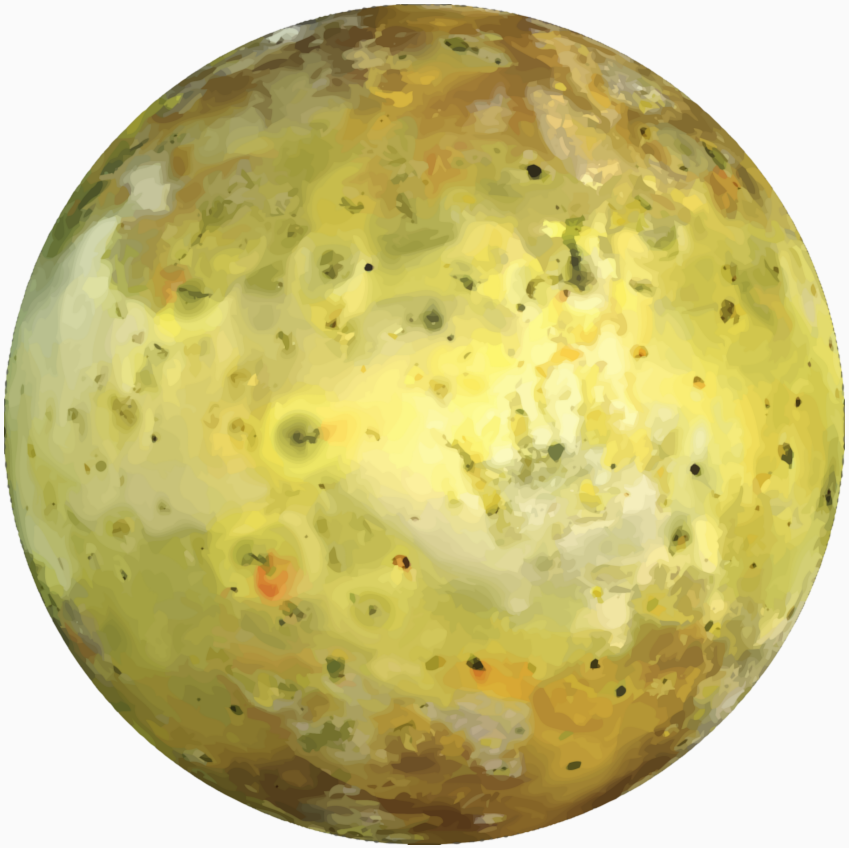
Vstup/výstup, formátování, soubory

PB161 Programování v jazyce C++

Nikola Beneš

2. května 2023

Vstup a výstup (IO)



Vstup a výstup v C++

Nízkoúrovňový vstup / výstup (Céčková knihovna)

- `std::FILE*`, `std::fopen`, …
- formátování: `std::printf`, `std::scanf` apod.
- nepoužívá RAII
- verze pro `wchar_t` v hlavičce `<cwchar>`

Proudový vstup / výstup

- klasická součást standardní knihovny C++
- abstrakce nad různými druhy zařízení
- trochu těžkopádná
- používá RAII

(C++23: `std::print`)

Motivace pro proudový vstup / výstup

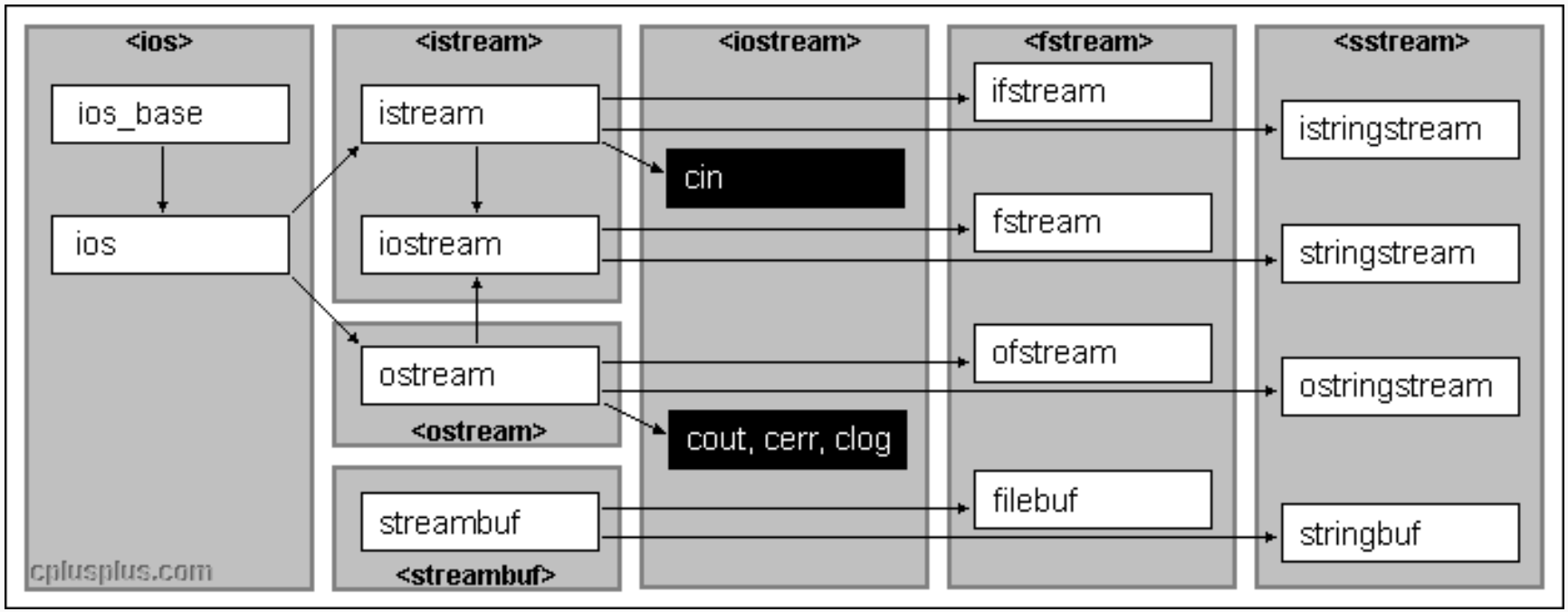
Vstup a výstup z různých druhů zařízení

- soubory
- klávesnice, obrazovka (terminál)
- tiskárna, skener, síťový socket, ...

Abstrakce konkrétního zařízení

- proudy (*streams*)
- data „plynou“ proudem od zdroje k cíli
- podtypový polymorfismus (dědičnost)

Hierarchie I/O proudů ve standardní knihovně



zdroj: https://cplusplus.com/reference/

Standardní vstup / výstup

Standardní instance

- `std::cin` – standardní vstup, typ `std::istream`
  - `stdin` v C
- `std::cout` – standardní výstup, typ `std::ostream`
  - `stdout` v C
- `std::cerr` – standardní chybový výstup, typ `std::ostream`
  - `stderr` v C
  - (implicitně) jediný nebufferovaný proud na tomto slajdu,
- `std::clog` – standardní logovací výstup, typ `std::ostream`
  - `stderr` v C

- verze pro `wchar_t` – `std::wcin`, `std::wcout`, …
- (jiné verze opět nejsou…)

Vztah mezi standardními vstupy / výstupy

Synchronizace s Céčkovými objekty `stdin`, `stdout`, `stderr`

- implicitně zapnutá
- I/O je (o něco) pomalejší, ale
  - míchání C++ a C stylu I/O se řadí správně
  - použití C++ proudů je bezpečné i vícevláknově
- dá se vypnout pomocí
  `std::ios_base::sync_with_stdio(false)`

Provázaní proudů

- `std::cin` je svázaný s `std::cout`
  - před jakýmkoli vstupem se zavolá `std::cout.flush()`
- `std::cerr` je svázaný s `std::cout`
  - před jakýmkoli výstupem se zavolá `std::cout.flush()`
- `std::clog` není (implicitně) svázaný s ničím
- dá se zjistit / změnit metodou `tie`

Operátory (formátovaného) vstupu a výstupu

basic_io.cpp

Operátor (formátovaného) vstupu >>

- u vestavěných typů a typů ze `std` ignoruje bílé místo na začátku

```
std::cin >> i >> d >> s;
```

Operátor (formátovaného) výstupu <<

```
std::cout << i << ' ' << d << ' ' << s << '\n';
std::cout << (i == d) << '\n';
```

- pozor na prioritu operátorů
  - << a >> jsou původně operátory bitového posuvu
- operátory vrací referenci na proud
  - umožňuje řetězení

Přetížení operátorů vstupu a výstupu

op_stream.cpp

`operator>>`, `operator<<`

- musí to být volné funkce (ideálně **`friend`**)
  - nemůžeme zasáhnout do tříd ze standardní knihovny
- vrací referenci na proud
  - umožňuje řetězení
- `>>` by měl nastavit `failbit` v případě špatného formátu
  - `in.setstate(std::ios::failbit)`

```
std::istream& operator>>(std::istream& in, X& x);
std::ostream& operator<<(std::ostream& out, const X& x);
```

Chybové stavy vstupu a výstupu

io_states.cpp

Stav proudu

- příznaky naznačující chybový stav proudu
  - `eofbit`, `failbit`, `badbit`
- metody pro testování stavu
  - `good()`, `fail()`, `eof()`
- typová konverze na `bool`
  - negace výsledku `fail()`
  - užitečné pro načítání vstupu v cyklu
- vyčištění příznaků: `clear()`

```
while (std::cin >> num) {
    std::cout << "Number: " << num << '\n';
}
```

https://en.cppreference.com/w/cpp/io/ios_base/iostate

Výjimky

- knihovna proudů implicitně výjimky nepoužívá
  - částečně historické důvody, ale
  - ne vždy je to vhodné: „došel jsem na konec souboru“ může a nemusí být výjimečná situace
- vynucení použití výjimek: metoda `exceptions()`
  - specifikace, pro které příznaky se má výjimka vyhodit
  - výjimka typu `std::ios_base::failure`

```
try {
    std::cin.exceptions(std::ios_base::failbit);
    // ...
} catch (std::ios_base::failure& ex) {
    std::cerr << "I/O Error: " << ex.what() << '\n';
}
```

Souborové proudy

files.cpp

Proudy pro vstup/výstup ze/do souborů

- typy `std::ifstream`, `std::ofstream`, `std::fstream`
- konstruktor se jménem souboru
- proud se automaticky zavře při ukončení existence objektu (v destruktoru)
- explicitní metody `open()`, `close()` (většinou nejsou potřeba)

```
{
    std::ofstream out("message.txt");
    out << "ZHOFRPH WR JUDYLWB IDOOV!\n";
}
```

Souborové proudy

Mód otevření souboru (nepovinný parametr konstruktoru)

- binární příznaky (*flags*), kombinujeme pomocí `|`
- typ otevření
  - `std::ios::in` (vstup, implicitní pro `std::ifstream`)
  - `std::ios::out` (výstup, implicitní pro `std::ofstream`)
  - `std::ios::in | std::ios::out` (obojí, implicitní pro `std::fstream`)
- způsob otevření
  - `std::ios::binary` (binární data, implicitní jsou textová data)
  - `std::ios::app` (*append*, výstup na konec souboru)
  - `std::ios::trunc` (vymaže obsah souboru, implicitní pro `std::ofstream`)

https://en.cppreference.com/w/cpp/io/basic_filebuf/open

sstream.cpp

Proudy v paměti

- typy `std::istringstream`, `std::ostringstream`, `std::stringstream`
- konstruktor může brát řetězec – počáteční stav proudu
- metoda `str()` nastaví obsah proudu
  - bez parametrů *vrací* aktuální obsah

```
std::ostringstream out;
out << "PI = " << 3.14 << '\n';
auto s = out.str();
std::cout << s << '\n';

std::istringstream in("X = 1");
in >> var >> dummy >> value;
std::cout << var << " = " << value << '\n';
```

Vstupní proudy (`std::istream` a potomci)

Neformátovaný vstup

- `read(buffer, length)`
  - přečte daný počet „znaků“
  - (typicky pro binární soubory – čtení po bajtech)
- `get(buffer, length)`, `get(buffer, length, delim)`
  - jako **`read`**, ale zastaví se před znakem **`delim`** (implicitně `'\n'`)
- `getline`
  - jako **`get`**, ale přečte i znak **`delim`** (ale neuloží ho)
- `get()` bez parametrů – načte jeden znak
  - vrací speciální hodnotu pro konec souboru
- `peek()` – náhled na další znak bez jeho přečtení
- `gcount()` – počet znaků načtených při posledním vstupu
- `ignore(count, delim)`
  - ignoruje **`count`** znaků ze vstupu (až po znak **`delim`**)
- a další … (viz dokumentaci)

Vstupní proudy (`std::istream` a potomci)

`sstream.cpp`

- načtení řádku do řetězce typu `std::string`
  - *volná* funkce
    `std::getline(std::istream&, std::string&)`
  - vrací referenci na vstupní proud

```
std::string line;
while (std::getline(std::cin, line)) {
    // ...
}
```

Výstupní proudy (`std::ostream` a potomci)

Neformátovaný výstup

- metoda `write(output, count)`
  - protiklad `read`
- metoda `put()`
  - zapíše jeden „znak“

Pozice a posun v proudu

- odkud se čte, kam se zapisuje?
  - „get“ ukazatel (`std::istream` a potomci)
  - „put“ ukazatel (`std::ostream` a potomci)
- `tellg()`, `tellp()` – získání pozice ukazatelů
- `seekg()`, `seekp()` – nastavení pozice ukazatelů;
  jeden parametr – pozice získaná `tell*()`;
  nebo dva parametry
  - offset: o kolik se posunout
  - `std::ios::beg` od začátku proudu
    `std::ios::cur` od aktuální pozice
    `std::ios::end` od konce proudu
- u některých proudů nedává smysl (`std::cin` apod.)
- pro soubory:
  - počáteční pozice v souboru – závisí na módu otevření
  - zápis na konci souboru soubor zvětšuje
  - zápis uvnitř souboru soubor přepisuje

Vyrovnávací paměť

Buffer

- data poslaná do proudu nemusí být ihned zapsána do cíle
- vyrovnávací paměť typu `std::streambuf`
- přenos z vyrovnávací paměti do cíle
  - při uzavření souboru (`close()`, destruktor)
  - při zaplnění bufferu
  - explicitně – pomocí manipulátorů (uvidíme za chvíli)
  - při jakékoli akci se svázaným proudem (pomocí metody `tie()`)

Proudové iterátory

stream_iter.cpp

`std::istream_iterator<typ>`

- jeden parametr – vstupní proud
- čtení z iterátoru je načítání hodnot ze vstupu
- bez parametru: iterátor na konec (jakéhokoli) proudu
- pohledová verze: `std::views::istream<typ>`

`std::ostream_iterator<typ>`

- parametry – výstupní proud, (nepovinný) oddělovač
- zapisování do iterátoru způsobí zápis na výstup
- oddělovač vypíše i za posledním prvkem

Formátování

Formátování

Nízkoúrovňové

- Céčková knihovna: `std::printf`, `std::scanf` apod.
- viděli jsme minule: `std::to_chars`, `std::from_chars`

Proudové

- tzv. I/O manipulátory

`std::format` apod.

- nová část knihovny
- formátovací řetězce ve stylu jazyka Python
- *podpora v gcc až od verze 13 :-(*
- (podpora v clangu nekompletní)

Manipulátory

- speciální hodnoty, které je možno předávat operátorům << a >>
  - některé z nich jsou *adresovatelné* funkce
    (`proud << funkce` pak má efekt jako `funkce(proud)`)
- většina souvisí s formátováním
- některé mění stav permanentně, jiné jen pro příští výstup
  - čtěte dokumentaci

Neformátovací manipulátory

- `std::flush` vyprázdní buffer
- `std::endl` vyprázdní buffer + zapíše konec řádku
  - chcete-li jen zapsat konec řádku, použijte normálně `'\n'`

Manipulátory

Formátování vstupu / výstupu

- `std::dec`, `std::hex`, `std::oct` – reprezentace celých čísel
- `std::boolalpha`, `std::noboolalpha` – reprentace `bool`
- `std::quoted(řetězec)` – řetězec v uvozovkách

Formátování výstupu

- `std::left`, `std::right` – zarovnání
- `std::setw`, `std::setfill`, `std::setprecision`
  - šířka, výplňový znak, přesnost

https://en.cppreference.com/w/cpp/io/manip

restore.cpp

Použití uvnitř vlastních operátorů <<, >>

- pokud měníte způsob formátování apod., vraťte po skončení proud do původního stavu
  - ideálně to zabalte do RAII objektu
- původní hodnoty možno získat / nastavit metodami
  - `width()` – šířka výstupu
  - `precision()` – přesnost floating-point čísel
  - `fill()` – výplňový znak
  - `flags()` – formátovací příznaky (všechno ostatní)

Nová formátovací knihovna

format.cpp

```
std::format(formátovací řetězec, argumenty)
```

- formátovací řetězec musí být literál
- formátování ve stylu Pythonovského `str.format`
  - výskyty `{}` ve formátovacím řetězci nahrazeny argumenty
  - specifikace:
    https://en.cppreference.com/w/cpp/utility/format/formatter
- vrací `std::string` nebo `std::wstring` podle literálu
- podpora pro vlastní typy pomocí specializace šablony
  (mimo záběr předmětu)

Podpora

- gcc: až od verze 13
- clang: nekompletní, příklad můžete vyzkoušet s přepínači
  `-stdlib=libc++ -fexperimental-library`

Pro zajímavost: Lokalizace

`locale.cpp`

`std::locale` (umístění, *locale*)

- sada parametrů, které definují různé varianty chování uživatelských rozhraní, mimo jiné i formátování
- `std::locale::classic()`
  - minimální POSIXové locale `"C"`, implicitní na začátku programu
- `std::locale(jméno)`
  - konstruktor dle zadaného jména
  - prázdný řetězec znamená „to, co má nastaveno uživatel“
- `std::locale::global(locale)`
  - nastaví globální locale, vrací předchozí hodnotu
  - používá je např. `std::to_string`
- každý I/O proud má vlastní locale, nastavení pomocí metody `imbue` (vrací původní locale)