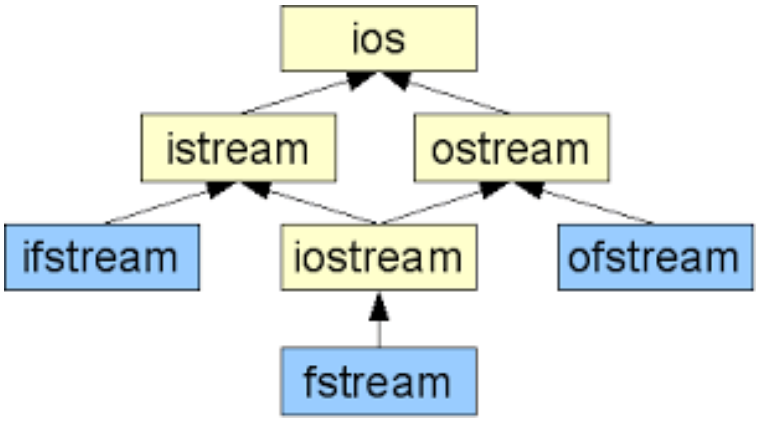
A programozás alapjai 2.

Hivatalos segédlet a hatodik laborhoz

C++ I/O

C++-ban az input és output kezelésére **stream**-eket használunk. Stream-ekre a program tud írni (adatot küldeni) és tud róluk olvasni (adatot fogadni). Az írás/olvasás mindig **szekvenciális**, azaz a **byte**-ok egymás után kerülnek a stream-re, és ugyanilyen sorrendben olvashatók le. Az alapvető header fájl, ami I/O műveletek végzését teszi lehetővé C++ programokban, az **iostream**, mely definiálja a következő, **std** névtérben lévő objektumokat (csak a legfontosabbak): cout, cin, clog, cerr



cout: standard output stream

- ostream típusú (output stream)
- alapértelmezetten a képernyővel (konzol) van összeköttetésben, erre fog írni
- használatához szükséges a **stream insertion operátor (<<)**, ami az előtte lévő stream-re szúrja be az őt követő karakter(eket)
- pl. cout << "Hello VIK World!";
- több stream insertion operátor összefűzhető, általában akkor használjuk ha változókat, függvény visszatérési értékét és szöveget vegyesen szeretnénk kiírni
- pl. cout << "Hello " << name << "! " << getSmiley();
- új sort kétféleképp kezdhetünk:
    - **\n** önmagában (inkább a következő pontban leírtakat használjuk) vagy szövegbe helyezve
    - std névtérben lévő **endl** használatával pl. cout << "Hello VIK World!" << endl;
        - endl hatására a stream-en végrehajtódik egy **flush** művelet, ami azt jelenti, hogy minden, a stream-ben lévő karakter kiírásra kerül. Ez azért hasznos, mert a stream buffer-ként működhet, tehát sokáig gyűjtheti a beszúrt karaktereket, mielőtt kiírná őket ténylegesen a konzolra (vagy ahová a stream irányítva van). Flush-sal kikényszerítjük ezt a kiírást.

(A fenti ábrán az év elején beszélt “is-a” hierarchia látható).

cin: standard input stream

- istream típusú (input stream)
- alapértelmezetten a billentyűzethez fér hozzá (mivel ez az alapértelmezett input eszköz)
- használatához szükséges a **stream extraction operátor (>>)**, ami az előtte lévő stream-ből kapott értékeket eltárolja a mögötte álló változóba
- pl. **int** pageNumber; cin >> pageNumber;

- ekkor a program addig **vár**, amíg nem kap inputot a cin-től
- billentyűzetről történő adatbevitelkor ENTER hatására fog a program olvasni az input stream-ről
- a >> operátor utáni **változó típusából** állapítja meg, hogyan kell **értelmeznie** a kapott byte-sorozatot
- **mindig ellenőrizni** kell, hogy sikerült-e a kapott byte-sorozat átalakítása
    - pl. nem sikerül, ha int-et vár és valamilyen szöveget kap
- több stream extraction operátor összefűzhető
    - több érték elválasztására használható bevitelkor: szóköz, új sor, tabulátor
    - pl. cin >> a >> b;
    - ekvivalens ezzel: cin >> a; cin >> b;
- string beolvasása
    - szavanként történik alapértelmezetten (szóközzel / tabulátorral / új sor karakterrel elválasztva)
    - egész sor beolvasása: getline(cin, stringVariable)

cerr: standard error (output) stream

- ostream típusú
- alapból a standard error eszközhöz van kötve, ami alapértelmezetten a képernyő (konzol)
- nincs bufferelve, tehát a stream-re kerülő karakterek azonnal megjelennek a kimeneten
- ugyanolyan szintaxissal használjuk, mint a cout-ot

clog: standard log (output) stream

- ostream típusú
- alapból a standard error eszközhöz van csatlakoztatva, ami alapértelmezetten a kijelző (konzol)
- bufferelt, tehát a stream-re írt karakterek akkor jelennek meg a kimeneten, ha a buffer betelik vagy flush műveletet hajtunk végre rajta
- ugyanolyan szintaxissal használjuk, mint a cout-ot

Szövegfájlok írása/olvasása

Két alap osztályt használunk szöveges fájlok (ASCII) kezelésére. Ha például egy fájlt szeretnénk olvasni, akkor példányosítunk egy ifstream típusú objektumot, aminek konstruktor paraméterként megadjuk a fájl nevét elérési útvonallal. Írásnál ugyanez a helyzet, csak ofstreammel játszuk el. Példányosítás után ezen az objektumon keresztül tudjuk a fájl tartalmát manipulálni adott irányú shift operátorral. A használat azért hasonlít például az std::cout, std::cin-re, mert az operációs rendszer mind a fájlműveleteket, mind a konzolos képernyőre való kiírást I/O fájlműveletként kezeli.

Ha egy fájlt nem sikerül megnyitni, a streamen a failbit flag lesz beállítva.

- **ifstream** *(input **f**ile **stream**)*
    - Példa:
        - ifstream myInputFile("myInput.txt");

- **ofstream** *(**o**utput **f**ile **stream**)*
  - Fájlmegnyitási módok:
    - ios::app // fájl végéhez való hozzáfűzés
    - ios::ate // fájlmutatót a fájl végére állítja
    - ios::trunc // törli a fájl tartalmát
  - Alapértelmezetten úgy nyitja meg a fájlt, hogy
    - ha még nem létezik a fájl a megadott útvonalon: létrehozza
    - ha már létezik: törli a tartalmát
  - Példa:
    - ofstream myOutputFile("myOutput.txt", ios::app);

stringstream

Az <sstream> headerben definiált típus, mely lehetővé teszi, hogy string-eket stream-ekként kezeljünk a cin és cout-hoz hasonlóan. Hasznos funkció például string-ek és számok közti konverzióhoz.

Pl.:

```
string ageStr("10");
int ageInt;
stringstream(ageStr) >> ageInt;
```

Operátorok túlterhelése

Az operátor túlterhelés témakör mindössze a korábban már megismert, **függvénynév túlterhelés lehetőségét egészíti ki**.

A problémakör jobb megértése érdekében vegyük például a következő esetet: komplex algebrában kardinális jelentőségű a komplex számok összeadása. Ha ezt reprezentálni szeretnénk C++-ban, egyből egy *Complex* (vagy hasonló nevű) osztály jut eszünkbe, amely tárol egy valós és egy képzetes (imaginárius) részt.

Mivel **gyakori** az ezeken való objektumokon való **műveletvégzés** (c1+c2, c1-c2, stb. ahol c1, c2 Complex objektumok), és nem mellékes ezeknek a minél **egyszerűbb** standard kimenetre való **kiíratás**a sem, minél rövidebb módon valósítjuk meg ezeket (std::cout << c1).

Pl. definiálhatnánk a Complex osztályban egy Add nevű függvényt, de sokkal kézenfekvőbb a megszokott módon végrehajtani az összeadást, c1+c2 használattal.

Ezeknek a problémáknak a megoldására lehetőségünk van **speciális (*tag*)függvények túlterhelésére**. Egy-egy műveletet egy-egy speciális függvény testesít meg.

Megvalósítás

Operátor tagfüggvény

<visszatérési érték típusa> ***operator*****<operációs jel>(<jobb oldali operandus típusa> <neve>){...}**

Példa:

```
Complex operator+(const double right) const
{ return Complex(real+right,imaginary); }
```

Ez lehetővé teszi a “c+10” típusú műveleteket. Jusson eszünkbe, hogyha az ilyen függvények tagfüggvények, bal oldali operandusnak alapértelmezetten azt az objektumot veszik, amin hívták az operációt.

Ez valójában azt jelenti, hogy c.operator+(10). Ugyan így is tudja értelmezni a fordító, gyakorlatban nyilván célszerűbb a “c+10” alakú változatot használni.

Globális operátor

friend **<visszatérési érték típusa>** ***operator*****<operációs jel>****(<bal oldali operandus típusa> <neve>, <jobb oldali operandus típusa> <neve>){...}**

Példa:

```
friend Complex operator+(const double left, const Complex& right)
{return Complex(left+right.real,right.imaginary);}
```

Erre azért van szükség, mert az első módszert alkalmazva nincs lehetőségünk “10+c” alakú műveleteket definiálni. A megoldást a *friend* kulcsszó biztosítja, ugyanis használatakor nem kapja meg 0. paraméterként a *this* mutatót (ami arra az objektumra mutat, amin hívjuk a tagfüggvényt). Így már az operátor mindkét operandusát (bal és jobb oldalit) meg tudjuk adni.

Láthatjuk, hogy a friend sérti az egységbezárás (encapsulation) elvét, hiszen közvetlenül hozzáférünk a privát tagváltozókhoz.

Láncolás

Az operátor túlterhelés lehetőséget biztosít **műveletek összeláncolására** is, mint például: “c1 + 10 + c2”.

Ismeretes, hogyha az operációk a precedencia szabány szerint egyenrangúak, a számítógép balról jobbra értékeli ki őket. Az előző példát például úgy, hogy részeredményként eltárolja a c1+10 visszatérési értékét, majd ezt adja össze a c2-vel. Így két függvényhívásra fordul át végül: (c1.operator+(10)).operator+(c2).

Standard kimenet/bemenet

- Tudjuk, hogy az std::cout **std::ostream**, míg az std::cin **std::istream** típusúak (output és input stream). Ezért ezt a két típust fel tudjuk használni operátor túlterheléskor, mint paramétertípus.
- Mivel itt is meg kell valósítani a láncolhatóságot (std::cout << c1 << “ and “ << c2;), mindenképp ostream-et, vagy istream-et kell visszaadnunk. Azonban **referenciának kell lenniük**, tekintve, hogy ezek az “iostream” objektumok nem másolhatók
    - probléma: van belső állapotuk, mint például a pozíció ahova írnak/ahonnan olvasnak
    - ha másolhatók lennének, két az eredeti és a másolat a stream-ben ugyanoda mutatnának és ugyanoda írnának
    - ez konfliktust jelentene
- A “<<” és “>>” operátorok jellegéből fakadóan nem lehetnek tagfüggvények sem, így *friend*-et kell használni.
    - Ha tagfüggvény lenne: std::ostream& operator<<(std::ostream& output)
    - viszont akkor így kellene használni: c << std::cout;

Egy komplex példa: Complex osztály

compex.h

```
#pragma once
#include <iostream>

class Complex {
	double re;
	double im;
public:
	Complex(double = 0.0, double = 0.0); // Complex(), Complex(1), Complex(1,2)
	Complex(const Complex&);  // Complex c1(c2), Complex c1=c2

	void setRe(double); // c.setReal(1)
	void setIm(double); // c.setImaginary(2)

	double getRe() const; // c.getReal()
	double getIm() const; // c.getImaginary()

	Complex operator+(const Complex&) const; // c1+c2
	Complex operator+(const double) const; // c1+10
	Complex operator-(const Complex&) const; // c1-c2
	Complex operator-() const; // -c (-re, -im)
	Complex& operator=(const Complex&); // c1=c2
	void operator+=(const Complex&); // c1+=c2;
	void operator+=(const double); // c1+=10;
	void operator-=(const Complex&); // c1-=c2;
	void operator-=(const double); // c1-=10;
	Complex operator--(); // --c1; // konjugált
	bool operator==(const Complex&) const; // c1==c2
	bool operator!=(const Complex&) const; // c1!=c2
};

Complex operator+(const double, const Complex&); // 10+c1
std::ostream& operator<<(std::ostream&, const Complex&); // 10+c1
std::istream& operator>>(std::istream&, Complex&); // 10+c1
```

complex.cpp

```
#include "complex.h"

using namespace std;

// Tesztelés céljából kiírjuk, hogy melyik függvény hívódott meg, de debuggerrel
is lehetne.
void w(const char* functionName){cout << "#CALLING " << functionName << endl;}

Complex::Complex(double re, double im):re(re),im(im){w(__FUNCSIG__);}
Complex::Complex(const Complex& other):re(other.re), im(other.im)
	{w(__FUNCSIG__);}
```

```
void Complex::setRe(double re){w(__FUNCSIG__); this->re = re;}
void Complex::setIm(double im){w(__FUNCSIG__); this->im = im;}
double Complex::getRe() const{w(__FUNCSIG__); return this->re;}
double Complex::getIm() const{w(__FUNCSIG__); return this->im;}

Complex Complex::operator+(const Complex& right) const
      { w(__FUNCSIG__); return Complex(re+right.re, im+right.im);}
Complex Complex::operator+(const double right) const
      { w(__FUNCSIG__); return Complex(re+right, im);}
Complex Complex::operator-(const Complex& right) const
      { w(__FUNCSIG__); return *this + -right; }
Complex Complex::operator-() const
      { w(__FUNCSIG__); return Complex(-re, -im);}
Complex& Complex::operator=(const Complex& right)
      { w(__FUNCSIG__); re = right.re;im = right.im;return *this;}
Complex Complex::operator--()
      { w(__FUNCSIG__); im *= -1; return *this; }
void Complex::operator+=(const Complex& right)
      { w(__FUNCSIG__); this->re += right.re;this->im += right.im;}
void Complex::operator+=(const double right)
      { w(__FUNCSIG__); this->re += right;}
void Complex::operator-=(const Complex& right)
      { w(__FUNCSIG__); this->re -= right.re;this->im -= right.im;}
void Complex::operator-=(const double right)
      { w(__FUNCSIG__); this->re -= right;}
bool Complex::operator==(const Complex& right) const
      { w(__FUNCSIG__); return re == right.re && im == right.im;}
bool Complex::operator!=(const Complex& right) const
      { w(__FUNCSIG__); return re != right.re || im != right.im; }

Complex operator+(const double left, const Complex& right)
      { w(__FUNCSIG__); return right+left; }
ostream& operator<<(ostream& os, const Complex& c)
      { w(__FUNCSIG__);
      os << noshowpos << c.getRe() << showpos << c.getIm() << "i" << noshowpos;
      return os;}
istream& operator>>(istream& is, Complex &z){
      w(__FUNCSIG__);
      double re, im; char c = 0;
      is >> re;
      is >> c; //+ vagy -
      if (c != '+' && c!='-')is.clear(ios::failbit);
      is >> im;
      is >> c; //i
      if (c != 'i')is.clear(ios::failbit);
      if (is.good())z = Complex(re, im);
      return is;
}
```

complexTest.cpp

```
#include <iostream>
#include "complex.h"
```

```
using namespace std;

int main()
{
	const Complex c1(-10, 20);
	Complex c2(c1); // Copy konstruktor. Complex c2 = c1; ugyenezt jelenti.

	// re=-10; im=20
	cout << "re=" << c1.getRe() << "; im=" << c1.getIm() << endl;
	cout << c2 << endl; // -10+20i

	Complex c3; // 0+0i
	cin >> c3; // 4+5i
	cout << c3 << endl; // 4+5i
	c3.setRe(11);
	c3.setIm(-44);
	cout << c3 << endl; // 11-44i

	c3 += 4;
	cout << c3 << endl; // 15-44i

	// senki se írja így!
	cout << c3.operator-(4) << endl; // 11-44i
	cout << c3 - 4 << endl; // 11-44i
	cout << -c3 << endl; // -15+44i
	c3 += (-c1) + c2 - (c3 + 10); // -10+0i

	cout << c3 << endl; // -10+0i
	cout << --c3 << endl; // -10-0i

	cout << 3 + c3 << endl; // -7+0i

	if (c1 == c2)
		cout << "c1==c2" << endl; // c1==c2

	if (c1 != c3)
		cout << "c1 != c3" << endl; // c1 != c3
	return 0;
}
```

Miért érhetők el a másik Complex privát tagváltozói?

Teljesen mindegy, hány Complex példányod van, a memóriában akkor is egy másolata lesz a függvény definíciónak. Ennek ellenére - mivel a this pointer minden híváskor átadódik (kivéve static-nél) -, úgy érzékeljük, hogy mégis több példány van a függvényből. Ez számunkra azt jelenti, hogy az other privát tagváltozóihoz is hozzá lehet férni. C++-ban ez nem sérti az encapsulation OO elvet.