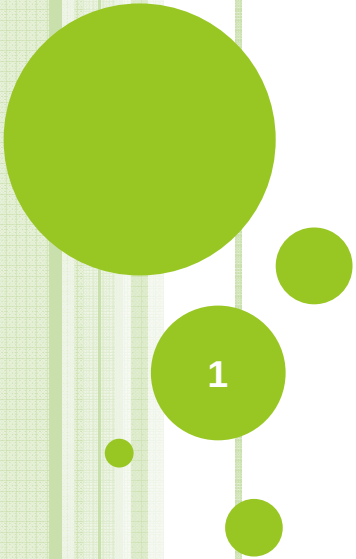
PB161 – Programování v jazyce C++
Objektově Orientované Programování



Dynamická alokace, polymorfismus

(NE)POUŽITÍ C++11 PRO DOMÁCÍ ÚKOLY

- Aktuální domácí úkoly jsou testovány s GCC 4.5.3
  - podpora C++11 v ní byla ve velmi experimentální fází
- Přechod na novější GCC uprostřed semestru není vhodný
  - možný vznik kompilačních problémů mimo testy
  - zapnutí překladu jen pro vybrané osoby je problematické (verze GCC se volí ještě předtím, než je známa osoba)
- Odevzdání s C++11 pro podzim 2013 tedy není možné
- Řešení (od dalšího semestru)
  - přejdeme na nové GCC 4.7.x
  - student si sám zvolí standard kompilace (C++98 nebo C++11)

SCRIBD – PŘIDÁNÍ POZNÁMEK

- Týden po začátku semestru přešel Scribd na nové rozhraní
  - funkčně stejné, ale vkládání poznámek se chová divně
  - vložené poznámky se zobrazí se jenom někdy
- Vypsal jsem bugreport, zkuste prosím také
  - at zdvihneme prioritu ☺

Dynamická alokace paměti

Dynamická alokace - motivace

- U statické alokace je velikost nebo počet známa v době překladu
  - `int array[100];`
  - `CClass a, b, c;`
- Často ale tyto informace nejsou známy
  - např. databáze se rozšiřuje
  - např. textový popisek má typicky 20 znaků, ale může mít i 1000
    - neefektivní mít vždy alokováno 1000 znaků
- Dynamická alokace umožňuje vytvořit datovou položku (proměnnou, pole, objekt) až za běhu

Statická vs. dynamická alokace

- Statická alokace alokuje na zásobníku (stack)
  - automaticky se uvolňuje po dokončení bloku kódu
    - typicky konec funkce, konec cyklu for…
  - výrazně rychlejší (obsah zásobníku je typicky v cache)
  - využití pro krátkodobé proměnné/objekty
- Dynamická alokace na haldě (heap)
  - zůstává do explicitního uvolnění (nebo konce aplikace)
  - využití pro dlouhodobé entity

Dynamická alokace v C

- malloc() & free()
- Alokuje potřebný počet bajtů
  - (musíme správně vypočíst)
  - typicky používáme počet_prvků*sizeof(prvek)
- Přetypujeme na požadovaný typ (je riziko chyby)
  - `int* array = (int*) malloc(100 * sizeof(int));`
- Po konci práce uvolníme
  - free(array);

Dynamická alokace v C++

```
A*     pTest = new A;
B*     pTest2 = new B(20);
A*     array = new A[100];
```

- Operátor new & delete & delete[]
- Dva základní typy alokace
  - alokace jedné entity (třída, jednoduchý typ, struktura)
  - alokace pole objektů (několik entit v paměti za sebou)
- Zadáváme jen počet prvků daného typy
  - potřebné místo v bajtech je automaticky dopočteno
  - tj. typicky nepoužíváme sizeof(prvek)
- Vzniká entita s přiřazeným datovým typem
  - typicky neprovádíme přetypování
- Je volán konstruktor objektu
  - pokud alokuje pole objektů, musí být bezparametrický

Dynamická alokace jednoho objektu

```
class A {
    int m_value;
public:
    A(int value) : m_value(value) {}
    void print() const {cout << m_value << endl; }
};
```

```
// Allocate A object dynamically
// If fail, std::bad_alloc exception is thrown
A*    pTest2 = 0;
pTest2 = new A(20);
pTest2->print();
if (pTest2) {
    delete pTest2;
    pTest2 = 0;
}
```

- Je zavolán konstruktor (new) i destruktor (delete)

Operátor -> vs. operátor .

- Operátor . se použije pro přístup k položkám objektu nebo struktury

```
A test1(20);
test1.print();
```

```
A* pTest2 = new A(20);
(*pTest2).print();
pTest2->print();
delete pTest2;
```

- Pokud objekt alokujeme dynamicky
  - máme proměnnou typu ukazatel
  - operátor . nelze přímo použít
  - musíme nejprve dereferencovat
- Pro zjednodušení je dostupný operátor ->
  - (*pObjekt).atribut == pObjekt->atribut

Chybové stavy při alokaci

- Pokud se alokace nepodaří, je vyvolána výjimka
  - std::bad_alloc
  - lze zachytit a reagovat (viz. přednáška výjimky později)
- Varianta operátoru new bez vyvolání výjimky
  - int * p2 = new (nothrow) int;
  - pokud se nezdaří, je vráceno NULL (=0)

Dynamická alokace objektu bez výjimky

```
class A {
    int m_value;
public:
    A(int value) : m_value(value) {}
    void print() const {cout << m_value << endl; }
};
```

```
// Allocate A object dynamically, without exception
A* pTest2 = new (std::nothrow) A(20);
if (pTest2) {
  pTest2->print();
  delete pTest2;
  pTest2 = 0;
}
```

- Je zavolán konstruktor i destruktor

Dynamické alokace bez volání konstruktoru

```
class A {
    int m_value;
public:
    A(int value) : m_value(value) {}
    void print() const {cout << m_value << endl; }
};
```

```
// Allocate A object dynamically without constructor called
A* pTest2 = static_cast<A*>(operator new (sizeof(A)));
pTest2->print();
if (pTest2) {
    delete pTest2;
    pTest2 = 0;
}
```

- Konstruktor není zavolán!
- Nepoužívá se často

Dynamické alokace pole objektů

```
class B {
    int m_value;
public:
    void setValue(int value) { m_value = value; }
    void print() const {cout << m_value << endl; }
};
```

```
// Allocate array of integers
const int NUM_ITEMS = 10;
int* array = new int[10];
delete[] array;

// Allocate array of A objects
// A* pTest2 = new A[NUM_ITEMS];  // wrong: Why?
B* pTest3 = new B[NUM_ITEMS];
for (int i = 0; i < NUM_ITEMS; i++) pTest3[i].setValue(5);
if (pTest3) {
     delete[] pTest3;
     pTest3 = 0;
}
```

DYNAMICKÁ ALOKACE A DĚDIČNOST

- Třídu A a B budeme používat v dalších ukázkách

```
class A {
protected:
    int m_value;
public:
    void setValue(int value) {
        m_value = value;
        cout << "A::setValue() called" << endl;
    }

    virtual int getValue() const {
        cout << "A::getValue() called" << endl;
        return m_value;
    }

    static void printValue(int value) {
        cout << "Value = " << value << endl;
        cout << "A::printValue() called" << endl;
    }
};
```

```
class B : public A {
public:
    void setValue(int value) {
        m_value = value;
        cout << "B::setValue() called" << endl;
    }

    virtual int getValue() const {
        cout << "B::getValue() called" << endl;
        return m_value;
    }

    static void printValue(int value) {
        cout << "Value = " << value << endl;
        cout << "B::printValue() called" << endl;
    }
};
```

Dynamická alokace a dědičnost

- Ukazatel na potomka můžeme přiřadit do ukazatele typu předka
  - `A* pObject = new B;`
  - na objekt typu B se přes rozhranní typu A
- Proběhne implicitní přetypování

```
// Retype B to A via pointers (compile time)
A* pObject = new B;
pObject->setValue(10);  // from A
pObject->getValue();    // from B (virtual)
pObject->printValue(15);// from A (static)
```

Uvolňování paměti

- Jednotlivé objekty uvolníme pomocí delete
  - nastavujte po uvolnění na = 0 pro zabránění opakovanému uvolnění
- Pole objektů uvolníme pomocí delete[]
  - musíme použít ukazatel na začátek pole
  - pokud použijeme ukazatelovou aritmetiku, tak uvolnění neproběhne
- Nepoužívaná, ale neuvolněná paměť => memory leaks
  - problém hlavně u dlouhoběžících resp. paměťově náročných apps

MEMORY LEAKS

- Dynamicky alokovaná paměť musí být uvolněna
  - dealokace musí být explicitně provedena vývojářem
  - (C++ nemá Garbage collector)
- Valgrind – nástroj pro detekci memory leaks (mimo jiné)
  - valgrind -v --leak-check=full testovaný_program
  - http://wiki.forum.nokia.com/index.php/Using_valgrind_with_Qt_Creator
- Microsoft Visual Studio
  - automaticky zobrazuje detekované memory leaks v debug režimu
- Detekované memory leaks ihned odstraňujte
  - stejně jako v případě warningu
  - nevšimnete si jinak nově vzniklých
  - obtížně dohledáte místo alokace

MEMORY LEAKS - DEMO

```
#include <cstdlib>
int main(int argc, char* argv[]) {
  int*  pArray = 0;
  int   bVar;              // Intensionally uninitialized

  pArray = new int[25];
  if ((argc == 2) &&  atoi(argv[1]) == 1) {
    pArray = new int[25];     // Will cause "100 bytes in 1 blocks are definitely lost in loss record" Valgrind error
  }
  if (pArray) delete[] pArray;

  int*  pArray2 = 0;
  pArray2 = new int[25];
  pArray2 += 20;                    // Pointer inside pArray2 was changed, delete[] will not work correctly
                          // Will cause "100 bytes in 1 blocks are definitely lost in loss record" Valgrind error
  if (pArray2) delete[] pArray2;   // Will not delete allocated memory as pArray2 is different from address returned by new[]

  if (bVar) {                       // Will cause "Conditional jump or move depends on uninitialised value" Valgrind error
    pArray[10] = 1;
  }

  return 0;
}
```

Automatický ukazatel std::auto_ptr

- Velice omezený garbage collection
  - #include <memory>
  - založeno na šablonách (bude probráno později)
- auto_ptr<typ_třídy> p(new typ_třídy);
- Speciální objekt, který “obalí” naši třídu
  - pokud zaniká, tak zavolá destruktor naší třídy a uvolní ji
  - přetížené operátory * a ->
    - s výsledkem pracujeme stejně jako s ukazatelem na třídu
- Využití pro neočekávané ukončení funkce
  - např. vyvolána výjimka => mohl nastat memory leak
- Nelze využít pro pole (volá se delete, ne delete[])
- Nepočítá zbývající reference (jako dělá Java, .NET)
- *Depricated* (nedoporučeno pro použití) v C++11

AUTOMATICKÝ UKAZATEL - UKÁZKA

```
class A {
    int m_value;
public:
    A(int value) : m_value(value) {}
    ~A() { cout << "~A() called" << endl; }
    void print() const {cout << m_value << endl; }
};
```

```
#include <iostream>
#include <memory>
void foo() {
    std::auto_ptr<A> pValue(new A(20));
    pValue->print();
    (*pValue).print();

    // No need to deallocate pValue
}

int main() {
    // foo() allocates dynamically, but uses auto_ptr
    for (int i = 0; i < 100; i++) {
        foo();
    }

    return 0;
}
```

Automatický ukazatel unique_ptr

C++11

- Další z chytrých ukazatelů, dostupný od C++11
  - https://en.wikipedia.org/wiki/Smart_pointer#unique_ptr
- auto_ptr nefunguje vhodně v případě, že kopírujeme
  - původní auto_ptr zůstává prázdný, umístěný ukazatel se přesune do nového

```
std::unique_ptr<int> p1(new int(5));
std::unique_ptr<int> p2 = p1; //Compile error.
//Transfers ownership. p3 now owns the memory and p1 is invalid.
std::unique_ptr<int> p3 = std::move(p1);

p3.reset(); //Deletes the memory.
p1.reset(); //Does nothing.
```

Dynamická alokace vícedimenz. polí

- Nemusíme alokovat jen jednorozměrné (1D) pole
- 2D pole je pole ukazatelů na 1D pole
- 3D pole je pole ukazatelů na pole ukazatelů na 1D pole

```
// Multi-dimensional array
// Let's allocate 100x30 array of integers

// 100 items array with elements int*
int** array2D = new int*[100];
for (int i = 0; i < 100; i++) {
    // allocated 1D array on position array2D[i]
    array2D[i] = new int[30];
}
array2D[10][20] = 1;

// You need to delete it properly
for (int i = 0; i < 100; i++) {
    delete[] array2D[i];
}
delete[] array2D;
```

Dynamická alokace - poznámky

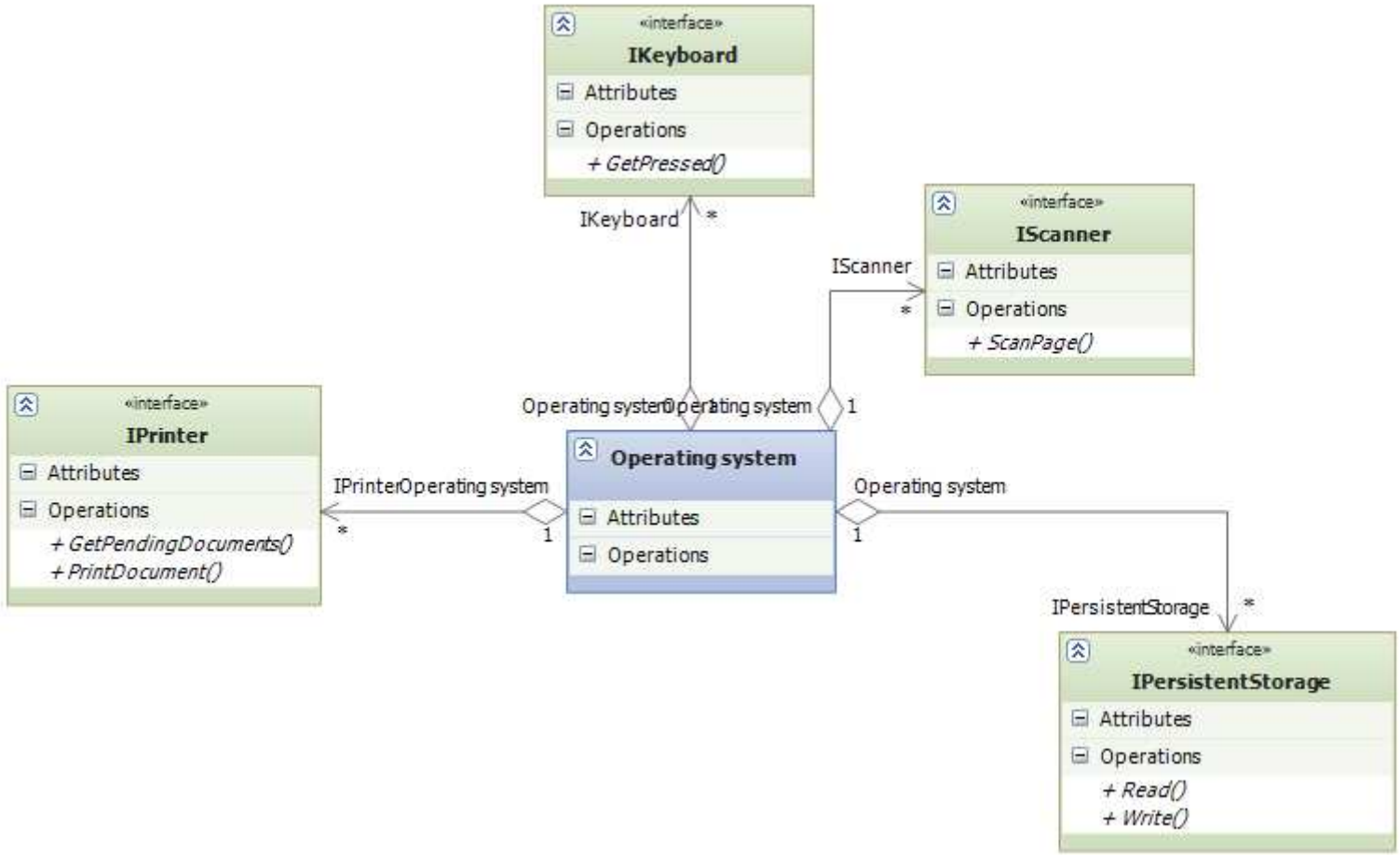
- Při dealokaci nastavte proměnnou zpět na NULL
  - opakovaná dealokace nezpůsobí pád
- Při dynamické alokaci objektu je volán konstruktor
  - u pole objektů je potřeba bezparametrický konstruktor
- Dynamická alokace je důležitý koncept
  - je nutné znát a rozumět práci s ukazateli (viz. PB071)

POLYMORFISMUS

Motivace pro polymorfismus

- Může Microsoft/Torvalds vyvinout operační systém bez znalosti konkrétního hardware na kterém bude provozován?
  - Tiskárny
  - Harddisk
  - CPU
  - Grafická karta
  - …

MOTIVACE PRO POLYMORFISMUS



Motivace pro polymorfismus

- Způsob jak v C++ realizovat rozhraní (interface)
- Možnost vývoje bez znalosti implementace
  - Windows/Linux bez znalosti všech budoucích tiskáren
- Možnost změny implementace bez změny používajícího kódu
- Typy polymorfismu
  - Přetížení – funkce se stejným jménem, ale různými argumenty
  - Překrytí – funkce se stejným jménem i argumenty
  - (Kombinace přetížení a překrytí)

POLYMORFISMUS - PRINCIP

- Zavolání metody se stejným jménem způsobí vykonání různého kódu
  - např. add(float real), add(float real, float imag)
  - např. metoda print() pro různé objekty
- K čemu je to dobré?
  - můžeme mít speciální metody pro různé varianty argumentů
  - nemusíme vědět, s jakým přesně objektem zrovna pracujeme
    - jen víme, ze které třídy je potomek
  - dokážeme pracovat s třídami, které nebyly známé v době psaní kódu
    - jsou potomky tříd, které známy byly

POLYMORFISMUS – STATICKÁ VAZBA

Typy polymorfismu - přetížení

- Více metod se stejným jménem, ale s různými typy anebo počty argumentů
- Během překladu poznáme, kterou metodu zavolat
  - jde typicky o statickou vazbu
  - ale může být i pozdní (viz. dále)

```
// Overloaded add method
void add(float realPart) { m_realPart += realPart; }
void add(float realPart, float imaginary) {
    m_realPart += realPart; m_imagPart += imaginary;
}
void add(const CComplexNumber &value) {
    setReal(m_realPart + value.getReal());
    setImaginary(m_imagPart + value.getImaginary());
}
```

VČASNÁ(STATICKÁ) VAZBA (STATIC BINDING)

- Včasnou vazbu může rozhodnout kompilátor při překladu
- Funkční volání je nahrazeno přímo adresou, kde je kód volané funkce
- Tuto adresu již nelze změnit později za běhu programu
- Rychlé!

```
 27          55
 28          56          // We will call different overloaded methods
 29          57          // based on argument type
 30          58          value1.add(10.3);
=> 31 0x00401385  <+65>:              mov    $0x4124cccd,%eax
 32 0x0040138a  <+70>:              mov    %eax,0x4(%esp)
 33 0x0040138e  <+74>:              lea    0x18(%esp),%eax
 34 0x00401392  <+78>:              mov    %eax,(%esp)
 35 0x00401395  <+81>:              call   0x41f600 <_ZN14CComplexNumber3addEf>
 36
```

Typy polymorfismu – přetížení (2)

- Překladač musí být schopen rozhodnout volanou metodu
- Může být ovlivněno implicitním přetypování
  - např. nemusíme mít metodu pro int, stačí pro float
    - int se automaticky přetypuje na float
  - souběžné metody pro int a float mohou způsobit problém
- Metody se v C++ nemohou lišit jen návratovým typem
  - nemuselo by být jasné, kterou metody použít

```
class A {
public:
    int MyFunction() { return 1; }
    const char* MyFunction() { return "1"; }
};
```

```
int main() {
    A object;
    // Which one to call?
    object.MyFunction();
    return 0;
}
```

  - lze obejít pomocí funkčních objektů, overloadingByValueTrickDemo.cpp

PŘETÍŽENÍ - POZNÁMKY

- Není omezeno na metody objektů, lze i pro obyčejné funkce

```
int divFnc(int a, int b) {
    return a / b;
}
float divFnc(float a, float b) {
    return a / b;
}
```

```
int main() {
    // Přetížení funkcí
    int c = divFnc(10, 5);
    // int c = divFnc(10.4, 5.4); // problém, proč?
    return 0;
}
```

- Zvolená funkce zůstává po kompilaci vždy stejná
  - dokud není provedena rekompilace
  - rychlejší a paměťově úspornější

Defaultní hodnoty parametrů metod

- Polymorfismus lze i přes defaultní parametry
- Částečná alternativa k několika přetíženým metodám
  - defaultní hodnota argumentu se uvádí v deklaraci metody
  - `int div(int first, int second, int* pRem = NULL);`
    - pokud uživatel chce, dostane zpět i zbytek po dělení
- Za prvním argumentem s defaultní hodnotou už musí mít všechny defaultní hodnotu
  - nelze `int div(int a , int b = 0, int c);`

Defaultní hodnoty parametrů - použití

- Často se používá při pozdějším přidávání argumentů funkce
  - není potřeba měnit stávající kód
  - stávající kód jakoby použije defaultní hodnotu
  - např. přidání možnosti získání zbytku po dělení
    - int div(int first, int second);
    - int div(int first, int second, int* pRem = NULL);
- Nově přidaný argument by ale neměl měnit původní chování funkce bez něj

PŘETÍŽENÍ – UKÁZKA

Ukázky

- overloadingDemo.cpp
- CComplexNumber
- Přetížení konstruktorů
- Přetížení metody add
- Problém při podobnosti datových typů
- Problém při přetížení jen návratovou hodnotou

POLYMORFISMUS – POZDNÍ VAZBA

Typy polymorfismu - překrytí

- Voláme metodu se stejným jménem i argumenty
- Metoda přesto může mít jinou implementaci (tělo)
  - jinou pro objekt A a jinak pro objekt B
    - pokud píšeme rychlejší XML parser, nemusíme vymýšlet nové jména metod od existujícího
  - jinak pro předka a jinak pro jeho potomka
    - potomek mění implementaci metody zděděné od předka
  - rozhodnutí na základě reálného typu aktuálního objektu
- Jde o pozdní vazbu

POZDNÍ VAZBA (DYNAMIC BINDING)

- Během překladu není ještě pevně zafixován volaný kód
- Přímé funkční volání je nahrazeno náhledem do tabulky virtuálních metod konkrétního objektu
  - teprve v tabulce je adresa cílové funkce, která se zavolá
  - tabulku lze modifikovat za běhu => implementace zvolena za běhu

```
47          78          // Call overriden add method
48          79          valueParentType.add(10.3);
49 0x00401398  <+84>:        mov    0x1c(%esp),%eax
50 0x0040139c  <+88>:        mov    (%eax),%eax
51 0x0040139e  <+90>:        mov    (%eax),%edx
52 0x004013a0  <+92>:        mov    $0x4124cccd,%eax
53 0x004013a5  <+97>:        mov    %eax,0x4(%esp)
               mov    0x1c(%esp),%eax
               mov    %eax,(%esp)
               call   *%edx
```

připomenutí – statická vazba

```
value1.add(10.3);
           mov    $0x4124cccd,%eax
           mov    %eax,0x4(%esp)
           lea    0x18(%esp),%eax
           mov    %eax,(%esp)
           call   0x41f600 <_ZN14CComplexNumber3addEf>
```

POZDNÍ VAZBA - SYNTAXE

- Klíčové slovo virtual
- Metodu označíme jako virtual v předkovi
- V potomcích už nemusíme
  - pro lepší čitelnost lze uvádět

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

- Nelze změnit v potomcích zpět na statickou vazbu!

VHODNOST UVÁDĚNÍ KLÍČOVÉHO SLOVA VIRTUAL

```
class base
{
public:
    virtual void foo();
};

class derived : public base
{
public:
    void foo();
}
```

```
class base
{
public:
    virtual void foo();
};

class derived : public base
{
public:
    virtual void foo();
}
```

- Uvedené zápisy jsou funkčně ekvivalentní
- Vhodnější je druhý zápis – z deklarace derived je zřejmé, že foo() je virtuální

Překrytí – ukázka

Ukázky

- overridingDemo.cpp
- CComplexNumber, CComplexNumberCommented
- Přidání klíčového slova virtual
- Překrytí metody add(float real) a print()
- Zastínění metody add(float real, float imaginary)

POLYMORFISMUS – VHODNOST POUŽITÍ

- Kdy použít statickou vazbu?
- Pokud je zřejmé, že potomci nebudou chtít překrýt dynamicky
  - potomci mohou stále předefinovat nebo skrýt
- Pokud je žádoucí vysoká rychlost
  - méně instrukcí pro zavolání
  - lze použít dohromady s inline metodami
    - namísto funkčního volání se vloží celý kód metody

POLYMORFISMUS – VHODNOST POUŽITÍ (2)

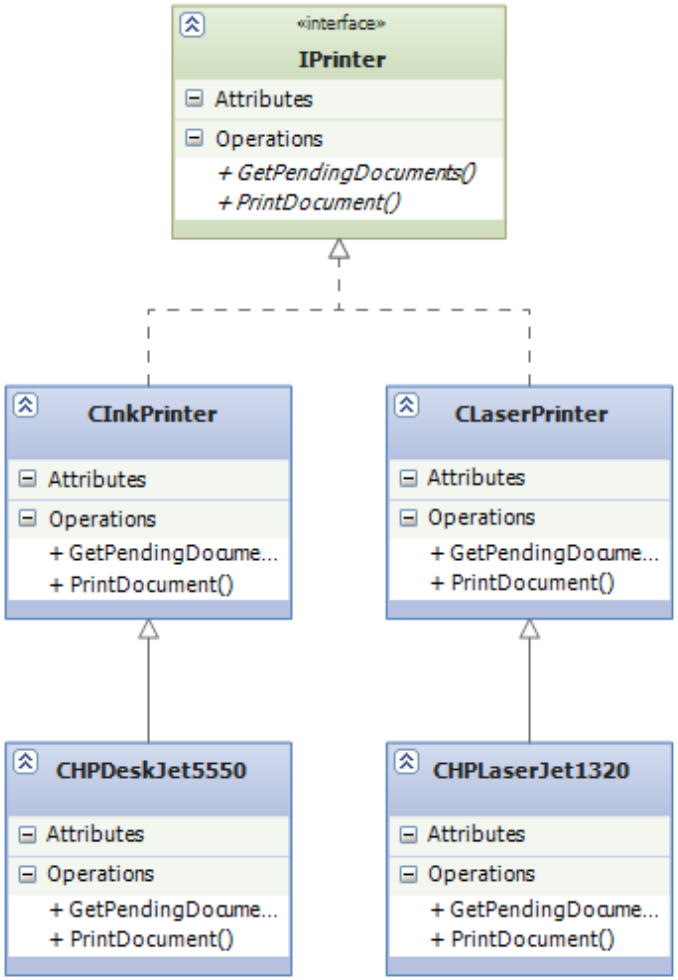
- Kdy použít pozdní vazbu?
- Pro metody, které budou reimplementovat potomci po svém
  - např. “vykresli” u grafického objektu
  - zlepšuje rozšiřitelnost
  - generická část kódu může být v předkovi
- Pokud typ objektu ovlivňuje chování třídy
  - použití virtuálních funkcí
  - jinak vhodnější použití šablony (template, viz. později)

POLYMORFISMUS – DEFAULTNÍ NASTAVENÍ

- V C++ jsou všechny metody defaultně nevirtuální
  - statická (včasná) vazba
  - výhodné z důvodu rychlosti (statická je rychlejší)
- Některé OOP jazyky mají právě naopak
  - v Javě jsou všechny metody defaultně virtuální
- Důvodem je kompromis mezi rychlostí a flexibilitou
  - C++ straní rychlosti

POLYMORFISMUS – POKRAČOVÁNÍ

HIERARCHIE

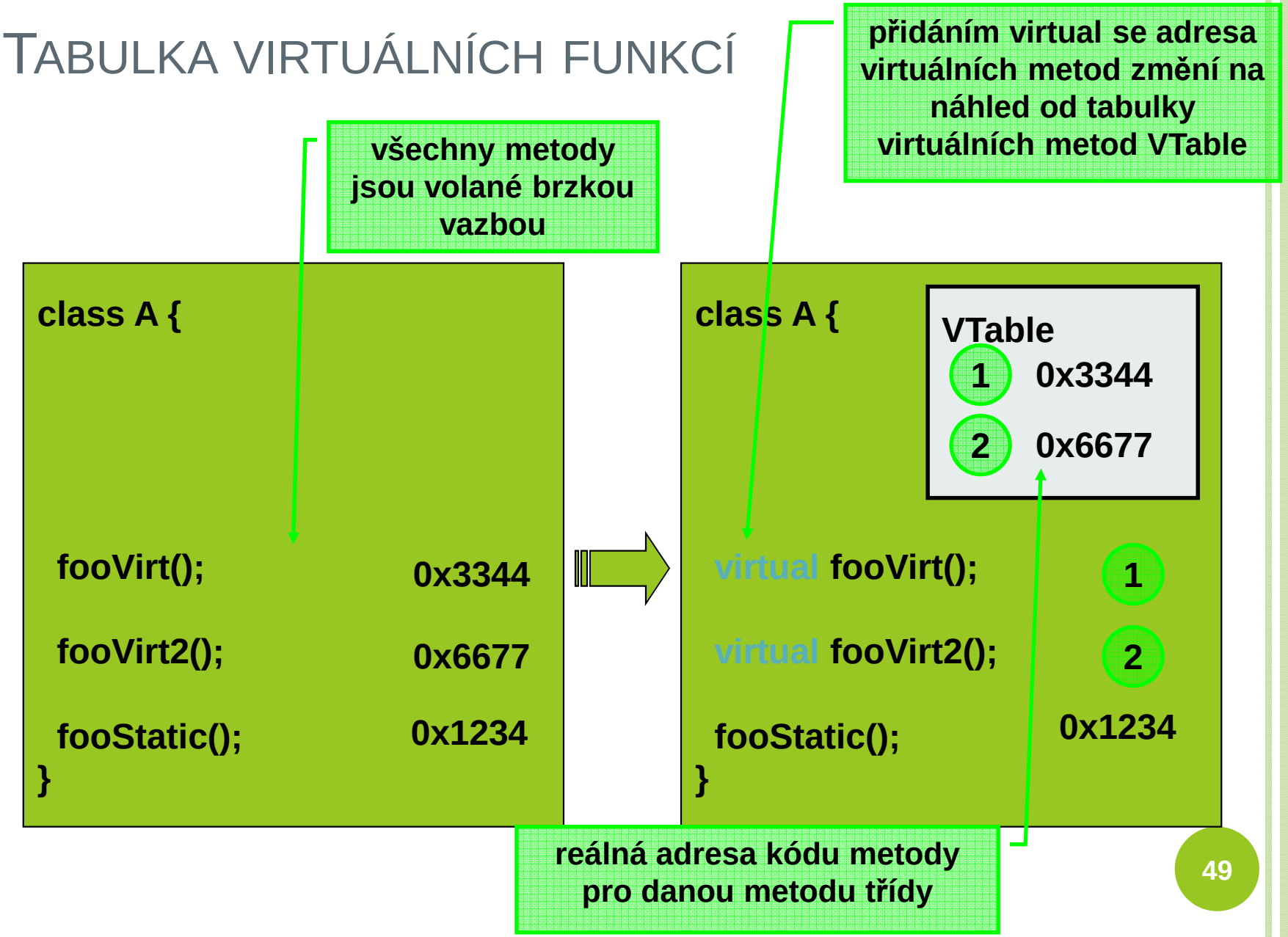


Čistě abstraktní třída (bez implementace)

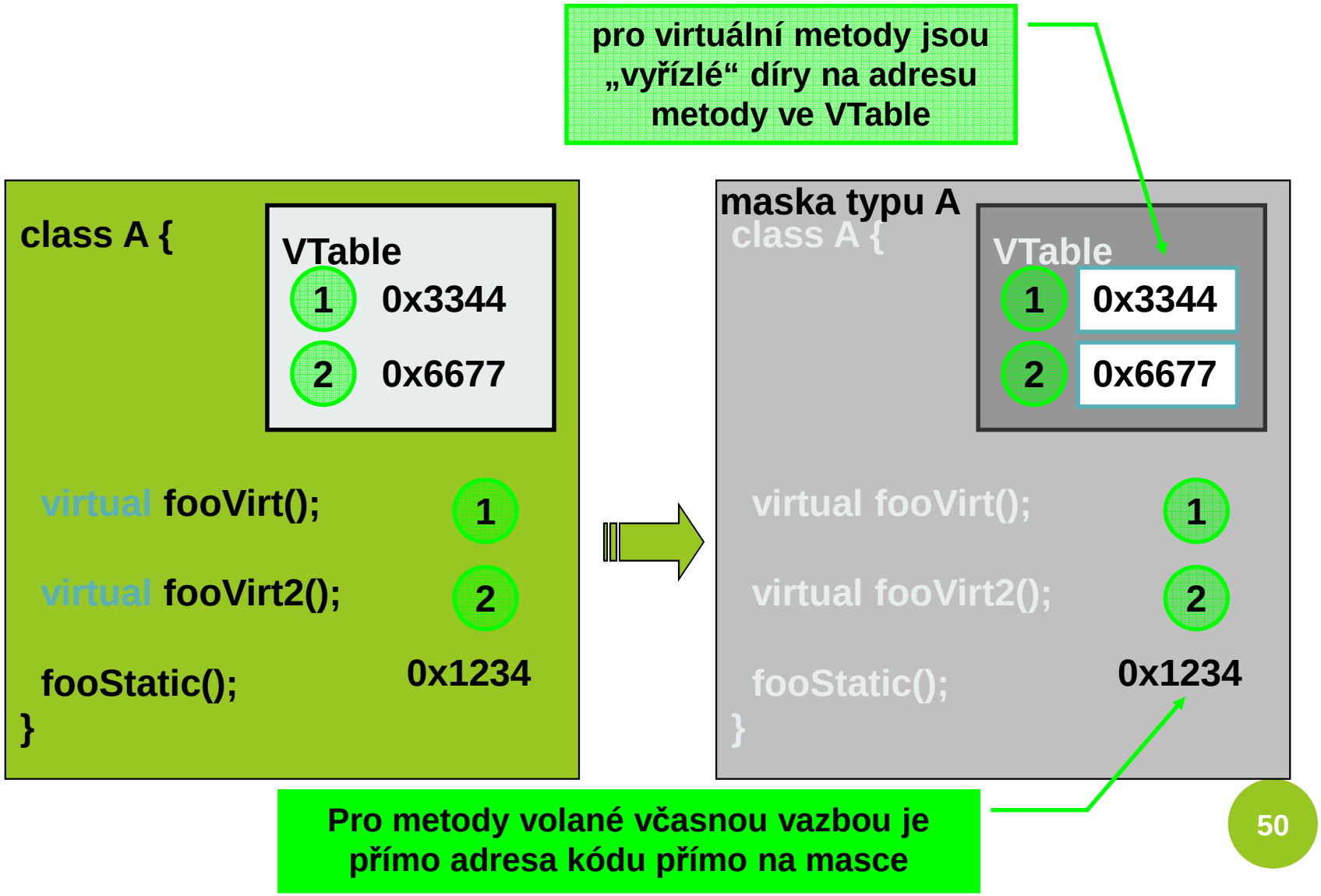
Abstraktní třída (může být část implementace)

Třída s implementací (děláme objekty)

TABULKA VIRTUÁLNÍCH FUNKCÍ

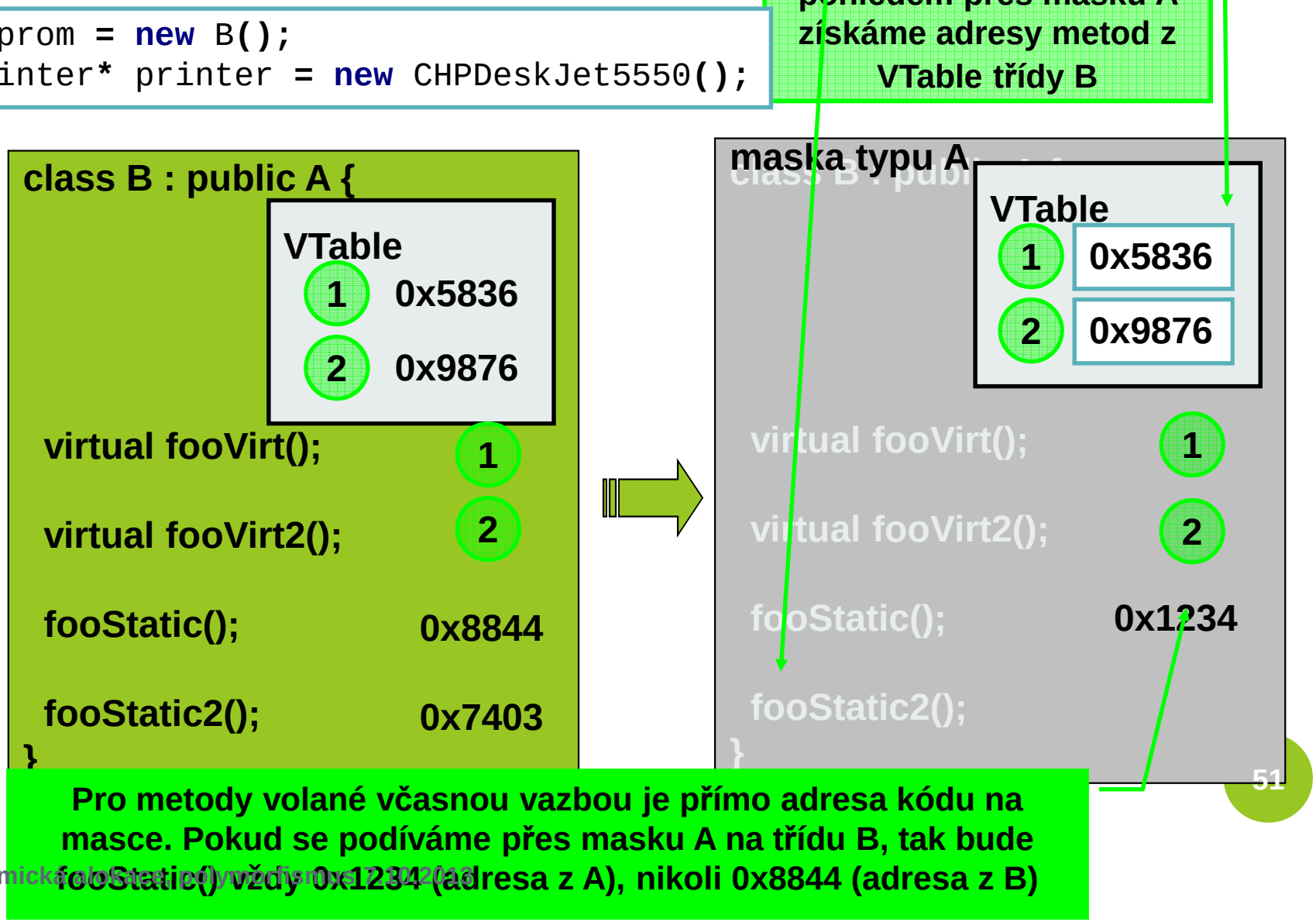


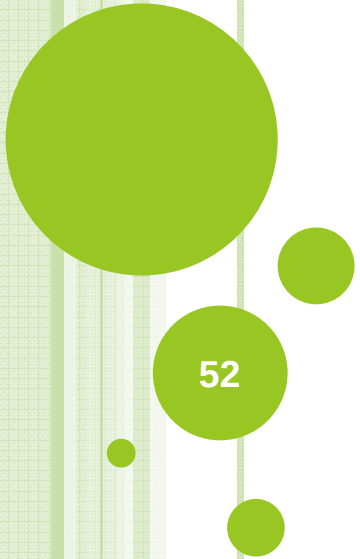
DATOVÝ TYP JE „MASKA“



PŘETYPOVÁNÍ B NA A

```
A* prom = new B();
IPrinter* printer = new CHPDeskJet5550();
```

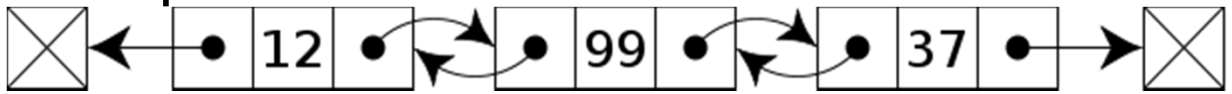




STD::LIST

KONTEJNER LIST

- Spojovaný seznam pro rychlé vkládání/odebírání
  - používán velice často
  - spojovaný seznam schránek
  - jednotlivé schránky obsahují zvolený datový typ
- Nepřistupuje se přes index, ale pomocí iterátorů
- Syntaxe použití




```
#include <list>
int main() {
  std::list<int> myList;
  return 0;
}
```

- http://www.cplusplus.com/reference/stl/list/

KONTEJNER LIST – UKÁZKA DOKUMENTACE

Kam vložit. Za zadaný iterátor.

Co vložit. Stejný datový typ s jakým je instance šablony vytvořena.

list::insert

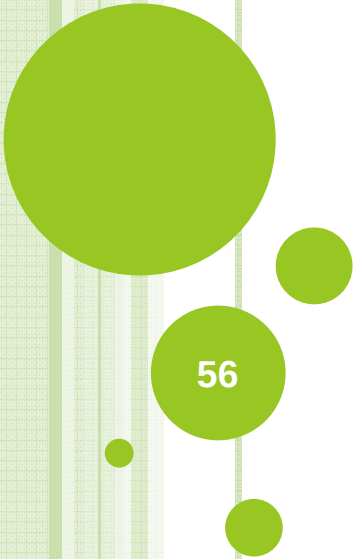
```
iterator insert ( iterator position, const T& x );
    void insert ( iterator position, size_type n, const T& x );
template <class InputIterator>
    void insert ( iterator position, InputIterator first, InputIterator last );
```

Lze vložit i sekvence prvků zároveň. Od iterátoru `first` po dosažení iterátoru `last`.

KONTEJNER LIST – UKÁZKA

```
std::list<int> myList;

myList.push_back(1);
myList.push_back(2);
myList.push_back(3);
myList.push_front(4);
myList.push_front(5);

cout << "List size: " << myList.size() << endl;
// Use iterator to output all value in list
std::list<int>::iterator iter;
for (iter = myList.begin(); iter != myList.end(); iter++) {
    cout << *iter << endl;
}

iter = myList.begin(); // get iterator to begin
iter++;                // jump to next item
myList.insert(iter, 10); // insert after this item

myList.clear();
cout << "List size: " << myList.size() << endl;
```

> ***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 DESTRUKTOR?

Problém s nevirtuálním destruktorem

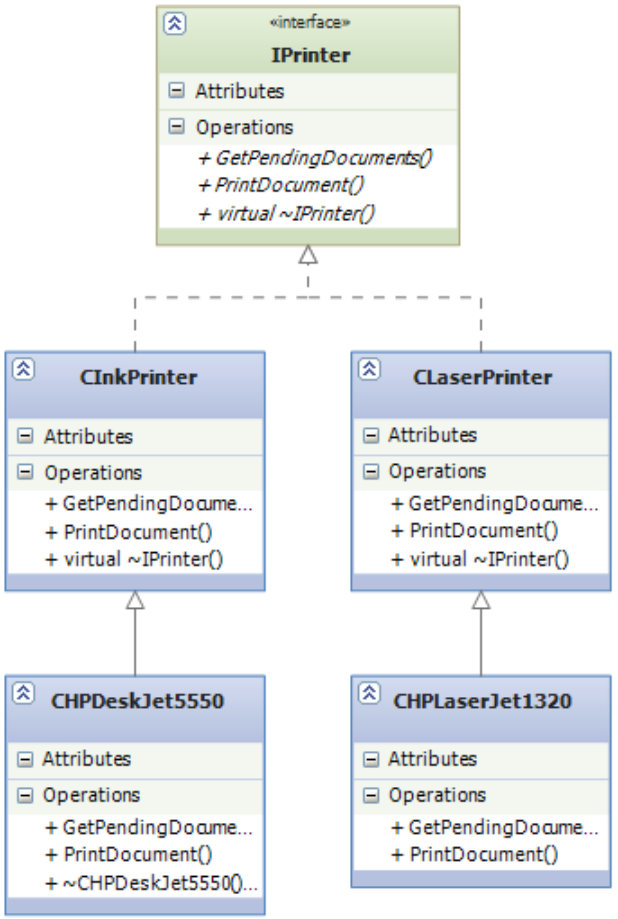
```
class IPrinter {
public:
    virtual string GetPendingDocuments() const = 0;
    virtual void PrintDocument(const string& document) = 0;
};

class CHPDeskJet5550 : public IPrinter {
    string m_printedDocument;
public:
    CHPDeskJet5550() {}
    virtual string GetPendingDocuments() const {
        return m_printedDocument;
    }
    virtual void PrintDocument(const string&
        m_printedDocument = document;
    }
    ~CHPDeskJet5550() {
        cout << "~CHPDeskJet5550() called";
    }
};
```

```
void demoNonVirtDestructor() {
    // Create new HPDeskJet5550 printer
    IPrinter* printer = new CHPDeskJet5550();

    // Use printer without knowledge about its type
    // We are using IPrinter interface
    printer->PrintDocument("secret");
    cout << printer->GetPendingDocuments() << endl;

    // ??? which destructor will be called?
    delete printer; // Why not ~CHPDeskJet5550()?
}
```

VIRTUÁLNÍ DESTRUKTOR

- Je velmi vhodné deklarovat už na úrovni rozhraní
  - virtual ~IPrinter() {};
  - nevíme předem, zda nebude některý potomek potřebovat
  - pokud neuděláme, nutíme potomka provádět úklid jiným způsobem
    - speciální „úklidová“ metoda (omezuje výhody dědičnosti)
    - využití automatických ukazatelů (nelze pro vše)
  - poskytujeme i prázdnou implementaci
- Typické použití u tříd vyžadující explicitní úklid
  - pokud proběhla dynamická alokace (dealokace)
  - pokud je potřeba uvolnit systémové prostředky (socket, kontext...)

HIERARCHIE DOPLNĚNÁ O VIRT. DESTRUKTOR



Čistě abstraktní třída (bez implementace)

Abstraktní třída (může být část implementace)

Třída s implementací (děláme objekty)

Shrnutí

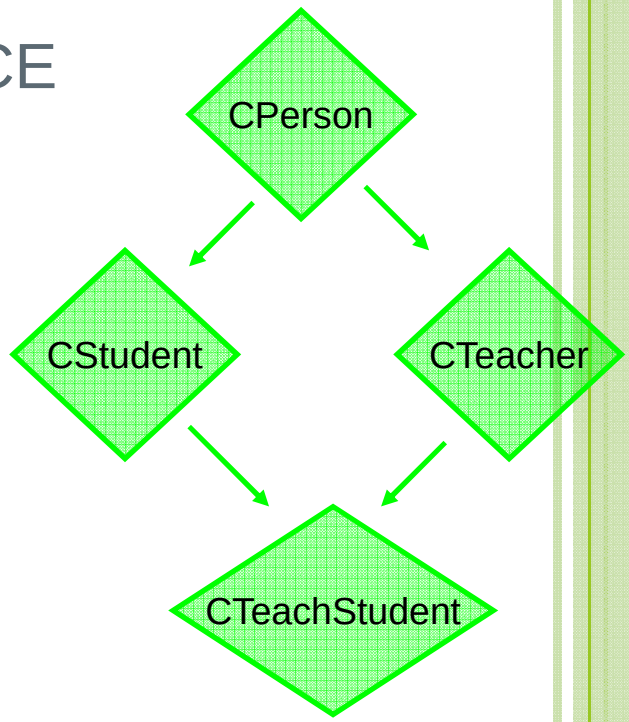
- Dynamická alokace
  - vznik objektů na haldě
  - nutnost dealokace
  - volání destruktoru
- Polymorfismus – důležitý koncept
  - přetížení (overloading)
  - překrytí (overridding)

prázdny slide - zmazať ???

SAMOSTUDIUM

Problém diamand, virtuální dědičnost

Problém typu diamant - motivace



- Třída CPerson
  - string email, getEmail()…
- Třída pro studenta
  - class CStudent: public CPerson;
  - getEmail() vrací formát xnovak@fi
- Třída pro učitele
  - class CTeacher: public CPerson;
  - getEmail() vrací formát novak@fi
- Cvičící, ale také student
  - class CTeachStudent: public CStudent, public CTeacher;
  - Co vrátí volání getEmail()?

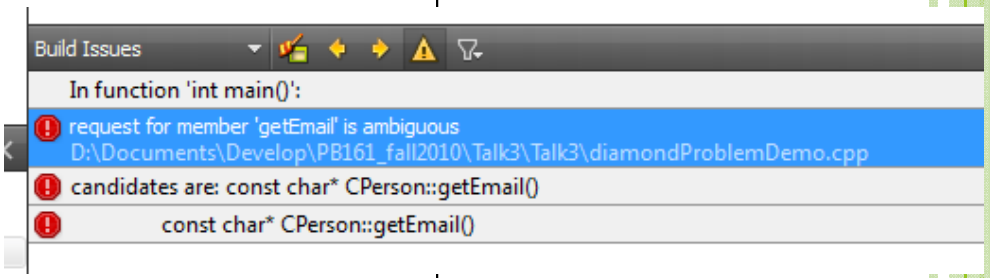
Problém typu diamant – kde je problém?

- Nelze zkompilovat
- Nevhodné použití dědičnosti
- (Vyskytuje se často)

DIAMANT – UKÁZKA

```
#include <iostream>
#include <string.h>
using namespace std;
#define MAX_EMAIL_LENGTH    30
class CPerson {
protected:
   char m_email[MAX_EMAIL_LENGTH+1];
public:
   CPerson(const char* email) {
      strncpy(m_email, email, MAX_EMAIL_LENGTH);
      m_email[MAX_EMAIL_LENGTH] = 0;
   }
   const char* getEmail() { return m_email; }
};

class CStudent : public CPerson {
public:
   CStudent(const char* email) : CPerson(email) {}
};

class CTeacher : public CPerson {
public:
   CTeacher(const char* email) : CPerson(email) {}
};

class CTeachStudent : public CTeacher, public CStudent {
public:
   CTeachStudent(const char* email) : CTeacher(email), CStudent(email) {}
};
```

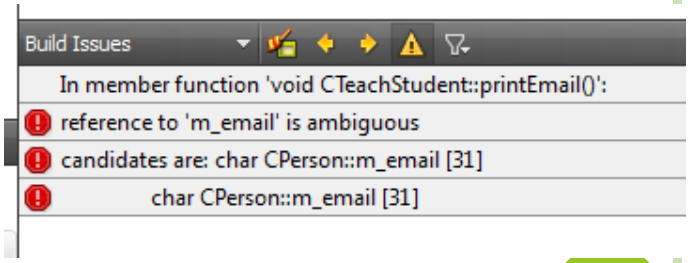
```
int main() {
   CTeachStudent teachStud("novak@fi.muni.cz");
   teachStud.getEmail();

   return 0;
}
```



Problém typu diamant

- Vzniká při nevhodném využití násobné dědičnosti
- Třída CTeachStudent zdědila jednu kopii metody getEmail od CStudent a druhou od CTeacher
- Při volání metody getEmail() třídy CTeachStudent není jasné, která kopie dat/metod se má použít
  - getEmail() z CStudent nebo CTeacher?
- Obdobný problém by vnikl ve funkci printEmail()

```
void CTeachStudent::printEmail() {
        cout << m_email;
}
```



DIAMANT – PLNÁ KVALIFIKACE

- Řešení pomocí plné kvalifikace jména metody/atributu
  - teachStud.CTeacher::getEmail();
  - CTeacher::m_email;
- Je nutné znát hierarchii, porušuje myšlenku

```
class CTeachStudent : public CTeacher, public CStudent {
public:
    CTeachStudent(const char* email) : CTeacher(email), CStudent(email) {}
    void printEmail() {
        //cout << m_email;
        cout << CTeacher::m_email;
    }
};
int main() {
    CTeachStudent teachStud("novak@fi.muni.cz");
    //teachStud.getEmail();
    teachStud.CTeacher::getEmail();

    return 0;
}
```

DIAMANT - VIRTUÁLNÍ DĚDIČNOST

- Pomocí klíčového slova virtual přikážeme jedinou kopii atributů a tabulky metod
  - používat opatrně
  - class CStudent: virtual public CPerson;
- Problémem je, že už při deklaraci CStudent musíme “tušit”, že později nastane diamant
- Pokud se mixuje virtuální a nevirtuální dědičnost
  - jen některá větev používá virtuální dědičnost
  - tak stále vzniká více kopií

Dědičnost a virtuální dědičnost

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

další prázdny slide