PB161 – Programování v jazyce C++
Objektově Orientované Programování

Podzim 2013

Dynamická alokace, I/O proudy

JMENNÉ PROSTORY

JMENNÉ PROSTORY - MOTIVACE

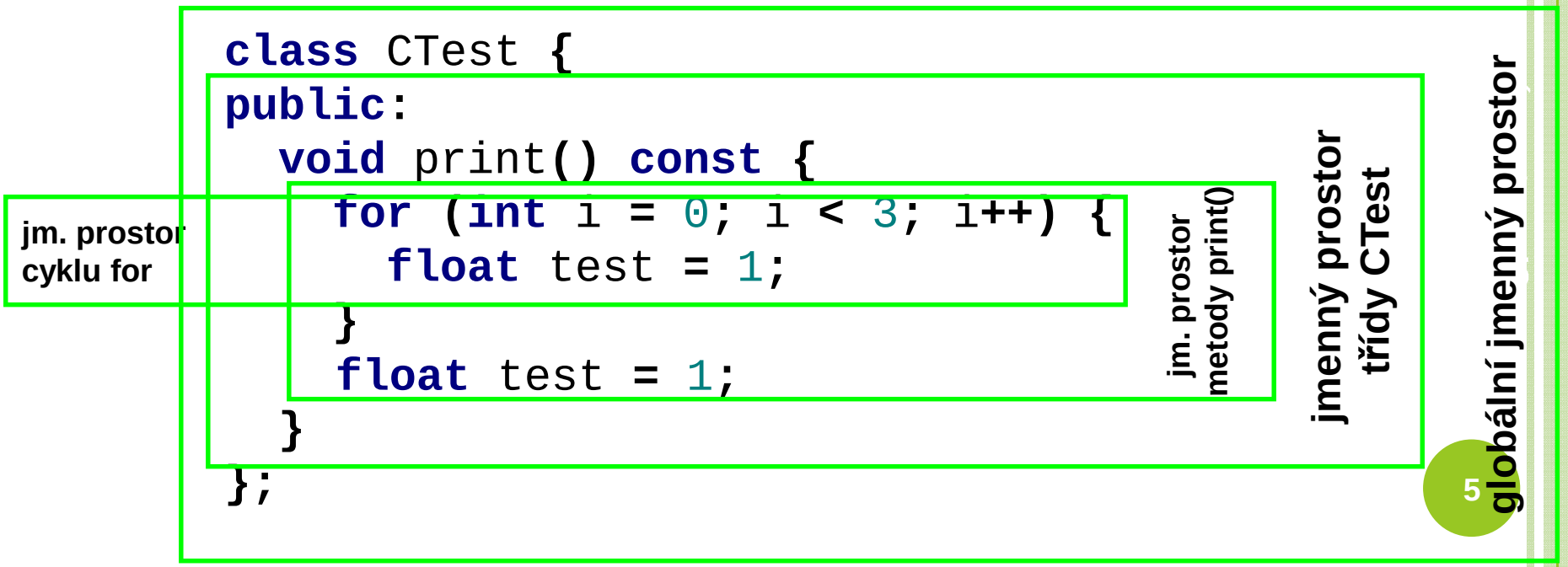
1. Výskyt dvou entit (např. tříd) se stejným jménem
   - většinou nenastane u našich vlastních tříd
     - tam pohlídáme a přejmenujeme
   - může ale nastat při použití dvou nezávislých knihoven
     - které obsahují stejnou třídu
2. Chceme se preventivně tomuto problému vyhnout
   - aby ostatní používající naši knihovnu neměli problém
3. Chceme používat knihovnu, která jmenné prostory už používá
   - např. <iostream>

Jmenné prostory

- Způsob jak zamezit kolizím jmen entit ve dvou nezávisle vyvíjených kusech kódu
- Způsob jak omezit viditelnost jména entity
  - třída, proměnná, funkce…
- Některé jmenné prostory vznikají automaticky
  - implicitní jmenné prostory, např. pro třídu
- Lze deklarovat vlastní jmenný prostor
  - explicitní jmenné prostory

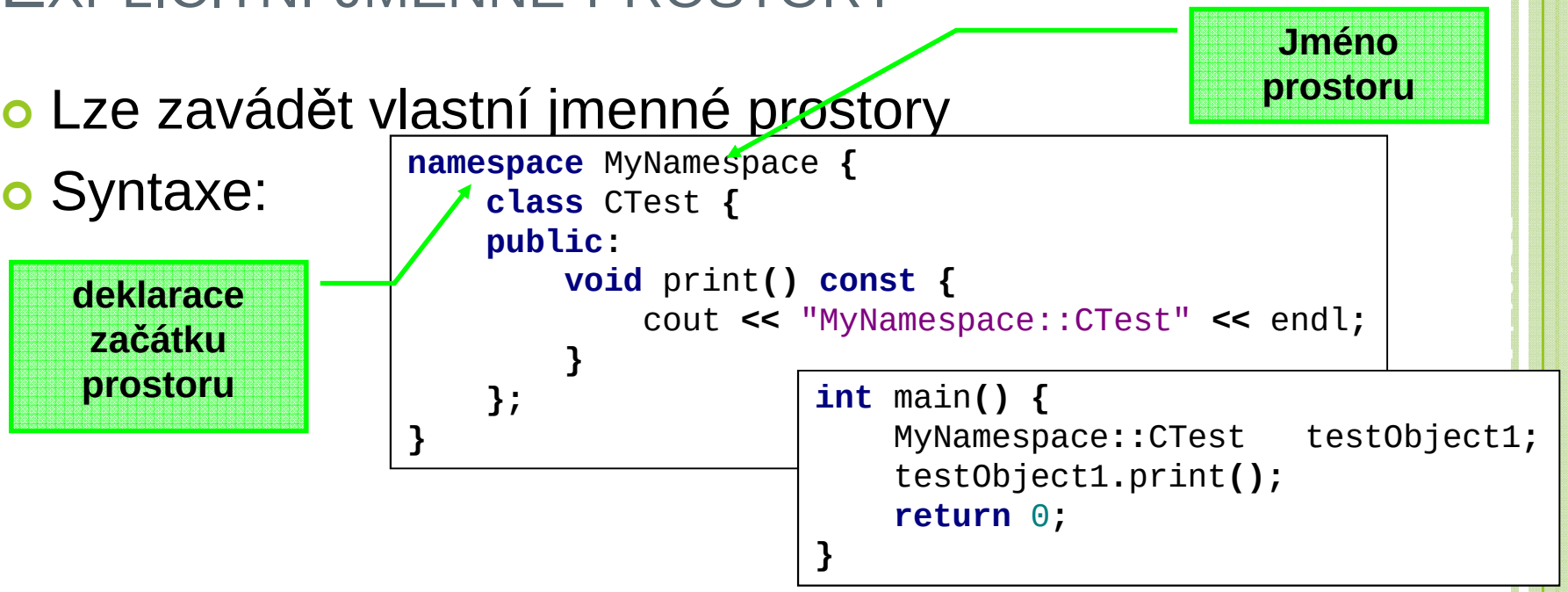
IMPLICITNÍ JMENNÉ PROSTORY

- Vznikají automaticky při deklaraci tříd, struktur, metod, funkcí, cyklů…
  - uzavřeny mezi složené závorky {}
  - mohou být pojmenované nebo nepojmenované



EXPLICITNÍ JMENNÉ PROSTORY

- Lze zavádět vlastní jmenné prostory
- Syntaxe:



- Jmenné prostory mohou být vnořené
  - MyName1:: MyName2::CTest

ZPŘÍSTUPNĚNÍ ENTIT ZE JMENNÉHO PROSTORU

1. Plná kvalifikace entity
   - např. `std::cout << "Test";`
   - znáte už z implementace metod
     - `Person::getAge() {}`

```
#include <iostream>
int main() {
    std::cout << "Hello world";
    std::cout << std::endl;
    return 0;
}
```

2. `using namespace jméno_prostoru;`
   - např. `using namespace std;`
   - tzv. *using-directive*
   - vloží obsah celého jmenného prostoru
   - analogie Javovského `import java.*;`
   - zvyšuje riziko jmenných konfliktů

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello world";
    cout << endl;
    return 0;
}
```

Zpřístupnění entit ze jmenného prostoru (2)

3. `using jméno_prostoru::jméno_entity;`
   - např. `using std::cout;`
   - tzv. *using-declaration*
   - vloží pouze odkazovanou entitu
   - analogie Javovského `import java.util.List;`
   - má prioritu před `using namespace`
   - použijte až po všech `#include`

```
#include <iostream>
using std::cout;
using std::endl;
int main() {
    cout << "Hello world";
    cout << endl;
    return 0;
}
```

ODDĚLENÍ KOLIDUJÍCÍCH JMEN TŘÍD

```
#ifndef LIBRARY1_H
#define LIBRARY1_H
class CSystem {
public:
    void foo() {
        cout << "CSystem::foo";
    }
};
#endif // LIBRARY1_H
```

```
#ifndef LIBRARY2_H
#define LIBRARY2_H
class CSystem {
public:
    void otherFoo() {
        cout << "CSystem::otherFoo";
    }
};
#endif // LIBRARY2_H
```

```
namespace Lib1 {
#include "library1.h"
}
namespace Lib2 {
#include "library2.h"
}

int main() {
    //CSystem systemObject;   // error: 'CSystem' was not declared
    Lib1::CSystem systemObject2;
    systemObject2.foo();

    Lib2::CSystem systemObject3;
    systemObject3.otherFoo();

    return 0;
}
```

Prevence před kolizí s jiným kódem

```
#ifndef LIBRARY1_H
#define LIBRARY1_H

namespace Lib1 {
  class CSystem {
  public:
      void foo() {
          cout << "CSystem::foo" << endl;
      }
  };
}

#endif // LIBRARY1_H
```

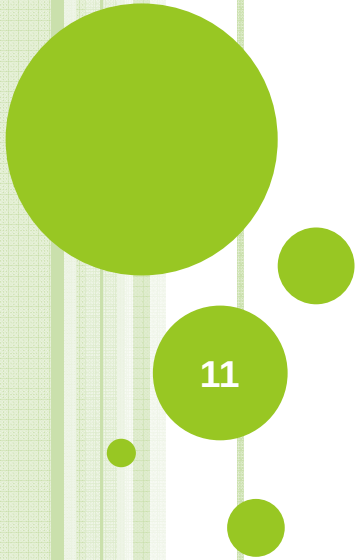
```
#include "library1.h"
#include "library2.h"

int main() {
    Lib1::CSystem systemObject2;
    systemObject2.foo();

    Lib2::CSystem systemObject3;
    systemObject3.otherFoo();

    return 0;
}
```

Abstraktní třída



GENERALIZACE

- Cílem je navržení takového rozhraní, které pod sebe schová chování více typů objektů
- Hledají se společné vlastnosti různých objektů
- Např. iterátor na procházení pole
  - není podstatné, že jde o `int[]` nebo `float[]` pole
- Např. zobrazení objektů na cílovou plochu
  - není podstatné, jaká přesně bude (obrazovka, tiskárna)
- Např. tiskárny
  - není podstatná technologie tisku

Motivace pro rozhraní

- Chceme podchytit, co všechno musí splňovat třída, aby se mohla vydávat za příslušníka dané skupiny
  - např. všechny grafické objekty je možné vykreslit
- V C++ implementujeme pomocí společného předka
  - `class IDrawable;`
  - požadavky na chování příslušníků zachytíme v jeho veřejných metodách
  - např. `virtual void paint();`
- Potomci si provádí vlastní implementaci těchto metod
  - `void CButton::paint() const {}`
- (Společný předek nemusí mít smysl jako instance)

Čistě virtuální metoda (pure virtual)

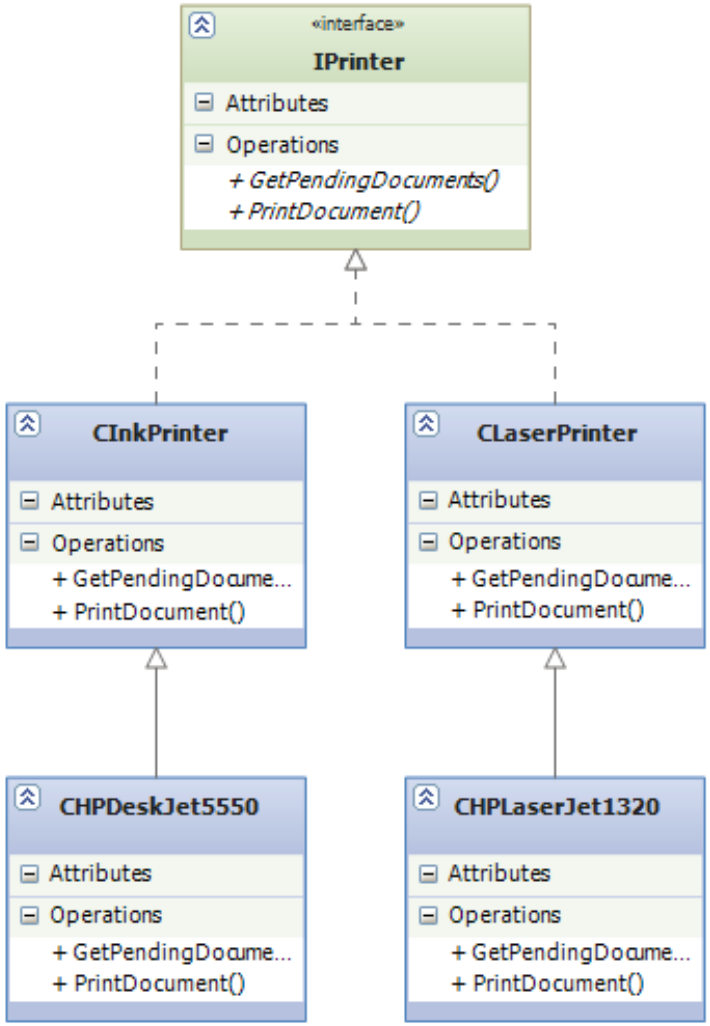
- Metoda, která má ve třídě pouze svou deklaraci
  - implementace je ponechána na potomky
- Syntaxe
  - `virtual návratový_typ metoda(parametry) = 0;`
- Potomci standardním způsobem implementují
  - překrývají čistě virtuální metodu předka

Abstraktní třída

- Třída s alespoň jednou čistě virtuální metodou
- Nelze z ní přímo vytvářet instance (objekty)
  - chyba při překladu
- Lze ale využít jako třídu pro dědění
  - potomci překrývají virtuální metody abstraktního předka
- Analogie rozhraní v Javě
  - může ale obsahovat implementaci některých funkcí
- Čistě abstraktní třída - všechny metody jsou čistě virtuální
  - opravdové rozhraní ve stylu Javy

```
class CPersonInterface {
public:
    virtual const char* getEmail() = 0;
    virtual void print() = 0;
};
```

ABSTRAKTNÍ TŘÍDA - OBRÁZEK



Čistě abstraktní třída

Abstraktní třída (může být část implementace)

Třída s implementací (děláme objekty)

ABSTRAKTNÍ TŘÍDA - UKÁZKA

```
// Abstract class, at least
// one method is pure virtual
class IPerson {
public:
    virtual const char* getEmail() = 0;
    virtual void print() = 0;
};

class CStudent : public IPerson {
    char m_email[MAX_EMAIL_LENGTH+1];
public:
    CStudent(const char* email) {
        strncpy(m_email, email, MAX_EMAIL_LENGTH);
        m_email[MAX_EMAIL_LENGTH] = 0;
    }
    virtual const char* getEmail() { return m_email; }
    virtual void print() {
        cout << "Student: " << m_email << endl;
    }
};
```

```
int main() {
    CStudent stud("novak@fi.muni.cz");
    stud.print();

    IPerson& person = stud;
    person.print();

    return 0;
}
```

ABSTRAKTNÍ TŘÍDA - UKÁZKA

- *abstractClassDemo.cpp*
- Čistě virtuální metoda
- Abstraktní třída
- Čistě abstraktní třída
- Potomek implementující abstraktní třídu
- Potomek implementující jen část čistě virtuálních metod

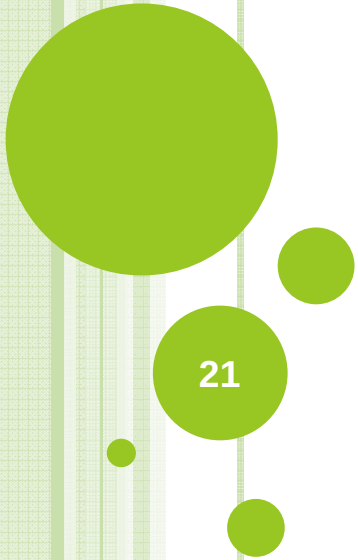
Psaní dobrého kódu

- Pokud přetěžujete metodu pouze pro různé varianty parametrů, ale většina kódu zůstává stejná, pak:
  - vytvořte jednu metodu X implementující vlastní kód
  - ostatní přetížené metody pouze předzpracují vstupní argumenty a zavolají X

Psaní dobrého kódu (2)

- Označte virtuální ty metody, které budou v potomkovi definovat jeho specializaci
  - `SerializeIntoFile()`
  - kód pro manipulaci zápis do souboru bude v předkovi
  - pro potomky virtuální funkce `SerializeIntoBuffer()`
- Nedělejte všechny metody virtuální
  - pokud není třída zároveň čistě abstraktní (rozhraní)
  - jinak svědčí spíš o špatném návrhu hierarchie

Proudy, vstup a výstup

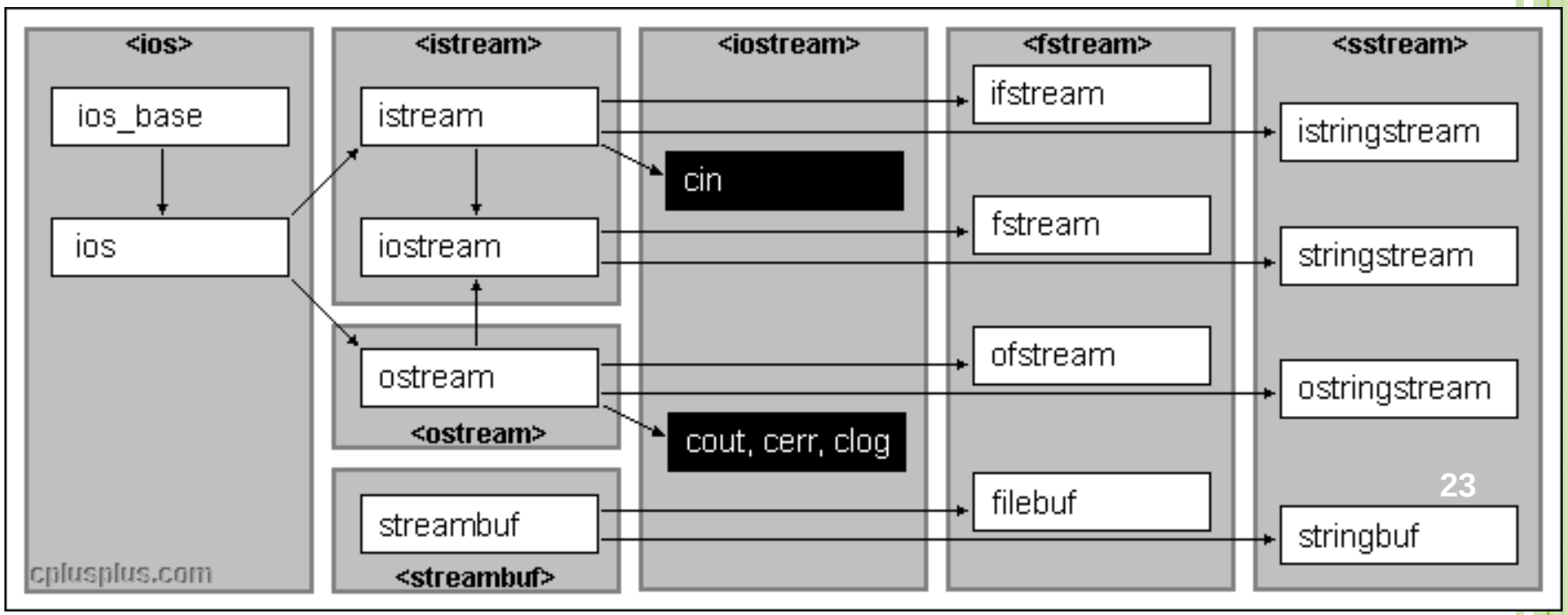


C++ I/O PROUDY

- Objektově orientovaná standardní knihovna C++
- Poskytuje funkčnost pro vstup a výstup založenou na proudech
- Proud je abstrakce nějakého zařízení s možností vyčítat anebo vkládat položky (znaky,bajty…)
  - data “plynou” proudem od zdroje k cíli
  - zásobník na stisknuté znaky (klávesnice->obrazovka)
  - soubor na disku (aplikace <->aplikace)
  - tiskárna (aplikace ->tisková hlava)
  - síťový socket (aplikace <-> vzdálená aplikace)
  - …

I/O PROUDY - HIERARCHIE

- Základní objekty obsaženy v souboru <iostream>
- Hierarchie zavedena pro možnost kombinace i oddělení proudů s různými vlastnostmi
  - a jejich specifické implementace



I/O PROUDY – HIERARCHIE (2)

- Základní předek je abstraktní třída `ios_base`
  - využívá se často násobná dědičnost
- Třída typu istream („input stream“)
  - rozhraní pro vstupní proudy
  - obsahuje operace pro získání existujících dat z proudu
  - např. čtení ze standardního vstupu (cin >> prom;)
- Třída typu ostream („output stream“)
  - rozhraní pro výstupní proudy
  - obsahuje operace pro zaslání nových dat do proudu
  - např. výpis na standardní výstup (cout << “Hello")
- Pozn. Proudy nelze kopírovat jako celek
  - protože proud může být proměnný (např. vstup z klávesnice, kdy uživatel ještě nezadal svůj vstup)

Proudy pro standardní vstup a výstup

- Třídy obsaženy v hlavičkovém souboru <iostream>
  - jmenný prostor `std::`
- Několik proudů má i standardní instance (objekty)
  - objekt cin – standardní vstup, potomek istream (stdin v C)
  - objekt cout – standardní výstup, potomek ostream (stdout v C)
  - objekt cerr – chybový výstup, potomek ostream (stderr v C), clog
- Speciální funkce pro manipulaci proudů (manipulátory)
  - vložení speciálních znaků známých z C (\n, \\, ...)
  - nastavení přesnosti a formátu dat
  - způsob fungování interní cache…
- http://www.cplusplus.com/reference/iostream/

Proud cout (standardní výstup)

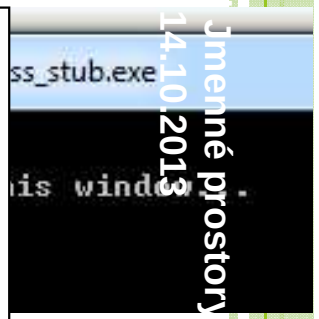
- Umožňuje zapsat data na (standardní) výstup
- Pro zápis se typicky využívá **operátor <<**
  - přetížen pro standardní typy, lze dodatečně i pro naši třídu

```
#include <iostream>
using std::cout;
using std::endl;

int main() {
    int    intValue = 5;
    float  floatValue = 4.56;
    char   charString[] = " is ";

    cout << intValue << " + " << floatValue << charString;
    cout << intValue + floatValue << endl;

    return 0;
}
```



int na výstup

pole znaků na výstup

fixní řetězec na výstup

konec řádku na výstup

Proud cout – další metody

- Operátor << můžeme přetížit i pro vlastní třídy
  - asociace zleva doprava
  - vrací referenci volajícího objektu - lze řetězit ve výrazu
  - (je vhodné to samé dělat i pro vlastní operátory)

```
cout << "Number: " << intValue << ", string: " << charString << endl;
```

- Členské metody pro vkládání znaků/hodnot
  - zděděné z třídy `ostream`
  - neformátované vkládání…
- Vložená data se mohou být ve vyrovnávací paměti (cache)
  - explicitní vyprázdnění proudu pomocí `flush()`

PŘETÍŽENÍ OPERÁTORU << - UKÁZKA

```
#include <iostream>
using std::cout;
using std::endl;
using std::ostream;

class CComplexNumber  {
    float m_realPart;
    float m_imagPart;
public:
    // …
    // Make some operators my friends
    friend ostream& operator <<(ostream& out, const CComplexNumber& complex);
};

/**
  Output operator as friend function
  */
ostream& operator <<(ostream& out, const CComplexNumber& complex) {
    out << "[" << complex.m_realPart << ", " << complex.m_imagPart << "]";
    return out;
}
```

```
int main() {
    CComplexNumber    value1(10, 20);
    cout << value1 << endl;
    return 0;
}
```

Přetěžování operátorů bude v další přednášce detailně.

Proud cin (standardní vstup)

- Umožňuje načíst data ze vstupu
  - např. zadané z klávesnice, oddělená bílým znakem (enter, tab)
  - načte se přímo do proměnné daného typu
- Využívá se typicky operátor >>

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

int main() {
    int    intValue = 0;
    char   charString[50];
    cout << "Type number: ";
    cin >> intValue;
    cout << endl << "Type string: ";
    cin >> charString;  // Note: Problem: if more then 49 characters are typed
    cout << "Number: " << intValue << ", string: " << charString << endl;
    return 0;
}
```

hlavičkový soubor pro cin, cout, cerr…

přesměrujeme vstup do proměnné intValue

přesměrujeme vstup do proměnné charString

PŘETÍŽENÍ OPERÁTORU >> - UKÁZKA

```
int main() {
    CComplexNumber    value1(10, 20);
    cout << value1 << endl;
    cin >> value1;
    cout << value1 << endl;
    return 0;
}
```

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

class CComplexNumber  {
    float m_realPart;
    float m_imagPart;
public:
    // …
    // Make some operators my friends
     friend istream& operator >>(istream& in, CComplexNumber& complex);
};

/** Input operator as friend function */
istream& operator >>(istream& in, CComplexNumber& complex) {
  if (in.good()) {
    in >> complex.m_realPart;
    in >> complex.m_imagPart;
  }
  return in;
}
```

Proud cerr (chybový výstup)

- Proud pro zápis chybových hlášek
- Standardně přesměrován na cout, ale nemusí
- Způsob použití je stejný jako u cout
  - je také potomek **ostream**

```
#include <iostream>
using std::cout;
using std::cerr;
using std::endl;

int main() {
   cout << “Hello world" << endl;
   cerr << “Error: hell world" << endl;

   return 0;
}
```

Chybové stavy proudů

- Proudy obsahují příznaky ukazující na chybu
  - `eofbit, failbit, badbit`
- Lze využít `operator!`
  - něco je špatně
- Metody pro testování stavu
  - `cin.good()`
  - `cin.fail()`
  - `cin.eof()`
- Metody pro vyčištění příznaků
  - `cin.clear()`

tu sa prekrýva obrázok s textom
- odstraniť text alebo zmenšiť obrázok???

PROUD CIN – ŘEŠENÍ CHYBNÉHO VSTUPU

- Zadaný vstup nemusí být kompatibilní s cílovým typem
- Mu

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

int main() {
    int    intValue = 0;
    cout << "Type number: ";
    cin >> intValue;
    if(!cin.fail()) {
        // Process correct numerical value
        cout << "Good, you typed: " << intValue << endl;
    }
    else {
        // Process incorrect numerical value
        cout << "Problem, you typed non-numerical value." << endl;
        cin.clear();
    }
    return 0;
}
```

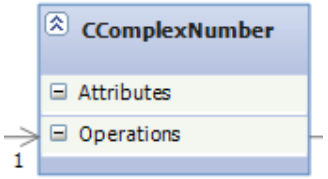
otestujeme stav proudu po pokusu o vložení do intValue

po problému vyčistíme

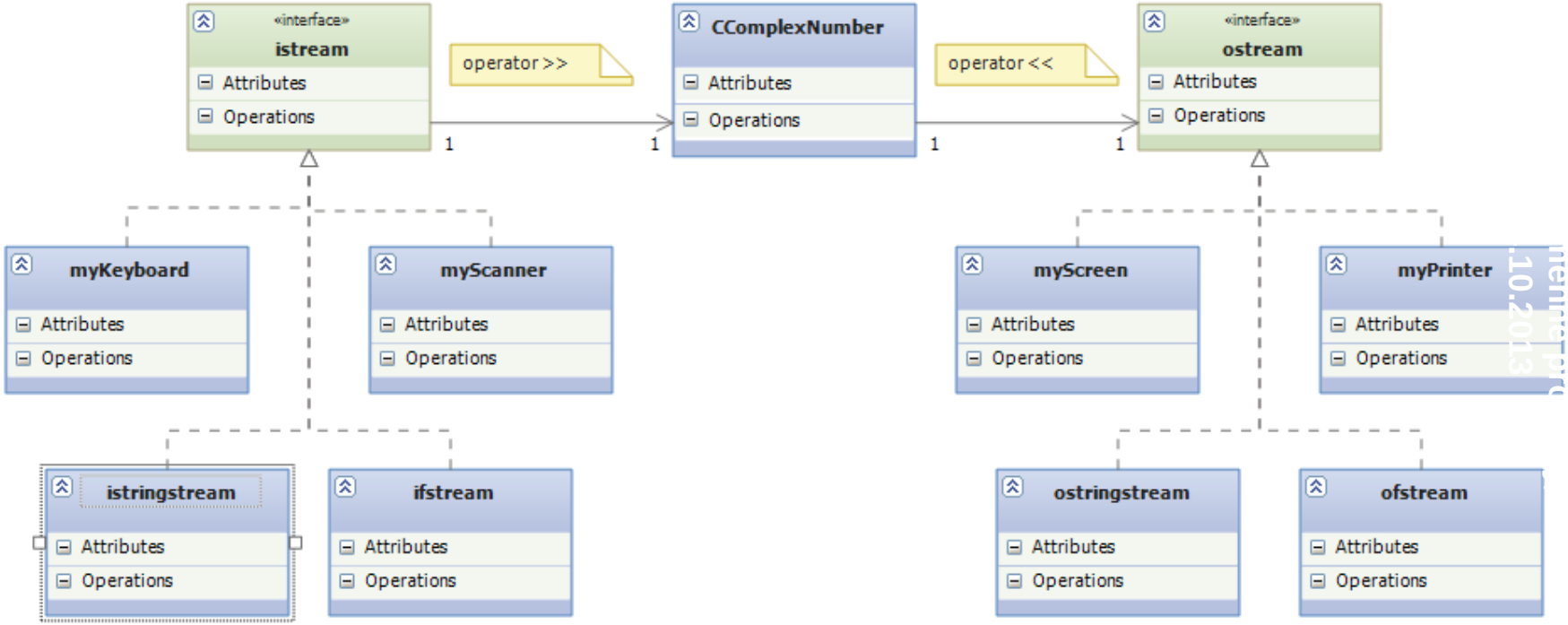
Souborové proudy

MYŠLENKA PROUDŮ – ABSTRAKCE

```
CComplexNumber  number;
cin >> number;
cout << number;
```



MYŠLENKA PROUDŮ – ABSTRAKCE



Operátory vstupu a výstupu

- Umožňují vložit resp. přijmout data z proudu
- (stream insertion resp. stream extraction)

```
istream& operator >>(istream& in, CComplexNumber& complex) {
    in >> complex.m_realPart;
    in >> complex.m_imagPart;
    return in;
}
```

```
ostream& operator <<(ostream& out, const CComplexNumber& complex) {
    out << "[" << complex.m_realPart << ", ";
    out << complex.m_imagPart << "]";
    return out;
}
```

JAK PŘEKLADAČ ZPRACUJE PROUDY?

opäť prekryv textu s obrázom

- Překladač hledá funkci se jménem 'operator<<'
  - Operand nalevo udává 1. argumentu
    - zde `ostream` (nebo potomek)
  - Operand napravo udává typ 2. argumentu
    - zde CComplexNumber

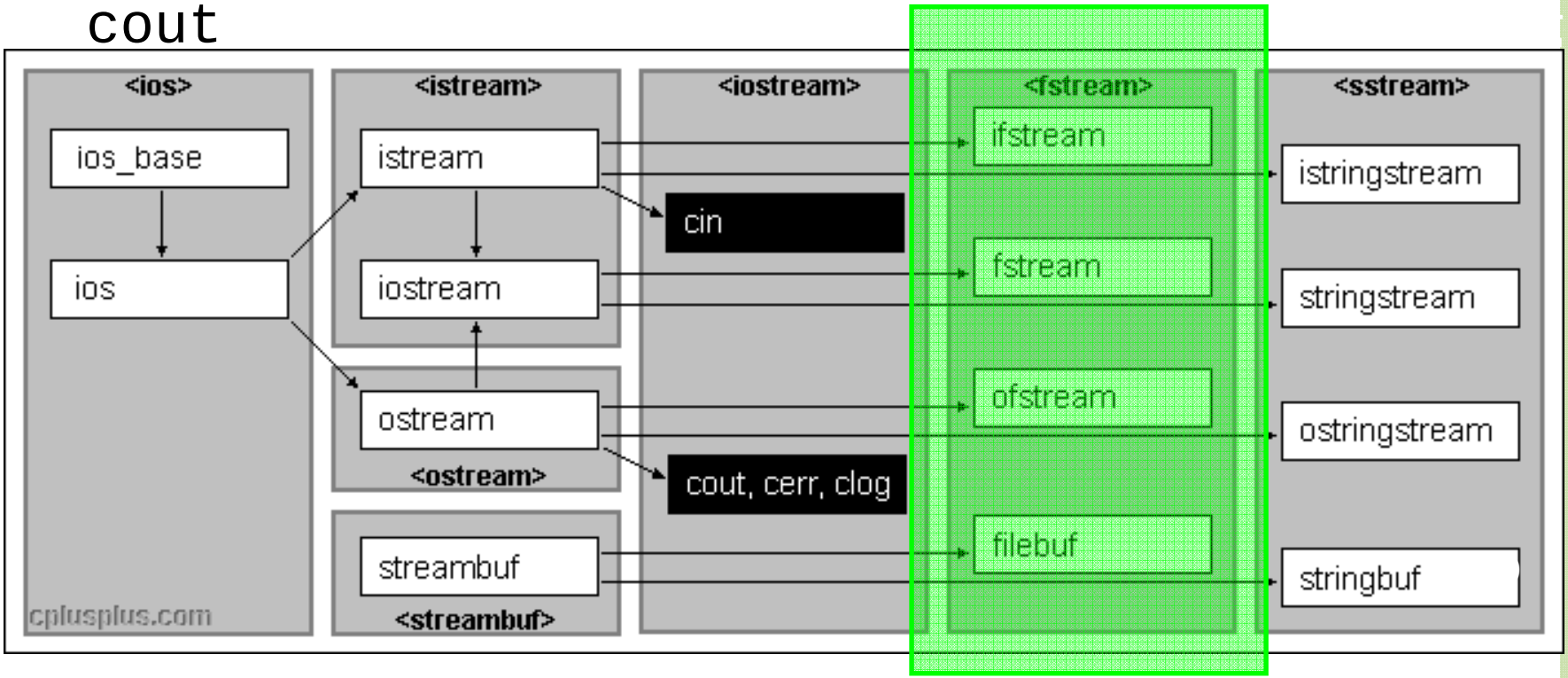
```
CComplexNumber number;
cout << number;
```

```
ostream& operator <<(ostream& out, const CComplexNumber& complex);
```

- Proč reference **&**?
  - nechceme vytvářet kopie objektů
- Proč **`const`**?
  - CComplexNumber nebudeme měnit (kontrola, optimalizace)
  - ostream naopak měnit budeme (vložení výpisu) => nemůže být const
- Proč vracet ostream?
  - výstupem vyhodnocení **cout << number;** bude **ostream**
  - lze řetezit: **cout << number << number2 << endl;**

SOUBOROVÉ PROUDY

- #include <fstream>
- http://www.cplusplus.com/reference/iostream/fstream/
- díky dědičnosti můžete využívat známé chování z `cin` a `cout`



Souborové proudy ifstream/ofstream

- Proudy pro práci se soubory
  - pokrývá funkčnost známou z C z knihovny <stdio.h>
- Potomci istream nebo ostream
  - dědí běžné metody, které znáte u objektů z <iostream>
- `ifstream` pouze pro vstup
  - ifstream inFile(“vstupni_soubor”);
- `ofstream` pouze pro výstup
  - ofstream outFile(“vystupni_soubor”);
- `fstream` pro vstup i výstup

IFSTREAM/OSTREAM/FSTREAM

- Svázání s konkrétním souborem
  - v konstruktoru: ofstream outFile(“vystupni_soubor”);
  - později metodou: outFile.open(“vystupni_soubor”);
- Typ otevření
  - `ios::in` (defaultní pro ifstream)
  - `ios::out` (defaultní pro ofstream)
  - `ios::in | ios::out` (defaultní pro fstream)
- Způsob otevření
  - `ios::binary` (binární přístup, default je textově)
  - `ios::append` (přidání na konec, obsah se ponechá)
  - `ios::trunc` (pokud soubor existuje, tak se vymaže)

SOUBOROVÝ PROUD – ELEMENTÁRNÍ UKÁZKA

Řešení pomocí jazyka C

```c
#include <stdio.h>
int main() {
 FILE* file = NULL;
 char fileName[] = "D:\\test.txt";
 if ((file = fopen(fileName, "r"))) {
    char value;
    while((value = getc(file)) != EOF) {
      putchar(value);
    }
    fclose(file);
 }
}
```

problém: prečo tu musí byť cstring?
V 2. úlohe problém, keď ako parameter inFile nebolo možné použiť premennu typu c++ string
- nutné otravne konvertovať
- ako čo najjednoduchšie konvertovať cstring na c++string a naopak?

Řešení pomocí jazyka C++

```cpp
#include <iostream>
#include <fstream>
using std::cout;

int main() {
    std::ifstream inFile("inputFile.txt");
    if (inFile.is_open()) {
        while (inFile.good()) {
            cout << inFile.get();
        }
        inFile.close();
    }
    return 0;
}
```

Čtení ze souboru - třída istream

- Metoda **`get()`**
  - čtení jednotlivých znaků včetně bílých znaků
  - čtení více znaků pomocí **`get(buff, 30)`**, ukončeno **`null`**
  - přetížená metoda, hodně variant
- Metoda **`getline()`**
  - jako **`get()`**, ale odstraní oddělovač z proudu

Čtení ze souboru - třída istream

- Použití operátoru vstupu **>>**
  - `inFile >> value;`
- Metoda **`read()`**
  - blokové čtení daného počtu bajtů
  - čte včetně bílých znaků
  - využito hlavně u binárních souborů

Čtení ze souboru – v cyklu

```
ifstream inFile("inputFile.txt", ios::binary);
// open output file and truncate it already exists
ofstream outFile("outputFile.txt", ios::binary | ios::trunc);

if (inFile.is_open() && outFile.is_open()) {
    const int dataLen = 100;
    char data[dataLen];
    while (inFile.good()) {
        // read all available characters
        inFile.read(data, dataLen);
        int dataReaded = inFile.gcount();
        outFile.write(data, dataReaded);
    }
}

inFile.close();
outFile.close();
```

Zápis do souboru

- Použití operátoru výstupu **<<**
  - `outFile << value;`
- Metoda **`put()`**
  - zápis jednotlivých znaků (včetně bílých znaků)
- Metoda **`write()`**
  - blokový zápis daného počtu bajtů
  - využito typicky u binárních souborů
- Při zápisu na konci se soubor zvětšuje
- Při zápisu uvnitř souboru se přepisuje!

Pozice a posun v souboru

- Odkud a kam se bude zapisovat?
  - „Get pointer“ – místo, odkud se bude číst
  - „Put pointer“ – místo, kam se bude zapisovat
- **`tellg(), tellp()`** – získání pozice get/put ukazatele
- Nastavení pozice pro *get* ukazatel **`seekg`**
  - `seekg(offset, pozice_odkud);`
  - `ios::beg` (začátek souboru), `ios::cur` (aktuální pozice), `ios::end` (konec souboru)
- Nastavení pozice pro *put* ukazatel **`seekp`**
  - `seekp(offset, pozice_odkud);`
- Počáteční pozice ovlivněna módem otevření

toto by bolo možno vhodne ešte odsadiť o jednu úroveň kedže sa to nevzťahuje priamo na metodu seekg ale na jej parameter pozice_odkud / ios_base::seekdir way

POZICE V SOUBORU - UKÁZKA

```
#include <iostream>
#include <fstream>
using std::cout;

int main() {
    std::ifstream inFile("inputFile.txt");
    inFile.open("inputFile.txt", ios::binary);
    if (inFile.is_open()) {
        inFile.seekg(0, ios::end);
        int dataLen = inFile.tellg();
        cout << "Data length is: " << dataLen << std::endl;
    }
   return 0;
}
```

Ukončení práce se souborem

- V destruktoru objektu
  - dochází k vyprázdnění vyrovnávacího pole a uzavření
- Explicitně metodou
  - `outFile.close();`
  - objekt samotný nezaniká, lze se připojit k dalšímu souboru

Problematika vyrovnávacího bufferu

- Data zaslaná do proudu nemusí být ihned zapsána do cíle
  - není efektivní zapisovat každý bajt do souboru
- Pro každý souborový proud automaticky vzniká vyrovnávací pole typu `streambuf`
- Přenos ze vyrovnávací pole do cíle nastává:
  1. při uzavření souboru
  2. při zaplnění vyrovnávacího pole
  3. explicitně pomocí manipulátorů nebo metod (`endl`, `nounitbuf`, `flush`, `sync`)

VYUŽITÍ SOUBOROVÝCH PROUDŮ - UKÁZKY

- *fileDemo.cpp*
- Otevření souboru
- Čtení a zápis do/ze souboru
- Vyprázdnění vyrovnávací paměti

Čtení/zápis do souboru - speciální

- Metoda **`ignore()`**
  - zahodí jeden/několik znaků ze vstupu
  - `inFile.ignore(7);`
- Metoda **`putback()`**
  - možnost využití namísto dat ze standardního vstupu v bash
  - předplníme si vlastní data do standardního vstupu
- Metoda **`peek()`**
  - náhled na další znak, např. testování konce `EOF`
  - zůstává v proudu
- Metoda **`gcount()`**
  - počet znaků načtených poslední operací (`get, read`)

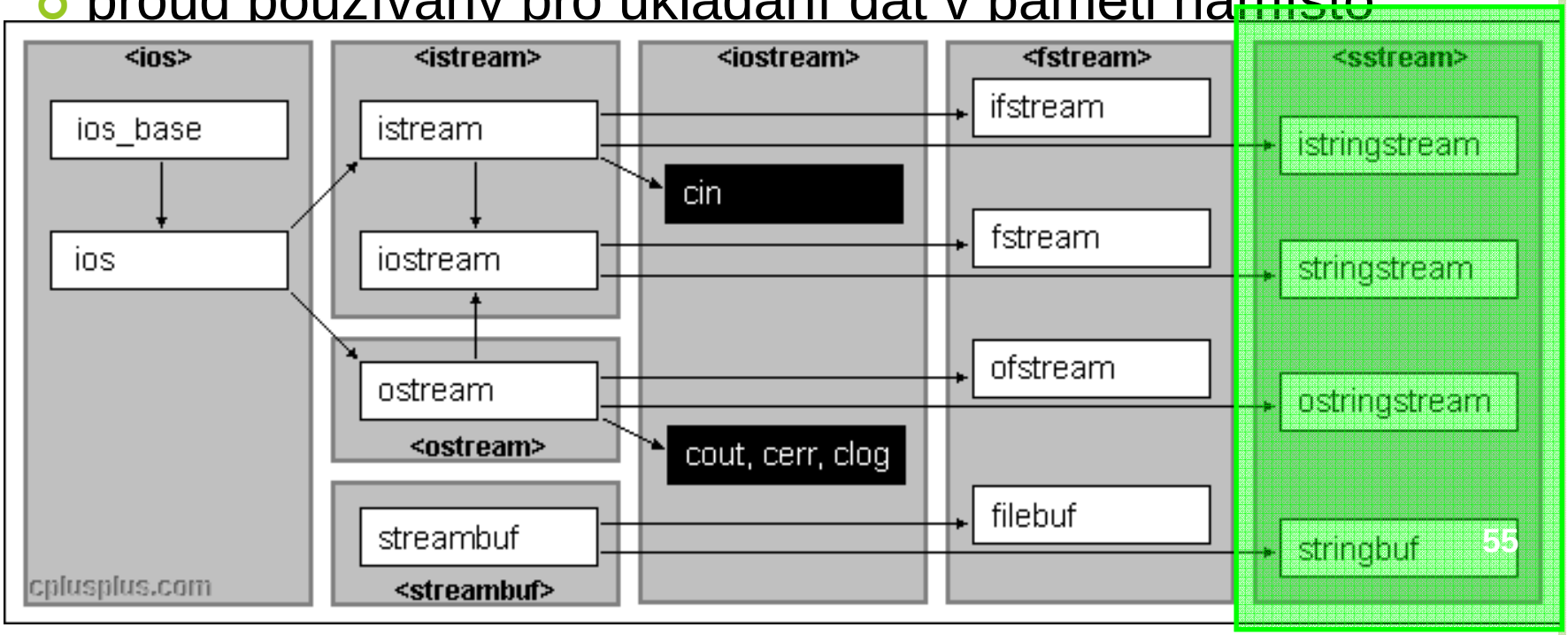
```
#include <iostream>
#include <fstream>
using std::cout;
int main() {
    std::ifstream inFile("inputFile.txt");
     if (inFile.is_open()) {
         // peek for next char, but leave it in stream
         char c;
         while (((c = inFile.peek()) != 't') && !inFile.eof()) {
             c = inFile.get();
             cout.put(c);
         }
         cout.flush();
         // discard two characters from input stream
         inFile.ignore(2);

         // read rest of the remaining data from stream
         const int dataLen = 100;
         char data[dataLen];
         while (inFile.good()) {
             // read all available characters
             inFile.getline(data, dataLen);
             cout << data;
         }
         inFile.close();
     }
   return 0;
}
```

Řetězové proudy

Řetězové proudy

- #include <sstream>
- http://www.cplusplus.com/reference/iostream/stringstream
- proud používaný pro ukládání dat v paměti namísto



Řetězové proudy – základní chování

- Potomci istream, ostream a iostream jako pro souborové proudy
  - dědí běžní chování proudů
  - operátory vstupu a výstupu
  - metody pro vkládání, vybírání a testování stavu proudu
- Proud je realizován v paměti jako:
  - textový řetězec `stringstream ss(ios::text);`
  - binární data `stringstream ss(ios::binary);`
- Lze jej transparentně nahradit např. za fstream
  - operátory jsou přetížené pro `istream` resp. `ostream`

Řetězové proudy - inicializace

- Proud s prázdným obsahem
  - `stringstream ss;`
- Proud pouze pro vstup resp. výstup
  - `istringstream` resp. `ostringstream`
  - `stringstream(ios::in)` resp. `stringstream(ios::out)`
- Proud počátečně inicializovaný řetězcem
  - stringstream ss("Hello world");

Řetězové proudy – čtení a zápis

- Běžné operátory vstupu >> a výstupu <<
- Běžné metody pro čtení a zápis (get(), write()…)
- Metoda `stringstream::str()`
  - vrátí obsah proudu naformátovaného do std::string
  - pozor na použití pro binární proudy (koncová nula)
- Metoda `stringstream::str(const string &s)`
  - vloží řetězec do proudu

STRINGSTREAM DEMO

```
void stringstreamDemo() {
    stringstream ss("Hello world");
    string mystr;
    ss >> mystr; cout << mystr;
    ss >> mystr; cout << mystr;

    ss.clear();
    ss.str("Hello Dolly");

    ss >> mystr; cout << mystr;
    ss >> mystr; cout << mystr;

    ss.clear(); ss.str("");

    ss << "Hello Hero";
    ss >> mystr; cout << mystr;
    ss >> mystr; cout << mystr;
}
```

inicializace obsahu přes konstruktor

inicializace obsahu přes str()

clear() a str("") vyčistí příznaky a obsah proudu

inicializace obsahu přes operator <<

Běžný způsob využití proudů

1. Vytvoříme naši třídu s atributy
2. Překryjeme operátory vstupu >> a výstupu <<
   - používáme rozhranní `istream` resp. `ostream`
3. Implementujeme načtení resp. zobrazení atributů do proudu
4. Připojíme objekt na vstupní nebo výstupní proud
   - např. `cin, cout`
5. Beze změn lze nahradit konkrétní vstupního nebo výstupní proud
   - nahradíme např. za `fstream` nebo `stringstream`

Problematika přístupového módu

- Jednotlivé třídy resp. metody mají defaultní hodnoty přístupového módu
  - `ifstream` má automaticky `ios::in`
  - ifstream (const char * filename, ios_base::openmode mode = ios_base::in);
- Explicitně specifikovaný mód přístupu může ovlivnit defaultní hodnotu
  - záleží na konkrétní třídě a metodě
  - např. ios::out u ifstream je ignorován (vždy zůstane in)
  - např. ios::binary u stringstream

Shrnutí

- Jmenné prostory
  - nevyhnete se jim, vznikají automaticky ☺
  - můžete zavádět svoje další dodatečně
- Proudy – významný koncept
  - využití dědičnosti
- Souborové proudy pro práci se soubory
  - dědí z iostream => známé operace