

PB161 – Programování v jazyce C++

Objektově Orientované Programování

Principy OOP – zapouzdření,
konstruktory, destruktory, const

Organizační - materiály

● Slidy a příprava

- http://cecko.eu/public/pb161_cviceni
- pro další týden typicky dostupné v pátek odpoledne
 - včetně zadání domácího úkolu
- odpoledne / večer po přednášce aktualizované

● Video nahrávky

- <https://is.muni.cz/auth/el/1433/podzim2014/PB161/um/vi/>
- do dvou dnu po přednášce (automaticky)

● Twitter

- <https://twitter.com/rngsec>
- zveřejnění přípravy a slidů, občasné info
- (opravdu důležité věci budou rozesílány hromadně na IS mail)
- (<http://www.robertrmorris.org/pavlovpoke> pro zavislé na FB, T...)

Studentští poradci

● Studentští poradci

- Hlavní počítačová hala, notebookové místo u kopírky
- dostupní pravidelně od tohoto týdne
- časy na <http://cecko.eu/public/pb161>

● Kudos

- pokud vám poradce dobře poradí, můžete mu udělit pochvalu:
<https://is.muni.cz/auth/cd/1433/podzim2014/PB161/kudos>

Vnitrosemestrální test – domluva termínu

- Forma testového papírového odpovědníku
 - stejně jako v PB071
 - celkově max. 20 bodů
 - 2 termíny pro přihlášení (16-17, 17-18)
- Bude obsahovat náplň předchozích přednášek
 - co udělá, vypíše, způsobí zadaný kód
- Možnosti termínu:
 - 27.10.
 - 3.11.



Co nás dnes čeká...

- Více o objektovém návrhu
- Koncept třídy v syntaxi C++
- Princip a implementace zapouzdření

Tvorba softwarové architektury

- Jde o proces tvorby aplikace
- Jak rozdělit zadaný problém do oddělených částí?
- Jak definovat rozhraní mezi těmito částmi?
- Jak definovat rozhraní mezi aplikací a okolím?
- Jak definovat komunikaci mezi částmi?
- Jak to všechno správně naprogramovat?
- Existuje více přístupů, OOP jedním z nich

Objektově orientované programování

- Programovací styl
 - pro zvýšení robustnosti, udržitelnosti a rozšiřovatelnosti kódu
- Centrováno kolem myšlenky „objektu“
 - kombinace dat (atributů)
 - a funkcí (metod) pro práci s nimi
 - umožňuje logicky svázat data a funkce, které s nimi pracují
- „Objekt“ je do jisté míry autonomní
 - obsahuje vše, co potřebuje ke své činnosti
 - s výjimkou interakce s ostatními objekty
 - např. konkrétní člověk v rámci lidské společnosti

Objektově orientované programování (2)

- Tento styl programování poskytuje
 - ochranu částí kódu a dat (zapouzdření)
 - implementace specializace rozhraní (dědičnost)
 - silnou typovost s rozšiřitelností (polymorfismus)
- Přímá podpora v syntaxi některých jazyků
 - čistě OOP jazyky (Smalltalk) vs. smíšené (C++)

Strukturovaný přístup k řešení problému

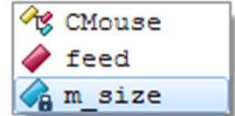
- Např. klasické C (ale pořád možno i v C++)
- Přemýšlíme o funkcích, které mají vstup a výstup
- Kroky při vývoji:
 - **Abstrakce** – navrhujeme procesy, které řeší problém
 - **Dekompozice** – rozdělíme procesy do menších podčástí (funkcí)
 - **Propojení** – implementujeme funkce a jejich vzájemné volání
- Nejprve definujeme procesy a datové struktury
- Poté definujeme sadu funkcí realizující proces a pracujících s datovými strukturami

Objektově orientovaný přístup k řešení

- Např. OOP v C++
- Přemýšlíme o objektech, které mezi sebou interagují
- Kroky při vývoji:
 - **Abstrakce** – navrhujeme nezávislé entity, které spolupracují
 - **Dekompozice** – rozdělíme problém na objekty zodpovědné za realizaci entit
 - **Propojení** – vytvoříme potřebné množství objektů a necháme je interagovat
- Nejprve definujeme chování a vlastnosti objektů
- Poté vytvoříme jejich instance a necháme je „spolupracovat“ (volají své metody)

Mapování OOP na C++

```
class CMouse {  
    int m_size;  
public:  
    CMouse(int initialSize) : m_size(initialSize) {}  
    int feed(int foodAmount) {  
        m_size += foodAmount / 10;  
        return m_size;  
    }  
};  
  
void oopCppMappingDemo() {  
    CMouse mouse1;  
    mouse1.feed(10);  
    mouse1.  
}
```



- **objekt** ~ instance C++ třídy
- **zpráva** ~ volání metody
- **metoda** ~ členská funkce
- **parametry zprávy** ~ parametry metody
- **stav** ~ hodnoty atributů

Třídy v C++

- Založeno na konceptu tříd a objektů
- Analogie se struct z C, ale s většími možnostmi
 - struktura doplněná o metody, kterými může svá data manipulovat
- Třída
 - podklad pro vytvoření objektu (alá struct XY)
 - atributy (data) a metody (implementace funkce)
 - např. výkresy pro Škoda Octavia
- Objekt
 - instance třídy (alá proměnná typu struct XY)
 - může být více objektů z jedné třídy (~ více struktur typu XY)
 - má svůj stav (hodnoty atributů) a definované chování (metody)
 - např. konkrétní auto Škoda Octavia s SPZ BMZ-4523

Uživatelský datový typ Třída - *class*

- Nové v C++, ale hodně podobností s C *struct*
- Atributy
 - datové položky v rámci třídy
 - stejné jako u *struct*, ale nejsou z venku viditelné
 - přístup pomocí speciální metody (getter/setter) nebo změna práv
- Tvorba instance třídy (objekt)
 - stejně jako u *struct*
 - (instance třídy X == proměnná s typem X)
 - lze inicializovat při vytváření instance (tzv. konstruktor)
- Možnosti manipulace stejně jako u *struct*
 - `COctavia autoBMZ4523;`
 - `COctavia* pAutoBMZ4523 = &autoBMZ4523;`
 - `COctavia* pAutoBMZ4523 = new COctavia("BMZ4523");`

Ukázka syntaxe class vs. struct v C

```
struct Mouse {  
    int size;  
};
```

Class je vylepšená struct

```
int feed(Mouse* pMouse, unsigned int foodAmount)  
{  
    if (pMouse) {  
        pMouse->size += foodAmount / 10;  
        return pMouse->size;  
    }  
    else return -1;  
};
```

Speciální metoda pro inicializaci

Funkce feed je přímo součástí třídy
není nutné dávat jako parametr

```
int feedMouseDemo()  
{  
    struct Mouse mouse1;  
    struct Mouse mouse2;
```

mouse1 a mouse2 jsou objekty třídy CMouse

```
    mouse1.size = 10;  
    mouse2.size = 50;  
    feed(&mouse1, 100);  
    feed(&mouse2, 30);  
    return 0;  
}
```

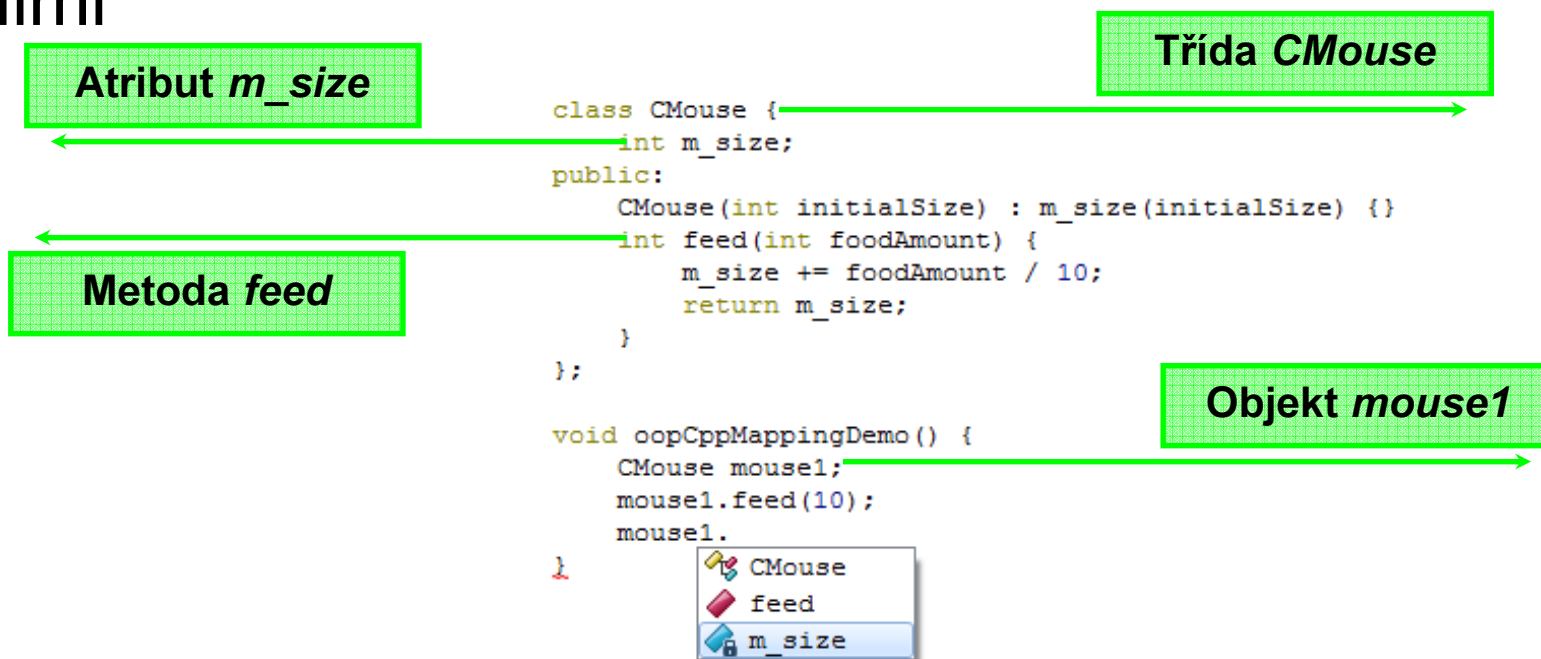
```
class CMouse {  
    int m_size;  
public:  
    CMouse(int initialSize) {  
        m_size = initialSize;  
    }  
    int feed(unsigned int foodAmount) {  
        m_size += foodAmount / 10;  
        return m_size;  
    }  
};  
  
int feedMouseDemo() {  
    CMouse mouse1(10);  
    CMouse mouse2(50);  
    mouse1.feed(100);  
    mouse2.feed(30);  
    return 0;  
}
```

Uživatelský datový typ Třída – *class* (2)

- Konstruktor, destruktork
 - metoda automaticky volaná při vytváření resp. rušení objektu
 - (detaily později)
- Deklaraci lze oddělit od definice
 - deklarace v hlavičkovém souboru (*.h)
 - definice (implementace) ve zdrojovém souboru (*.cpp)
- Dopředná deklarace s neúplným typem
 - stejně jako u struct
 - `class CMouse;`
 - `CMouse` musí být později dodefinována (jinak chyba při linkování)

Připomenutí

- Třída je podklad pro tvorbu objektů
- Objekt je paměťová instance třídy
- Třída obsahuje datové atributy a metody pro práci s nimi



Metoda třídy

- Metoda třídy je funkce (stejná syntaxe i chování)
- Metoda třídy typicky pracuje s vnitřním stavem
 - čte/modifikuje atributy
- Metoda třídy má vždy jeden skrytý argument
 - „nultý“ argument
 - ukazatel na instanci třídy, jejíž metoda se volá

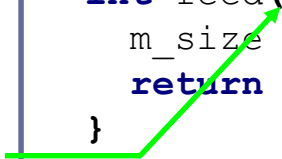
struct + funkce

```
struct Mouse { int size; };  
int feed(Mouse* pMouse, int foodAmount) {  
    pMouse->size += foodAmount / 10;  
    return pMouse->size;  
}
```

skrytý
parametr *this*

class + metoda

```
class CMouse {  
    int m_size;  
    int feed(int foodAmount){  
        m_size += foodAmount / 10;  
        return m_size;  
    }  
};
```



Ukazatel this

- Ukazatel na aktuální objekt
 - `this->m_size;`
- Automaticky jako parametr metod třídy
 - **není** explicitně deklarován v hlavičce metody
 - je ale dostupný pro použití
- Použití při konfliktu atributu třídy a parametru metody
 - parametr metody má stejné jméno jako vnitřní atribut
- Použití pro test přiřazení sebe sama
 - u přiřazovacích operátorů (viz. později)
- Použití pro možnost řetězení příkazů
 - u přiřazovacích operátorů (viz. později)

```
class CTest {  
    int value;  
public:  
    int getValue2() const {  
        int value = 1;  
        return value;  
    }  
};
```

Ukazatel this - ukázka

```
class CTest {
    int value;
public:
    CTest() : value(0) {}
    int getValue() const {
        int value = 1; // local variable
        return value; // What will be returned? 1 or 0?
    }
    int getValue2() const {
        int value = 1;
        return this->value; // Use this to distinguish
    }
    void testSame(const CTest& test) const {
        if (&test == this) cout << "Same" << endl;
        else cout << "Different" << endl;
    }
};
```

problém s
rozlišením

rozlišení
pomocí *this*

testování sebe
sama

```
#include <iostream>
using std::cout;
using std::endl;
int main() {
    CTest test;
    cout << test.getValue() << endl;
    cout << test.getValue2() << endl;

    CTest test2;
    test.testSame(test2);
    test.testSame(test);

    return 0;
}
```

Deklarace vs. definice

- **Deklarace** zavádí nové jméno entity do programu
 - typicky v hlavičkovém souboru (*.h)
- **Definice** poskytuje unikátní popis (implementaci) entity
 - funkce, typ, třída, instance...
 - typicky ve zdrojovém souboru (*.cpp)
- Jen jedna definice (implementace), možno více deklarací
- Deklarace může být zároveň definice
 - např. deklarace metody přímo doplněná jejím tělem

Ukázka třídy rozdělené do H a CPP

mouse.h

```
#ifndef MOUSE_H
#define MOUSE_H

class CMouse {
    int m_size;
public:
    CMouse();
    int getSize() const { return m_size; }
    bool feed(const unsigned int foodAmount);
private:
    bool increaseSize(const unsigned int foodAmount);
};

#endif // MOUSE_H
```

deklarace i
definice

deklarace

deklarace

import deklarací

mouse.cpp

```
#include "mouse.h"
CMouse::CMouse() {
    m_size = 10;
}
bool CMouse::feed(const unsigned int foodAmount) {
    // Check basic limits of foodAmount
    if (foodAmount > 0 && foodAmount < 100) {
        return increaseSize(foodAmount);
    }
    else return false;
}
bool CMouse::increaseSize(const unsigned int foodAmount) {
    m_size += foodAmount / 10;
    return true;
}
```

definice

definice

Zamyšlení: Co je vlastně všechno třída?

- What's in a class? – The Interface Principle
 - <http://www.gotw.ca/publications/mill02.htm>
- Třída není jen to, co je obsaženo v její definici
 - tj. nejen obsah `class { }`
- “Třída” jsou i volné funkce pracující s danou třídou
 - pouze ty obsažené v hlavičkovém souboru definujícím třídu
 - “předpis” jak se má daná třída využívat
 - tvůrce třídy vytváří celý *.h soubor

Konstruktor

Konstruktor - motivace



- Motivace: Naše třída má několik atributů
 - Jaká bude jejich hodnota při vytvoření objektu?

```
class CExampleClassDefConstructor {  
    int m_atrib;  
public:  
    int getAtrib() const { return m_atrib; }  
};  
int main() {  
    CExampleClassDefConstructor object1; // Invoke default constructor  
    cout << "object1::m_atrib = " << object1.getAtrib() << endl; // ???  
    return 0;  
}
```

- většinou neinicializovaná => nepříjemné
- Možným řešením je speciální metoda „clean()“
 - ale „zbytečné“ volání hned po vytvoření objektu
- C++ nabízí elegantní řešení - konstruktor

Konstruktor

- Metoda, která je **automaticky volána při vytváření** objektu
- Zajistí, že objekt bude od začátku v konzistentním stavu
 - můžeme inicializovat atributy na defaultní hodnotu
 - můžeme je nastavit na speciální hodnoty
 - můžeme otevřít spojení na server...
- Konstruktor může mít **argumenty** a může být **přetížen**
 - inicializace uživatelem zadanými hodnotami
 - více konstruktorů s různými argumenty (přetížení, více později)

Konstruktory - syntaxe

```
class CExampleClass {  
    int m_atrib1;  
    int m_atrib2;  
  
public:  
    CExampleClass();  
    CExampleClass(int atribut1);  
    CExampleClass(int atribut1, int atribut2);  
  
    int getAtrib1() const { return m_atrib1; }  
    int getAtrib2() const { return m_atrib2; }  
};
```

jméno metody jako třída

bez návratové hodnoty

Deklarace konstruktorů s 0, 1 a 2 parametry

// Constructor with no parameters
// Constructor with one parameter
// Constructor with two parameters

Volání konstruktorů

```
CExampleClass object2;  
CExampleClass object3(10);  
CExampleClass object4(10, 15);
```

```
CExampleClass::CExampleClass() {  
    m_atrib1 = 0;  
    m_atrib2 = 0;  
}  
  
CExampleClass::CExampleClass(int atribut1) {  
    m_atrib1 = atribut1;  
    m_atrib2 = 0;  
}  
  
CExampleClass::CExampleClass(int atribut1, int atribut2) {  
    m_atrib1 = atribut1;  
    m_atrib2 = atribut2;  
}
```

Implementace konstruktorů

Defaultní konstruktor

- Co když není definován žádný konstruktor?
 - automaticky existuje defaultní konstruktor
 - nemá žádné argumenty a neinicializuje atributy
 - *CClass object;*
- Definováním uživatelského konstruktoru se odstraní defaultní konstruktor bez argumentů
 - pokud chceme konstruktor bez parametrů, musíme ho znova definovat
 - tzv. bezparametrický konstruktor
- V C++ se bezparametrické konstruktory volají pouze pro třídy a struktury
 - pro nativní (built-in) typy se nevolá nic

Konstruktor – inicializační sekce

- Inicializační sekce konstruktoru
 - inicializace atributů
 - předání parametrů pro konstruktor rodiče

```
class X {  
    int m_atrib1;  
    int m_atrib2;  
public:  
    X(int atrib1) : m_atrib1(atrib1), m_atrib2(1) {}  
};
```

```
class Y : public X {  
public:  
    Y() : X(33) {}  
};
```

- Preferujte inicializační sekci před přiřazením
 - nelze jinak předat argumenty pro konstruktor předka
 - nelze jinak inicializovat atributy s referenčním typem
 - nemusí se vytvářet lokální kopie argumentů

Konstruktory - ukázka

- *constructorDemo.cpp*
- použití konstruktoru
- neinicializovaná proměnná
- defaultní konstruktor
- konstruktor s argumenty
- přetížení konstruktoru

Quiz

- Co znamenají následující řádky?
 - `class CClass;`
 - `class CClass {};`
 - `CClass obj1;`
 - `CClass obj1(10);`
 - `CClass obj1();`
- `CClass* obj1 = new CClass(10);`
 - `delete obj1;`

Bezparametrický konstruktor a C++11

- Co znamená `CClass object()` ;



- Nelze deklarovat, je to považováno za deklaraci funkce `object` bez argumentů vracející `CClass`
- V C++11 došlo ke sjednocení syntaxe
 - `CClass object{}; // default constructor`
 - `int i{}; // default ctor, i==0`

C++11 default a delete

- Součástí tříd obecně je několik důležitých metod
 - Bezparametrický konstruktor
 - Kopírovací konstruktor – details pozdější přednáška
 - Move konstruktor – details na konci semestru
 - Přiřazovací operátor
- Lze definovat, že některá z těchto metod je defaultní (**default**) nebo zakázaná (**delete**)

```
class CClass {  
public:  
    CClass() = default;  
    CClass &operator=( CClass ) = delete;  
    ...  
};
```


C++11 inicializační seznam

- Je zavedená možnost použití inicializačního seznamu
 - Pomocí složených závorek je možné také vytvářet instance i přes jiný konstruktor

```
#include <initializer_list>
#include <algorithm>
class MyClass {
    int *data;
public:
    // inicializacní seznam se bere vždy hodnotou
    MyClass( std::initializer_list< int > l ) : data( new int[ l.size() ] )
    {
        std::copy( l.begin(), l.end(), data );
    }
    ...
};
MyClass object{1, 2, 3, 4, 5, 42, 1001};
```

Destruktory

- Určeno pro úklid objektu
 - uvolnění polí a dynamických struktur
 - uzavření spojení apod.
- Automaticky voláno při uvolňování objektu
 - statická i dynamická alokace
- Může být pouze jediný (nelze přetěžovat)
- Nemá žádné parametry

Destruktor - syntaxe

- Syntaxe ~jméno_třídy()
- Stejné jméno jako třída
- Nevrací návratovou hodnotu
- U C++ vždy voláno při zániku objektu
 - konec platnosti lokální proměnné
 - dealokace dynamicky alokovaného objektu
 - odlišnost od Javy (Garbage collection)

```
class CExampleClass {  
public:  
    CExampleClass(); // Constructor with no parameters  
    ~CExampleClass(); // Destructor  
    virtual ~CExampleClass(); // Destructor  
};
```

Make base class destructors public and virtual, or protected and nonvirtual

To delete, or not to delete; that is the question: If deletion through a pointer to a base Base should be allowed, then Base's destructor must be public and virtual. Otherwise, it should be protected and nonvirtual. –C++ Coding Standards

Proč virtuální destruktorky?



Zapouzdření

Proč je výhodné zapouzdření

```
struct Point {  
    int x, y;  
    void Draw() {...}  
};
```

```
int main(void) {  
    Point myPoint;  
    myPoint.x = 10;  
    myPoint.y = 20;  
    myPoint.Draw();  
    return 0;  
}
```

```
class CPoint {  
    int m_x, m_y;  
public:  
    CPoint() { m_x = 0; m_y = 0;}  
    void setPoint(int x, int y) {  
        m_x = x;  
        m_y = y;  
    }  
    void Draw() {...}  
};
```

```
int main(void) {  
    CPoint myPoint2;  
    myPoint2.setPoint(10, 20);  
    myPoint2.Draw();  
    return 0;  
}
```

Zapouzdření – změna zadání

- Naši třídu už někdo používá, šedý kód z minulého slidu je velmi nepraktické/nemožné měnit
- Změna zadání: body v 3D
- Změna zadání: `int` nepostačuje pro zachycení rozsahu souřadnice

- změna typu u x a y na float?
- změna typu u x a y na řetězec?
- změna typu u x a y na BigInt?

- Jaké jsou důsledky pro předchozí kód?

```
int main(void) {  
    Point myPoint;  
    myPoint.x = 10;  
    myPoint.y = 20;  
    myPoint.Draw();  
    return 0;  
}
```

- Zapouzdřením omezujeme viditelnost vnitřního stavu
 - můžeme lépe kontrolovat interakce se stavem (setter)
 - můžeme měnit reprezentaci stavu

Jak poznat, která komponenta je více zapouzdřená?

- Stupeň zapouzdření komponenty je nepřímo úměrný množství ostatního kódu, který přestane fungovat, pokud komponentu změníme
 - Metrika zapouzdření: počet funkcí, které mohou vyžadovat změnu následkem změny komponenty
- Pokud změníme u Point datový typ proměnných na string, přestane fungovat šedý kód
- U CPoint změníme datový typ a upravíme příslušně `setPoint()`
 - pro nové aplikace vyžadující větší přesnost přidáme další metodu `setPoint()` s jinými argumenty
 - šedý kód zůstane fungovat!

```
void setPoint(int x, int y) {  
    m_x = convertToString(x);  
    m_y = convertToString(y);  
}
```


Zapouzdření v C++

- C++ poskytuje nástroje pro zapouzdření dat
 - ale umožňuje i porušit (tj. přímý přístup)
- Realizováno prostřednictvím **přístupových práv**
 - k atributům
 - k metodám
- Základní přístupová práva
 - *public* – všichni mohou číst/modifikovat/používat
 - *private* – nikdo kromě vlastní třídy nemůže číst nebo přímo používat
 - *protected, friend* – specializovanější (později)

Syntaxe přístupových práv

● Struktura

změna z public
na private

```
..
struct exampleStruct {
    // all attributes (and methods) are public, until said otherwise
    int atribPublic;

    private:    // switch to private access rights
    int atribPrivate;

    public:     // switch to public access rights
    int atribPublic2;
};
```

● Třída

změna z private
na public

```
class CExampleClass {
    // all attributes (and methods) are private, until said otherwise
    int m_atribPrivate;

    public:    // switch to public access rights

    // Public methods manipulating value of private attribute
    int getAtribPrivate() {
        return m_atribPrivate;
    }
};
```

Přístupová práva

- Právo platí, dokud není nastaveno jiné
 - viz. předchozí ukázka kódu
- `struct` je to samé jako `class`, rozdíl právě v defaultních právech
- `struct` v C++ má všechny položky defaultně `public`
 - z důvodu zpětné kompatibility s C
 - lze přenastavit na `private`
- `class` v C++ má všechny položky defaultně `private`
 - ponechte pro atributy `private`
 - je nutné explicitně nastavit `public` pro veřejné metody

Přístupová práva - public

- K položce s právem *public* má přístup kdokoli
 - atribut může být čten a měněn kýmkoli
 - metoda může být volána „zvenčí“
- Jako *public* typicky neoznačujeme atributy
 - podporujeme zapouzdření
- Jako *public* označujeme metody, které jsou součástí rozhraní
 - deklarace existujících *public* metod by se neměly měnit
 - někdo je nejspíš používá
 - implementaci měnit můžeme (tělo je skryto)

Přístupová práva - private

- K položce s právem *private* má přístup **pouze sama třída**
 - atribut nebo metoda nemůže být použit/volána „zvenčí“
 - výjimkou je objekt/metoda s právem *friend* (viz. později)
 - pokus o použití metody definované jako *private* vyvolá chybu už během překladu
- Jako *private* **označujeme** typicky všechny **atributy**
 - podporujeme zapouzdření
- Jako *private* označujeme metody, které **nejsou** součástí rozhraní
 - nechceme, aby na nich někdo závisel

Přístupová práva - ukázka

- *accessRightsDemo.cpp*
- deklarace veřejných a privátních atributů
- rozdíly class vs. struct
- chyby překladače
 - přístup k privátnímu atributu
 - přístup k privátní metodě

Pro zamyšlení: Nečlenské metody pro zlepšení zapouzdření?

- Členské metody (např. setter) zlepšují zapouzdření oproti situaci s přímo přístupnými atributy
- Mohou nečlenské metody také zlepšit zapouzdření?
- Metrika zapouzdření: počet funkcí, které mohou vyžadovat změnu po změně komponenty
 - struct – veškerý kód, který struct používá
 - class – všechny členské metody (N)
 - změna jedné členské metody na nečlenskou => N - 1
- Pozor, týká se jen ne-členských ne-friend funkcí
 - tj. funkce, které pro svoje vykonání využijí pouze veřejné metody třídy
- Přečte si: How non-member functions improve encapsulation” (Scott Meyers)
 - <http://www.drdobbs.com/cpp/how-non-member-functions-improve-encapsu/184401197>

Zapouzdření a delegace požadavků

- Může-li třída vykonat úkol, jež ji byl zadán, vykoná jej, jinak jej deleguje tomu, kdo má zodpovědnost za vykonávání daného úkolu
- Třída nemá zjišťovat informace (které by měly nejlépe zůstat zapouzdřené) od jiných tříd ve snaze se na jejich základě rozhodnout jak úkol provést
- Danou záležitost má rozhodnout a provést třída, které požadované informace patří

Zapouzdření – klíčové slovo `const`

Klíčové slovo *const*

- Zavedeno pro zvýšení robustnosti kódu proti nezáměrným implementačním chybám
- Motivace 1:
 - potřebujeme označit **proměnnou, která nesmí být změněna**
 - typicky konstanta, např. počet měsíců v roce
- Motivace 2:
 - chceme deklarovat, že naše **funkce nebude měnit** vstupní parametr
 - přestože by mohla (např. předání referencí, ukazatelem)
- Motivace 3:
 - chceme deklarovat, že daná **metoda nemění vnitřní stav** objektu
 - Lze ji tedy volat i nad konstantním objektem
- A chceme mít **kontrolu přímo od překladače!**

Klíčové slovo *const* (2)

- Explicitně vyznačujeme proměnnou/objekt, která nebude měněna
 - jejíž hodnota (nebo hodnota atributů) by neměla být měněna
 - argument, který nemá být ve funkci měněn
- Explicitně označujeme funkci, která může být volána i na konstatním objektu
 - protože nebude měnit jeho vnitřní stav
 - je kontrolováno při překladu!
- Lze mít dvě identické funkce lišící se pouze v **const**
 - funkce s const má nižší prioritu, použije se pouze pokud bude argument také const
 - např. **begin()** a **end()** u STL kontejnerů

Klíčové slovo *const* (3)

- Používejte co nejčastěji
 - zlepšuje typovou kontrolu a celkovou robustnost
 - kontrola že omylem neměníme konstantní objekt
 - umožňuje lepší optimalizaci překladačem
- Proměnné s `const` jsou lokální v daném souboru
- Pozor na `const int a, b = 0;`
 - raději každá proměnná na samostatném řádku

Klíčové slovo *const* - ukázka

- *const* proměnná
- *const* argument
- *const* metoda
- chyby překladače

Klíčové slovo *const* - proměnná

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

void konstConstantDemo() {
    //const int a, b = 0; // error, uninitialized const 'a'
    const int numMonthsInYear = 12;
    cout << "Number of months in year: " << numMonthsInYear << endl;
    //numMonthsInYear = 13; // error, assignment of read-only variable
}
```

proměnná *b* není
const

konstanty nelze
dodatečně měnit

```
char* konstReturnValueDemo() {
    return "Unmodifiable string";
}

const char* konstReturnValueDemo2() {
    return "Unmodifiable string";
}

int main() {
    char* value = konstReturnValueDemo();
    value[1] = 'x'; // read-only memory write - problem

    char* value2 = konstReturnValueDemo2(); // error: invalid conversion
    return 0;
}
```

vracíme řetězec
(konstantní)

explicitně označíme řetězec
jako nemodifikovatelný

zde se jej ale
snažíme modifikovat
– chyba za běhu

problém ohlásí už překladač

Klíčové slovo *const* - metody

```
class CMouse {  
    int m_size;  
public:  
    void setSize(int newSize) {  
        m_size = newSize;  
    }  
    int getSize() const {  
        return m_size;  
    }  
};  
  
void konstFunctionDemo() {  
    CMouse mouse1;  
    //const CMouse mouse2;    // error: uninitialized const mouse2  
    const CMouse* pMouse2 = &mouse1;  
  
    mouse1.getSize();  
    mouse1.setSize(10);  
    cout << pMouse2->getSize() << endl;  
  
    // Let's try to call non-constant method of constant object  
    //pMouse2->setSize(10);    // error: no matching function for call  
}
```

metoda `setSize()` nemůže být konstantní
(mění vnitřní stav)

metodu `getSize()` označíme jako
volatelnou i nad konstantním objektem

vytvoříme si konstantní
ukazatel na objekt

nemůžeme volat nekonstantní
metodu konstantního objektu

Ukazatel na konstantu, konstatní ukazatel, konstantní ukazatel na konstantu

- **const char** * myPtr = &char_A;
 - Ukazatel na konstatní hodnotu typu znak
 - Tento ukazatel nelze využít pro změnu této hodnoty (kam ukazuje myPtr)
- **char** * **const** myPtr = &char_A;
 - Konstantní ukazatel na znak
 - Hodnotu ukazatele nelze změnit (obsah myPtr)
- **const char** * **const** myPtr = &char_A;
 - Konstatní ukazatel na konstantní hodnotu
- http://www.codeguru.com/cpp/cpp/cpp_mfc/general/article.php/c6967/Constant-Pointers-and-Pointers-to-Constants.htm

Výpis na standardní výstup

- Z C znáte `printf("Dnes je %d. zari", den);`
- V C++ na výstup zapíšete takto:

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

int main() {
    int den = 27;
    cout << "Dnes je " << den << ". zari";
    return 0;
}
```

The diagram illustrates the C++ code for standard output. It includes three callout boxes with green borders and arrows pointing to specific parts of the code:

- Analogie <stdio.h>**: Points to the `#include <iostream>` line.
- Standardní výstup je objekt cout**: Points to the `cout` object in the output statement.
- Operátor << poskytuje objektu cout řetězcovou reprezentaci argumentu těsně za ním**: Points to the `<<` operator in the output statement.

- Práci se standardním výstupem budeme dělat detailněji v 4. přednášce

Shrnutí

- Konstruktor je metoda pro inicializaci objektu
 - pozor na defaultní konstruktor
- Zapouzdření skrývá vnitřní data a logiku
 - umožňuje abstrahovat uživatele od aktuální implementace
- Používejte co nejčastěji **const**
 - číselné konstanty
 - parametry funkce/metody, celá metoda

Zdroje

- [StackOverflow](#) – Q&A
- [C++ FAQ](#) – Odpovědi na často kladené otázky.
- [cplusplus.com](#) – Dokumentace C++ a standardní knihovny.
- [cppreference.com](#) – Dokumentace C++ a standardní knihovny, obsahuje i dokumentaci k C++11.
- [C++ Coding Standards](#) – “101 Rules, Guidelines, and Best practices”
- [Effective C++](#) – “55 Specific Ways to Improve Your Programs and Designs”
- [Exceptional C++](#) – “47 Engineering Puzzles, Programming Problems, and Solutions”
- [Modern C++](#) – “Generic Programming and Design Patterns Applied”