

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

**Načítání a zápis PDB souboru**

# Čistě virtuální metody, abstraktní třídy

- Při použití polymorfismu někdy v rodičovské třídě potřebujeme metodu, která nic nedělá, ale všechny odvozené objekty ji musí překrýt (např. Shape::draw())
- Tuto metodu můžeme deklarovat jako **čistě virtuální** (*pure virtual*) použitím **= 0;** na konci deklarace (místo těla metody). Volání čistě virtuální metody vede k chybě.
- Pokud je ve třídě nějaká nepřekrytá čistě virtuální metoda, jde o **abstraktní třídu**, jejíž instance nelze vytvářet

```
class Shape {
public:
    virtual void draw() const = 0;
};

class Circle : public Shape {
public:
    virtual void draw() const; // Korektní překrytí
}

class Rectangle : public Shape {
public:
    virtual void draw(); // Nic nepřekrývá, protože se od
                        // zděděné metody liší (není const)
}

int main()
{
    Shape s;           // CHYBA, Shape je abstraktní třída
    Circle c;         // OK
    Rectangle r;      // CHYBA, Rectangle je abstraktní kvůli
                    // nepřekryté draw() const
}
```

# Klíčové slovo `override`

- Od C++11 je k dispozici klíčové slovo `override`, kterým můžeme zaručit, že námi definovaná metoda v odvozené třídě skutečně překrývá nějakou zděděnou metodu
- Pokud metoda označená `override` nic nepřekrývá (např. kvůli různým typům parametrů či konstantnosti), kompilátor ohlásí chybu

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

class Circle : public Shape {
public:
    virtual void printClass() const override // OK, korektní překrytí
    { cout << "Circle" << endl; };
}

class Rectangle : public Shape {
public:
    virtual void printClass() override // CHYBA, tato metoda nic nepřekrývá!
    { cout << "Rectangle" << endl; };
}
```

# Kompilace programů přes Kate

- Editor Kate dokáže přímo spouštět nástroj Make či kompilátor, aniž bychom to museli dělat ručně v terminálu
- Výhodou je, že Kate zpracuje případná chybová hlášení a naváže je na dotčená místa v kódu
- Zapnutí podpory: [Settings](#)→[Configure Kate](#)→[Plugins](#)→[Build Plugin](#)
- Nastavení: V panelu [Build Output](#) na záložce [Target Settings](#) vytvoříme novou sadu cílů (tlačítkem [Create new set of targets](#)), pokud v seznamu žádná není
- Potom můžeme dvojklikem do pole vpravo na řádku Build nastavit požadovaný příkaz make
- Řádek Build může být třeba povolit zaškrtnutím políčka vlevo
- Ostatní řádky (Clean, Configure) můžeme smazat (tlačítkem [Delete current target](#))
- Nakonec tlačítkem [Build selected target](#) spustíme kompilaci

# Použití debuggeru GDB

- Ladění složitějších programů si můžeme usnadnit použitím debuggeru
- Kompilátoru předáme argumenty **-g** a případně **-Og** (optimalizace)
- Debugger spustíme příkazem **gdb ./navez\_programu**
- Potom můžeme debugger ovládat následujícími příkazy (lze zkracovat, v závorce je plné znění příkazu):

**run *argumenty*..** – spustí laděný program

**break *jmeno\_funkce*** – definuje breakpoint (zastaví program na dané funkci)

**bt (backtrace)** – vypíše řetězec volaných funkcí k aktuálnímu místu

**cont (continue)** – pokračuje v běhu programu po přerušení

**n (next)** – provede aktuální řádek programu a zase zastaví

**l (list)** – vypíše zdrojový kód kolem aktuálního místa

**p *jmeno\_promenne* (print)** – vypíše hodnotu proměnné

**kill** – ukončí laděný program

**quit** – ukončí celý debugger

# Integrace GDB do editoru Kate

- Debugger GDB můžeme ovládat i grafickou cestou přes Kate
- Zapnutí integrace: [Settings](#)→[Configure Kate](#)→[Plugins](#)→[GDB](#)
- Nastavení: V panelu [Debug View](#) na záložce [Settings](#) vytvoříme “target” (sada nastavení pro jeden laděný program) a nastavíme název spustitelného souboru, pracovní adresář a argumenty
  - Vyžaduje-li program vstup od uživatele, zapneme [Redirect IO](#) (vstup pak píšeme do záložky IO)
  - Nevidíme-li v panelu nástrojů ladicí tlačítka (Step, Continue), musíme panel zapnout přes [Settings](#)→[Toolbars Shown](#)→[GDB Plugin](#)
- Ladění pak spustíme pomocí [Debug](#)→[Start Debugging](#)
- Dále krojujeme pomocí [Step In](#) (vstupuje do volané funkce) / [Over](#), nastavujeme zarážky pomocí [Toggle Breakpoint](#), [Continue](#) pokračuje v běhu k další zarážce
- V panelu [Locals and Stack](#) vidíme hodnoty proměnných a zásobník volání
- Ladění ukončíme přes [Debug](#)→[Kill](#)

# Třída string

- Typ `string` není základním vestavěným typem, ale je implementován jako třída ve standardní knihovně C++
- Třída `string` obsahuje některé užitečné metody:
  - `length()` - vrátí počet znaků v řetězci
  - `size()` - totéž jako `length()`
  - `operator[](int pos)` - vrátí znak na pozici `pos`
  - `substr(int pos, int n)` - vrátí podřetězec dlouhý `n` znaků začínající na pozici `pos`
  - `c_str()` - vrátí řetězec, jak je používán v C, tj. typ **char\*** ukončený nulovým znakem (např. pro předání řetězce funkcím, které akceptují pouze klasické řetězce **char\*** a nepodporují `string`)
  - `begin()`, `end()` - standardní iterátory
- Podrobnější informace: <https://cplusplus.com/reference/string/string/>
- Uvedené metody nekontrolují velikost řetězce, tj. pokus o přístup k znakovým pozicím mimo meze řetězce vede k pádu programu. Musíme tedy sami předem kontrolovat velikost řetězce.
- Při práci s ne-ASCII znaky (Unicode) jsou všechny délky a pozice v bajtech, ne ve znacích (písmena s diakritikou zabírají >1 B)

# Formátování výstupu

- Pro formátování výstupů používáme tzv. **manipulátory**
- Pro použití manipulátorů je třeba v záhlaví zdrojového souboru deklarovat `#include <iomanip>` a `using namespace std;`
- Manipulátory se používají ve spojení s operátorem `<<`
- Manipulátory nastavují formátování a různé parametry pro načítání vstupu a výstupu

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

int main()
{
    // Vypise cele cislo zarovnanе doprava, minimalne 5 znaku,
    // (zleva se doplni mezery)
    cout << right << setw(5) << 234 << endl;

    // Nasledujici prikaz vypise realne cislo s platnosti na 2
    //desetinna mista, tj. 456.15
    cout << fixed << setprecision(2) << 456.15738 << endl;

    return 0;
}
```



# Manipulátory k I/O formátování

- Následující manipulátory lze použít pro formátování **výstupu**:

**fixed** – výstup reálného čísla ve formátu 123.45

**scientific** – výstup reálného čísla ve formátu 1.2345e2

**defaultfloat** (výchozí nastavení) - vybere *fixed* nebo *scientific* dle potřeby, v závislosti na velikosti vypisované hodnoty

**left** – zarovnává výpis doleva

**right** – zarovnává výpis doprava

**setw(n)** – nastaví minimální počet vypisovaných znaků n **pro nejbližší následující operaci** výpisu (pro čísla nebo řetězce).

**setfill(c)** – nastaví použití znaku c pro vyplnění výpisu, pokud je šířka (nastavená pomocí `setw()`) větší, než je třeba

**setprecision(n)** – nastavuje počet číslic za desetinnou tečkou pro vypisovaná reálná čísla (pro výpis *fixed* a *scientific*) nebo maximální počet vypisovaných platných číslic (při výchozím nastavení, tedy bez *fixed* a *scientific*)

# Manipulátory k I/O formátování

- Následující manipulátory lze použít při načítání ze **vstupu**:
  - skipws** – nastaví přeskakování bílých znaků (mezera, tabulátor, konec řádku) při načítání (toto je výchozí nastavení)
  - noskipws** – nastaví, že bílé znaky nebudou přeskakovány
  - ws** – načítá bílé znaky tak dlouho, dokud nenarazí na nebílý znak
- Podrobný výčet manipulátorů:  
<https://cplusplus.com/reference/library/manipulators/>

# Načítání PDB souboru v C++

- PDB formát se používá pro ukládání struktur biomolekul v PDB databázi [www.rcsb.org](http://www.rcsb.org)
- Data PDB souboru jsou organizována jako *fixed format*, tj. každá položka má přesně udanou pozici na řádku
- Dokumentace k PDB formátu je dostupná na: <https://www.wwpdb.org/documentation/file-format>
- Každý řádek PDB souboru obsahuje na začátku 6 znaků identifikujících typ řádku
- Souřadnice atomů se načítají z řádků ATOM (pro standardní residua) a HETATM (pro nestandardní residua)
- Načítání PDB souboru v C++ je podobné jako v C, ale využívají se proměnné typu `string` a odpovídající metody (např. `substr()`, `length()`), proudy `stringstream` a jejich metody (operátor `>>`, `str()`, `clear()`, `fail()`)
- Zápis PDB souboru v C++ je podobný jako v C, zapisuje se do proudu pomocí operátoru `<<` a pro formátování se používají manipulátory (`fixed`, `left`, `right`, `setw()`, `setprecision()`)

# Struktura záznamu ATOM a HETATM

ATOM.....7..CG2·THR·A.....1.....18\_159·11.531...6.124·1.00·10.28.....C  
1 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75

1	-	6	"ATOM"	
7	-	11	serial	Atom serial number
13	-	16	name	Atom name
17			altLoc	Alternate location indicator
18	-	20	resName	Residue name
21				
22			chainID	Chain identifier
23	-	26	resSeq	Residue sequence number
27			iCode	Code for insertion of residues
28	-	30		
31	-	38	x	Coordinates for X in Angstroms
39	-	46	y	Coordinates for Y in Angstroms
47	-	54	z	Coordinates for Z in Angstroms
55	-	60	occupancy	Occupancy
61	-	66	tempFactor	Temperature factor
77	-	78	element	Element symbol, right-justified
79	-	80	charge	Charge on the atom

# Třídy pro načítání PDB souboru

- Informace načítané z řádků ATOM a HETATM ukládáme do třídy Atom, která obsahuje datové členy pro uložení informace o jednom atomu a odpovídající metody

```
// Pro možnost vypisu na standardni vystup (cout) vlozime:
#include <iostream>
// Pro praci se souborovymi proudy (ifstream a ofstream) vlozime:
#include <fstream>
// Pro praci s retezcovymi proudy (stringstream) vlozime:
#include <sstream>
// Pro praci s manipulatory vlozime:
#include <iomanip>
// Pro praci s kontejnerem vector<> vlozime:
#include <vector>
// Pro primy pristup ke jmenum promennych a funkci standardni knihovny
// deklarujeme jmenny prostor std:
using namespace std;

class Atom
{
    // Cleny tridy Atom
};

int main(int argc, char *argv[])
{
}
```

# Třída Atom - datové členy

```
class Atom
{
public:
    // Tady budou verejne metody
private:
    int recordType;    // Rozliseni radku ATOM a HETATM
    int atomNumber;
    string atomName;
    char alternateLocation;
    string residueName;
    char chainId;      // Znak identifikujici retezec
    int residueNumber; // Cislo residua
    char iCode;
    double coordX, coordY, coordZ; // Kartezske souradnice atomu
    double occupancy;
    double tempFactor;
    string elementName;
    string formalCharge;
    bool hasOccupancy; // Byla nactena occupancy?
    bool hasTempFactor; // Byl nacten teplotni faktor?
};
```

# Třída Atom - výchozí hodnoty

```
class Atom
{
public:
    // Nasledujici staticke konstatntni promenne slouzi k nastaveni
    // hodnoty v recordType
    static const int RECORD_UNKNOWN = 0;
    static const int RECORD_ATOM = 1;
    static const int RECORD_HETATM = 2;
private:
    int recordType = RECORD_UNKNOWN;
    int atomNumber = 0;
    // Retezce je vhodne v tomto pripade inicializovat mezerami,
    // tj. pro jmeno atomu 4 mezery, pro jmeno residua 3, pro jmeno prvku 2
    string atomName = "    ";
    char alternateLocation = ' ';
    string residueName = "    ";
    char chainId = ' '; // Znak identifikujici retezec
    int residueNumber = 0; // Cislo residua
    char iCode = ' ';
    double coordX = 0, coordY = 0, coordZ = 0; // Kartezske souradnice atomu
    double occupancy = 0;
    double tempFactor = 0;
    string elementName = "    ";
    string formalCharge;
    bool hasOccupancy = false; // Byla nactena occupancy?
    bool hasTempFactor = false; // Byl nacten teplotni faktor?
};
```

# Třída Atom - metody

```
class Atom
{
    Public:
        // Konstruktor není třeba, vše inicializujeme v těle třídy

        // Metoda pro načtení řádku
        void readLine(const string &line);

        // Metoda pro výpis řádku do PDB souboru či na terminal
        void writeLine(ostream &out) const;

        // Zde mohou být další metody, např. pro přístup k datovým členům
        // např. getAtomNumber(), getAtomName(), getResidueName()
};
```



# Načítání řádků z PDB souboru

```
string line, recordName;
vector<Atom> atoms;
string inputPdbFileName = "1jxy_noal.pdb";

ifstream ifile(inputPdbFileName);
// Zde musi byt osetreni chybného otevreni souboru

while (getline(ifile, line)) {
    if (line.length() >= 6) {
        // Zkopirujeme prvni 6 znaku do recordName
        recordName = line.substr(0, 6);
        if (recordName == "ATOM  " || recordName == "HETATM") {
            Atom atom;
            //Metoda readLine() nacte z radku vsechna data pro dany atom
            atom.readLine(line);
            atoms.push_back(atom); // Atom vlozime do kontejneru
        }
    }
}
```

# Načtení záznamu ATOM a HETATM - část 1

```
void Atom::readLine(const string &line)
{
    string recordName, s; // s je pomocna retezcova promenna
    istringstream sstream;

    if (line.length() < 53) { cout<<"Prilis kratky radek!"<<endl; return;}
    // Zkopirujeme prvnych 6 znaku do recordName
    recordName = line.substr(0, 6);
    if (recordName == "ATOM ") recordType = RECORD_ATOM;
    else if (recordName == "HETATM") recordType = RECORD_HETATM;
    else return; // Neni-li to ATOM ani HETATM, nelze pokracovat

    // Nacteme cislo atomu
    s = line.substr(6, 5); // 5 znaku od pozice 6 se zkopiruje do s
    sstream.str(s); // Do retezcoveho proudu nastavime retezec s
    sstream.clear(); // Odstranime pripadny chybovy stav proudu
    sstream >> atomNumber; // Nacitame cislo atomu
    if (sstream.fail()) { /*Nahlasime chybu a vyskocime z metody.*/ }

    // Nacteme jmeno atomu
    // z retezce line se zkopiruji 4 znaky od pozice 12 do atomName
    atomName = line.substr(12, 4);

    // Nacteme alternate location indicator
    alternateLocation = line[16]; // Zkopiruje se znak na pozici 16
    // Pokracovani na dalsi strance
```

# Načtení záznamu ATOM a HETATM - část 2

```
// Pokracovani metody readLine() z predchozi stranky

// Zde se nactou data do promennych residueName, chainId,
// residueNumber, iCode

// Nacteme souradnice - zopakujeme pro coordX, coordY, coordZ
s = line.substr(30, 8);
sstream.str(s);
sstream.clear();
sstream >> coordX;
if (sstream.fail()) { /*Nahlasime chybu a vyskocime*/ }

// Nacteme occupancy
if (line.length() >= 60) { // Pouze pokud je radek delsi nez 60 znaku
    s = line.substr(54, 6);
    stringstream.str(s);
    stringstream.clear();
    stringstream >> occupancy;
    if (!stringstream.fail())
        hasOccupancy = true; // Occupancy byla uspesne nactena
    // Occupancy je nepovinna, takze i pri nenacteni muzeme pokracovat
}

// Zde nacteme tempFactor, elementName, formalCharge
// Nezapomeneme predtim vzdy zkontrolovat delku radku
} // Konec metody readLine()
```

# Zápis do PDB souboru - část 1

```
// Nekde ve funkci main nebo vhodne funkci/metode bude umisten kod, který otevře  
// vystupni soubor a potom bude pro jednotlivé atomy volat nasledující metodu
```

```
void Atom::writeLine(ostream &out)  
{  
    // Zapiseme jmeno zaznamu ATOM nebo HETATM  
    if (recordType == RECORD_ATOM)  
        out << "ATOM  ";  
    else if (recordType == RECORD_HETATM)  
        out << "HETATM";  
    else return; // Pripadne navíc zahlasíme chybu  
  
    // Zapiseme číslo atomu, 5 znaku zarovnaných doprava  
    out << right << setw(5) << atomNumber;  
    out << ' '; // Zde je v PDB vždy mezera  
    // Zapiseme jmeno atomu, 4 znaky zarovnané doleva  
    out << left << setw(4) << atomName;  
    // Zapiseme jeden znak alternate location  
    out << alternateLocation;  
    // Zapiseme jmeno residua, 3 znaky zarovnané doleva  
    out << left << setw(3) << residueName;  
    out << ' '; // Zde je v PDB vždy mezera  
    out << chainId; // Zapiseme jeden znak identifikující řetězec  
    // Zapiseme číslo residua, 4 znaky zarovnané doprava  
    out << right << setw(4) << residueNumber;  
    out << iCode; // Zapiseme jeden znak insert code  
    out << "  "; // Zde jsou v PDB vždy 3 mezery  
    // Pokracovani na dalsi strane
```

## Zápis do PDB souboru - část 2

```
// Pokracovani metody write_line() z predchozi stranky

// Zapiseme kartezske souradnice atomu zarovnanne doprava
// s pevnym poctem cislic za desetinnou teckou (manipulator fixed),
// pocet desetinnych mist bude 3 a celkovy pocet zapsanych
// znaku je 8 (vc. desetinne tecky)
out << right << fixed << setprecision(3);
out << setw(8) << coordX;
out << setw(8) << coordY;
out << setw(8) << coordZ;

if (hasOccupancy) // Zapiseme occupancy
    out << right << fixed << setprecision(2) << setw(6) << occupancy;
else
    out << "    ";

if (hasTempFactor) // Zapiseme teplotni faktor
    out << right << fixed << setprecision(2) << setw(6) << tempFactor;
else
    out << "    ";

out << "          "; // Zde je v PDB souboru vzdy 10 mezer
out << right << setw(2) << elementName;
out << left << setw(2) << formalCharge;

out << endl; // Zapiseme znak konce radku
} // Konec metody writeLine()
```

# Třída Application

- V jazyce C++ často používáme třídu nazvanou například Application, která je hlavní třídou programu a soustřeďuje nejdůležitější data a metody
- Ve funkci main() pak pouze vytvoříme objekt typu Application a zavoláme jeho metodu run(), která obsahuje veškerý hlavní kód

```
class Application
{
    public:
        // V metode run() bude veskery hlavni kod, ktery jsme
        // drive umistovali do funkce main()
        int run(int argc, char *argv[]);
        bool readPdbFile();
        void writePdbFile();
    private:
        string inputPdbFileName;
        string outputPdbFileName;
        vector<Atom> atoms;
};

int main(int argc, char *argv[])
{
    Application app;
    return app.run(argc, argv);
}
```

# Dodržujte následující pravidla

- **Vždy inicializujte datové členy třídy** (obvykle se inicializují jen proměnné základních typů, proměnné objektových typů obvykle není třeba inicializovat, protože jsou inicializovány svými konstruktory).
- **Nemáte-li dobrý důvod** definovat třídě vlastní konstruktor(y), **žádné konstruktory neuvádějte** (kompilátor je vygeneruje sám).
- **Používejte přednostně inicializaci v těle třídy**, ve složitějších případech inicializační seznam (v záhlaví konstruktoru za dvojtečkou). Přiřazování v těle konstruktoru používejte jen v dobře odůvodněných případech.
- Program vytvářejte postupně, nejdříve vytvořte definice tříd a metody, jejich těla ponechejte prázdná a program přeložte. Potom přidávejte kód, vždy po dokončení důležité části program přeložte a otestujte.
- Funkcím pro výstup předávejte proud jako **ostream &**. Bude je pak díky polymorfismu možno volat s `ofstream` i `cout`, což se hodí pro testování (cvičný výpis načteného řádku na terminál).

# Cvičení

1. Vytvořte program který **načte atomy z PDB souboru** (řádky ATOM a HETATM) a **zapíše je do jiného souboru**.
  - Jména vstupního a výstupního souboru budou specifikována **na příkazovém řádku**
  - Program otestujte se souborem `/home/tootea/C3220/data/1jxy_noa1.pdb`. Dbejte na správné formátování výstupu (zarovnání sloupců).
  - Objekty typu Atom ukládejte do kontejneru `vector<Atom>`
  - V programu **použijte třídu Application**, do níž umístíte všechny kód, který byste jinak měli v `main()`.
  - Chybové hlášky upravte tak, aby nahlásily, **na kterém řádku ve vstupním PDB souboru chyba nastala** (vytiskněte číslo řádku a jeho obsah). K implementaci **nepoužívejte globální proměnné**, ale vhodně upravte argumenty či návratový typ metody `Atom::readLine()`.

**4 body**