A programozás alapjai 2.

Hivatalos laborsegédlet hallgatók számára.

Alapprobléma

Legyen adott a következő program:

```
#include <iostream>
class Animal {
public:
	// nem virtuális függvény
	void eat() { std::cout << "I'm eating generic food." << std::endl; }
};

class Cat : public Animal {
public:
	// a viselkedést felülírjuk
	void eat() { std::cout << "I'm eating a rat." << std::endl; }
};

void func(Animal* xyz) { xyz->eat(); }

int main() {
	Animal *animal = new Animal;
	Cat *cat = new Cat;

	// itt jól működik
	animal->eat(); // kimenet: "I'm eating generic food."
	cat->eat(); // kimenet: "I'm eating a rat."

	// probléma: nem működik jól
	func(animal); // kimenet: "I'm eating generic food."
	func(cat);    // kimenet: "I'm eating generic food."
}
```

Figyeljük meg a fenti kódsort! A köztes func(Animal*) függvény meghívásakor **ugyanazt a viselkedést várnánk el, mint nélküle**, de sajnálatos módon mégsem úgy működik, ahogy kellene. Magyarán, ha egy leszármazott osztály felüldefiniálja az ősosztály viselkedését valamely tekintetben, akkor minden esetben a leszármazott új viselkedése érvényesüljön, attól függetlenül, hogy milyen interfészen keresztül hivatkozunk rá (tehát legyen független a viselkedése mind az ősosztály, mind a saját interfészétől).

A problémát az okozza, hogy túl korán (fordítási időben) rendeljük hozzá az egyes metódusokhoz a definíciót.

A megoldáshoz vezető út: binding

Amikor egy programot lefordít a fordító (compiler), minden utasítást (statement) a gép által futtatható nyelvre írja át. Ekkor minden gépi nyelvi sorhoz egy egyedi, szekvenciális címet rendel. Ebből következik, hogy minden függvénynek lesz egy saját azonosítója.

Bindingnak hívják azt a folyamatot, amikor az azonosítók (változó -és függvénynevek) címekké alakulnak át. Ennek az egyik változata az **early binding**, amikor a compiler fordítási időben el tudja dönteni, hogy az egyes függvényhívásokkor milyen címre ugorjon tovább.

Azonban néhány programban előfordulhat, hogy csak futási időben derül ki, hogy melyik függvénydefiníciót kell meghívni. Vegyük például a következő esetet:

```
int add(int a, int b){return a+b;}
int subtract(int a, int b){return a-b;}
/// ...
int (*pFcn)(int, int);
if(RANDOM)
      pFcn = add;
else
      pFcn = subtract;
std::cout << pFcn(20,10);
```

Mivel véletlenszerű, hogy melyik függvényre fog végül mutatni a pFcn függvény pointer (függhet pl. felhasználói interakciótól), nem lehet eldönteni fordítási időben, hogy a kiíratásnál melyik függvény címére ugorjon tovább. Ezért van létjogosultsága a binding másik fajtájának, a **late binding**nak, ami a futtatási idejű hozzárendelést jelenti.

Megoldás: virtuális tagfüggvények

Ahhoz, hogy általunk kívánatos működésre bírjuk az alapproblémánál bemutatott kódot, late bindingra van szükségünk. Erre nyújt lehetőséget a **virtual** kulcszó, amit a függvénydeklaráció elejére kell biggyeszteni.

Ennek megfelelően az Animal eat()-je a következőre módosul:

```
virtual void eat() { std::cout << "I'm eating generic food." << std::endl; }
```

A virtuális tagfüggvények implementálásához a C++ a late binding egy speciális formáját használja, amit virtuális táblának (*virtual table*) neveztek el. Ez tömören egy LUT (lookup table), amely függvénynevek alapján történő címlekérésre szolgál late binding alatt.

Tisztán virtuális tagfüggvény

Oké, szóval ezzel megoldottuk a problémát, nem? Nos, végülis igen, mert a macska mindig patkányt eszik. Viszont ha jobban belegondolunk, annak már semmi értelme, hogy az állat általános ételt fogyaszt. Szóval ezt a szépséghibát még jó lenne megoldani... Valahogy úgy kellene, hogy az Animal eat()-je lehetőleg ne is rendelkezzen definícióval, mert láthatóan felesleges lenne. Szimplán rábízzuk ezt a feladatot a leszármazottakra (de nekik már kötelező ezt megtenniük [de csak akkor, ha nem absztrakt osztályok, ld. később]).

Az ilyen definíció nélküli, de leszármazottban definiálásra váró függvényeket hívják **tisztán virtuális tagfüggvényeknek**.

Megvalósítása pedig a következőképp történik: a **virtuális függvény deklarációját egyenlővé tesszük nullával és nem definiáljuk** (jelezvén, hogy a virtuális táblában nem lehet innen hova ugorni).

Tehát az Animal eat()-je így módosul:

```
virtual void eat() = 0;
```

Tisztán virtuális tagfüggvény következménye: absztrakt osztály

Egy osztályt akkor és csak akkor lehet példányosítani, ha minden metódusa definiálva van. Viszont, ha egy adott osztály rendelkezik tisztán virtuális tagfüggvénnyel, akkor definíciója sincs. Leszármazottból (ha van neki egyáltalán) meg nyilván nem „mászhat fel” a definíció, mert az öröklés egyirányú (öröklési fa). Ez maga után vonja azt a következményt, hogy a tisztán virtuális tagfüggvénnyel rendelkező osztályokat nem lehet példányosítani. Az ilyen osztályokat **absztrakt osztályoknak** nevezzük.

Fontos, hogy ha például valamilyen **kívülről hozzáférhetetlen láthatóságot adnánk a konstruktornak** (pl. private, protected) akkor **nem számítana absztrakt osztálynak**, hiszen ugyanúgy lehetne példányosítani, azonban azzal a megkötéssel, hogy csak osztályon belül. Erre példa a korábban már bemutatott Meyers’ Singleton tervezési minta.

Triviális, ám mégis megjegyzendő, hogy a **tisztán virtuális tagfüggvényen kívül lehet olyan tagfüggvénye, amely rendelkezik definícióval**. Ekkor a definiált függvény által megvalósított viselkedéssel rendelkezi fognak a leszármazottak is.

Kitérő: interfész (interface)

C++-ban nem definiáltak külön nyelvi elemet az olyan absztrakt osztályoknak, aminek minden függvénye tisztán virtuális, azonban sok más nyelvben (pl.: C#, Java) létezik ilyen, amelyet nemes egyszerűséggel **interfésznek** (*interface*) neveztek el. Ennek megfelelően pedig az *interface* kulcsszóval lehet őket definiálni más nyelvekben. Az érdeklődők az ajánlott irodalomban olvashatnak erről bővebben.

Egy hasznos alkalmazás: heterogén kollekció

Tegyük fel, hogy ki szeretnénk rajzolni N db kört és M db téglalapot (a 3. labor Circle és Rectangle osztályai) grafikusan. A jelen állás szerint a legpraktikusabb az lenne, ha definiálnánk egy Circle és Rectangle tömböt, amelyekben a kívánt objektumok vannak, végigmegyünk mindkettőn és meghívjuk a *draw()* függvényüket (szintén tegyük fel, hogy ilyen létezik).

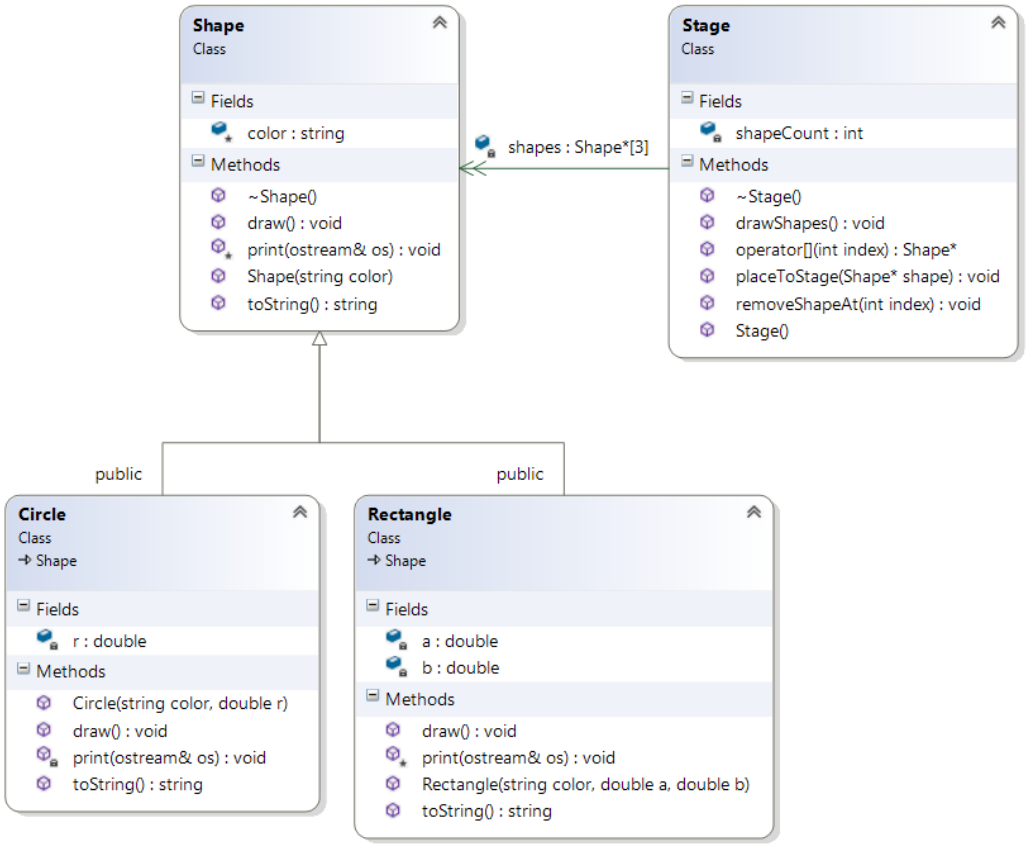
Ez a módszer még két osztálynál úgy-ahogy “elmegy”, de egy komolyabb szoftvernél vállalhatatlan, hiszen ha bővítenénk a kódbázisunkat egy új osztállyal, mindig definiálni kellene egy új tömböt, új ciklust kellene írni, mindig módosulna a régi kódunk, ráadásul tele lenne ismétlődéssel (DRY). Magyarán **a kirajzolás felelősségét megvalósító rész rugalmatlan lenne a változások felől nézve**.

A problémát úgy oldjuk meg, hogy írunk egy *Shape* közös viselkedést leíró (*draw() tisztán virtuális tagfüggvény*) absztrakt osztályt és ebből *örököltetjük* a Circle és Rectangle osztályokon kívül az új osztályokat is. Ezt követően már nem kell külön-külön tömböket készíteni az egyes osztályoknak, hanem elég egyetlen Shape interfésszel rendelkező objektumokra mutató tömb, amiben lehet Circle és Rectangle is vegyesen azért, mert a *Circle és a Rectangle is Shape*. Ez maga után vonja azt is, hogy elég egyetlen egy ciklus, amit **nem kell módosítanunk újabb Shape-ből való örököltetésekkor**. Ezt a megoldást hívjuk **heterogén kollekciónak**, mivel heterogén (különböző) elemeket kollekcióban (gyűjteményben, tömb, stb.) tárolunk.

Van viszont még egy probléma: ha az egyik leszármazott dinamikus memóriakezelést valósít meg, akkor abban nyilván meg kell írnunk a destruktort, hogy a lefoglalt memóriaterületet fel tudjuk szabadítani. Ehhez azonban **virtuállissá kell tennünk az ős absztrakt osztály destruktorát**, hogy late bindingkor mindenképp a leszármazott destruktora fusson le.

Egyszerű példa heterogén kollekció megvalósítására

Mindenek előtt szolgáljon a következő UML diagramm a gyorsabb áttekintés végett:



Grafikus alkalmazások esetén gyakori, hogy a kirajzolandó objektumokat (legyenek azok akár síkidomok akár testek) tároló objektumot Stage-nek (színpadnak) nevezik.

Most a Stage Shapeket tárol úgy, hogy igyekszik csak a számára legszükségesebb információkat tudni az általa tárolt objektumokról, mint például azt, hogy ki lehet őket rajzolni (draw()), illetve diagnosztikai kiíratót függvényt lehet hívni rajtuk. A Stage-re rá tudunk rakni Shape-eket (paceToStage()), le tudunk venni róla (removeShapeAt()), meg tudunk indexelni egy konkrét tárolt objektumot (operator[]), ki tudjuk rajzolni a tárolt alakzatokat (drawShapes()). A Circle és a Rectangle pedig továbbspecifikálja az általánosan megfogalmazott Shape-t.

A heterogén kollekciónak többek között van egy nagyon fontos gyakorlati alkalmazása a nyilvánvalón kívül: házi feladatban remekül lehet alkalmazni :)

main.cpp

```
#include <iostream>
#include "stage.h"
#include "shape.h"
#include "circle.h"
#include "rectangle.h"
```

```
int main()
{
	Stage stage; // no shapes on stage
	stage.placeToStage(new Circle("red", 6)); // red
	stage.placeToStage(new Circle("green", 7)); // red, green
	stage.placeToStage(new Rectangle("blue", 10)); // red, green, blue

	stage.drawShapes(); // red, green, blue
	stage.placeToStage(new Rectangle("yellow", 10, 4)); // no more room
	stage.drawShapes(); // red, green, blue

	stage.removeShapeAt(1); // red, blue
	stage.drawShapes(); // red, blue

	// dereferálás => operator[] => late binded print
	std::cout << *stage[0] << std::endl;
	std::cout << *stage[1] << std::endl;

	// a program végén meghívódik a stage destruktora
	// és felszabadítja a dinamikusan foglalt objektum területeit
}
```

shape.hpp

```
#pragma once
#include <string>

class Shape
{
	// protected, hogy a leszármazottak is
	// hozzá tudjanak férni és kívülről ne lehessen meghívni
protected:
	std::string color; // enum is lehetne
	// - a printet önmagában ne lehessen meghívni
	// - ha kiíratásra van szükség, arra ott van a frienddé tett operator<<
	// - minden másra (főleg diagnosztikára) ott a toString
	// - virtual: ha egy leszármazott felüldefiniálja a printet,
	//   akkor late binding miatt a leszármazott definíciója fog lefutni
	//   - viszont lehetőség van a leszármazottban meghívni
	//     az ősosztály definícióját: Shape::print(os)
	virtual void print(std::ostream& os) const {
		os << toString();
	}
public:
	Shape(std::string color = "transparent"): color(color) {}

	// virtual ~Shape() = 0; // így is lehetne, de:
	// - mivel pl. a Circle-ben nincs dinamikus adattag,
	//   nincs mit felszabadítani, ezért ott üres a destruktor
	//   - hasonlóan a Rectangle-nél és az összes ezekhez hasonló osztályoknál
	// - így mivel ez elég gyakori, az ősosztályban praktikus nem
	//     tisztán virtuálissá tenni a destruktort, helyette definiáljuk üres
függvényként
	virtual ~Shape() {}
```

```
	virtual std::string toString() const
	{
		// - továbbra sincs szükség sortörésre
		// - inkább rábízzuk a fejlesztőre, hogy hol akarja megtörni
		return "color=" + color;
	}

	// Shape-t önmagában nincs értelme kirajzolni
	virtual void draw() = 0;
	friend std::ostream& operator<<(std::ostream& os, const Shape& right){
		right.print(os);
		return os;
	}
};

std::ostream& operator<<(std::ostream&, const Shape&);
```

stage.hpp

```
#pragma once
#include <iostream>
#include "shape.h"

class Stage
{
	int shapeCount; // éppen hány Shape-t tárolunk
	Shape* shapes[3]; // legfeljebb 3 Shape-re mutatunk (jobb: dinamikus tömb)
public:
	Stage():shapeCount(0){} // kezdetben 0 db Shape-ünk van
	void placeToStage(Shape* shape) {
		// ha még fér a tömbbe, a végére illesztjük
		if (shapeCount < 3)
			shapes[shapeCount++] = shape;
		else
			// - különben hibát írunk ki
			// - (szebb alak: try{...}catch(...){...}, de ezt majd később :))
			std::cout << "No more room left on stage." << std::endl;
	}

	void removeShapeAt(int index) {
		// csak akkor tudunk törölni Shape-t, ha van egyáltalán
		if (!shapeCount)
			std::cout << "No shapes on stage." << std::endl;

		// figyelünk, hogy ne indexeljünk alul vagy felül
		if (index >= shapeCount || index < 0)
			std::cout << "Index is out of range." << std::endl;

		delete shapes[index]; // kitöröljük az indexelt elemet
		shapes[index] = shapes[--shapeCount]; // a helyére illesztjük a végén lévő
Shape-t
		shapes[shapeCount] = NULL; // a végét NULL-ra állítjuk
	}
```

```
	void drawShapes() {
		// végigmegyünk a tárolt Shape-ken és kirajzoljuk őket
		for (int i = 0; i < shapeCount; i++)
			shapes[i]->draw();
	}

	Shape* operator[](int index) const {
		// figyelünk a helyes indexelésre
		if (!shapeCount || index >= shapeCount || index < 0)
			return NULL;

		// visszaadjuk az indexelt Shape-re mutató pointert
		return shapes[index];
	}

	~Stage() {
		// fel is kell őket szabadítani!
		for (int i = 0; i < shapeCount; i++)
			delete shapes[i];
	}
};
```

circle.hpp

```
#pragma once
#include <sstream>
#include <iostream>
#include "shape.h"

class Circle: public Shape
{
	double r;
	// már nem kell, hogy protected legyen, de kívülről
	// még mindig ne legyen látható
	void print(std::ostream& os) const {
		os << "I'm a circle; " << toString() << "; ";
		// - mivel a print virtuális volt az ősosztályban és
		//   a leszármazottban van hozzá definíció, alap esetben
		//   az ősosztály definíciója nem hívódna meg
		// - mi viszont most úgy döntünk, hogy meghívjuk
		Shape::print(os);
	}
public:
	Circle(std::string color, double r = 0): Shape(color), r(r) {}
	std::string toString() const {
		std::stringstream ss;
		ss << "r=" << r;
		return ss.str();
	}
```

```
	void draw() {
		std::cout << "This circle is " << color << " (believe me)" << std::endl;
		float pr = 2; // a karakteres felületet alkotó cellák méretének aránya
		for (int i = -r; i <= r; i++) {
			for (int j = -r; j <= r; j++)  {
				float d = ((i*pr) / r)*((i*pr) / r) + (j / r)*(j / r);
				if (d>0.9 && d<1.5) // közelítés tapasztalati konstansokkal :)
					std::cout << '*';
				else
					std::cout << ' ';
			}
			std::cout << std::endl;
		}
	}
};
```

rectangle.hpp

```
#pragma once
#include <sstream>
#include <iostream>
#include "shape.h"

class Rectangle: public Shape
{
	double a, b;

	// már nem kell, hogy protected legyen, de kívülről
	// még mindig ne legyen látható
	void print(std::ostream& os) const {
		os << "I'm a rectangle; " << toString() << "; ";
		// - mivel a print virtuális volt az ősosztályban és
		//   a leszármazottban van hozzá definíció, alap esetben
		//   az ősosztály definíciója nem hívódna meg
		// - mi viszont most úgy döntünk, hogy meghívjuk
		Shape::print(os);
	}
public:
	Rectangle(std::string color, double a=1, double b=1):Shape(color),a(a),b(b) {}

	std::string toString() const {
		std::stringstream ss;
		ss << "a=" << a << "; b=" << b;
		return ss.str();
	}
```

```
	void draw() {
		std::cout << "This rectangle is " << color << " (believe me)" << std::endl;
		// téglalap teteje
		std::cout << '+';
		for (int i = 0; i < a-2; i++)
			std::cout << '-';
		std::cout << '+' << std::endl;

		// teteje és alja közötti rész
		for(int i=0;i<b;i++)
		{
			std::cout << '|';
			for (int j = 0; j < a-2; j++)
				std::cout << ' ';
			std::cout << '|' << std::endl;
		}

		// téglalap alja
		std::cout << '+';
		for (int i = 0; i < a-2; i++)
			std::cout << '-';
		std::cout << '+' << std::endl;
	}
};
```

Ajánlott irodalom

- http://stackoverflow.com/questions/2391679/why-do-we-need-virtual-functions-in-c
- http://www.learncpp.com/cpp-tutorial/124-early-binding-and-late-binding/
- http://www.learncpp.com/cpp-tutorial/125-the-virtual-table/
- http://www.learncpp.com/cpp-tutorial/126-pure-virtual-functions-abstract-base-classes-and-interface-classes/
- https://www.tutorialspoint.com/csharp/csharp_interfaces.htm