny grafické objekty (kružnice, obdélníky, vyplněné kružnice) a sledujte výpis
  - Všechny metody `printValues()` změňte na virtuální a porovnejte výpis s předchozím
  - Program dále modifikujte tak, že grafické objekty budou alokovány dynamicky a ukládány do pole ukazatelů **2 body**
2. Předchozí program modifikujte následovně:
  - Všechny grafické objekty budou alokovány dynamicky a ukládány do jednoho pole ukazatelů `Graphic*`
  - Zároveň metody `readFile()`, `printValues()` a `draw()` budou implementovány jako virtuální **2 body**
3. Program modifikujte tak aby namísto pole ukazatelů používal kontejner `vector` s ukazateli. **1 bod**