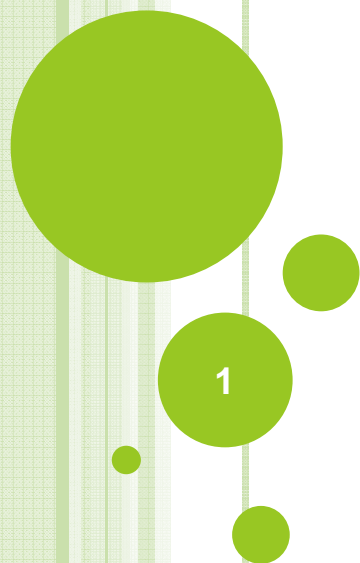


PRINCIPY OOP, DĚDIČNOST



CO NÁS DNES ČEKÁ...

- Operátor reference &
- Dědičnost a kompozice
- Kopírovací konstruktory, destruktory
- `std::string`



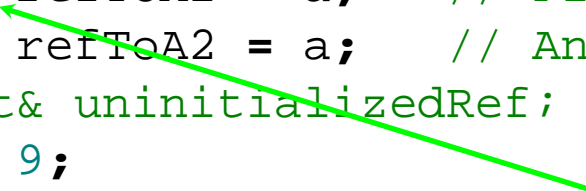
OPERÁTOR REFERENCE &

3

REFERENCE NA PROMĚNNOU

- Alternativní jméno pro objekt/proměnnou (~alias)
 - operátor reference je &

```
int a = 1;
int b = 5;
int& refToA1 = a;    // First reference to a
int& refToA2 = a;    // Another reference to a
//int& uninitializedRef; // error: uninitialized reference
a += 9;
refToA1 += 9;
refToA2 += 11;
```



operátor
reference

- Určitá analogie s ukazatelem na proměnnou
 - není ukazatel na objekt, ale má podobné použití
 - není proměnná s adresou na původní

PŘEDÁVÁNÍ ARGUMENTŮ REFERENCÍ

- Alternativa k předávání argumentů funkce ukazatelem

- volání hodnotou:
- volání odkazem:
- volání referencí:

```
void byValue(int A) { A = 10; }
```

```
void byPointer(int* pA) { *pA = 10; }
```

```
void byReference(int& A) { A = 10; }
```

- Zavolání funkce vypadá jako volání hodnotou

- při předávání referencí není vytvářena kopie objektu
- změna argumentu uvnitř funkce se ale projeví i mimo funkci

KONSTANTNÍ REFERENCE

○ Konstantní reference

- void foo(const typ& param)
- umožňuje specifikovat záměr programátoru o nakládání s objektem (referencí, ale nebude měněn)

○ Kontrolováno během překladu

- konstatní reference na nekonstantní objekt - chyba
- změna nekonstantního objektu přes konstantní referenci - chyba

```
void constReferenceDemo(const int& A, const int* pB) {  
    // We can read values  
    cout << "a + b: " << A + *pB << endl;  
    // But we can't change them  
    A = 1; // error: assignment of read-only reference 'A'  
    *pB = 1; // error: assignment of read-only location '* pB'  
}
```

KONSTANTNÍ REFERENCE – DETAILS

- Pro parametry v hlavičce funkce/metody platí, že konstantní reference je "catch-all"
- Oproti normální referenci, která umí "chytat" pouze lvalue (to, co někde bydlí, má to adresu), tak konstantní umí "chytat" všechno, i dočasné objekty
- Právě proto, že je konstantní reference "catch-all", tak má při určování, která fce se zavolá (při stejné signatuře) nižší prioritu, než funkce pouze s referencí

REFERENCE - UKÁZKA

- referenceDemo.cpp
- více referencí na jedinou proměnnou
- nutnost inicializace reference
- předání argumentu referencí
- změna hodnoty mimo funkci
- konstantní reference



DĚDIČNOST

9

DĚDIČNOST V OOP

- Dědičnost je nástroj pro podporu abstrakce
 - potomci dodržují rozhraní zavedené předkem
 - mohou ale měnit chování (implementaci)
 - můžeme mít generický kód, který bude pracovat s budoucími implementacemi
- Dědičnost je nástroj pro omezení duplicity v kódu
 - duplicita v kódu je nepříjemná
 - snižuje přehlednost
 - zvyšuje náročnost úprav (nutno na více místech)
 - zvyšuje riziko chyby (někde zapomeneme upravit)
 - dědit z třídy jen pro využití části funkčnosti ale není dobré (viz. dále)
- Zlepšuje možnost znovuvyužití existujícího kódu

“DĚDIČNOST” V C

- V C se abstrakce a znovuvyužití kódu dosahuje:
 - dobrým návrhem funkčního rozhraní
 - umístěním do samostatných hlavičkových souborů
 - přechod na jinou implementaci ideálně jen změnou hlavičkového souboru
- V C se duplicita kódu odstraňuje:
 - vytvořením nové funkce
 - a vložením funkčního volání na původní místa

DĚDIČNOST V C++

- V C++ existuje systematičtější podpora
 - lze odstranit duplicitu i pro proměnné
 - navíc podporuje silnou typovou kontrolu
- Mechanismus umožňující vytvořit další třídu (potomek) s využitím předlohové třídy (předek)
 - potomek zdědí možnosti předka (atributy a metody)
 - může je rozšiřovat a předefinovat (překrývat)

DĚDIČNOST – POSTUP PŘI ABSTRAKCI

- Máme dvě (nebo víc) entit se společným chováním
- Snažíme se vytvořit společnou logiku (metody), které budou potřebné chování pro všechny entity popisovat
 - aniž bychom museli vědět, se kterou právě pracujeme
 - tvoříme rozhraní
- Vytvoříme novou třídu (předka, rozhraní)
 - obsahující popis společného rozhraní (veřejné metody)
- Pro jednotlivé entity vytvoříme nové samostatné třídy, které budou implementovat definované rozhraní
- Nové třídy budou potomci třídy definující rozhraní

DĚDIČNOST – POSTUP PŘI ODSTRAŇOVÁNÍ DUPLICITY

- Máme dvě (nebo víc) tříd se společným chováním
- Identifikujeme společnou logiku (metody)
- Identifikujeme společná data (atributy)
- Vytvoříme novou třídu (předka)
 - obsahující společné atributy a logiku
- Odstraníme přesunuté atributy&metody z původních tříd
- Původní třídy nastavíme jako potomky nového předka

DĚDIČNOST - PŘÍKLAD

- V laboratoři máme domácí a polní myš. Rozdíl mezi druhy je jen v počáteční velikosti a rychlosti přibírání po požití potravy.
- Nové třídy CHouseMouse a CFieldMouse
- Společné vlastnosti přesunuty do CMouseBase
- CHouseMouse a CFieldMouse potomci CMouseBase

DĚDIČNOST - SYNTAXE

Hlavičkový
soubor předka

```
#include "cmousebase.h"
```

Potomek

```
class CFieldMouse : public CMouseBase {  
public:  
    CFieldMouse();  
protected:  
    bool increaseSizeByFood(const unsigned int foodAmount);  
};
```

Předek

Modifikátor definující
způsob dědění metod
a atributů

PŘÍSTUPOVÁ PRÁVA - PROTECTED

- K položce s právem protected má přístup pouze potomek
 - atribut může být čten a měněn potomkem
 - metoda nemůže být volána „zvenčí“
- Jako protected typicky označujeme metody
 - které nemají být dostupné všem, ale potomkům ano
 - často jde o virtuální přetěžované metody (později)
 - méně často atributy – raději protected „setter“ metodu

TYPOVÁ HIERARCHIE

- Dědičnost vytváří hierarchii objektů
 - od nejobecnějšího k nejspecifičtějším
- Na místo předka může být umístěn potomek
 - proměnná s typem předka může obsahovat potomka
 - potomek může být argumentem funkce s typem předka
 - zároveň zachována typová bezpečnost
- Při dědění lze omezit viditelnost položek předka
 - specifikace práv při dědění

SPECIFIKÁTORY PŘÍSTUPOVÝCH PRÁV DĚDĚNÍ

- `public (class B : public A {};`)
 - zděděné položky dědí přístupová práva od předka
 - práva zůstanou jako předtím
- `private (class B : private A {};`)
 - zděděné položky budou `private`, odvozená třída však bude mít přístup ke položkám, pokud byly v předkovi `public` nebo `protected`
 - nebude přístup k položkám `private` u předka
 - v potomcích potomka už nebude přístup
 - používáme, pokud nechceme být předkem

SPECIFIKÁTORY DĚDĚNÍ PŘÍSTUPOVÝCH PRÁV (2)

- `protected (class B : protected A {};`)
 - položky `private` a `protected` zůstanou stejné, z `public` se stane `protected`
- pokud neuvedeme `(class B : A {};`)
 - `class` jako `private`, u `struct` a `union` jako `public`
- `virtual (class B : virtual A {};`)
 - lze kombinovat s jedním z předchozích, přikazuje pozdní vazbu – (viz později)

PRÁVA PŘI DĚDĚNÍ - UKÁZKA

- inheritanceRightsDemo.cpp
- přístup k private metodě
- přístup při dědění public/private/protected
- změna práv pro přístup při opakovaném dědění
- způsob znepřístupnění původně public metody

KÓD Z INHERITANCERIGHTSDEMO.CPP

```
//  
// Inheritance rights demo  
//  
class A {  
private:  
    void privateMethodA() {}  
protected:  
    void protectedMethodA() {}  
public:  
    void publicMethodA() {}  
};  
  
/**  
    Public inheritance - all rights for inherited methods/atributes stay same  
    */  
class B : public A {  
public:  
    void test() {  
        //privateMethodA();    // error: 'void A::privateMethodA()' is private  
        protectedMethodA(); // OK  
        publicMethodA();     // OK  
    }  
};  
  
/**  
    Private inheritance - all rights for inherited methods/atributes changes to private  
    */  
class C : private A {  
public:  
    void test() {  
        //privateMethodA();    // error: 'void A::privateMethodA()' is private  
        protectedMethodA(); // OK, but protectedMethodA is now private in C  
        publicMethodA();     // OK, but publicMethodA is now private in C  
    }  
};
```

KÓD Z INHERITANCE RIGHTS DEMO.

```
/**
 * Public inheritance from C (C was inherited privately from A)
 */
class D : public C {
public:
    void test() {
        //privateMethodA(); // error: 'void A::privateMethodA()' is private
        //protectedMethodA(); // error: 'void A::protectedMethodA()' is protected
        //publicMethodA(); // error: 'void A::publicMethodA()' is inaccessible
    }
};

/**
 * Protected inheritance - all rights for inherited methods/atributes changes to private
 */
class E : protected A {
public:
    void test() {
        //privateMethodA(); // error: 'void A::privateMethodA()' is private
        protectedMethodA(); // OK, protectedMethodA stays protected in E
        publicMethodA(); // OK, but publicMethodA is now protected in E
    }
};

/**
 * Public inheritance from E (E was inherited as protected from A)
 */
class F : public E {
public:
    void test() {
        //privateMethodA(); // error: 'void A::privateMethodA()' is private
        protectedMethodA(); // OK
        publicMethodA(); // OK
    }
};
```

```
class C : private A {
public:
    void test() {
        //privateMethodA();
        protectedMethodA();
        publicMethodA();
    }
};
```

PŘETÝPOVÁNÍ PRIVATE/PROTECTED POTOMKA

- “Lze získat přístup k public metodám předka z instance potomka, který dědil pomocí private/protected pomocí jeho přetypování na předka?”
- nelze, viz. brokenRightsDemo.cpp
- <http://stackoverflow.com/questions/9661936/inheritance-a-is-an-inaccessible-base-of-b>

PŘETÝPOVÁNÍ PRIVATE/PROTECTED POTOMKA (2)

```
class A {
public:
    void foo() {
        cout << "foo called" << endl;
    }
};

class B : protected A {

};

int main() {
    A a; a.foo();

    B b;
    //b.foo();      // error: 'void A::foo()' is inaccessible

    // Let's try to retype B to A to get access to originally public method
    A& refB = b;    // error: 'A' is an inaccessible base of 'B'
    refB.foo();
}
```

JAK „DĚDIT“ Z VÍCE EXISTUJÍCÍCH TŘÍD?

- Nová třída má mít vlastnosti více entit
- Novou třídu lze přetypovat na více různých předků
 - v Java se řeší pomocí interfaces
- V C++ lze řešit
 - pomocí násobné dědičnosti (třída má více předků)
 - pomocí kompozice objektu (třída má více podčástí)

SYNTAXE NÁSOBNÉ DĚDIČNOSTI

```
class CRAMMemory {  
    int m_ramSize;  
public:  
    CRAMMemory(unsigned int size) { m_ramSize = size; }  
    int getRAMSize() const { return m_ramSize; }  
};  
  
class CCPU {  
    int m_clockFrequency;  
public:  
    CCPU(unsigned int freq) { m_clockFrequency = freq; }  
    int getCPUFreq() const { return m_clockFrequency; }  
};  
  
class CNotebookInherit : public CRAMMemory, public CCPU {  
public:  
    CNotebookInherit(unsigned int ramSize, unsigned int cpuFreq)  
        : CRAMMemory(ramSize), CCPU(cpuFreq) {  
        this->  
            CCPU::getCPUFreq();  
    }  
    int getCPUFreq() const { return m_clockFrequency; }  
};
```

Předek 1

Předek 2

Dědíme z
obou předků

- Syntakticky správně, je ale vhodné?

DĚDIČNOST VS. KOMPOZICE

- Dědičnost je „Is-A“ vztah (chová se je jako A)
 - potomek má všechny vnější vlastnosti předka A
 - potomka můžeme přetypovat na předka
 - (je správné se na notebook dívat jako na případ CPU?)
- Kompozice je „Has-A“ vztah
 - třída může mít jako atribut další třídu A
 - hodnotou, referencí, ukazatelem
 - třída obsahuje vlastnosti A a další
 - třída může mít víc tříd jako své atributy
 - (je vhodnější se na notebook dívat jako na složeninu CPU a RAM?)

KOMPOZICE NAMÍSTO NÁSOBNÉ DĚDIČNOSTI

```
class CNotebookInherit : public CRAMMemory, public CCPU {  
};
```

Násobná
dědičnost

Kompozice

```
class CNotebookComposition {  
    CRAMMemory    m_ram;  
    CCPU           m_cpu;  
public:  
    /**  
     * Initialize attributes in constructor. As params are passed into constructors  
     * of attributes, initialization list section needs to be used  
     */  
    CNotebookComposition(unsigned int ramSize, unsigned int cpuFreq)  
        : m_ram(ramSize), m_cpu(cpuFreq) {}  
    int getCPUFreq() const { return m_cpu.getCPUFreq(); }  
    int getRAMSize() const { return m_ram.getRAMSize(); }  
};
```

DĚDIČNOST VS. KOMPOZICE - UKÁZKA

- inheritanceCompositionDemo.cpp
- násobná dědičnost
- kompozice
- využití inicializační sekce konstruktoru
- přetypování na předka

DĚDIČNOST – VHODNOST POUŽITÍ

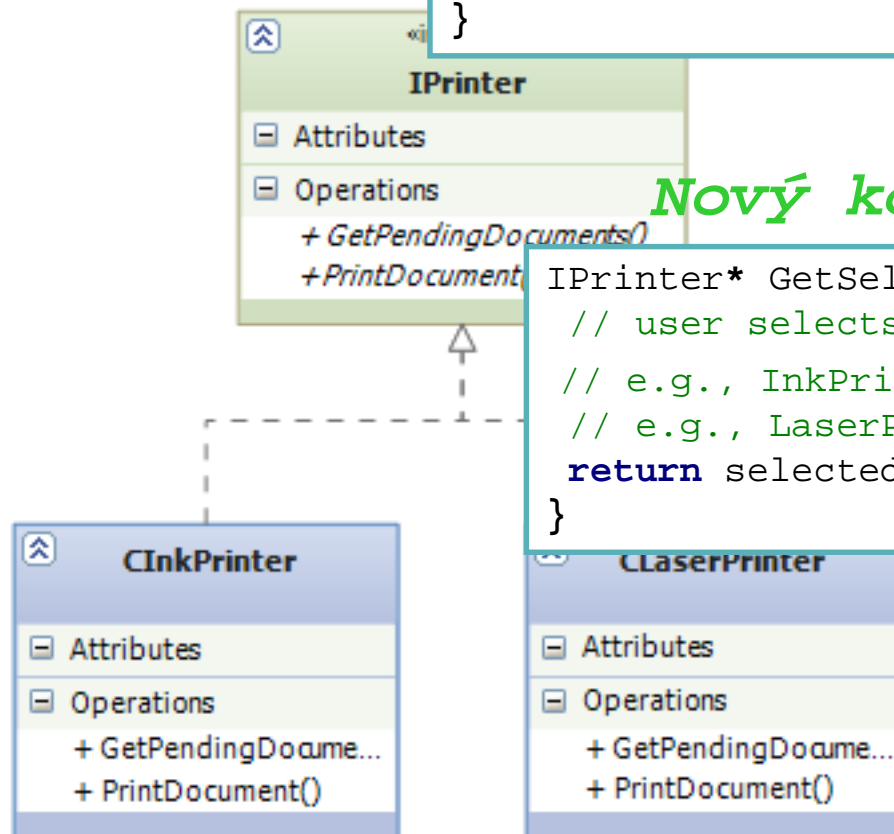
- Dle použití lze volit mezi dědičností a kompozicí
- Obecně preference kompozice před dědičností
 - násobná dědičnost může být nepřírozená
 - ale kompozice může být kódově rozsáhlejší
- Možná i kombinace
 - objekt obsahuje kompozicí třídy jako atributy
 - jednotlivé atributy mohou mít hierarchii dědičnosti

Existující kód

```
int main() {  
    IPrinter* printer = GetSelectedPrinter();  
    printer->PrintDocument();  
    printer->GetPendingDocuments();  
  
    return 0;  
}
```

Nový kód

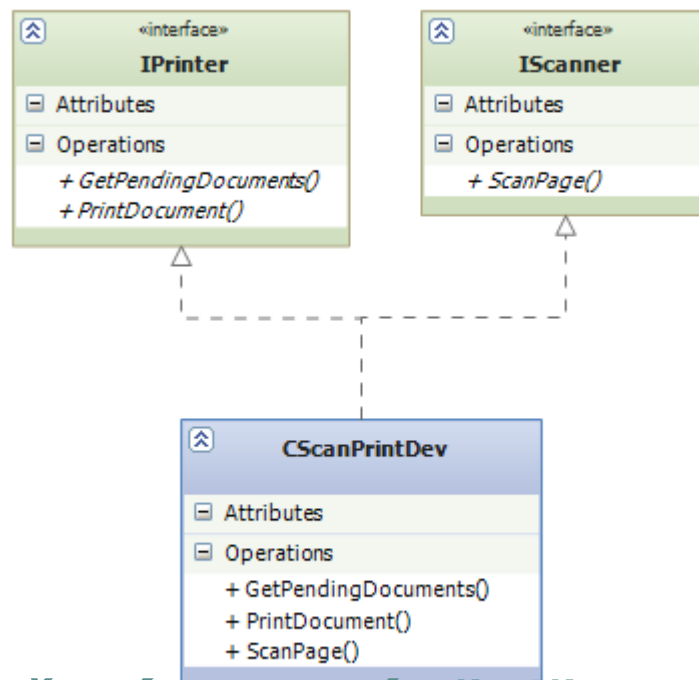
```
IPrinter* GetSelectedPrinter() {  
    // user selects printer via GUI  
    // e.g., InkPrinter -> selectedPrinter  
    // e.g., LaserPrinter -> selectedPrinter  
    return selectedPrinter;  
}
```



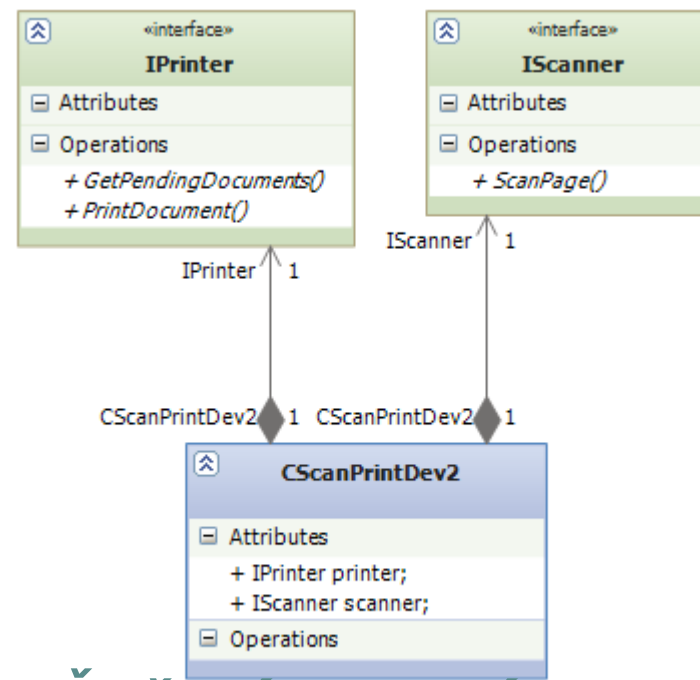
DĚDIČNOST – SPRÁVNÉ POUŽITÍ (1)

- Z nového kódu můžeme vždy volat kód existující
 - aniž bychom přepisovali existující kód
 - (víme jméno a parametry existující funkce, nový kód přizpůsobíme)
- Dědičnost nám umožňuje volat z existujícího kódu kód, který teprve bude napsán
 - existující kód pracuje s předkem (např. Printer)
 - (Printer deklarován v době psaní existujícího kódu)
 - nový kód vytváří potomky (např. InkPrinter)
 - existující kód pracuje s Printer
 - InkPrinter lze přetypovat na Printer
 - existující kód může používat InkPrinter (jako Printer)
- Potomek by neměl měnit logiku chování (rozhraní) předka!
 - InkPrinter pořád přijímá dokument na tisk
 - „pouze“ tiskne specializovaným způsobem

MULTIFUNKČNÍ ZAŘÍZENÍ (TISKÁRNA + SCANNER)



Řešení pomocí dědičnosti



Řešení pomocí kompozice

34

MULTIFUNKČNÍ ZAŘÍZENÍ (2)

- Je lepší použít dědičnost nebo kompozici?
- Pokud se jednotlivý předkové funkčně nepřekrývají, tak lze vícenásobná dědičnost
 - vhodné je dědit z čistě abstraktních tříd (viz. dále)
 - pak je stejné jako rozhraní v Javě (interfaces)
 - IPrinter a IScanner pokrývají odlišnou funkčnost
- Dědičnost používáme, pokud předpokládáme přetypování potomka na předka
 - u ClnkPrinter bude nastávat
 - bude nastávat i u CScanPrintDev?
- Dědičnost používáme, pokud předpokládáme později vznik potomků z naší třídy

DĚDIČNOST – SPRÁVNÉ POUŽITÍ (2)

- Dědičnost je velice silný vztah
 - => jeho nevhodné použití může přinést problémy
- Potomci při dědičnosti mají specializovat, ne rozšiřovat funkčnost základního objektu
 - CStudentTeacher není jen speciální případ studenta
 - CStudentTeacher je student a učitel zároveň
 - => víc než jen student => dědičnost není vhodná
 - vhodnější je zde kompozice (CStudentTeacher obsahuje atributy CStudent a CTeacher)
- Platí že potomek je substituovatelný za předka?
 - pokud ano, použijte dědičnost
- Preferujte kompozici před dědičností

DĚDIČNOST A VIRTUÁLNÍ DĚDIČNOST

- Problém typu diamand vzniká typicky při nevhodné OO hierarchii
- Virtuální dědičnost umožňuje obejít, ale vhodnější je přímo odstranit problém změnou hierarchie
 - (pokud je možné)



KONSTRUKTORY & DESTRUKTORY

38

KONSTRUKTOR - PŘIPOMENUTÍ

- Metoda automaticky volaná při vytváření objektu
- Možnost více konstruktorů (přetížení)
- Možnost konstruktoru s parametry

KONSTRUKTOR – POŘADÍ VOLÁNÍ V DĚDIČNOSTI

```
class X {  
    public: X() { cout << "Constructor X" << endl; }  
};  
class Y : public X {  
    public: Y() { cout << "Constructor Y" << endl; }  
};  
class Z : public Y {  
    public: Z() { cout << "Constructor Z" << endl; }  
};  
int main() {  
    Z object1;  
    return 0;  
}
```



```
D:\Documents\Develop\PB161_fall2010\T  
Constructor X  
Constructor Y  
Constructor Z  
D:\Documents\Develop\PB161_fall2010\T
```

- V jakém pořadí budou konstruktory volány?
 - konstruktory předka se volají před konstruktory potomka

KOPÍROVACÍ (COPY) KONSTRUKTORY

- Konstruktor pro vytváření kopie existujícího objektu
 - volány ve více situacích (přiřazení, funkční argument...)
- Pokud nevytvoříte vlastní, překladač vygeneruje automaticky vlastní!
- Dva základní typy konstruktorů: plytký a hluboký

KDY JSOU VOLÁNY KOPÍROVACÍ KONSTRUKTORY?

- Při použití funkcí s argumenty typu třída
 - pokud je argument předáván hodnotou
 - `void foo(CClass X) {};`
- Když má funkce návratovou hodnotu typu třída
 - `CClass foo() { CClass X; return X; }`
- Když je objekt použit pro inicializaci jiného objektu
 - třída je parametr konstrukturu druhého objektu
 - `CClass x; CClass y(x);`
- Není volán při předávání referencí/odkazem

SYNTAXE KOPÍROVACÍCH KONSTRUKTORŮ

- jméno_třídy(const jméno_třídy & předloha)
- Využívá se const
 - neměníme původní objekt
- Využívá se předání referencí &
 - nechceme volat opakovaně znovu kopírovací konstruktor

```
Person(const Person& copy) : m_name(0) {  
    cout << "Copy constructor Person called" << endl;  
    setName(copy.getName());  
}
```

PLYTKÝ (SHALLOW) COPY KONSTRUKTOR

- Kopíruje pouze hodnoty atributů
 - defaultní varianta – vytvářena překladačem
- Není problém pro jednoduché datové typy (int)
- Problém při atributu typu ukazatel nebo složitější objekty
 - např. při atributu typu `char*` zkopíruje se pouze hodnota ukazatele, ale nevytvoří se celé druhé pole

HLUBOKÝ (DEEP) COPY KONSTRUKTOR

- Je nutné explicitně implementovat
- Proveden dodatečný uživatelský kód
 - typicky vytvoření a kopírování dynamických polí atp.
- Zajistí vytvoření kompletní kopie původního objektu

KOPIROVACÍ KONSTRUKTOR - UKÁZKA

```
class Person {
    char* m_name;
public:
    Person(const char* name) : m_name(0) {
        cout << "Constructor Person(" << name << ") called";
        cout << endl;
        setName(name);
    }
    Person(const Person& copy) : m_name(0) {
        cout << "Copy constructor Person called" << endl;
        setName(copy.getName());
    }
    ~Person() {
        if (m_name) delete[] m_name;
    }
    const char* getName() const { return m_name; }
    void setName(const char* name) {
        if (m_name) delete[] m_name;
        m_name = new char[strlen(name) + 1];
        strncpy(m_name, name, strlen(name) + 1);
    }
};
```

```
int main() {
    //
    // Demo for copy constructor
    //
    Person object1("Prvni");
    cout << object1.getName() << endl;
    Person object2(object1);
    cout << object2.getName() << endl;

    //
    // Demo for assignment
    //
    // No copy constructor called, just primitive values assignment
    // (pointer to char array)
    object2 = object1;
    // Seems good at the moment, but...
    cout << object2.getName() << endl;
    // Changing name in object2 will also change name in object1
    object2.setName("Problem");
    cout << object1.getName() << endl;

    return 0;
}
```

OPERÁTOR PŘÍŘAZENÍ =

- Při použití pro primitivní hodnoty zkopíruje samotnou hodnotu
- Při použití pro třídní datový typ provede kopii hodnot všech atributů
- Defaultní operátor přiřazení se chová stejně jako defaultní kopírovací konstruktor
 - operátor přiřazení lze také předefinovat (později)

CO JE KONSTRUKTOR S `EXPLICIT` ?

- Každá třída má alespoň implicitní konstruktor
 - bez parametrů (bezparametrický konstruktor)
 - pokud deklarujeme další konstruktor (typicky s argumenty), tak se implicitní konstruktor nevytváří
- Objekt ale můžeme vytvářet i pomocí argumentů s typem, který neodpovídá typu v konstruktoru. Proč?
 - pokud je to možné, dojde k implicitní typové konverzi
 - použije se kopírovací konstruktor
- Klíčovým slovem `explicit` říkáme, že daný konstruktor může být použit jen v případě, že přesně odpovídá typ argumentů
 - ochrana proti nečekané implicitní typové konverzi



EXPLICIT - UKÁZKA

```
class MyClass {
public:
    MyClass(float X) {}
};

class MyClass2 {
public:
    explicit MyClass2(float X) {}
};

int main() {
    MyClass object = 10;    // OK
    MyClass2 object2 = 10;  //error
    //error: conversion from 'int' to non-scalar type 'MyClass2' requested
    return 0;
}
```

PORUŠENÍ ZAPOUZDŘENÍ?

```
class X {  
private:  
    int x;  
public:  
    const X& operator = (const X& src); {  
        this->x = src.x; return *this;  
    }  
};
```

- Platí, že třída X má přístup ke všem svým atributům
 - tedy i atributům pro jiné instance této třídy (zde src)
- Porušení zapouzdření to není, protože třídu X a metodu, která ve třídě X pracuje s jinou instancí X píše "jeden" autor.

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)



STD::STRING

53

MOTIVACE PRO CHYTŘEJŠÍ ŘETĚZCE

- Práce s C řetězcí není nejsnazší
 - musíme používat speciální funkce pro manipulaci
 - musíme hlídat koncovou nulu
 - musíme hlídat celkovou velikost pole
- Chtěli bychom
 - nestarat se o velikost (zvětšení, zmenšení)
 - používat přirozené operátory pro manipulaci
 - mít snadnou inicializaci, kopírování

STL STD::STRING

- C++ nabízí ve standardní knihovně třídu `std::string`
 - přesněji, jde o instanci šablony třídy (viz. později)
- Umístěno v hlavičkovém souboru `<string>`
 - pozor, `<string.h>` je hlavička pro C řetězec, ne pro string
- Jedná se o kontejner obsahující sekvenci znaků
 - smysl použití stejný jako pro C řetězec (`char []`)
 - máme ale k dispozici řadu užitečných funkcí a operátorů
 - nemusíme se starat o velikost pole
 - automatické zvětšení/zmenšení
- <http://www.cplusplus.com/reference/string/string/>

STL STD::STRING – ZÁKLADNÍ POUŽITÍ

- Deklarace řetězce
 - `string s1;`
 - `string s2("Hello");`
- Přiřazení hodnoty
 - `s3 = "world";`
- Spojení řetězců
 - `s1 = s2 + s3; // "Helloworld"`
 - `s1 = s2 + " " + s3; // "Hello world"`
- Spojení z jednotlivým znakem
 - `s1 = s2 + 'o';`
- Přístup k jednotlivým znakům
 - `char a = s1[10]; // 10th character - No bounds checking!`
 - `char a = s1.at(10); // 10th character - with bounds checking`
- Zjištění délky řetězce
 - `int len = s1.length(); // length without ending zero`


```

#include <iostream>
#include <string>
using std::cout;
using std::cin;
using std::endl;
using std::string;

int main() {
    string s1;           // empty string
    string s2("Hello");
    string s3;
    s3 = "world";
    s1 = s2 + s3;        // "Helloworld"
    s1 = s2 + " " + s3;  // "Helloworld"

    cout << s1;
    cout << s2 + s3;

    cout << s1[10];      // 10th character - No bounds checking!
    cout << s1.at(10);   // 10th character - with bounds checking

    //cout << s1[100];    // 100th character - No bounds checking!
    //cout << s1.at(100); // 100th character - exception

    s1 = s2 + 'o';       // Append single character

    cout << "Length is " << s1.length();

    return 0;
}

```

STL STD::STRING – KONSTRUKTORY

- Řada konstruktorů pro počáteční inicializaci
- `string()` ... prázdný
- `string(const string& str)` ... kopírovací
- `string(const string& str, int start, int end)` ... podřetězec
- `string(const char* s)` ... z céčkového řetězce
- A další
 - <http://www.cplusplus.com/reference/string/string/string/>

STL STD::STRING – POROVNÁVACÍ OPERÁTORY

- K dispozici jsou běžné porovnávací operátory
 - `>`, `>=`, `<`, `<=`, `==`
 - význam stejný jako u céčkové funkce `strcmp()`
- Porovnává se na základě lexikografického uspořádání
 - `"ahoj" < "zlato"`
 - `"ahoj" == "ahoj"`
 - `"ahoj" > "achoj"`
- K dispozici přetížená metoda `compare()`
 - lze namísto operátorů
 - umožňuje i porovnání podčástí apod.
 - <http://www.cplusplus.com/reference/string/string/compare/>

```
main.cpp:14: Chyba:no matching function for call to 'std::basic_string<char>::replace(const char [6], const char [6])'  
s1.replace("Hello", "World");  
      ^
```

STL STD::STRING – DALŠÍ UŽITEČNÉ METODY

○ Vyhledávání v řetězci pomocí find()

- `s1 = "Hello world";`
- `int posWorld = s1.find("world"); // return 6`
- <http://www.cplusplus.com/reference/string/string/find/>

○ Nahrazení v řetězci pomocí replace()

- `s1 = "Hello world";`
- `s1.replace("world", "dolly"); // "Hello dolly"`
- <http://www.cplusplus.com/reference/string/string/replace/>

○ Vložení podřetězce na pozici pomocí insert()

- `s1 = "Hello world";`
- `s1.insert(6, "bloody "); // "Hello bloody world"`
- <http://www.cplusplus.com/reference/string/string/insert/>

zle, nenasiel som prototyp metody replace s iba 2 argumentmi, prekladac hlasi chybu, podľa cplusplus.com potrebuje minimalne 3 argumenty

Jediná možnosť je:

```
s1.replace(s1.find("world"), s1.length(), "dolly")
```

```
////
```

`s1.replace("world", "dolly")` je zle použitie

STL STD::STRING – KONVERZE NA C-ŘETĚZEC

- Lze konvertovat na Céčkový řetězec
- Metoda `c_str()`
 - `const char* c_str () const;`
 - včetně koncové nuly `\0`
- Metoda `data()`
 - `const char* data () const;`
 - bez koncové nuly!

STL STD::STRING – DALŠÍ POZDĚJI

- std::string je STL kontejner
- Vlastnosti a interní chování probereme u STL

SHRNUTÍ

- Dědičnost umožňuje využít kód předka v potomkovi
 - potomek může vystupovat jako datový typ předka
- Násobná dědičnost vs. kompozice
- Kopírovací konstruktor
 - definujte, pokud máte složitější atributy
- `std::string`