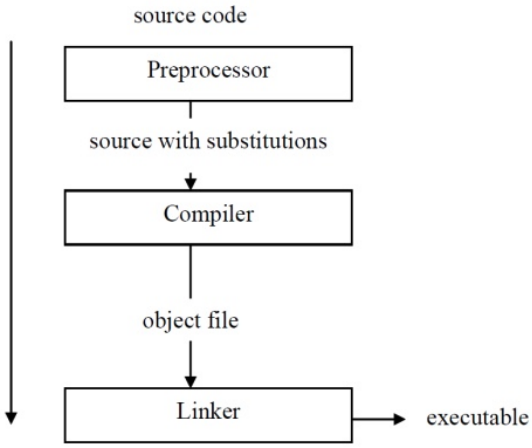
A programozás alapjai 2

BME AUT Programozás alapjai 2 tárgyi segédletek oldala

static kulcsszóról bővebben (angol)

1. előadás

A fordítás menete



Preprocesszor

Végig nézi a kódot és minden neki szóló utasításnál (preprocesszor direktíva), melynek mindegyike #-el kezdődik (pl. `#include`) változtatást végez a kódon. Az így kapott fájlok C/C++ nyelvűek és már nem tartalmaznak preprocesszor direktívát. Emellett speciális jelöléseket tesz a kódba a compiler számára, amiből visszakövethető, hogy melyik behelyettesített kódrészlet honnan származik, ez később hiba elhárításnál pontosan meg tudja mondani honnan ered a hiba.

Szemantika (kód nyelvtan)

`#include <fájl>` esetén a hivatkozott fájl egész tartalmát bemásolja a kódba.

`#define azonosito helyettesites` (pl. `#define NICE 69`) esetén a kódban az “azonosito” **minden** előfordulásakor lecseréli a megadott értékre.

Header fájlokban:

```
#ifndef VALAMI //ha nincs definiálva “VALAMI”, akkor…
#define VALAMI //definiálunk egy “VALAMI” direktívát
//itt vannak a változók és függvények deklarációi…
#endif //az ifndef vége
```

Ez arra jó, hogyha esetleg egy nagyobb szoftver esetén többször `#include`-olnánk be egy header fájlt, akkor csak egyszer másolja be a kódba.

`#pragma` Az előbbit helyettesíti, ha a header fájl első sorában van.

Compiler

A preprocesszor kimenetéből object fájlt készít. A C/C++ nyelvű kódot átírja előbb assembly-be, majd az assembler bináris kódba. Ebben a fájlban még lehetnek olyan hivatkozások, melyeket a fordító nem ismer, például ha egy függvény definícója egy másik forrásfájlban van. Ez a compilernek nem gond, csak a nevét, visszatérési értékének típusát és paraméterezését, azaz deklarációját kell ismernie, azt nem, hogy ez hol található fizikailag. A deklarációt úgy is elképzelhetjük, mint egy igéretet a compilernek, hogy ez bizony valahol létezik, nyugodtan használja. Tehát az object fájl referenciákat, hivatkozásokat tartalmaz más fájlokban lévő elemekre - függvényekre, változókra. Minden forrásfájlhoz külön object fájl jön létre, melyeknek kiretjesztése .o (linuxon) vagy .obj (windowson). Ebben a lépésben fordulhat elő pl. syntax error.

Linker

A fordítás végső kimenetét állítja elő, egy futtatható fájlt (.exe) vagy dinamikusan linkelhető fájlt (.dll), melyhez
felhasználja az összes object fájlt, ami a programunkhoz tartozik. Behelyettesíti az előbb említett
hivatkozásokat a memóriacímükkel a többi object fájlt vagy hivatkozott libraryt felhasználva.
Ha egy függvény
vagy változó definíciót egyik object fájlban sem találja, vagy többet is talál, hiba keletkezik.

C és C++ fordítója

- gcc: GNU C Complier
- g++: GNU C++ Complier
- g++ a c nyelvű forrásálományokat is c++ kódként fordítja
- több forrás állományunk van → először külön fordítani őket, aztán linkelni

---

extern

- Változó vagy függvény deklaráció előtt áll pl. `extern int i;`
- Azt mondja meg a fordítónak, hogy a függvény/változó “valahol létezik, definiálva van”, nem kell tudnia,
  hogy hol, csak a típusát.
- Minden függvényen kívül (file scope) definiált változó alapértelmezetten extern is. File scope a `main()`
  függvényen is kívül van.
- Az extern változónak/függvénynek pontosan egyszer kell definiálva lennie. File scope-ban kell lennie.
- Ha több helyről is tud definíciót szerezni, a linker hibát dob, mert nem tudja, hogy melyiket kell
  használnia, ha pedig nincs definiálva akkor azért.
- extern: “Compiler, hidd el, hogy ez valahol márpedig létezik, Linker, te pedig keresd meg a definícióját!”

Deklaráció	Definíció
compiler-nek van rá szüksége	linker-nek van rá szüksége
bevezet egy azonosítót, és megadja a típusát (függvény esetén a visszatérési értékét, és paramétereinek típusát)	- teljesen specifikál egy entitást pl. mit csinál egy függvény - ekkor jön létre a memóriában
prototípus: függvény deklarációja	implementáció: függvény definíciója

Deklaráció	Definíció
`int add(int, int)`	`int add(int a, int b) { return a+b; }`

akárhányszor lehet deklarálni
 	
`double f(int, int);`
`double f(int, int); // megengedett`
	pontosan egyszer lehet és kell definiálni, különben
	linkage errort kapunk

nem definíció	egyben deklaráció is

- függvény prototípus (nincs törzse)
 	
`double func(double);`
- extern szerepelhet előtte, és nincs inicializálva / nincs megadva függvény törzs
 	
`extern double a;`
 	
`extern double func(double);`
-
`typedef ...` // lehet típust definiálni belőle
- statikus adattag osztálydeklaráción belül
- osztálynév deklaráció, ha nem követi definíció
 	
`class A;`
// az utolsó két pontról később tanulunk, nem kell
ezeket tudni, amíg nem ismerkedtünk meg az osztályokkal, csak a teljesség kedvéért szerepelnek
itt
	- `double func(double a) { return a*a; }`
	-
	`extern double a = 10;`
	-
	`int c = 5;`
	-
	`class A {}`

Statikus könyvtárak

- Változók, függvények gyűjteménye, melyeket programjaink felhasználhatnak, ha a statikus library
  hozzájuk van linkelve. A programunk futtatható fájljának része lesz. A program futtatásához a futtatható
  fájlon kívül nincs szükség másra. A használt statikus könyvtár kicseréléséhez az

egész futtatható fájlt
le kell cserélni.

- (Dinamikus könyvtárak esetén a benne található függvények betöltése futási időben történik, a
  könyvtár nem része a futtatható fájlnak. Egyszerűen lecserélhető, a futtatható fájl lecserélése nélkül.
)
- Kiterjesztése: .lib (Windows) vagy .a (Linux)

Létrehozása

Ha nem solution-ön belüli statikus programkönyvtárat szeretnénk használni:
Jobb gomb a projekt nevén -> Properties -> Linker -> Input / Additional Dependencies -> meg kell adni az
előbb generált *.lib fájl nevét (ez általában a projekt neve - a kiterjesztést ne felejtsük el).

Nullterminált string-ek

Karaktertömb, ami az érvényes karakterek után egy null (`0x00`) karaktert tartalmaz. Ez jelzi a string végét. A string méretét úgy tudjuk kiszámolni, hogy a karaktertömb elejétől kezdve megkeressük az első null karaktert. Ebből következik, hogy a string nem tartalmazhat (a végét leszámítva) null karaktert. Fontos, hogy ne felejtsünk el helyet foglalni a terminálo karakternek. Ha pl. egy 50 karakter hosszú tömböt szeretnénk, akkor valójában 51 karakternek kell helyet foglalni a memóriában. További hibalehetőség, ha elfelejtjük kiírni a null karaktert a tömb végére. Ekkor, ha le szeretnénk kérdezni a string hosszát, a null karaktert kereső algoritmus (mivel nem találja a karaktert) kifut a string-ünknek lefoglalt memóriaterületből. Konstans string literálok (idézőjelek között megadott szöveg) típusa explicit jelzés nélkül is nullterminált string. Pl. “Hello World” végén ott van 12. karakterként a ‘`\0`’, de ezt nem kell kiírnunk. Tehát ha egy legalább 12 méretű karaktertömbnek a “Hello World” string-et adjuk értékül, akkor nem kell külön törődnünk a null karakter meglétével.
pl.

```
char hello[20] = "Hello World";
```

Ezzel ellentétben, ha karakterenként adunk értéket egy karaktertömbnek, a végén

nekünk kell gondoskodni arról, hogy a végére odakerüljön a null.
pl.
`char bye[10] = {'B', 'y', 'e', '\0'};`

sizeof() vs strlen()

Először is, a `sizeof()` nem szöveghossz számolásra való, ellentétben a címben említett másik két függvénnyel.

Egyik példa

`char month[] = "october";`

`char month[8] = "october";` // 7+1 null terminátor; teljesen megegyezik az előző sorban lévő definícióval

- `sizeof(month)` értéke 8, hiszen 8 byte-ot foglalunk a memóriában (konkrétan a stack-en) mindkét
  esetben
- `strlen(month)` értéke 7, mert 7 hasznos hosszúságú maga a string (null terminátort nem számolja bele)

Másik példa

`char month[100] = "october"`

- `sizeof(month)` értéke 100
- `strlen(month)` értéke 7

Referencia

A referencia egy alias, tehát egy másik név egy már létező másik változóra.

Tegyük fel, hogy létrehozunk egy x nevű, int típusú változót. Gondoljunk úgy a változó nevére, mint egy felirat, amit a változó memóriaterületéhez kapcsoltunk (‘tessék, ezt a memóriaterületet x-nek fogom hívni’). Amikor létrehozunk erre az x nevű változóra egy y nevű referenciát, akkor a változóhoz tartozó memóriaterületre még egy feliratot teszünk (‘tessék, ezt a memóriaterületet most már y-nak is hívom’). Innentől kezdve ugyanhhoz

a memóraterülethez hozzáférhetünk x-en és y-on keresztül is (és mindketőn keresztül lehet módosítani is, nem csak olvasni).

`int x = 0;`

`int& y = x;` // y egy integer referencia x-re inicializálva

Paraméter átadása függvénynek referenciával

Amennyiben egy változót referenciaként adunk át egy függvénynek, abban a függvényben a referencián keresztül módosítható az eredeti változó értéke. Megvan az az előnyük a pointer-ekkel szemben, hogy paraméterátadáskor nem kell a változó címét képezni, így ezt nem is tudjuk lefelejteni (ami pedig sok fejfájást okozhat, hogy miért nem működik a program). Továbbá nem kell bonyolult pointer-szintaktikákat alkalmazni a függvényen belül (`*p`)

```
void func(int& number){
number++;
}
// a függvény hívása
int num = 0;
func(num);
```

Fontos! Függvény soha ne adjon vissza referenciát lokális változójára! Amikor a függvény visszatér, a lokális változói törlődnek a stack-ből, így egy olyan memóraterületre fog hivatkozni a referencia, ami már nem a programunké.

Pointer vs referencia

	Pointer	Referencia
NULL	lehet	nem lehet
inicializálás	nem kell	kötelező
változtatás	lehet	nem lehet
használat	dereferálható (pl. *pa = 5)	transzparens, nem kell dereferálni,

	Pointer	Referencia
		ugyanúgy használható, mint a változó (pl. ra = 5)
referálható-e	igen (**pa, &pb)	nem (&ra a változó címe)
	Muszáj használni, ha: - tömbről van szó - át kell állítani máshova - lehetséges, hogy nem mutat sehova - C nyelvű függvényt kell hívni C++-ból, ami csak ezt érti	- kívülről olyan, mint egy változó - használatában olyan, mint egy pointer - függvényen belül a változótól használatban nem különbözik (nem memóriacím, hanem maga a változó)

Mikor használjunk referencia/érték szerinti átadást?

- beépített típusú változó (pl. int, double)
  - ha nem akarjuk megváltoztatni az eredeti példányt → érték szerint, mert kicsi, a másolása gyors
  - ha meg akarjuk változtatni az eredeti példányt → referencia
- nem beépített típusú → referencia, mert a másolás lassú
  - ha nem akarjuk megváltoztatni az eredeti példányt → konstans referencia, hogy ténylegesen megakadályozzuk
  - ha meg akarjuk változtatni az eredeti példányt → referencia

Ajánlott irodalom

- http://www.diffen.com/difference/C_vs_C%2B%2B
- http://www.cprogramming.com/tutorial/c-vs-c++.html
- https://msdn.microsoft.com/en-us/library/0kw7hsf3(v=vs.100).aspx
- https://msdn.microsoft.com/en-us/library/xhkhh4zs.aspx
- https://msdn.microsoft.com/en-us/library/3awe4781.aspx
- https://www.visualstudio.com/vs/compare/

- https://www.jetbrains.com/student/
- https://msdn.microsoft.com/en-us/library/ms175759.aspx

2. előadás

C vs C++

Szintaktika

C	C++

`int *x = malloc(sizeof(int));`
`int *x_array = malloc(sizeof(int) * 10);`
`// ...`
`free(x);`
`free(x_array);`
	`int *x = new int;`
	`int *x_array = new int[10];`
	`// ...`
	`delete x;`
	`delete[] x_array;`

`#include <stdio.h>`
`int main() {`
`foo();`
`return 0;`
`}`
`int foo() {`
`printf("Hello world");`
`}`
	`#include <stdio.h>`
	`int foo() {`
	`printf("Hello world");`
	`}`
	`int main() {`
	`// kötelező deklarálni használat előtt`
	`foo();`
	`return 0;`
	`}`

`struct MyStruct {`
`int x;`
`};`
`struct MyStruct MyStructInstance;`
`// ...`
`typedef struct DummyStructName {`
`int y;`
`} MyAmazingStruct;`
 	
`MyAmazingStruct MyAmazingStructName;`
	`struct MyStruct {`
	`int x;`
	`};`
	`MyStruct MyStructInstance;`
	`// ...`
	`typedef struct DummyStructName {`
	`struct MyStruct`
	`MyStructBestInstance;`
	`// struct_name_t instance2;`
	`// invalid!`
	`} MyAmazingStruct;`

`gcc foo.c -lm` `// -lm: Math library`	`g++ foo.cc` `// C++-ban már nem kell külön linkelni egyes libraryket.`
`typedef enum {FALSE, TRUE} bool;`	`// van beépítve bool típus` `bool b; // deklaráció`

C	C++

	`bool b1 = true; // copy inicializáció`
	`bool b2(false); // direkt inicializáció`
	`bool b3 { true }; // uniform inicializáció (C++11)`
	`b1 = false; // értékadásbool b4 = !true; // false`
	`bool b5(!false); // true`

`int main() {`
`printf("Hello, World");`
`return 0;`
`}`
	`int main() {`
	`printf("Hello, World");`
	`}`

`int i;`
`for(i = 1; i< 20; i++)`
`printf("i=%d\n", i);`
	`for(int i = 1; i < 20; i++)`
	`std::cout << "i=" << i << std::endl;`

Egyéb különbségek

	C	C++
Platform	bármi, amire implementálták a compilert	bármi, amire implementálták a compilert
Szöveg típus (bővebben)	nincs, de reprezentálható char tömbként	van, std::string
OOP (Objektumorientált Programozás) támogatás (bővebben)	nincs, de imitálható struktúrákka	van
Egységbezárás (Encapsulation) (bővebben)	nincs	van
Osztályok (class) (bővebben)	nincs	van
Polimorfizmus (bővebben)	nincs	van
Input/Output	input: scanf output: printf	input: std::cin output: std::cout, std::cerr, std::clog (bővebben)
Genericitás (bővebben)	nem	igen
Procedurális (bővebben)	igen	igen

	C	C++
Reflexió (bővebben)	nincs	van
Inline kommentezés	//	//
Blokk kommentezés	/* valami */	/* valami */
Operátor túlterhelés (bővebben)	nincs	van
Utasítás végjelzés	;	;
Névtér (bővebben)	nincs	van
Hibakezelés (bővebben)	nincs	van

(Az ismeretlen fogalmakkal a félév további részeiben fogtok megismerkedni, itt most csak
ismeretterjesztésként szolgál).

Konstans típus

Konstans változót *mindig inicializálni kell.*

```
const int i; // nem OK
const int i = 20; // OK
```

Konstans típusú változó *értéke közvetlenül nem változtatható.*

A következő példa eszerint **hibás**:

```
const int i = 20; // OK
i++; // nem OK
```

Konstansra nem mutathat olyan pointer, *amin keresztül megváltoztathatnánk a változót.*

A pointernél jelölni kell majd, hogy konstans értékre mutat. Egy példa a **helytelen** használatra:

```
const int i = 20;
int* p = &i;
(*p)++;
```

Tömb is lehet konstans, de akkor nem változtatható!

```
const int array[] = {10, 20, 30};
array[2]++; // nem OK!
```

Konstans int típussal lehet tömböt deklarálni, mivel fordítási időben kiértékelődik.

Ezért ez a kódrészlet **jó**:

```
const int i = 20;
char array[i];
```

Konstans pointer

`int* const` egy konstans pointer egy nem konstans `int`-re → a mutatott változó módosítható, a mutató nem

`int const*` vagy `const int*` egy nem konstans pointer egy konstans `int`-re → a mutatott változó nem
módosítható, de a mutató igen.

`const int* const` egy konstans pointer egy konstans `int`-re

```
char s[]="Szia";
const char pc=s; // const char-ra mutató pointer
```

```
pc[0]='s'; // nem OK, konstansra mutat
pc++; // OK, a pointer nem konstans, így változtatható

char s[]="Szia";
char const cp=s; // char-ra mutató konstanspointer
cp[0]='s'; // OK, nem konstansra mutat
cp++; // nem OK, a pointer konstans

char s[]="Szia";
const char* const cpc=s;// const char-ra mutató konstanspoint
er
cpc[0]='s'; // nem OK, konstansra mutat
cpc++; // nem OK, a pointer konstans…
```

Konstans referencia = referencia egy konstansra

Ez amiatt van, mert a referenciát alapból nem lehet megváltoztatni. Referencia esetén a `const` kulcsszó a hivatkozott értékre vonatkozik.

```
int j = 0;
int const &i = j;
i = 1; // nem OK, a hivatkozott érték konstans
const int &k = j;
k = 1; // nem OK, a hivatkozott érték konstans
```

Konstans referencia használata paraméterátadásnál

Ha **nagy objektumot** szeretnénk egy függvénynek paraméterül átadni 3 lehetőségünk van:

- érték szerinti átadás - lassú
- referencia szerinti átadás - gyors, de a függvény megváltoztathajta a hivatkozott objektumot
- konstans referencia szerinti átadás - gyors, a függvény nem módosíthatja az objektumot, ezt

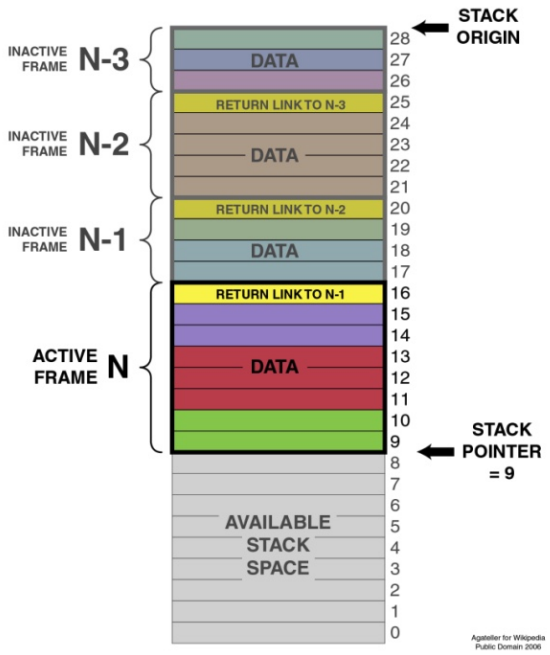
preferáljuk

Memóriatípusok

A statikus/globális memóriaterület

A statikus memóriaterületen helyezkednek el a globális változók, ezek a program egész futása során léteznek. Még a main() függvény végrehajtásának kezdete előtt létrejönnek.

```
int num; // Létrehoz egy integert a globális memóriaterületen
char str[10] = "door"; // Létrehoz egy 10 elemű karaktertömbö
t
char p = "chair";
/* Létrehoz egy névtelen karaktertömböt a globális memóriater
ületen ("chair") és deklarál egy char* típusú pointert, ami e
rre a névtelen karaktertömbre mutat. A pointer csak kezdeti é
rtékként mutat erre  tömbre, átállíthatjuk máshova, de onnant
ól "chair" karaktertömböt nem érjük el*/
int main() {
    num = 10;
    printf("%d", num);
    /*a formátum string is a globális memóriaterületen jön lé
tre, a printf() pedig egy pointert kap erre*/
    return 0;
}
```



A verem/stack

A függvények **lokális változói** kerülnek ide. Tartalma változik, függvényhíváskor a stack tetején létrejönnek annak lokális változói, amikor pedig **visszatérünk a függvényből, a lokális változói megszűnnek**. Minden függvény csak a **saját lokális változóit látja**, a többi függvényét nem. **Rekurzió esetén** (amikor a függvény saját magát hívja meg), egymástól **független lokális változók** keletkeznek a különböző függvénypéldányokhoz, egymását nem tudják elérni. Egy függvény hívásakor annak paraméterei is lemásolódnak a verembe.

A dinamikus memóriaterület: heap

Olyan terület, melyből a program **futása közben egy adott nagyságú részt kérhetünk**, és ha már nincs rá szükségünk, **visszaadhatjuk azt**. Ez teszi lehetővé, hogy akkora méretű memóriát foglaljunk le, melynek **nagyságát** a program **fordításakor még nem ismerjük**. Lefoglaláskor egy pointert kapunk a kért méretű memóriaterületre, ha talált ilyet. Ne felejtsük el, hogy nem biztos, hogy az operációs rendszer tud nekünk memóriát foglalni, ekkor a `malloc()` NULL-lal tér vissza, ezért **mindig ellenőrizni kell** a visszaadott pointer értékét. Amikor felszabadítunk egy memóriaterületet, a pointerünk továbbra is arra a területre fog mutatni (amíg el nem állítjuk onnan), de már tilos arra hivatkozni. Még akkor sem, ha azóta nem foglaltunk új memóriát, mivel akár egy másik program is használhatja azt. Mi történik ha elfelejtünk felszabadítani egy dinamikusan lefoglalt memóriaterületet? A program lefordul, ha nincsen más hiba, jó eredménnyel lefuthat. Viszont a felszabadítatlan memóriaterület a program futásának befejeződése után is felszabadítatlan marad. Ennek eredményeként lecsökken a felhasználható memória mérete, többszöri futtatás után egyre kevesebb és

kevesebb szabad memóriája marad a gépnek. Ezt általában csak akkor vesszük észre, ha lassulni kezd a gép. Ezért ez a hiba nehezen felderíthető, úgyhogy jegyezzük meg, hogy a programban pont annyi free() függvényhívásnak kell szerepelni, ahány malloc()-nak. (“Aki malloc()-ot mond, mondjon free()-t is” /Dr. Czirkos Zoltán/). Vigyázat! Ez alatt nem azt értjük, hogy megszámoljuk, hányszor lett leírva a programban a malloc() és free() utasítás, hanem hogy a futás során hányszor hívódik meg. Pl. ha egy ciklus törzsében foglalunk memóriát malloc()-kal 50-szer, akkor ehhez összesen 50 darab free() hívás fog tartozni, valószínűleg szintén ciklusban. Elágazásnál szintén figyelni kell, hogy a különböző ágakban hány malloc() hívás történik, és a későbbiekben ugyanennyi free() hívás történjen.

Tömb átadása paraméterként

Tömböket paraméterként függvényeknek átadni a kezdőcímükkel lehet. Mivel a függvény a tömbből csak a kezdőcímét látja, nem tudja annak méretét. Próbáljuk meg a sizeof() függvénnyel megtudni a függvényben? Nem fog sikerülni, mivel csak egy pointert kap a függvény, így a sizeof()-fal ennek a pointernek a méretét kapjuk meg, nem a tömbbét. A megoldás, hogy át kell adnunk a tömb méretét is paraméterként.

Miért nem szeretjük a rekurzív függvényeket?

A függvényeket meghívásukkor úgynevezett stack frame-ekbe (verem keret) helyezzük. Az adott függvény
befejezésekor a stack frame megsemmisül. Ez három részből áll:

- a függvény argumentumai
- minden nem-statikus lokális változó a függvényben
- a visszatérési cím, ahova vissza kell ugorni, miután lefutott a függvény

Ez az elv lehetővé teszi a rekurziót. Például, ha van egy ilyenünk:

```
int factorial(int n) {
    if (n <= 1)
        return 1;
    else
```

```
        return factorial(n - 1) * n;
}
```

Aztán meghívjuk így: factorial(3); ilyen szerkezetet kapunk:
---------- ESP
n = 1
RA = <in factorial()>

---

n = 2
RA = <in factorial()>

---

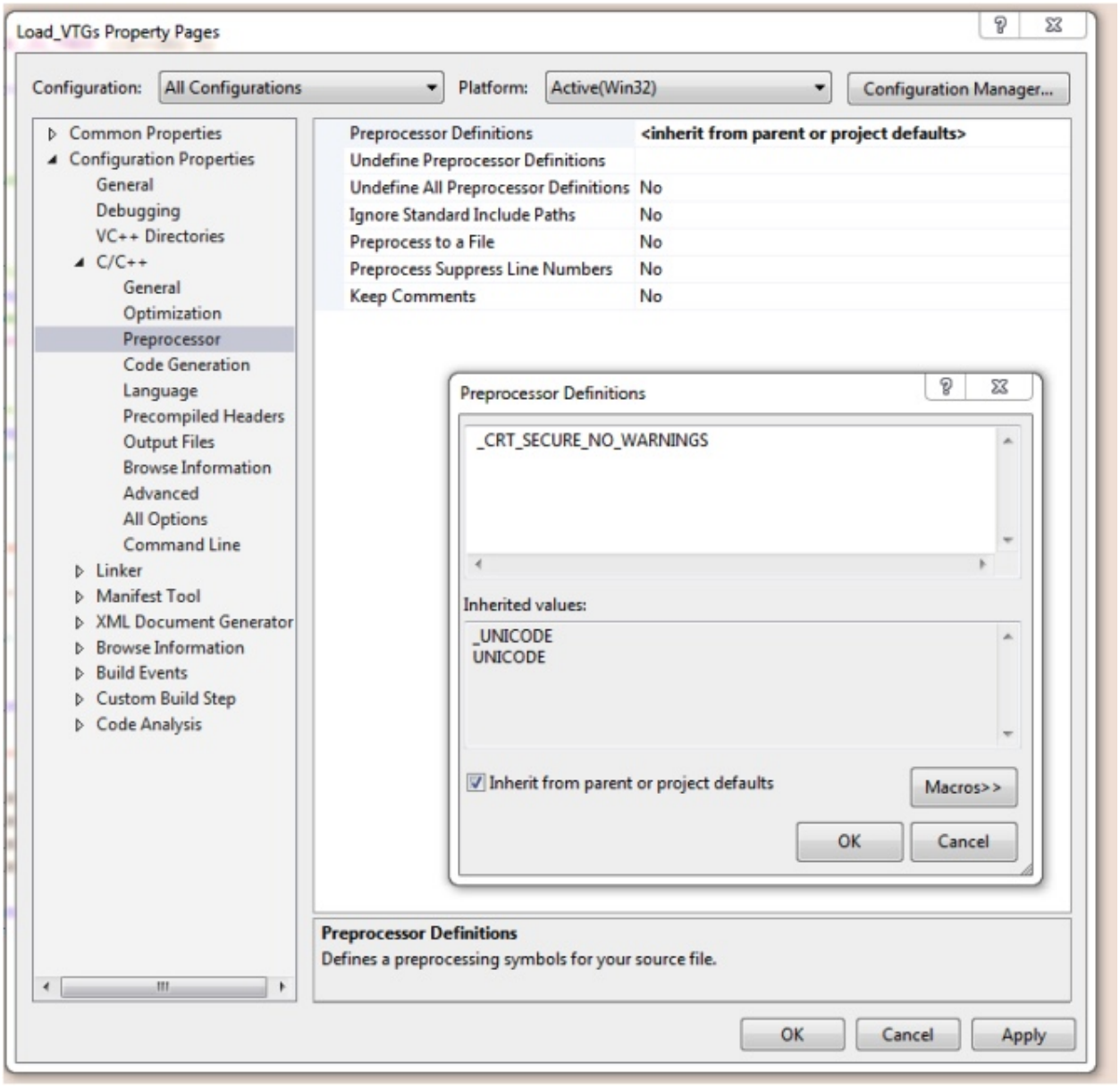
n = 3
RA = <in main()>

A függvény definíciójából látszódik, hogy 3 stack frame-nek kell létrejönnie. Ez ugyan ebben az esetben nem visz el számottevő memóriát, de ha nagyobb számokat adunk meg paraméternek, már jól kivehető, hogy egyáltalán nem helytakarékos, ráadásul még lassú is. Azonban tudjuk, hogy minden rekurzív függvény megírható procedurálisan, ciklusokkal. Lehetőség szerint inkább írjuk meg így a programunkat.

Buffer overflow attack

Ha scanf() függvényt akartok használni a programotokban, a következő warningot fogjátok kapni Visual Studioban: `Error C4996 'scanf': This function or variable may be unsafe. Consider using scanf_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS`. Ezzel az üzenettel jelzi nekünk a fordító, hogy a scanf() nem biztonságos. Na de mit is jelent ez? A scanf()-ben található egy olyan bug amit ha kihasználnak (exploit), akkor a programunk azonnyomban leáll. Ez az úgynevezett buffer overflow attack. A probléma abból fakad, hogy a scanf()-nek senki nem mondja meg, hogy hány bájtot olvasson. Ebből fakadóan addig olvas, amíg azt “hiszi”, hogy olvasnia kell. Egy karakter tömb esetében ez egy null terminátor ‘\0’ vagy ‘\n’. Azonban ha valaki több karaktert akar eltárolni, mint amennyit lefoglalt, a függvény olyan helyre fog írni, ami nem az övé. Ekkor a program segmentation fault hibával leáll. Viszont van egy modernebb - C++11 szabványban létező - függvényünk, a scanf_s (_s = secure), ami lehetővé teszi, hogy

megadjuk hogy hány bájtot akarunk olvasni. Ez a funkció garantálja, hogy több bájtot olvassunk, mint amennyire szükségünk van. Ha viszont valamiért mégis használni szeretnénk a scanf()-et, a következő beállítást kell végrehajtanunk a compileren:



Vagy másik megoldásként kikapcsolhatjuk a warning-ot: `#pragma warning(disable:4996)` (és ez egyéb, nem biztonságosnak minősített műveletek esetén is megtehető, különböző hibakódokkal)

Hogyan kell használni a scanf_s()-t?

```
printf("Enter your name: ");
char name[5]; // OK: "Alex\0", NOT OK: "Alexander\0"
if (scanf_s("%s", name, sizeof(name)) == 1)
    printf("Your name: %s\n", name);
else { // kezeljük a keletkezett szituációt (kivételkezelés a
lapja...)
    printf("Invalid length of data!\n");
    strcpy(name, "????");
    int c;
    while ((c = getchar()) != '\n' && c != EOF); // stdin buf
fer űrítése (flush)
}
```

Miért úgy használjuk a scanf() függvényt, ahogy?

Honnan tudja a scanf(), hogy honnan kell olvasnia?

Ha megnézzük a scanf() definícióját, látni fogjuk, hogy teljesen megegyezik a fscanf() függvénnyel. Az
fscanf()-ről azt kell tudni, hogy bármilyen streamről (folyam) tud olvasni. Minden operációs rendszer
alapértelmezetten hármat szolgáltat:

- `stdin`: standard input; Pl. a leütött billentyűk betűi ennek a bufferén jelennek meg. Innen lehet őket
  kiolvasni.
- `stdout`: standard output; ha a konzolra írunk ki, akkor valójában erre a streamre írunk (#protipp: `printf()`
  =
  `fprintf()`)
- `stderr`: standard error; kimenete a stdout-ra irányul át alapértelmezetten, de ha szeretnénk, akár fájl
  streambe is átirányíthatjuk.

Fájl stream? Igen, ilyen is van, ugyanis az operációs rendszer ugyanolyan adatfolyamként kezeli mint a stdin,
stdout, stderr mester hármast. Ez azért jó, mert egyszerűen tudunk fájlt kezelni (az igazi erejével Java-ban
találkozhatunk, mert ott akár át is irányíthatunk egymásba streameket: ami az egyiknek kimenet, az a
másiknak a bemenete lesz és a kettő közé beékelhetünk valamilyen “filtert” ami mondjuk vagy átenged adatot,
vagy nem valamilyen szempont szerint, vagy akár módosíthat rajtuk. Alkalmazása pl.: real-time titkosítás).

Szóval a lényeg:

- `scanf("%d", &valtozo) = fprintf(stdin, "%d", &valtozo)`
- `printf("Num: %d", n) = fprintf(stdout, "Num: %d", n)`

Mi az a függvénypointer?

Pointerekkel nem csak változókra illetve tömbökre tudunk mutatni, hanem akár függvényekre is. Gondoljunk bele: a függvényeket is a memóriában tároljuk, tehát kell lennie címüknek is. Na de milyen típussal hivatkozzunk rájuk? Esetleg void*? Nos, saját szintaktikájuk van. Tekintsük az alábbi C kódot:

```
#include <stdio.h>
int add(int a, int b) {
    return a + b;
}
main() {
    int (*plus1)(int, int) = add; // jó
    int (*plus2)(int, int) = &add; // jó
    printf("%d, %d", plus1(1, 2), plus2(1, 2)); // 3, 3 a kim
enet
    getchar();
}
```

Azért lehetséges függvénypointert létrehozni, mert a függvények is a memóriában tárolódnak, így van memóriacímük is, amire tudunk hivatkozni.

> 💡 Szintaktika: `visszatérési_típus(*fv_pointer)(paraméterek típusa);`

És ez miért jó?

TFH írunk egy rendező függvényt, ami alapértelmezetten tud növekvő sorrendbe rendezni, de azt szeretnénk, hogy ha úgy kívánja a helyzet, akkor tudjon csökkenőbe is, szóval a függvényünknek adunk egy bool (0/1) paramétert hogy ha az igaz (1), akkor növekvőbe, különben meg csökkenőbe rendezzen.

Aztán ahogy telik az idő, igényt formálunk arra, hogy ne csak ebben a kétféle módban rendezzen, hanem bonyolultabb algoritmus szerint. Szóval azt csináljuk, hogy 0/1 érték helyett egy saját kiértékelő függvényt (predikátumot) adunk át paraméternek.

Függvénynév túlterhelés

Lehetővé teszi, hogy ugyanahhoz a függvénynévhez több definíciót hozzunk létre. Az ugyanolyan nevű
függvényeknek deklarációban különbözniük kell, tehát a paramétereik száma vagy azok típusa más. Viszont
nem számít túlterhelésnek, és a fordító el sem fogadja, ha két függvény csak és kizárólag a visszatérési
értékük típusában tér el egymástól.

Mire jó ez? Tegyük fel, hogy írtunk egy olyan függvényt, mely összead két double típusú számot. Legyen ez a

`double add(double a, double b)` nevű és paraméterezésű függvényünk. Aztán később szükségünk lenne egy
olyan függvényre, ami int típusú számokat ad össze. Adhatnánk a függvényünknek más nevet is, de az “add”
elnevezés egyszerű és elegendően kifejező, ezért egyszerűen írhatunk egy `int add(int a, int b)` függvényt.
Ebben az esetben, amikor meghívjuk a programunkban az “add” függvényt, a fordító

kikeresi, hogy melyik
függvényt is kell most meghívni a megadott paraméterektől függően. Ha két `double` értékkel hívjuk meg, akkor az első függvény hívódik meg, ha két `int` értékkel hívjuk meg, akkor pedig a második. Tegyük fel, hogy több “add” nevű függvényt nem írtunk. *Ekkor, ha egy double és egy int paraméterrel hívjuk meg, hibát kapunk, mivel nem létezik “add” függvény ilyen típusú paraméterekkel.*

Parancssori paraméterek

C++-ban lehetőség van a main függvényben parancssori argumentumok fogadására a következő paraméterezéssel:

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

- `argc`: A kapott paraméterek száma.
- `argv[]`: A kapott paraméterek karaktertömbként

A legelső paraméter mindig a futtatható fájl elérési útvonala és neve.

A `main()` függvényt megírhatjuk paraméterek átvétele nélkül is, amit a függvénytúlterhelés (ld. fentebb) tesz lehetővé.

Inline függvények

Az `inline` függvények abban különböznek a hagyományos függvényektől, hogy a fordító minden hívás helyére
bemásolja a függvény törzsét, ha a függvény kicsi. Az “
`inline`” megjelölés valójában egy ajánlás a fordítónak,
hogy tegye meg ezeket a lépéseket, azonban dönthet úgy, hogy ezt figyelmen kívül hagyja. Hogy megértsük,
mikor hogy dönt, nézzük meg a hátterét:

Az egész elképzelés mögött az áll, hogy ha van egy rövid függvény (pl. Két szám összeadása vagy kivonása),
és azt nagyon sok helyen hívjuk, akkor ez sok felesleges idővel jár. Minden függvényhíváskor például kezelni
kell a stack-et. Ekkor sok időt spórolunk, ha függvényhívás helyett a kódba másoljuk a függvény törzsét. Pont

ez történik
`inline` függvények esetén.

A sok bemásolgatással viszont nő a kód mérete. Amikor a compiler úgy véli, hogy túl hosszú az `inline`-nak
megjelölt függvény törzse és így nagyon megnőne a kód mérete, egyszerűen nem végzi el a másolásokat,
hanem marad hagyományos függvény.

Alapértelmezett függvényparaméterek

- Egyszer lehet definiálni. Még akkor sem lehet többször (pl. külön a header és *.c vagy *.cpp fájlban),
  ha ugyanazt az értéket adnánk meg.
  - `int magicFunction(int a, int b);` // header-ben
  - `int magicFunction(int a, int b = 0) {...}` // *.c vagy *.cpp fájlban
- Lehetővé teszik, hogy egy függvényt minden paraméterének megadása nélkül meghívjuk. Ilyen
  esetben a nem megadott paraméterek értékét helyettesítik a definiált értékkel.
- A paraméterlista vége felől folytonosnak kell lenniük.

`void add(int a, int b = 3, int c, int d = 4);` // nem OK

`void add(int a, int b = 3, int c, int d);` // nem OK

`void add(int a, int b = 3, int c = 2, int d = 4);` // OK

Hogyan működik a tömbindexelés?

```
int tomb[10];
int *p = tomb; // a p pointert a tömb elejére állítjuk (0. el
emre)
```

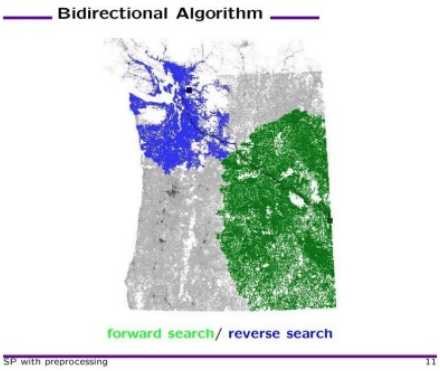
```
int i = 2;
*(tomb+i) = 3;
tomb[i] = 3;
*(p+i) = 3; // p+i az i. elemre mutató pointer, *(p+i) pedig
maga az i. elem
p[i] = 3;
```

Tömbök elemének elérésekor két művelet hajtódik végre (mindig): az elem címének kiszámítása, majd az
elem elérése (indirekcióval). Ezt a kétlépcsős elérést jobban szemlélteti a `*(tomb+i) = 3;` és `(p+i) = 3;` *sor a fenti kódban. A tömb nevére hivatkozva a fordító megadja, hogy hol található a tömb, egy pointert kapunk rá, melyen az indexelő [] operátort alkalmazva címszámítást és dereferálást (*) is végzünk. A *(tomb+i) ugyanazt* jelenti, mint a t[i]. Az indexelő operátor [] csak egy rövidítés, az emberi gondolkodást könnyíti: könnyebben
olvasható forma, ezért általában ezt használjuk.

Vermes labor feladat problémái

Tegyük fel, hogy írtunk N db C függvényt, ami mind-mind ahhoz kell, hogy stack (verem, LIFO) struktúrákat
tudjunk kezelni (push, pop, …). Írni szeretnénk M db queue-t (sor, FIFO) kezelő függvényt is (push, pop).

Első probléma: milyen névvel illessük az új függvényeket? Push, pop már van, szóval prefixet illesztünk a
nevük elé, hogy ne legyen névütközés: stack_push, stack_pop, queue_push, queue_pop. Az analógiát
követve nyilvánvalóan belátható, hogy egy komolyabb méreteket öltő szoftver alkalmazásnál ez a módszer
nem alkalmazható.



Második probléma: összefüggő, komplex adatstruktúrát nem tudunk modellezni. Például: ismert probléma, hogy Dijkstra útkereső algoritmusának időbeli komplexitása már elég alacsony ahhoz, hogy implementálásával egy “jó” rendszert lehessen építeni, de nem elég jó például egy Google Mapshez ahol rengeteg csúcs és él van, így még mindig lassúnak számít. Ezért mérnökök (a matematikusok általában az előző lépésnél már megálltak), el kezdtek kísérletezni. Az egyik próbálkozás az volt, hogy ha ismert a start és a cél, akkor indítsák mindkét irányból a Dijkstrat és ahol legelőször találkoznak, összekötik a két szakaszt és megkapják a legkisebb költségű utat. Remélem ezután mindenki kapisgálja, hogy itt bizony két azonos adatstruktúra van a háttérben, azonban különböző értékekkel. Ilyenkor természetes késztetést érzünk arra, hogy az ugyanazon célt szolgáló változókat, tulajdonságokat, metódusokat egységbezárjuk (encapsulation). Ezt az új adatmodellt (adatok és kapcsolataik + a rajtuk végezhető műveletek definiálását) osztályoknak hívják. Ha konkretizálni szeretnénk az osztályt és az egyes változóknak értéket szeretnénk adni, példányosítjuk egy (vagy több) objektumba.

Harmadik probléma: nem tudunk adatot elrejteni. Tegyük fel, hogy többen dolgoznak egy szoftveren (értsd:
mindig), nem teljesen egyértelmű, hogy melyik adatot kell és hogyan módosítani. Például ha lemodelleztünk
egy személyt (Person osztály), akkor nem szeretnénk ha valaki más módosítaná a quentin nevű Person típusú
objektumunk email címét asdqwerew-re, hiszen ez nem egy értelmes adat. Előtte validálnunk kell, hogy
tényleg megfelel-e az elvárt email formátumnak. A megoldás, hogy a Person osztályon

kívüli világ elől
elrejtünk minden változót és tagfüggvényeken keresztül módosítjuk (setEmail).

Ajánlott irodalom

- https://en.wikipedia.org/wiki/Standard_streams
- http://stackoverflow.com/questions/23378636/string-input-using-c-scanf-s
- http://stackoverflow.com/questions/21434735/difference-between-scanf-and-scanf-s
- http://www.eet.bme.hu/~czirkos/cpp.php
- http://www.eet.bme.hu/~czirkos/cpp_iras/memoria.pdf
- http://stackoverflow.com/questions/840501/how-do-function-pointers-in-c-work
- http://stackoverflow.com/questions/7410296/why-is-bounds-checking-not-implemented-in-some-of-thelanguages
- https://en.wikipedia.org/wiki/Function_pointer
- http://c.learncodethehardway.org/book/ex18.htm
- http://stackoverflow.com/questions/654754/what-really-happens-when-you-dont-free-after-malloc
- http://stackoverflow.com/questions/31372892/running-malloc-in-an-infinite-loop
- https://www.programiz.com/cpp-programming/default-argument

3. előadás

Mik azok az osztályok és objektumok?

Osztály (class): egy új típus C++-ban

- függvények és tulajdonságok gyűjteménye

- a tagfüggvények viselkedést definiálnak
- a tagváltozók állapotot írnak le
  - tagváltozó típusa lehet egy másik osztály → tartalmazás kapcsolat
    - pl. számítógépben van CPU, RAM → Computer osztályban lesz egy Cpu és egy vagy több Ram típusú tagváltozó
    - pl. a szobának része az ajtó, ablak stb.
- egyszer kell megírni

Objektum (object): egy osztály egy példánya

- osztály definiálja a struktúrát
- különböző objektumok tagváltozóinak értékei különbözők
- annyiszor van létrehozva, ahányszor szeretnénk (memória azért gátat szab ennek, de nagyon sok lehet belőle)

Class vs Struct

	Class	Struct
Használható C-ben?	nem	igen
Használható C++-ban?	igen	igen
Alapértelmezett láthatóság	private	public

Objektum-orientált tervezési elvek 1.

- **Absztrakció (abstraction)**
  - a feladat szempontjából lényegtelen részleteket figyelmen kívül hagyjuk
    - pl. alkalmazottak nyilvántartásakor nem szükséges tudni (általában) a személyek kisujjainak hosszát
  - a világban létező dolgok leképezhetők a program objektumaira
- Osztályozás (classification)

    - az ugyanolyan viselkedéssel és tulajdonságokkal rendelkező dolgokat csoportosítjuk
    - ezeknek a viselkedése és tulajdonságai ugyanazzal az osztállyal írhatók le
- Beágyazás (encapsulation)
    - az összetartozó adatokat és a rajtuk értelmezett műveleteket egy osztályba helyezzük, így együtt egy egységet alkotnak
- Adatrejtés (information/data hiding)
    - egy osztály nem enged közvetlen hozzáférést az tagváltozóihoz
    - csak függvényeken keresztül érhetők el
    - minden tagváltozónak privátnak kell lennie így elkerülhető az objektum inkonzisztens állapotba kerülése (csak olyan változtatást hajtunk végre a függvényben, ami engedélyezett és értelmes)

Konstruktor

Az osztályok speciális tagfüggvénye, ami akkor hívódik meg, amikor az osztályból példányosítunk egy új objektumot. Neve megegyezik az osztály nevével és nincs semmilyen visszatérési értéke (még void sem). Általánosan az objektum kezdőértékeinek beállítására használható.

Egy osztály konstruktora túlterhelhető, többféle paraméterezésűt létre lehet hozni (pl. ha a Rectangle osztályt két paraméterrel hívjuk, akkor téglalapként viselkedik, ha csak eggyel, akkor a magassága és szélessége is a megadott paraméterrel lesz egyenlő, és négyzetet reprezentál). Alapértelmezett értékek beállíthatók a paramétereinek a korábban tanult szabályok alapján.

Explicit nem lehet meghívni, az objektum létrehozásakor automatikusan hívódik meg a megfelelő paraméterezésű konstruktor, és egyszer fut le.

Alapértelmezett (default) konstruktor

Ha a konstruktornak nincs egy paramétere sem, **default konstruktor**-nak nevezzük.

Ha mi magunk nem írunk egyetlen konstruktort sem, alapértelmezetten lesz egy default konstruktora az osztályunknak. Amint írunk bármilyen másik konstruktor (például egy

másoló konstruktort), de default konstruktort nem, akkor nem fog létrejönni automatikusan default konstruktor.

Megjegyzendő, hogyha nem definiálunk külön default konstruktort, de van olyan konstruktor, amelyben minden paraméternek van default értéke (tehát meghívható akár paraméter nélkül), akkor nem kell külön default konstruktort írni.

Másoló konstruktor

Paraméterként olyan típusú objektumot vesz át, mint amelyik osztály konstruktoráról van szó. Ekkor a paraméterül kapott objektum tagváltozóinak értékét le kell másolni az új objektum tagváltozóiba. Fontos, hogy mindenhol **deep copy**-t kell készíteni. Ez azt jelenti, hogy ha van pl. egy pointer tagváltozónk, akkor az új objektumban nem egyszerűen létrehozunk egy pointert, amivel rámutatunk a már meglévő memóriaterületre. A helyes megoldás, ha lefoglalunk egy ugyanakkora memóriaterületet, mint amekkora az eredetiben van, és minden értéket átmásolunk az új memóriaterületre is. Az új objektumban lévő mutató erre az új memóriaterületre fog mutatni.

Destruktor

A destruktor egy olyan speciális tagfüggvény, ami akkor hívódik meg automatikusan, amikor az adott objektum életciklusa végetér:

- *delete*-t hívnak rajta
- program vagy scope (pl. függvény) terminál
- kezeletlen exception (kivétel) keletkezik
- közvetlenül is meghívható, de ilyet nem szoktak (helyette delete)

Nem írunk destruktort, a compiler mindig készít egyet `inline public` láthatósággal.

Ha az

- osztályon belül egy tagváltozónak `new`-val foglaltok típusos memóriaterületet,
- fájlt vagy hálózati kapcsolatot nyittok meg (ezek drága erőforrások, hiszen más processek várakozhatnak rájuk → ha lehetőség van rá, még a destruktor lefutása előtt fel kell szabadítani őket!)

akkor a **destruktorban mindig fel kell szabadítani**. A fentebbi bekezdésben említett, a compiler által automatikusan létrehozott destruktor ezt nem végzi el, mindig nekünk kell megírnunk.

Láthatóság

Az osztályok tagváltozói és tagfüggvényei rendelkeznek ún. láthatósággal. Ez azt adja meg, hogy az adott tagváltozót ki érheti el közvetlenül és az adott tagfüggvényt ki hívhatja meg. A két alapvető láthatóság a következő:

public

Mindenhonnan hozzáférhető a programban közvetlenül, osztályon kívüli függvényből is. Ez azt jelenti, hogy nem kell függvényt hívni ahhoz, hogy megkapjuk vagy megváltoztassuk az értékét egy tagváltozónak.

private

Osztályon kívülről nem érhető el, nem is látható. Ez az alapértelmezett láthatóság az osztályokban. Az osztály függvényeiből viszont elérhető.

- `getter`: Olyan metódus, amely visszaadja egy privát láthatóságú tagváltozó értékét. A metódus maga publikus. Elnevezése “get” prefix-szel kezdődik, és a tagváltozó nevével folytatódik. pl. privát “age” tagváltozó getter metódusának fejléce: `int getAge()`
- `setter`: Olyan metódus, amely megváltoztatja egy privát láthatóságú tagváltozó értékét. A metódus maga publikus. Elnevezése “set” prefix-szel kezdődik, és a tagváltozó nevével folytatódik. Ebben a függvényben lehetőségünk van a paraméterül átvett érték ellenőrzésére. Ha nem megfelelő értéket adott meg a hívó, nem végezzük el a módosítást. Így biztosítható pl. hogy az e-mail címnek e-mail cím formátuma legyen.

  pl. privát “age” tagváltozó setter metódusának fejléce: `void setAge(int newAge)` → ekkor, ha a “newAge” értéke negatív, nem végezzük el a változtatást, mert az nem érvényes kor.

Léteznek más láthatóságok is, melyekről a későbbiekben esik majd szó.

A *this* pointer

Osztályok tagfüggvényein belül érhető el, 0. paraméterként automatikusan megkapja. A tagfüggvényen belül arra az objektumra mutat, melyen a függvényt meghívták. Ezen keresztül elérhetők az aktuális objektum tagváltozói és tagfüggvényei (privát láthatóságúak is).

C++ elnevezési konvenciók

- **típus**: UpperCamelCase
  - szavak között nincs elválasztó karakter
  - az minden kezdőbetű nagybetű
  - főnév
  - pl. BookStore
- **függvény**: lowerCamelCase
  - ige
  - szavak között nincs elválasztó karakter
  - első betű kisbetű
  - minden más kezdőbetű nagybetű
  - pl. getOldestBook()
- **változó/tagváltozó**: lowerCamelCase (ld. függvény elnevezése)
  - pl. firstName
- **konstans**: UPPER_CASE
  - csupa nagybetű
  - szavak aláhúzással elválasztva
  - pl. MAX_TABLE_SIZE

Clean Code 1. - Általános

- **Miért tartsuk be** ezeket és a későbbiekben bemutatott elveket (azon felül, hogy a házi feladatban elvárt ezeknek az elveknek a betartása)?
    - könnyebb, gyorsabb megérteni a programot (készítő és más személy által is), így **időt takarítasz meg** magadnak és megkönnyíted a saját életed (a befektetett odafigyelés 10-szeresen térül meg a későbbiekben)
    - **könnyebb változtatni** valamit a programon (és egy program soha sincs kész, mindig változik)
    - **könnyebb** lesz **kijavítani a bug-okat**
- Nagyon hosszú sorokat **tördeld** (max. 80 karakter legyen egy sor)
- Használj **behúzásokat**, ne minden egy oszlopban kezdődjön
- Használj **értelmes nevek**et, amik felfedik a változó/függvény/osztály célját
    - kereshető
    - öndokumentáló
- **Különböző célokra különböző változók**at használj
- Don’t Repeat Yourself (**DRY**): Ne ismételd magad!
    - a duplikáció minden probléma gyökere
    - ha valamin változtatni kell később, mindenhol változtatni kell (pl. bug javítás)
    - valószínű, hogy valahol elfelejted megváltoztatni, és nehéz lesz kideríteni, miért nem működik mégsem a kód

Clean Code 2. - Függvények

- Egy függvény **max 20 sor** legyen
    - különben bontsd fel több függvényre
- **Egy függvény egy felelősség**et valósít meg (ha több szekcióra osztható, akkor valószínűleg több dolgot csinál)
- Függvényeknek **max 3 paramétere** legyen (különben nehéz tesztelni minden kombinációt)

- **Logikai értéket átadni függvénynek csúnya**, valószínűleg több dolgot csinál a függvény, ezért kerüljük messziről
- Függvényeknek **ne legyen váratlan mellékhatás**a ○ pl. tömb elemeit kiíró függvény ne változtassa meg a tömb elemeit
- Egy függvény **vagy változtat valamin vagy információt szolgáltat** róla, egyszerre a kettőt nem csinálja
- Védd magad: **ellenőrizd** a bemeneti paramétereket, ne feltételezd, hogy jók

Clean Code 3. - Kommentek

Milyet használjunk?

- I**nformatív** komment: bonyolult kifejezés magyarázata
- **Szándék** magyarázata: miért csinálja a kód azt, amit ○ főleg ha érdekes, furcsa megoldásról van szó vagy bugfix
- **Figyelmeztetés** a következményekről ○ pl. sok ideig tart a futtatás, készüljön fel rá, aki használja
- **Pontosítás**: bonyolult szintaxis, mit jelent egy bizonyos visszatérési érték ○ pl. compare függvény visszatérési értékei jelentésének magyarázata -1, 0, 1
- **TODO** komment: később még át kell írni, vagy még nincs kész az, amit használna
- Nem egyértelmű rész **fontosságának** jelölése ○ pl. bináris keresés előtt muszáj rendezni az adatokat
- Függvény paramétereinek elvárt **értéktartománya**, visszaadott értékek tartománya, milyen hiba léphet fel
- Mik az **alapértelmezett értékek**

Milyet NE használjunk?

- **Redundáns** komment, ami könnyen kiolvasható a kódból ○ for(i = 0; i < 10; i++){} // ciklus 0-tól 9-ig

- Komment jól elnevezett változó/függvény helyett ○ int asd = 12; // hónapok száma egy évben HELYETT int monthsInYear = 12;
- **Kódrészlet** kikommentezve
- **Túl hosszú** komment
- **Nem teljes** komment: egyes részek meg vannak magyarázva, mások nincsenek

Clean Code 4. - Osztályok

- **adatrejtés** elve miatt minden **tagváltozó** legyen **private**, amihez biztosíts getter metódusokat, ha szükséges, akkor setter-t is
- egy osztálynak **egy felelőssége** legyen
- bővítés során régi kód ne változzon
- **ne** használj olyan osztályokat, amiknek csak **tagváltozóik** és **getter/setter** metódusaik vannak
  - nincs viselkedése
- a **kapcsolódó tagváltozók és viselkedés** egy osztályban legyen ű**kevés publikus függvénye** legyen az osztálynak
  - különben nehéz megtalálni azt, amire szükség van
  - ugyanannak a dolognak az elvégzése csak egyféleképp legyen végrehajtható
- implementálj **standard függvényeket**
  - egyenlőség-vizsgálat
  - kiírás
  - deep copy

4. előadás

Dinamikus memóriakezelés

Az osztályok bevezetésével egy komoly problémával szembesülünk. A malloc-cal történő memóriafoglalás nem hívja meg az osztály konstruktorát.

Ez nekünk miért baj? A memóriafoglalás után külön be kell állítanunk minden tagváltozó értékét külön-külön. Lehet akár nagyon sok tagváltozónk is, amiknek a beállítgatása rengeteg felesleges művelettel jár, és ezt minden egyes objektumnál meg kéne csinálnunk. Na nem! Pont erre lehetne használni a konstruktort, hogy csak átadogatjuk neki a paramétereket, és minden tagváltozónak kezdőértéket adjon.

A megoldás erre a `malloc`-`free` páros helyett két új művelet: a `new` és a `delete`. Szintaxisuk:

```
int *x = new int;
int *x_array = new int[10];
```

```
delete x;
delete[] x_array;
```

Figyelem, `new`-val foglalt memóriaterületet mindig `delete`-tel szabadítunk fel, `new[]`-val foglalt memóriaterületet pedig mindig `delete[]`-tel! Ha ettől eltérünk, a program viselkedése nem definiált, bármi történhet (felrobbanni azért nem fog a géped, de érdekes hibák jöhetnek elő).

Érdemes kiemelni: C++ programon belül sose használjátok a `malloc`-`free` párost, helyette `new`-`delete`.
Mindezek mellett, a
`new` típusbiztos, miközben a `malloc` egyáltalán nem az.

Összefoglalás: new/delete vs malloc/free

Tulajdonság	new/delete	malloc/free
Honnan foglal memóriát?	Free Store	Heap
Visszatérési érték	teljesen típusos pointer	void*
Hiba esetén mit csinál?	hibát dob (ld. később: std::bad_alloc); sose ad vissza NULL-t	NULL-t ad vissza

Tulajdonság	new/delete	malloc/free
Honnan tudja a szükséges méretet?	compiler számolja ki	byte-okban kell megadni (sizeof(típus)*elemszám)
Hogyan kezel tömböket?	közvetlenül megadható tömbméret (pl. new int[10])	manuális számítást igényel (ld. előző pont)
Átméretezés (reallocating)	nem beépített (nincs is rá szükség)	reallocate() függvénnyel megoldható, de nem hív copy konstruktort
Túlterhelhető? (ld. köv. labor: viselkedés újradefiniálható)	igen	nem
Meghívja a konstruktort, destruktort?	igen	nem

A dinamikus memóriaterület egy másik fajtája: Free Store

Egyike a két dinamikus memóriafoglalást biztosító memóriaterületeknek. Különlegessége, hogy az objektum életciklus ideje kevesebb is lehet, mint amennyi eltelik a `new` és a `delete` között. A többi időben (tehát amikor nem él az objektum, de a memória le van foglalva) `void*`-on keresztül ugyan lehet módosítani a területet, de az objektum nem-statikus változóin, metódusain keresztül nem.

Dinamikus tagváltozó

Egy osztály rendelkezhet dinamikusan foglalt tagváltozókkal. Ezeknek memóriát általában a konstruktorban foglalunk (ha már tudjuk, mekkora memóriaterület szükséges - pl. default konstruktorban nem adott meg semmit a felhasználó). A memória felszabadítását mindig a destruktorban végezzük el.

Ha egy osztályban van dinamikusan kezelendő tagváltozó, mindig írjunk másoló konstruktort.

Másoló konstruktor - Miért is deep copy?

Tegyük fel, hogy van egy “MyString” osztályunk, amiben dinamikusan tárolunk egy karaktertömböt. Legyen egy példány belőle a “magic” nevű MyString, aminek a dinamikus tömbjében “my magic” szerepel.

Hozzunk létre másoló konstruktorral egy “bigMagic” nevű MyString típusú objektumot. Most ehhez ne deep copy-t használjunk, hanem a “bigMagic” pointer-ét állítsuk át a “my magic” tömb kezdőcímére.

Nyugodtan használhatjuk így őket minden gond nélkül egy darabig. Konkrétan addig, amíg meg nem szűnik pl. a “magic” nevű objektumunk, mivel ilyenkor fel kell szabadítani a dinamikusan foglalt memóriaterületet is, hogy ne legyen memóriaszivárgás. Ekkor viszont a “bigMagic” objektum már olyan memóriaterületre fog hivatkozni, ami nem tartozik a programunkhoz, bárkinek kioszthatja az operációs rendszer. Ez ugyebár tilos.

Na de akkor mit lehetne tenni? Ne szabadítsa fel a “magic” nevű objektum a dinamikus memóriaterületet? Akkor ki fogja? Annak kéne, aki utoljára használja, mivel akár 10 objektum is másolhatta volna. De honnan tudjuk, hogy ki az utolsó?

Na ezeket a problémákat elkerülendő használunk deep copy-t. Amikor a “bigMagic” objektumunk másoló konstruktora meghívódik, lefoglal egy teljesen új memóriaterületet, és átmásolja bele a “my magic” karaktertömböt. Innentől kezdve a két objektum által használt tömbök teljesen függetlenek. Ha az egyik objektum megszűnik, a másik a saját tömbjén nyugodtan dolgozhat, ahogy az elvárt.

Másoló konstruktor paramétere miért mindig konstansreferencia?

Amikor paraméterül egyszerűen adunk át egy objektumot egy függvénynek (nem pointerként és nem referenciaként), az objektumról másolat készül, és azt kapja meg a függvény.

Ha a másoló konstruktor nem referenciaként kapná meg a paramétert, akkor ahhoz meg kellene hívni a paraméter objektum másoló konstruktorát. Ami ugyanez a függvény, tehát megint csak nem referenciaként veszi át a paramétert, ezt is le kell másolni. Végtelen másoló konstruktort hívó rekurzióba futottunk, ami elég kellemetlen.

`MyClass(MyClass otherObject) {}` → paraméter átvételéhez kellene a másoló konstruktor, aminek a paraméterének az átvételéhez kellene a másoló konstruktor, aminek a

paraméterének az átvételéhez …

Emellett azért konstans, mert azt az objektumot, amit lemásolunk, nem szeretnénk még véletlenül sem módosítani.

Ajánlott irodalom

- http://stackoverflow.com/questions/240212/what-is-the-difference-between-new-delete-and-malloc-free

5. előadás

Konstans tagváltozók, tagfüggvények

Osztályok **konstans tagváltozóit** kezdőérték-adás után nem lehet megváltoztatni. Továbbá olyan osztályra, mely rendelkezik konstans tagváltozóval, nem alkalmazható értékadás operátor, viszont copy konstruktor igen (létrehozáskor lehet kezdőértéket adni neki, és soha többé).

A **konstans taggfüggvények** garantálják, hogy nem fogják megváltoztatni, illetve nem hívnak meg olyan függvényt, mely megváltoztathatja az **objektum állapotát** (tehát nem változtatja az objektum tagváltozóinak értékét). Pl. getter függvények konstansok lehetnek (és legyenek is mindig), mivel csak visszaadják egy tagváltozó értékét, nem módosítják azt. Ahhoz, hogy egy tagfüggvény konstans legyen a “const” kulcsszót kell a paraméterlista és függvénytörzs közé helyezni. Ezt mind a függvény deklarációjánál, mind a definíciójánál jelölni kell (a header és .cpp fájlban is).

Ha egy konstans függvényben olyan utasítás szerepel, mely módosítja az objektum állapotát, vagy nem konstans függvényt hív, **fordítás idejű hiba** keletkezik.

Konstruktor soha nem lehet konstans, mivel inicializálja a tagváltozókat, tehát változtatja őket, és ezt nem tilthatjuk meg neki.

Konstans objektumon csak konstans tagfüggvény hívható.

Statikus tagváltozók, tagfüggvények

A statikus tagváltozó olyan tagváltozó, mely nem egy objektum sajátja, hanem az egész osztályra közösen vonatkozik. Tehát nem az objektum tulajdonsága, hanem az osztályé. Értéke minden objektum számára ugyanaz.

Egy osztály statikus tagváltozója akkor is létezik és elérhető, ha az adott osztályból még egyetlen objektum sem lett példányosítva.

Statikus taggfüggvények az osztályhoz tartoznak, nem kapnak 0. paraméterként this pointert. A statikus tagváltozókon dolgoznak, nem érik el egy objektum saját, nem statikus tagváltozóit és nem hívhatnak nem statikus függvényeket.

Elérhetőek egy példányon keresztül vagy az osztály nevén keresztül.

Mire lehet ezt használni például?

Egyedi objektum azonosító

Gyakori feladat, hogy az objektumokat egyedi azonosítóval szeretnénk ellátni. Ezt meg lehetne úgy oldani, hogy az objektum konstruktorában átadjuk neki paraméterül. Ekkor viszont semmi sem garantálja, hogy az objektumok azonosítói egyediek lesznek, a létrehozót nem tudjuk kötelezni arra, hogy létrehozás előtt ellenőrizze az egyediséget.

Random szám generálás sem tökéletes megoldás, mivel előfordulhat ismétlődés. Ráadásul nem tudjuk előre, hogy hány objektum lesz példányosítva, így nehéz megbecsülni, hogy mekkora intervallumon kéne random számot generálni.

A legjobb (és egyetlen igazán elfogadható megoldás), ha az osztály rendelkezik egy 0 kezdőértékű statikus változóval, és egy ugyanolyan típusú egyedi azonosítót tároló tagváltozóval (jó, ha konstans). Amikor egy új objektum példányosodik, a konstruktorában az egyedi azonosítójának a statikus változó értékét adjuk értékül, és a statikus változó értékét növeljük eggyel.

Figyelem! Másoló konstruktorban az egyedi azonosítót nem másoljuk, hanem a statikus változóból vesszük az értéket, majd növeljük, mint más konstruktornál. Hiába másoló konstruktor, az egyedi azonosítót nem másolhatjuk.

Meyers’ singleton pattern (tervezési minta)

Tegyük fel, hogy egy adott objektumot kell használnunk az alkalmazás egész életciklusa alatt. C++-ban lehetőségünk lenne egy globális példányt létrehozni, amit a programban bárhol használhatnánk. Azonban egy jó objektum orientált design esetén nem engedhető meg globális változó vagy függvény, mivel ellentmondásban lenne a beágyazás (encapsulation) és az adatrejtés (data hiding) elvekkel. A másik probléma, hogyha át szeretnénk állni egy másik programozási nyelvre, például C#-ra, ahol már minden objektumnak számít (még egy egyszerű int is), ott már nincs is lehetőség globális változók létrehozására. Ezért találták ki a Singleton tervezési mintát (pattern), amit minden OOP-t támogató nyelvben meg lehet valósítani (bár ennek is több változata létezik). Egyébként az ilyen univerziális praktikákból nagyon sok van (akit érdekelne bővebben: http://www.oodesign.com/).
Példák alkalmazásra:

- Logger singleton osztály
  - különböző logolási szinteket szeretnénk megvalósítani
  - grafikusan és konzolosan is szeretnénk logolni
    - nem elégszünk meg a printf-fel, cout-tal és társaival
  - ha nem lenne, nagyon sok overheaddel járna
    - mindig copy-pastelni kellene X sor kódot csak azért, hogy tudjunk logolni egy adott helyen
- több Logger osztálynak nem lenne értelme
- PrintSpooler singleton osztály

  nyomtatási sor kezelő
  - erőforrás hozzáférési konkurrencia kezelés
    - két processz/szál szeretne egyszerre nyomtatni
    - versenyhelyzet alakul ki köztük
    - arbitrációs folyamat: annak eldöntése, hogy ki nyomtathasson először

Fontos: ne Singleton osztály köré szervezzük az egész programunkat, különben Clean Code elvet sértünk (“god class”). Magyarán irányító szerepet ne töltsön be. Megsértése esetén a program egyes részei a függőség miatt nem lesznek újrafelhasználhatók. Ez azt jelenti, hogya nekikezdetek egy másik szoftver projektnek, az alapoktól kell

felépíteni az egészet (korábban megírt modulok nélkül), ami rengeteg időt elvisz feleslegesen. Javallott, hogy **Singletont csak a fentebb említett példákhoz hasonló szolgáltatásokhoz használjatok.**

Felmerülhet a kérdés, hogy miért nem olyan osztályt készítünk inkább, amelyikben minden függvény és változó statikus. Ekkor ugyan jobb teljesítményt érténk el (hiszen fordítási idő végére már használhatók lesznek), azonban számos olyan feature-t elveszíthetünk, amit a Singleton biztosít: függvénynév túlterhelés, könnyebb tesztelni (“dummy”/”mock”/teszt adattal feltöltött Singleton), könnyebb belső állapotnyilvántartás (objektumoknak van állapota). (Van még pár fontos előnye (polimorfizmus és öröklés), de ezekről majd később fogtok hallani).

```
// singleton.h //
#pragma once
#include <iostream>
class Singleton {
    // - priváttá tesszük a konstruktorokat
    // - így csak osztályon belül lehet példányosítani
    Singleton() {
        someValue = 10;
    }
    Singleton(const Singleton&){}
    int someValue;
public:
    // privát konstruktor miatt
    // az osztályon beüli scope-ot csak statikus metóduson keres
    // ezen belül kell elérni, hogy csak egy példány létezhesse
    static Singleton& getInstance();
    // funkcionalitást megtestesítő függvény
    void doStuff();
};
```

```
// singleton.cpp //
#include "singleton.h"
```

```
Singleton& Singleton::getInstance() {
    // csak egyszer jön létre, bárhányszor hívjuk meg a getInsta
    static Singleton s;
    return s;
}
void Singleton::doStuff() {
    std::cout << "Method of a singleton class. Value of someValu
}
```

```
// main.cpp //
#include "singleton.h"
int main() {
    // kívülről akár teljesen különböző példányoknak is lehetne
    Singleton& mySingleton1 = Singleton::getInstance();
    Singleton& mySingleton2 = Singleton::getInstance();
    mySingleton1.doStuff(); // kimenet: Method of a singleton cl
    mySingleton2.doStuff(); // kimenet: Method of a singleton cl
    return 0;
}
```

Osztály tagváltozóinak inicializálása

Inicializálásnak a változók/tagváltozók létrehozásakor történő kezdőérték-adást nevezzük. Egy egyszerű értékadás nem számít inicializálásnak.

pl. `int x; x = 0; // definíció + értékadás`

pl. `int x = 0; // definíció + inicializáció`

Egy osztály tagváltozóinak inicializációját a **konstruktorok** teszik lehetővé. Az értékadásokat elvégezhetjük a konstruktor **törzsében** vagy **inicializációs** listán. Az összes tagváltozónak a memóriafoglalás és az inicializációs lista műveletei még a konstruktor törzsének lefutása előtt végrehajtódnak. Ha egy tagváltozó nem szerepel az inicializációs listában, alapértelmezett értékkel foglalódik le számára a memóriaterület. Ha szerepel, akkor a megadott értékkel.

Tegyük fel, hogy van egy “Computer” osztályunk, aminek két int típusú tagváltozója van, a “cpu” és “ram”.

Inicializáció a konstruktor törzsében: `Computer(int c, int r){ cpu = c; ram = r; }`

Inicializációs listával: `Computer(int c, int r): cpu(c), ram(r) {}`

Beépített, **egyszerű** típusú tagváltozók esetén mindegy, hogy azokat milyen módon inicializáljuk (ld. fenti mindkét példa jó).

Ezzel ellentétben, ha egy **objektum** tagváltozóról van szó (valamilyen osztály típusú), és az inicializációs listában nem szerepel, akkor default konstruktorral fog létrejönni, mielőtt a konstruktor törzse lefutna. Ha valamilyen paraméterekkel szeretnénk létrehozni, akkor ezt mindenképp az inicializációs listában tegyük meg, különben feleslegesen jön létre belőle egy default példány. Ha pedig nincs is default konstruktora a tagváltozó osztályának, akkor muszáj inicializációs listában valamilyen paraméterrel inicializálni.

Referencia tagváltozó inicializációja csak és kizárólag inicializációs listában hajtható végre, és itt kötelező is, mivel referenciának muszáj értéket adni.

Konstans tagváltozónak mindig inicializációs listában adunk kezdőértéket.

Statikus tagváltozókat az osztályon kívül kell definiálni és inicializálni. Ezt nem lehet header fájlban megtenni, mivel ekkor minden fájlnak, mely include-olja ezt a header fájlt, lenne egy definíciója a statikus változóra. Összességében több definíciója lenne ugyanannak a változóban, ami tiltott (pontosan egy lehet neki), és a linker hibát fog dobni. Tehát *.cpp fájlban, mindenen kívül (global scope) kell definiálni.

pl. `int MyClass::staticVar = 10;` → osztályhoz tartozik, így hivatkozunk rá (mivel több osztálynak is lehetne ugyanilyen nevű statikus változója, meg kell adni, melyikre gondoltunk)

Általános szabályok az előbbi, egyedi eseteken kívül:

- Ha egy tagváltozónak **nem alapértelmezett értéket** szeretnénk beállítani (pl. egy objektumnak a default konstruktora által biztosított értékeket), akkor használjunk **inicializációs listát**. E mögött az a motiváció, hogy ha a konstruktor törzsében végezzük az értékadásokat, akkor addigra már alapértelmezett értékkel lefoglalódott számára a memória, és még azt felülírjuk (2 művelet), ahelyett hogy egyetlen, jól paraméterezett kezdőérték-adással oldottuk volna meg (1 művelet) és időt takarítanánk meg.

- Minden olyan típusú tagváltozónak **inicializációs listában** kell kezdőértéket adni, melynek **nincs értékadás operátora** vagy **default konstruktora**.

Friendship

Egy osztályban deklarálhatunk olyan függvényeket, melyek nem tagfüggvények, mégis engedélyt kapnak arra, hogy a private (és protected - ld. később) láthatóságú tagváltozókat és tagfüggvényeket elérjen. Ezt úgy tehetjük meg, ha a külső függvény deklarációját felvesszük az osztályba és elé tesszük a “friend” kulcsszót.
Egy osztály is lehet friend, mely a friend függvényekhez hasonlóan hozzáfér egy másik osztály privát (és protected) tagváltozóihoz és tagfüggvényeihez. Ezt az engedélyt mindig explicit ki kell írni, tehát egy osztály “barátjának a barátja nem a barátunk”.
Friend osztályt deklarálni kell a “friend” kulcsszóval abban az osztályban, aminek a privát (és protected) tagjaihoz hozzáférhet.

pl.

```
class Person; // Person osztály deklarációja
class Address {
    friend class Person; // a Person osztály hozzáfér a privát t
    // ...
}
class Person { Address live;
    // Address típusú tagváltozó
    int age;
public:
    friend void aging(Person); // külső függvény, ami hozzáfér a
    // ...
}
void aging(Person p){
    p.age++;
}
```

Névterek

Előfordulhat az az eset (bármikor), hogy amikor egy vagy több fájlt/libraryt include-olunk, ugyanazzal az azonosítóval létezik több függvény vagy osztály. Például írhatunk egy olyan függvényt, aminek printf() a neve, és ha include-oljuk az stdio.h könyvtárat, abban szintén létezik egy printf() metódus. Ilyen névütközés esetén a fordítás folyamán hibát kapunk, mert nem tudja kitalálni, hogy mi melyikre gondoltunk, amikor meghívtuk azt a függvényt, amiből kettő is van (pl. a mi saját printf() vagy az stdio.h-ban lévő printf() metódust hívja meg), vagy melyik osztályt szeretnénk példányosítani, ha több lehetőség is van. Ezeknek a névütközéseknek a feloldására használhatóak a névterek.

Egy névtér egy kódrészt fog egybe, melyen belül garantálni kell az azonosítók (a függvény, változó és osztálynevek) egyediségét. Egy névtér elemei egymást elérik. Névteren belül található függvényekhez/osztályokhoz csak úgy férhetünk hozzá, ha megadjuk a a névteret is, amelyen belül definiálva lettek. Ehhez a hatókör-feloldó (scope resolution) operátort (::) használjuk. Ennek a használatával tudjuk megmondani a fordítónak, hogy egy bizonyos névtérben található függvényt/osztályt használjon.

Tegyük fel, hogy az előbb említett printf() függvényünket egy MagicNamespace nevű névtérbe helyeztük át. Most, ha meghívjuk a printf() függvényt, akkor az stdio.h-ban lévő függvény fog lefutni. Ha pedig a sajátunkat szeretnénk meghívni, ahhoz a MagicNamespace::printf() függvényhívás szükséges.

Névterekkel nem csak a névütközéseket oldhatjuk meg, hanem logikai egységet is létrehozhatunk összetartozó osztályoknak, függvényeknek stb. Pl. egy névtérbe helyezhetjük a geometriai alakzatok osztályait.

Egy névtér több fájlba is felbontható, és névterek egymásba ágyazhatók.

using

Amikor egy bizonyos névtérben lévő függvényeket, osztályokat stb. sűrűn használjuk, ezért nagyon sokszor ki kell írnunk a névtér nevét is, a C++ lehetőséget ad ennek egyszerűsítésére.

Meg tudjuk mondani a fordítónak, hogy ha egy bizonyos azonosítót használunk, akkor azt melyik névtérben keresse.

- szintaxis: `using nevter::azonosito;`
- beágyazott névterek esetén: `using nevter1::nevter2::azonosito;`

Abban az esetben viszont, amikor nagyon sok függvényt stb. szeretnénk használni egy névtérből, nem hatékony ennek a használata. Ezt a problémát úgy tudjuk megoldani, hogy a fordítónak azt az utasítást adjuk, hogy egy adott névtérből használjon minden azonosítót.

- szintaxis: `using namespace nevter;`

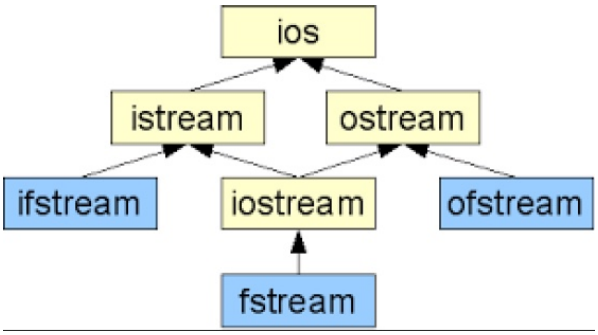
> 💡 Figyelem, using namespace használatával legyünk óvatosak, mert az eredetileg megoldott névütközés probléma visszatérhet, ezért kerüljük! Pl. két olyan névtérre is alkalmazzuk, melyekben szerepel ugyanaz az azonosító. Ekkor az elsőként bemutatott módszerrel lehet a compilernek megmondani, melyik névtérből szeretnénk használni az azonosítót vagy explicit kiírjuk a névteret.

Using-ot használhatunk globálisan és függvényeken belül is.

6. előadás

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** (***i**nput **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 kiíratása 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 szabáj 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(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+c
std::istream& operator>>(std::istream&, Complex&); // 10+c1
```

```
complex.cpp
#include "complex.h"
using namespace std;

// Tesztelés céljából kiírjuk, hogy melyik függvény hívódott meg
void w(const char* functionName){cout << "#CALLING " << function
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"
    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; ugyene

    // 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.

8. előadás

Öröklés

Az objektum-orientált tervezés egy fontos koncepciója. Lehetővé teszi a szoftver egyszerűbb elkészítését és karbantartását, mivel egy osztály funkcionalitása, viselkedése újrafelhasználható.

Öröklés két osztály között állhat fenn, az egyiket ősosztálynak (base class), a másikat leszármazott (derived class) osztálynak nevezzük. A leszármazott osztály rendelkezik az ősosztály tulajdonságaival, és azokat ki is bővítheti.

A leszármazott osztály az ősosztály minden, nem privát tagját eléri.

Új láthatósági típus: protected

Az ősosztályban protected láthatóságú tagváltozók a leszármazott osztályok számára láthatók, de más, külső osztályok számára nem.

Az öröklés típusa

Többféle öröklés valósítható meg, melyek abban különböznek egymástól, hogy az ősosztály különböző láthatóságú tagjai milyen láthatóságúak lesznek a leszármazott osztályban. Ezt a következő táblázat foglalja össze:

Ősosztályban lévő láthatóság	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	nem látható	nem látható	nem látható

Magyarázat pl. (public-public): Az ősosztályban public láthatóságú tagváltozó/tagfüggvény public öröklés esetén a leszármazott osztályban public láthatóságú lesz.

Általánosítás (generalization)

Tegyük fel, hogy van két (vagy több) osztályunk, melyek rendelkeznek hasonló tulajdonságokkal, viselkedéssel (metódussal). Ekkor létrehozhatunk egy általánosabb ősosztályt, melyből mindegyik öröklődik. Ezt a folyamatot nevezzük generalizációnak. A közös tulajdonságok az ősosztályban tagváltozóként, a közös viselkedés az ősosztályban tagfüggvényként jelennek meg.

Pl. van egy Horse és egy Dog osztályunk, mindegyikben megvalósíthatnánk az eat(), run() stb. függvényeket. De minek megírni kétszer ugyanazt (pl. növeli az energiáját az evés), a hasonló viselkedés kerülhetne egy általános ősosztályba. És miért ne csináljunk a kettőnek egyetlen osztályt, pl. Animal néven. Mivel vannak speciális tulajdonságaik is, pl. a ló tud versenyezni, a kutya pedig vadászni. Tehát eljutottunk

odáig, hogy legyen egy Animal ősosztályunk a közös tulajdonságokkal és viselkedéssel, maradjon meg a Horse és Dog osztály a speciális tulajdonságaikkal, és mindkettő öröklődjön az Animal osztályból.

Specializáció (specialization)

Specializáció az általánosítás ellentettje, egy már meglévő osztályból hozunk létre leszármazott osztály(oka)t. Ezt akkor tesszük meg, amikor az osztálynak vannak olyan tagváltozói, tagfüggvényei, asszociációi, melyek az osztály objektumainak csak egy részére érvényes.

Pl. van egy Vechicle nevű osztályunk, de csak bizonyos járművekre igaz, hogy tudnak repülni, másokra, hogy tudnak vízen úszni, így külön-külön tulajdonságokkal rendelkeznek (persze vannak közösek is, pl. max sebesség). Így a Vechicle osztályból öröklődik a Plane és a Ship osztály.

Behelyettesíthetőség / Polimorfizmus (polymorphism)

A leszármazott osztály minden helyen használható, ahol az ősosztály, tehát az ősosztály helyettesíthető a leszármazott osztállyal. Pl. mivel egy kutya állat is, egy Animal objektum helyén használhatunk Dog objektumot is (ha öröklődik belőle, és most tegyük fel, hogy így van), mivel rendelkezik annak minden publikus tulajdonságával.

Mit is jelent ez? Hozzunk létre egy függvényt, ami átvesz egy Animal objektumot. Ennek a függvénynek átadhatunk egy Dog típusú objektumot is. Miért nem baj ez? Tegyük fel, hogy szeretnénk hívni egy függvényt (Eat) az Animal típusú kapott objektumon. Mivel a Dog típusú objektum öröklődik az Animal osztályból, ő is rendelkezik ezzel a függvénnyel, így meghívható rajta. Amikor a paraméterül kapott objektumon függvényt hívunk, nem tudjuk, hogy az egy Animal vagy egy Dog típusú objektum-e, de nem is érdekel minket, csak meg szeretnénk etetni.

Ezzel a későbbiekben részletesebben foglalkozunk.

Tartalmazás vs Öröklés

A kettő közötti különbséget egy öröklésre való rossz példával szemléltetjük: legyen egy Car (autó) és egy Wheel (kerék) osztályunk. Érezzük, hogy valamilyen kapcsolatban vannak egymással. Nos, ha egy kerék nélküli kocsi és egy kerék halmazát vesszük, akkor ez a halmaz rendelkezni fog a kerék osztály minden tulajdonságával, nem? Ez azt jelenti, hogy örököltetjük a kocsit a kerékből, nem? És akkor például a kocsi egy függvényén belül fogjuk tudni módosítani a kerék egy kerületét, ugye? Eddig oké, de mi van akkor, ha a kocsinak (például) négy kereke van? Akkor négyszer örököltetünk? Egyáltalán hogyan hivatkozunk az egyes kerekekre?

Ezen a problémán kívül a már korábban említett polimorfizmus is probléma, miszerint bárhol ahol az ősosztály egy példánya szerepel, kicserélhetjük egy leszármazott osztály példányára (nyilván nem cserélünk ki egy kereket hirtelen kocsira).

Ezt a problémát tipikusan kicseréljük az intuitívabb tartalmazásra: a kocsi osztály egy vektorban tárol négy kereket.

Általános ökölszabály, hogy örököltetni csak akkor örököltetünk, ha egyrészt megvalósítható vele a polimorfizmus, másrészt csak viselkedést bővítünk, adatot újra nem használunk fel. Így tehát öröklésre szintén rossz példa, ha például egy vektorból örököltetünk.

Clean Code 5. - Öröklés

- az öröklés mindig a viselkedés újrahasznosítása miatt használható
  - adat-újrahasznosítás lenne: inkább tartalmazás
- közös viselkedés közös ősosztályban legyen
  - az öröklés-hierarchiában mozgassuk minél magasabbra
  - különben ismétlődések lennének (és nem szeretjük a duplikációt)
- egy osztály ne függjön a leszármazottaitól
  - pl. ne tartalmazzon leszármazott típusú objektumot
  - különben nem lehet újrafelhasználni nélkülük
- protected láthatóságú csak függvény legyen, tagváltozó ne
- ne legyen túl mély az öröklési hierarchia (max. 7 szint)

- soha ne vizsgáld egy objektum típusát, használj polimorfizmiust
- ne használj öröklést, ha a leszármazott osztály nem bővíti az ősosztály viselkedését
  - ekkor a leszármazott osztály inkább az ősosztály egy objektuma
- ha a leszármazott osztályban felülírod az ősosztályban már meglévő viselkedést üres viselkedéssel, rossz az öröklési hierarchia
  - valószínűleg fel kell cserélni az öröklési sorrendet

9. előadás

Alapprobléma

Legyen adott a következő program:

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

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).

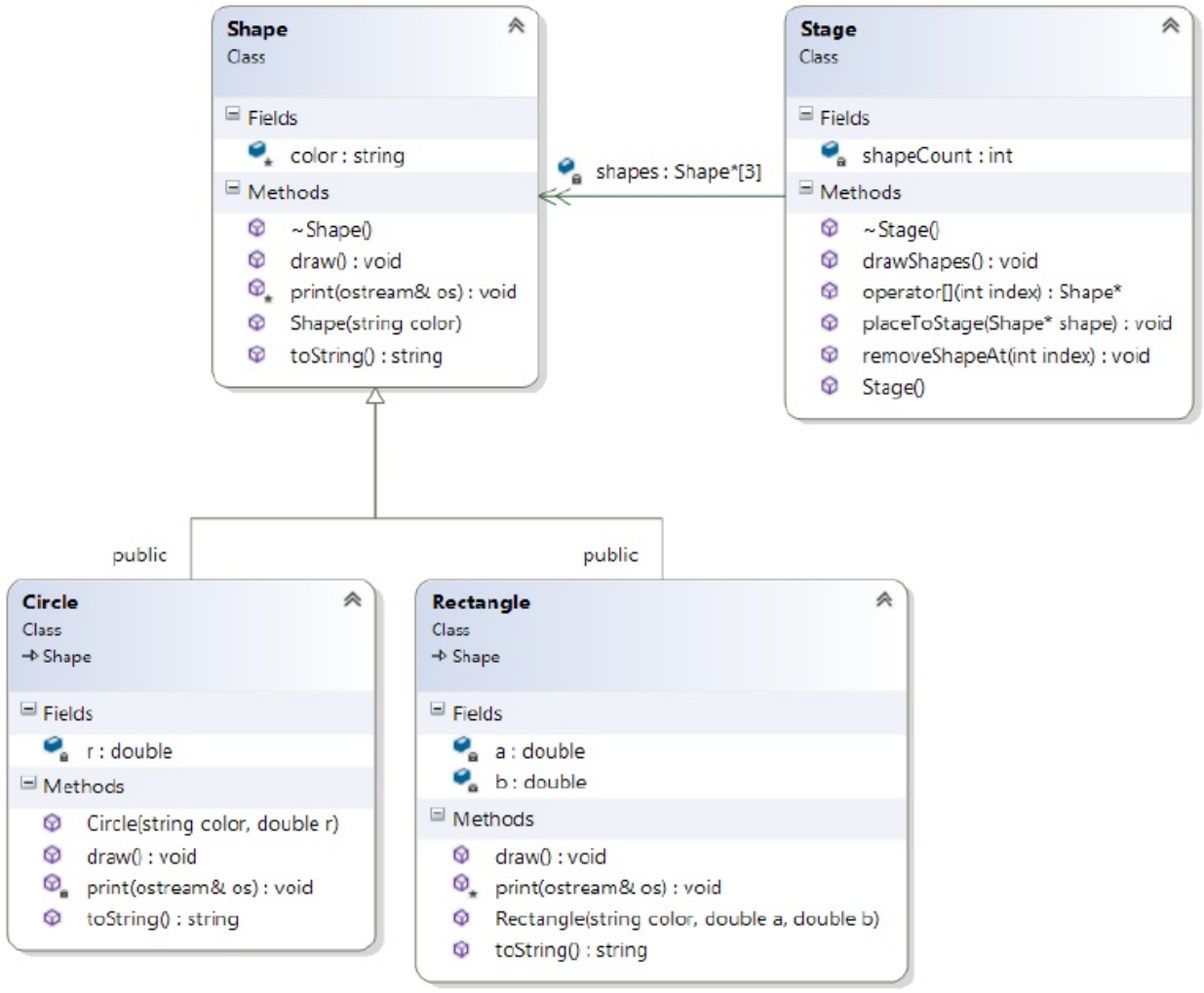
Ez a módszer még két osztálynál úgy-ahogy “elmegy”, de egy komolyabb szoftvernél vállalhatatlan, hiszen ha bővítenénk a kódbázisunkat egy új osztállyal, mindig definiálni kellene egy új tömböt, új ciklust kellene írni, mindig módosulna a régi kódunk, ráadásul tele lenne ismétlődéssel (DRY). Magyarán **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 öröklt etésekkor.** 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 (placeToStage()), 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, gree
    stage.drawShapes(); // red, green, blue
    stage.placeToStage(new Rectangle("yellow", 10, 4)); // no m
    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 területe
}
```

shape.cpp

```
#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
    // - minden másra (főleg diagnosztikára) ott a toString
    // - virtual: ha egy leszármazott felüldefiniálja a printet,
    // akkor late binding miatt a leszármazott definíciója fog l
    // - 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
    // - így mivel ez elég gyakori, az ősosztályban praktikus ne
    // tisztán virtuálissá tenni a destruktort, helyette definia
    virtual ~Shape() {}
    virtual std::string toString() const
    {
        // - továbbra sincs szükség sortörésre
        // - inkább rábízzuk a fejlesztőre, hogy hol akarja megt
        return "color=" + color;
    }
    // Shape-t önmagában nincs értelme kirajzolni
    virtual void draw() = 0;
    friend std::ostream& operator<<(std::ostream& os, const Shap
        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: c
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 ma
            std::cout << "No more room left on stage." << std::e
    }
    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 illes
        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
        float pr = 2; // a karakteres felületet alkotó cellák me
        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
                if (d>0.9 && d<1.5) // közelítés tapasztalati ko
                    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):Sha
        std::string toString() const {
            std::stringstream ss;
            ss << "a=" << a << "; b=" << b;
            return ss.str();
        }
    void draw() {
        std::cout << "This rectangle is " << color << " (believe
        // 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

10. előadás

Alapprobléma

Tegyük fel, hogy olyan programot írunk, amiben különböző járműveket reprezentáló osztályokra van szükségünk. Rendelkezünk eddig Car és Plane osztályokkal (ne legyen közös ősosztályuk az egyszerűség kedvéért, ezzel a problémával később foglalkozunk).

A programunk tervezésekor tegyük fel, hogy már nagyon elterjedtek a repülő autók, így van igény arra, hogy ezt is lemodellezzük. Vegyük bele a repülőkre jellemző funkcionalitást a Car osztályba? Vagy fordítva, az autó funkcionalitását a Plane osztályba? Dehogy, hiszen nem minden autó tud repülni, és nem minden repülő használható autóként. Akkor hozzunk létre egy új osztályt FlyingCar néven. És ebbe hogy kerülnek bele a repülőkre és autókra jellemző tulajdonságok? Leimplementáljuk újból az egészet (esetleg copy-paste)?

Mielőtt erre válaszolnánk, egy kis emlékeztető: **nem duplikálunk** (copy-paste az ellenségünk)! Ha mégis így tennénk, és később kiderül, hogy valamit változtatni kell a Car vagy Plane funkcionalitásán, bővíteni kell vagy hibát találtunk, akkor ugyanezt a változtatást elvégezhetnénk a FlyingCar osztályban is. Ha pedig elfelejtjük, akkor rosszul fog működni a program, és sok **időbe** kerül kideríteni, hogy egy **elfelejtett** módosítás volt a baj.

DUPLIKÁCIÓT OLYAN MESSZIRŐL KERÜLJÜK, AMENNYIRE CSAK LEHET!

Lehetséges megoldás: többszörös öröklés

Szóval nem, nem írjuk meg az összes funkcionalitást elölről egy új osztályban. Ehelyett minden szükséges osztályból örököltetjük egyszerre a FlyingCar osztályunkat. Kibővíthetjük további tagváltozókkal, tagfüggvényekkel is (épp, mint az egyszerű öröklésnél), valamint rendelkezni fog minden ősosztály minden nem privát tulajdonságával. Minden ősosztályhoz külön megadható, hogy milyen láthatósággal örököltetünk belőle.

Pl. lehetne: `class FlyingCar: public Car, protected Plane {...}`

De most legyen mindkét öröklés public, csak megmutattuk, hogy akár ilyen is lehetne.

Virtuális többszörös öröklés

A közös ős

Írjuk tovább a programunkat, aztán feltűnik nekünk egy nagy hasonlóság az osztályaink között. Mindegyikkel lehet utazni, feltölteni valamilyen szükséges üzemanyaggal, javítani amikor elromlik stb., tehát van hasonló viselkedésük. Korábban tanultuk, hogy ha bizonyos osztályok közös tulajdonságokkal rendelkeznek, hozzunk létre nekik egy **közös ősosztályt**, és ebben implementáljuk a közös tulajdonságokat. (Az autónak, hajónak, repülő autónak stb. egyébként is van közös összefoglaló neve, ilyenkor gyanakodhatunk, hogy valószínűleg közös ősosztályuk is lesz).

Tehát hozzunk létre egy osztályt Vechicle néven, ami rendelkezik minden közös viselkedéssel. Ebből öröklődjön a Car és a Plane osztály. A FlyingCar osztálynak nem kell közvetlenül öröklődnie belőle, mivel már öröklődik pl. a Car-ból, így rajta keresztül a Vechicle-ből is közvetetten.

1 FlyingCar = 2 Vechicle? Nagy baj van!

Viszont itt belefutunk egy új problémába. A FlyingCar osztályunk jelenleg közvetlenül 2 osztályból öröklődik, ezek mindegyike öröklődik a Vechicle-ből. Ezzel az a nagyon nagy baj, hogy így a FlyingCar összesen kétszer öröklődik közvetetten a Vechicle-ből. Nézzük meg, mit is jelent ez, és mi fog történni. Amikor létre szeretnénk hozni egy

FlyingCar objektumot, először az ősosztályok konstruktorai futnak le. Legyen ez a sorrend most Car, majd Plane.

- Tehát először lefut a Car konstruktora. Ebben először a Car ősosztályának a konstruktora fut le (Vechicle). Így a memóriában lesz hely foglalva Vechicle-nek és Car-nak is.
- Aztán ugyanezt eljátsszuk a Plane ágon is: itt is foglalódik hely a Plane ősosztályának, a Vechicle-nek, és a Plane-nek is.
- Végül lesz hely foglalva a Car plusz adatainak, függvényeinek.
  Így összesen kettő Vechicle-nek foglalódott hely a memóriában. Megkérdezhetjük, hogy na és ez miért baj?
- Memóriát nem pazarlunk.
- Semmi értelme, hogy egy repülő autó példány kettő jármű példány legyen egyszerre
- Amikor olyan függvényt szeretnénk hívni FlyingCar objektumon, ami a Vechicle osztályhoz tartozik, a fordító nem fogja tudni, hogy a kettő közül mi melyik Vechicle függvényére gondoltunk. Ez a felállás **fordítási hibát** okoz.

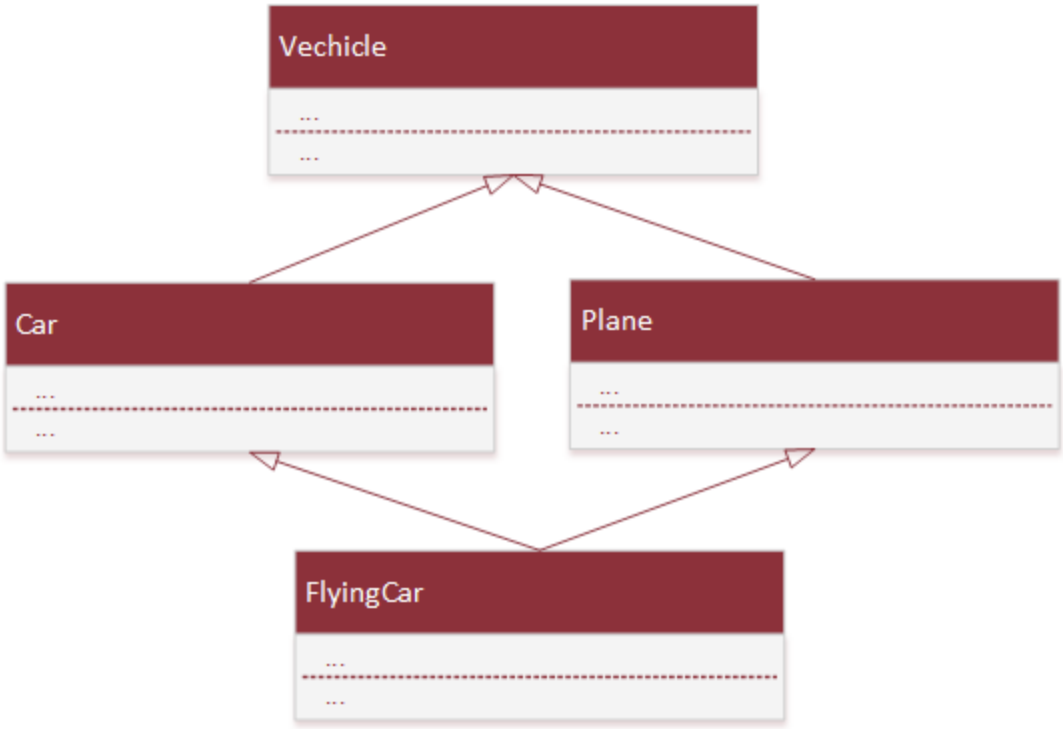
Megoldás

Azt szeretnénk, hogy 1 FlyingCar-hoz 1 Vechicle tartozzon. Ebbe már a compiler sem köthetne bele, tudná melyik függvényt kell hívni, amikor Vechicle osztálybeli függvényt hívunk.

Pont ezt teszi lehetővé a **virtuális öröklé**s. A Car és a Plane nem hoz létre külön-külön Vechicle memóriaterületeket, hanem egyen osztozkoznak.

Végső soron az öröklés így alakul:

```
class Vechicle {...};
class Car: public virtual Vechicle {...};
class Plane: public virtual Vechicle {...};
```

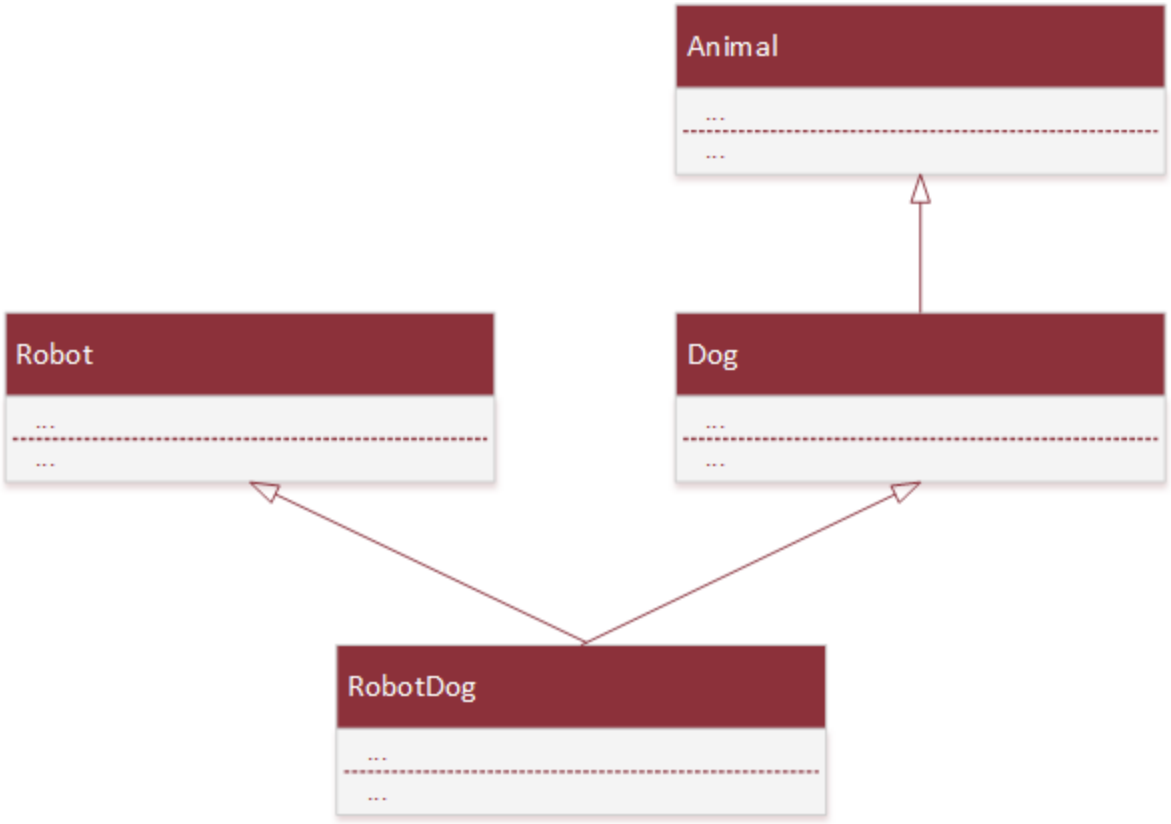



```
class FlyingCar: public Car, public Plane {...};
```

Összefoglalás

Ha egy osztály olyan több, különböző osztályból öröklődik, melyeknek van közös ősosztálya, virtuális öröklésre van szükség. Könnyen felismerhető, mivel ha felrajzoljuk az öröklési hierarchiát, egy gyémánt-szerű alakot kapunk, ezért is hívják “**diamond problem**”-nek, vagy “**deadly diamond of death**”-nek.

Az viszont, ha két független, azaz különböző öröklési hierarchiából származó osztályból örököltetünk, nem jár ilyen gondokkal. Például ha van egy Robot osztályunk, és egy Dog osztályunk, melyeknek nincs közös őse, és ezekből örököltetünk egy RobotDog osztályt, nem szükséges virtuális öröklés, ez nem a diamond problem esete.



Miért ne használjunk többszörös öröklést?

Többszörös öröklés használata előtt nagyon gondoljuk át, hogy tényleg ezt használva lehet-e a legjobban megoldani a problémát. Nem lehet hogy tartalmazás kellene helyette? Vagy valaminek egyáltalán nem kéne osztálynak lennie, hanem egy osztály egy tulajdonságának?

Egy olyan programot ami többszörös öröklést használ, nagyon nehéz fenntartani, bővíteni, debuggolni, vagy bármilyen minimális módon belenyúlni. Főként a diamond problem miatt keletkezhetnek olyan hibák, amik megkeseríthetik az életünket (azt pedig ugyebár nem szeretnénk). Természetesen előfordulhat, hogy muszáj használni, ekkor viszont nagyon nagyon figyeljünk oda a virtuális öröklésekre.

Ha nem is áll fenn a diamond problem, még lehet baj, amennyiben egy osztály több őse is definiál ugyanolyan nevű és paraméterezésű függvényeket. Ekkor nem egyértelmű, hogy ha ilyen függvényt hívunk a leszármazott osztály egy objektumán, melyik ősosztályban definiált függvényre gondoltunk. Szerencsére ezt explicit tudjuk egyértelműsíteni. Tegyük fel, hogy a fentebb említett Robot-Dog-RobotDog öröklési hierarchia esetén a Robot és Dog osztályban is definiáltunk run() függvényt. Ekkor, ha

egy RobotDog példányon run() függvényt hívunk, a fordító nem fogja tudni, hogy a Robot vagy a Dog osztályban definiált függvényt kell-e itt meghívni. Így tudjuk egyértelműsíteni, hogy mit szeretnénk: robotDogObject.Robot::run() VAGY robotDogObject.Dog::run().

De mikor érdemes mégis használni?

Ha már **előre adott osztályok** vannak egy (third party) keretrendszerben, akkor sokszor azt **írják elő**, hogy származtassunk le belőlük. Így implementálhatjuk azokat a műveleteket, amiket a keretrendszer szeretne meghívni. Vagyis úgy illesztünk a keretrendszerhez, hogy **több osztályból származtatunk**! Mivel így csak a művelet deklarációját adják meg, a **leszármazott egyfajta hívható felületet, interfészt implementál**.

A gyakorlat az, hogy maximum egy osztályból öröklünk, viszont több interfészt implementálunk. Magasabb szintű nyelvek, mint például a C# vagy a Java nem is engedik a többszörös öröklést, helyette egy külön nyelvi elemet vezettek be, az interface-t, ami analóg azon C++-os absztrakt osztályokkal, amelyek csak tisztán virtuális, publikus tagfüggvényeket tartalmaznak.

Ezzel részletesebben a laboron ismerkedhettek meg.

11. előadás

Alapprobléma - függvények

Tegyük fel, hogy írtunk egy rendező függvényt `int` típusra, mely átvesz egy tömböt és a tömb elemszámát paraméterként. Legyen ez a függvény a következő:

```
void sort(int* array, int count) {
    for (int i = 0; i < count-1; i++) {
        int min = i;
        for (int j = i + 1; j < count; j++) {
            if (array[j] < array[min])
            min = j;
        }
```

```
        if (min != i) {
            int c = array[i];
            array[i] = array[min];
            array[min] = c;
        }
    }
}
```

Aztán szükségünk van egy double tömböt rendező függvényre. Meg egy string tömböt rendező függvényre. Meg egy saját Computer típusú elemekből álló tömböt rendező függvényt. És ez így mehetne tovább nagyon sok típusra.

Meg lehetne oldani úgy a problémát, hogy lemásoljuk a megírt függvényt, kicseréljük a típusokat a megfelelőre (az elemszám azért marad `int`), és kész.

Na ez az elfogadhatatlan megoldás, mivel pont a duplikáció esete, amit nagyon nem szeretünk, így minél inkább el szeretnénk kerülni.

Függvénytemplate

A tamplate-ek a generikus programozás alapjai, lehetővé teszik, hogy típusoktól független függvényeket írjunk. Egyszer kell megírni a függvényt, mégis több típussal használható. A template paraméterek lehetővé teszik, hogy a használandó típus(oka)t is átadjuk paraméterként.

A függvény törzsétől függően a felhasznált típusnak bizonyos kritériumoknak meg kell felelnie. Például ha a függvényünkben `cout` stream-re kiírjuk az átvett paramétert, csak akkor működik a függvény, ha a paraméter típusára meg van írva az `operator<<`.

A fentebb említett függvényünk a következőképp írható meg függvény template-ként:

```
template <class T>
void sort(T* array, int count) {
    for (int i = 0; i < count-1; i++) {
        int min = i;
        for (int j = i + 1; j < count; j++) {
            if (array[j] < array[min])
                min = j;
```

```
        }
        if (min != i) {
            T c = array[i];
            array[i] = array[min];
            array[min] = c;
        }
    }
}
```

Ebben az esetben csak akkor használható a függvény T típusú tömbbel, ha a T típusra meg van írva az `operator<` és az `operator=`.

Template paraméter

A rendezőfüggvény template paramétere T. Ez a template paraméter egy típust reprezentál, ami még nem lett definiálva, de a függvényben használhatjuk úgy, mint egy szokványos típust.

Amikor egy adott típussal meghívjuk a függvényt, a compiler automatikusan generál egy függvényt (ha még nem tette meg korábban), ahol a használt típus már nem a template paraméter, hanem a hívásban szereplő paraméter típusa. A template-ek fordítási időben fejtődnek ki azokra a típusokra, amelyekkel használjuk, a többivel nem.

Csak fordítási időben derülnek ki a hibák.

Pl. ha a rendező függvényt mi meghívjuk `double` típusú tömbbel, létrehoz egy olyan függvénypéldányt, ahol minden T típus helyén `double` áll.

Függvénytemplate használata

Vezessünk be egy új függvényt, ami kiírja két, paraméterként átvett változó négyzetösszegét:

```
template <class T>
void squareSum (const T a, const T b) {
    cout << a * a + b * b << endl;
}
```

Implicit példányosítás

Függvénytemplate-eket használhatunk úgy, mint egy egyszerű, szokványos függvényeket.

Pl.:

```
squareSum(2, 3); squareSum(1.0, 2.0);
```

Mivel mindkét paraméter ugyanolyan típusú a különböző hívásokkor (az elsőben int, a másodikban double), a fordító automatikusan ki tudja találni, hogy mi a template paraméter, milyen típussal kell példányosítania a függvényt.

Explicit példányosítás

Választhatóan

Az előző két példában mutatott esetben kiírhatnánk konkrétan, hogy milyen típussal hívjuk a függvényt a következő módon:

```
squareSum<int>(2, 3); squareSum<double>(1.5, 2.5);
```

Ezt nem kötelező megtenni ezekben az esetekben.

Ezek a függvények már teljesen olyanok, mint a szokványos függvények.

Kötelezően

Bizonyos esetekben azonban muszáj konkrét típust megadni, aminek fő oka, hogy a template függvény nem végez semmilyen konverziót. Például nem tudja végrehajtani a fordító a squareSum(2.5, 3); hívást, mivel a két paraméterének típusa eltérő, és a függvény mindkét paraméternek egy közös, T típust vár. Nem tudja kitalálni, mi legyen ebben az esetben.

Ekkor két lehetőségünk van. Az első, hogy az egyik paramétert átkonvertálom híváskor a másik típusára, hogy ki tudja találni a fordító a típust. A második, hogy explicit megadjuk, milyen típusú paramétereket várjon, így ha mégsem olyat kapna, megpróbálja átkonvertálni a paramétereket erre (ha nem sikerül, hibát kapunk). Így a két lehetséges megoldás:

```
squareSum(2.5, double(3)); squareSum<double>(2.5, 3);
```

Ha a paraméterek konstansok helyett változók lennének, ugyanezek a szabályok lennének érvényesek.

Alapprobléma - osztályok

Nagyjából ugyanaz a probléma, mint korábban, csak az osztály tagjainak kell tudnia használni több típust.
Például szeretnénk írni egy Stack osztályt, amiben dinamikus tömbként tárolhatunk ugyanolyan típusú elemeket. Viszont szeretnénk külön csak `int`, `double`, `string` stb. típusú elemeket tároló Stack-et. Ekkor írhatnánk külön-külön osztályt pl. StackInt, StackDouble, StackString stb. néven, de ez nagy pazarlás lenne. Mindegyiknél külön-külön az összes függvényt megírni minek? Csak időpazarlás, karbantarthatatlan és csúnya.

Parametrizált osztályok

Erre a problémára adnak megoldást a template osztályok. Pl.:

```
template <class T>
class Stack{
    T* pData;
    int elementNum;
public:
    int push(T newElement); // visszatérési érték=0 ha sikeres;
    //…
};
```

Ekkor a pData dinamikus tömb által tárolandó elemek típusát a Stack osztály példányosításakor adhatjuk meg, de az osztály definícióját úgy írhatjuk meg, hogy ezt nem tudjuk.

Amennyiben egy függvény definícióját az osztályon kívül írjuk meg, minden definíció elé ki kell írni a template-et. Pl.:

```
template <class T> int Stack::push(T newElement) {..}
```

Osztálytemplate használata

A Stack osztály használata pl.:

```
Stack<int> integers;
Stack<double> doubles;
Stack<Stack<int>> complicated;
```

A felparaméterezett osztály már egy közönséges osztály, bárhol használható, ahol egyszerű osztály is.

Default template argumentum

Template paraméternek megadható default érték, pl. ha azt szeretnénk, hogy alapértelmezetten int-eket tároló Stack jöjjön létre: template <class T = int>

Ekkor, ha int típusú elemeket tároló Stack-et szeretnénk, létrehozhatjuk ilyen módon is:

`Stack<> s;`

Mi lehet template paraméter

Template paramétek lehetnek a következők:

- típus
  - int, double, string
  - felparaméterezett template osztály: pl. Stack<Stack<int>> s; jó, ekkor a paraméter Stack<int>
- típusos konstans
  - konstans
  - konstans változó
- template osztály
  - nem felparaméterezett template osztály: pl. Vector<Stack> s;

- ehhez a Vector osztály deklarációjának a következő template-tel kell rendelkeznie:

```
template <template <class V> class T>
class Vector {
    T a;
    V b;
};
```

Tagfüggvény template

Akár template osztályról, akár egyszerű osztályról van szó, a tagfüggvényeknek lehetnek külön templateparaméterei. Template osztálynál ezt az indokolhatja különösebben, ha az osztály template paramétertől különböző template paramétert szeretnénk átvenni.

Pl.:

```
template <class T>
class MyClass {
    //…
    template <class V>
    V myFunc(V param);
};
```

Annyi verzió generálódik belőle automatikusan, ahányféle különböző típussal használjuk.

Öröklés template osztályból

Csak osztályból lehet örököltetni. Egy template osztály addig template osztály, amíg van nem rögzített template paramétere. Amint minden templateparaméter rögzítve lett, a template osztályból általános osztály lesz.

Pl.:

```
template <class T, class U> class Base {...}
class Derived: public Base<int, V> {...}
```

Template vs. Öröklés

Ha ugyanazt a viselkedést szeretnénk leírni több típusra, használjunk template-et.

Ha felüldefiniálható viselkedésre van szükségünk, használjunk öröklést.

template<class T> vs template<typename T>

Technikailag a két kulcsszó ugyanazt jelenti. Azért létezik mégis két kulcsszó rá, mert az ősidőkben Stroustroup jónak látta, hogy egy szóhoz több jelentést rendelünk (static-nél még rosszabb a helyzet).

Van azonban pár speciális eset, amikor különböznek, mint például:

```
template<template<class> typename MyTemplate, class Bar> class F
template<template<class> class MyTemplate, class Bar> class Foo
```

Amikor template templateket készítesz, a class kulcsszót kell használnod.

Template osztály dekompozíció

Template osztályt nem lehet két fájlra bontani olyan módon, hogy az osztály deklarációja egy header, a definicíója pedig egy .cpp fájlban van, a fordítás sikertelen lesz.

Ennek oka a következő: A fordítási folyamat második lépésében a compiler a template osztályt szeretné lefordítani egyszerű osztállyá minden olyan értékre, mellyel használták. Pl. ha a Stack template osztályt használjuk int és double típusokkal, akkor két osztályt generálna le. Amikor egy helyen lát egy ilyen példányosítást, a beinclude-olt header fájlból (muszáj include-olva lennie, különben nem lehetne használni) látja, hogy milyen tagváltozói és tagfüggvényei vannak.

El szeretné készíteni az osztályhoz tartozó memóriastruktúrát, de mivel csak a header fájlt látja, fogalma sem lesz róla, hogy a függvényeket hogyan kell elkészíteni. Mivel a

.cpp fájlt ő nem ismeri (fájlonként fordít, a fájlok közti kapcsolatokat a következő lépésben a linker végzi el). A 2. lépésben még nem keletkezik hiba, a fordító feltételezi, hogy valahol már léteznek a függvények a szükséges típusokkal (a példában double és int típussal). Viszont amikor a template osztályhoz tartozó .cpp fájlt fordítja (függetlenül az előző fájl fordításától), mivel nincs megadva típus a template osztálynak, nem tudja a függvényekből létrehozni a típusos függvényeket (ebben a fájlban senki nem mondja meg neki, mik kellenének).

A probléma a linkeléssel van. Mivel a compiler nem tudta példányosítani a függvényeket a megfelelő típusokkal, a linker nem talál a generált típusos osztályok függvényeinek definíciót és linkage error keletkezik.

Megoldás: Tegyük egy fájlba az egy osztályhoz tartozó deklarációkat és definíciókat, így az include-oláskor a függvények definíciója is elérhetővé válik. Ettől még nem kell az osztályon belül, inline megírni a függvények törzsét, lehet az osztálydeklaráció után.

12. előadás

Impicit, explicit konverzió

Beépített típusok

C++-ban definiálva vannak standard konverziók a beépített, egymással kompatibilis típusokra. Kisebb méretű típusból nagyobb méretű típusra való konverzió probléma nélkül végrehajtható.

Pl.:

- `float` → `double`
- `unsigned` → `int`
- `short` → `int`
- `int` → `long`

Ilyen esetben három lehetőségünk van a konverzióra:

```
float f = 1.2; double d1 = f; // implicit konverzió
double d2 = (double)f; // explicit konverzió
double d3 = double(f); // konstruktorral
```

Ellenkező irányba viszont, amikor nagyobb méretűből kisebb méretűre szeretnénk konvertálni, előfordulhat, hogy adatot vesztünk. Ezt a compiler warning-gal jelzi (warning: possible loss of data), amit explicit konverzióval kerülhetünk el.

Pl.: `double`-ból `float`-ra konvertálva csökken a számábrázolás pontossága, warning elkerülése:

```
double d = 3.14;
float f = (float)d;
```

Bizonyos esetekben pedig nem is biztos, hogy ugyanazt az értéket kapjuk vissza konverzió után, ezekre nagyon figyeljünk:

- Negatív int érték konvertálása usigned int-re:
  - általános szabály: `int` → `int MOD UINT_MAX+1`
  - -1 → unsigned int által reprezentálható legnagyobb érték
  - -2 → unsigned int által reprezentálható 2. legnagyobb érték
  - stb.
- A bool típus false értékének a 0 vagy null pointer felel meg, a true értéknek minden más
  - bool típusra történő konverzió ez alapján egyszerű
    - `0` → `false`
    - `null` → `false`
    - minden más → `true`
  - bool típusból történő konverzió
    - false → 0
    - true → 1

- Lebegőpontos számról egész számra történő konverziókor a tört rész levágásával képződik az egész szám (nincs kerekítés alapértelmezetten, ehhez külön függvény kellene). Ha a keletkező egész szám nem ábrázolható a céltípusban (nem fér bele abba, amivé szeretnénk konvertálni), nem definiált, mi fog történni (undefined behavior).

Pointerek

`Null` pointer bármilyen típusú pointer-ré konvertálható, és bármilyen típusú pointer konvertálható `void` pointerré.

Osztályok

Ősosztály típusú pointerrel mutathatunk egy leszármazott osztály típusú objektumra.

Pl. ha Vechicle őse a Car-nak:

```
Car car;
Vechicle* vechicle = &car;
```

Leszármazott osztályra mutató pointer átkonvertálható az ősosztályára mutató pointerré (ld. heterogén kollekciók).

Leszármazott objektumnak nem adható értékül ősosztály típusú objektum, mert nem tudná mivel kiegészíteni, viszont fordítva működik:

```
Car car;
Vechicle vechicle;
car = vechicle; // nem jó
car = (Car)vechicle; // nem jó
vechicle = car // jó
vechicle = (Vechicle)car // jó, ugyanaz
```

Két, nem kapcsolódó típusú objektum közötti konverzió nem hajtható végre. Pointereikkel lehet, de teljesen értelmetlen. Pl.:

```
Dog dog;
Car car;
```

```
car = (Car)dog; // hiba
Car* pCar;
Dog* pDog = &dog;
pCar = (Car*)pDog; // nincs hiba
pCar->tankFuel(); // hiba, kutya típusú objektumra mutat valójá
```

Konverziós konstruktor

Minden olyan konstruktor, ami előtt nem áll explicit kulcsszó (ld. később), és egy paraméterrel hívható. Lehet több paraméterű konstruktor is, de ekkor az elsőn kívül mindegyiknek kötelező rendelkeznie default paraméterrel.

```
class Complex {
    double im;
    double re;
public:
    Complex(double re, double im = 0.0);
}; // ...
Complex c1 = 5.5;
Complex c2(5.5); // ugyanaz, mint az előző
Complex c3(1.0, 1.0)
```

Ennél a példánál maradva, ha lenne olyan függvényünk, ami Complex objektumot vár, és double-t kap, az nem lenne gond, mivel át tudja konvertálni Complex objektummá. Viszont ha Complex referenciát vár, azzal gond van. Csak akkor hívható double értékkel, ha nem egyszerű Complex referenciát vár, hanem konstanst. Ilyen függvény double értékkel történő hívásakor létrejön egy temporális Complex objektum a double értéket átkonvertálva, és erre kap referenciát a függvény. Pl.:

```
void func1(Complex c) {...}
void func2(Complex& c) {...}
void func3(const Complex& c) {...}

func1(2.0); // jó
```

```
func2(2.0); // nem jó
func3(2.0); // jó
```

Ilyen jellegű konverzióknál csak egy lépés lehet automatikus, a többit explicit ki kell írni. Például van automatikus konverzió int-ről double-re, double-ről Complex-re, de int-ről double-re már nincs. Az int → double → Complex láncban csak az egyik automatikus. Pl.:

```
func1(1); // hiba
func1((double)1); // jó, csak a double -> Complex automatikus
```

Ha egy művelet végrehajtására több út is rendelkezésre áll, azt fogja választani, amihez nincs szükség konverzióra. Konverziót csak akkor használ, ha nem talál konverzió nélküli egyező módot.

Pl.:

```
func(double d) {...}
func(Complec c) {...}
func(1.0); // az első függvény hívódik meg
```

explicit kulcsszó

Konverziós konstruktor elé explicit kulcsszót írva, az adott konverzió nem hajtható végre implicit módon, mindig explicit módon kell jelezni a konverziót.

```
class Complex {
    double im;
    double re;
public:
    explicit Complex(double re, double im = 0.0);
};

void func(Complex c) {...}

//...
```

```
Complex c(5.0); // jó
func(2.0); // hiba
func((Complex)2.0); // jó
```

Konverziós operátorok

A konverziós konstruktor arra volt jó, hogy valamilyen más típusból hozzunk létre sajátot. A konverziós operátor pedig pont ennek az ellenkezőjét fogja csinálni, tehát saját típusból hoz létre egy másikat.

```
class Complex {
    double im;
    double re;
public:
    Complex(double re, double im = 0.0);
    operator const double() const {return re;}
};
// ...
Complex c(2.5, 3.0);
double d = c;
```

Erre is igaz, hogy automatikusan, implicit módon csak egy lépést tud végrehajtani, a többi lépést explicit cast-olással kell.

```
Complex c(2.5, 3.5);
float f1 = c; // nem jó
float f2 = (double)c; // jó
float f3 = (float)(double)c; // jó
```

A konverziós operátor öröklődik, tehát a leszármazottban is használható felüldefiniálás nélkül is.

```
class A : public Complex {...}
A a(2.0, 5.0);
```

```
double d = a; // jó
```

Lehet virtuális az ősosztályban, és a leszármazott osztályban felüldefiniálhatjuk.

Próbáljunk meg inkább konverziós konstruktort használni. Ha nagyon muszáj konverziós operátort, akkor figyeljünk a többértelműség elkerülésére (többféleképp is végre lehet hajtani ugyanazt).

Összefoglalás

```
class Complex {
    double im;
    double re;
public:
    Complex(double re, double im = 0.0);
Complex& operator=(const double& d);
operator double() const;
};
// ...
double d = 2.5;
Compex c = d; // konverziós konstruktor
c = d; // = operátor
d = c; // konverziós operátor
```

C++ saját típuskonverziói

A C++ saját konverziós szintaxisai, melyek nagyobb biztonságot nyújtanak, használatukkal megadható a konverzió célja is.

static_cast<target_type>(expression)

Nem alkalmaz futás idejű típusellenőrzést, nem garantálja, hogy a konverzió eredményeképp a célobjektum teljes lesz. Pl. ha leszármazott osztállyá konvertálunk egy ősosztály típusú objektumot, attól még a leszármazott osztály plusz függvényei nem lesznek hívhatók. Nem távolítja el a konstans tulajdonságot.

reinterpret_cast<target_type>(expression)

Bármely pointer típust képes átkonvertálni bármely más pointer típussá, még teljesen független típussá is. Nem ellenőrzi sem a pointerek típusát, sem a pointerek által mutatott objektumok típusát. Nem távolítja el a konstans tulajdonságot. Megengedi az egész típusok és pointerek közti konverziót, illetve pointerek helyett használható referencia is.

Annyit garantál és semmi többet, hogy ha egy A* típusú pointert castolunk B* típusúvá (A és B helyén bármilyen típus szerepelhet), majd az eredményt visszacastoljuk A*-gá, akkor ennek az eredménye egy érvényes A* típusú pointer lesz.

dynamic_cast<target_type>(expression)

Az öröklési hierarchián felfele illetve lefele történő cast-oláshoz használható polimorf típusok között. Privát öröklés esetén felfele nem használható, futás idejű hibát kapunk. Nagyon biztonságos, biztosítja, hogy az eredményként kapott objektum teljes és érvényes legyen. Ha nem sikerül a cast-olás, null pointert ad vissza, ezért az eredményét mindig ellenőrizni kell használat előtt. Nem távolítja el a konstans tulajdonságot.

```
Base * pBase1 = new Derived;
Base * pBase2 = new Base;
Derived * pDerived;
pDerived = dynamic_cast<Derived*>(pBase1); // sikeres, nem null
pDerived = dynamic_cast<Derived*>(pBase2); // nem sikeres, null
```

const_cast<target_type>(expression)

Az egyedüli cast-olási mód, mely képes konstans típust nem konstanssá konvertálni. Csak erre képes, de mivel veszélyes művelet, külön jelölést kapott. Ha az így kapott, már nem konstans objektumot nem módosítjuk, nincs gond. Viszont ha módosítjuk, nem definiált a további működés (undefined behavior).

Kivételkezelés

Miért használjunk kivételkezelést?

A program futása során keletkező problémák, hibák kezelésére egyszerű és tiszta lehetőségeket biztosít, és használatukkal kevésbé valószínű, hogy egy hibát nem kezelünk le. Hiba esetén a vezérlés a hibakezelő részre ugrik.

A régi módszer a hibakódok visszaadása és hibaüzenetek kiírása több szempontból is előnytelen. Ekkor keverednek a programunk funkcionalitását megvalósító kódrészek a hibakezelő kódrészekkel. Ennek következtében a kód zavaros lesz, nehéz lesz karbantartani és nagy a valószínűsége az elfelejtett hibakezelésnek. Konstruktorban például más módon lehetetlen is hibát jelezni, hiszen nem tud hibakódot visszaadni.

A hibakódok kezelése nem jól skálázódik, az exception-használat viszont igen. Ez egy 10 soros kódban nem fog látszódni, de 10000 sornál már látszódik a különbség.

Gyakori, hogy a hibát nem abban a függvényben kell kezelni, ahol ténylegesen keletkezett, hanem egy hosszú hívás lánc másik végén. Ezt exceptionökkel könnyű kezelni, viszont a hibakódok vizsgálatát minden egyes sziten el kell végezni, ami ráadásul teljesen felesleges. A különbséget az alsó táblázatban látható példakódok szemléltetik:

Exception	Error code

`void f1() {`
`try {`
`// ...`
`f2();`
`// ...`
`} catch (const some_exception& e) {`
`// ...hiba kezelése...`
`}`
`}`
`void f2() { ...; f3(); ...; }`
`void f3() { ...; f4(); ...; }`
`void f4() { ...; f5(); ...; }`
`void f5() {`
`// ...`
`if (/`
`...hiba feltétel...`
`/)`
`throw some_exception();`
`// ...`
`}`
	`int f1() {`
	`// ...`
	`int rc = f2();`
	`if (rc == 0) {`
	`// ...`
	`} else {`
	`// ...hiba kezelése...`
	`}`
	`}`
	`int f2() {`
	`// ...`
	`int rc = f3();`
	`if (rc != 0)`
	`return rc;`
	`// ...`
	`return 0;`
	`}`
	`int f3() {`
	`// ...`
	`int rc = f4();`
	`if (rc != 0)`

Exception	Error code

	`return rc;`
	`// ...`
	`return 0;`
	`}`
	`int f4() {`
	`// …`
	`int rc = f5();`
	`if (rc != 0)`
	`return rc;`
	`// ... return 0;`
	`}`
	`int f5() {`
	`// ...`
	`if (/`
	`...hiba feltétel...` `/)`
	`return some_nonzero_error_code;`
	`// ...`
	`return 0;`
	`}`

Viszont soha ne használjunk exceptiont visszatérési érték átadására!

Az exception használata

A throw kulcsszóval tudunk exceptiont dobni bárhol a kódban, amikor egy probléma felmerül. A catch-csel jelölt blokkban kaphatjuk el és kezelhetjük a dobott exceptiont. A try egy védett blokkot jelent. Az itt keletkező hibákat a try blokkot követő catch blokk(ok)ban kezeljük.

Ha a hívási fában találunk egy hibafeltételt, throw-val hibát dobunk. A throw-nak paramétert kell adni, ami lehet beépített típusú változó vagy tetszőleges osztálybeli objektum. A throw olyan, mint egy return, addig ugrál felfelé a hívási fában, amíg egy megfelelő catch el nem kapja (aminek a paramétere kompatibilis a dobott típussal).

A catch blokkban definiált kódrész csak akkor fut le, ha történt hiba. Ha azt szeretnénk, hogy minden hibát elkapjon, akkor ezt a catch(...) szintaxissal tehetjük meg. Ennek a használata viszont a hibakezelési elveket betartva ritka, mivel információt kell szolgáltatnunk a felhasználónak a hiba keletkezésének okáról és lehetséges megoldásának módjáról. Ezt viszont nehéz úgy megtenni, hogy nem tudjuk, milyen hibát kaptunk el.

Ha nem keletkezik hiba a try blokkban, vagy hiba esetén lefut egy catch blokk, a végrehajtás a try-catch blokk utántól folytatódik. Try-catch blokkot helyezzünk minden olyan kódrész köré, ami hibát dobhat, és az adott helyen azt szeretnénk kezelni.

Példa

```
try {
    // védett kód
    func1();
    func2();
} catch( const ExceptionName1& e1 ) {
    // catch block 1 ...
} catch( const ExceptionName2& e2 ) {
    // catch block 2 ...
} catch( const ExceptionName3& eN ) {
    // catch block N ...
}
func3();
```

A fenti kódrészletben ha a try blokk függvényhívásai során nem keletkezik hiba, a catch blokkok lefutása nélkül a program végrehajtása a func3() hívásra ugrik. Tehát lefut a func1(), func2(), func3().

Tegyük fel, hogy most a func1() függvényben exception dobódik. Ekkor a func1() azonnal visszatér, és a func2() lefutása nélkül a megfelelő catch blokkra ugrik. Ha viszont egy ExceptionName2 típusú hiba keletkezett, a második catch blokk fog lefutni, ahol pl. kiírjuk a megfelelő hibaüzenetet a felhasználónak. Ezt követően a végrehajtás a func3() hívással folytatódik. Tehát lefut a func1() valamennyi része (a hibáig), a 2. catch blokk és a func3().

Egymásba ágyazás és hiba újradobása

Lehetőségünk van try-catch blokkok egymásba ágyazására, tehát egy try blokkon belül lehet másik try blokk is. A belső catch blokkból tovább tudunk dobni egy hibát a throw; szintaxissal. Ekkor nem kell paramétert megadni a throw-nak. Ilyet csak catch blokkon belül tehetünk meg. Catch blokkon belül is lehet akár új hibát is dobni, de ha egy try

blokkhoz tartozó valamelyik catch blokkba már beléptünk, akkor az ugyanahhoz a try blokkhoz tartozó másik catch blokkba már nem fog belépni.

Ennek legfőképp akkor van gyakorlati haszna, ha a belső try-catch blokkban bekövetkező hiba esetén is szeretnénk lefuttatni egy kódrészletet, de azt is védett try-catch blokkban.

Például:

```
try {
    try {
        // kód
    }
    catch (int n) {
        throw;
    }
    // kód
}
catch (int e) {
 // hiba kezelés
}
```

Típusosság

A C++ kivételek típusosak, a típusuk alapján tudjuk elkapni őket. Hiba esetén a try blokk utáni catch blokkok sorrendben lesznek ellenőrizve, hogy a paraméter típusa kompatibilis-e a hiba típusával.

Megáll a végrehajtás és belép a catch blokkba, ha catch(...)-ot talál, ami mindent megfog. Nagy hátránya, hogy nem kapjuk meg paraméterben az exception objektumot.

A catch paraméter típusa egyezhet pontosan a kivétel objektum típusával (int, double, Person). Ez úgy viselkedik, mint egy paraméterátadás függvénynek, másolat készül az objektumról.

A catch paramétere lehet referencia vagy konstans referencia ugyanarra a típusra vagy az osztály ősére. Ezt a megoldást preferáljuk általában, mert például 20 catch blokkot le

tudunk fedni egy catch blokkal, ami egy olyan paraméterű hibaüzenetet vár, amiből az előző 20 catch blokk paramétere öröklődik.

Ennek megfelelően van egy beépített osztályunk (std::exception), amiből sok más, hasznos exception osztály öröklődik. Ezeket itt találjátok, érdemes jó alaposan megnézni a listát: http://www.cplusplus.com/reference/exception/exception/

Vegyük például az std::out_of_range hibaosztályt. Akkor szokás használni, ha valamilyen adatokat tároló struktúrát (például vektort) túl, illetve alul indexelünk. Laboron többször volt ilyen szituáció, amit így kellett volna megoldani.

Fontos, hogy ha saját hibaosztályt szeretnénk létrehozni, akkor érdemes az std::exceptionből örököltetni, hiszen ha valaki olyan try-catchet ír, ami std::exceptiont kap el, jó okkal feltételezhetjük, hogy a mi custom exceptionünket is el szeretné kapni. Egyébként nem érdemes konkrétan std::exceptiont elkapni, mert az olyan, mintha a szőnyeg alá söpörnénk a problémát.

Triviális, de a félreértés elkerülése végett gondoljuk ezt át: mivel a hibaosztályok is osztályok, nem csak egy szintes, de akár többszintes öröklési fát is készíthetünk velük. Ekkor a fa teteje az std::exception, majd ahogy haladunk le a fában, egyre hibaspecifikusabb hibaosztályokkal találkozhatunk., például: std::exception → network_error → packet_error → invalid_padding_format.

Általános gyakorlat, hogy adat osztályokat (például Person, Vehicle stb.) nem dobunk és nem kapunk el, mert nem hibát képviselnek. Emellett számokkal se szokás dobálózni (pl. throw 5;), mert éppen az volt a célunk az egésszel, hogy megszabaduljunk a semmitmondó hibakódoktól. Ezek helyett érdemes például egy mappát készíteni a projektünkön belül és oda megírni a különböző hibákat reprezentáló hiba osztályainkat (network_error, packet_error stb.), amik persze tartalmazhatnak extra változókat, mint például egy hibaüzenetet vagy érintett változónevet stb. Ezáltal a kód is sokkal olvashatóbbá válik és mindig tudni fogjuk, hogy mit tudunk egyáltalán elkapni és mit nem (nem kell felkészülnünk a 40 féle adat osztályunk repkedésére).

Alapvető szabály, hogy amit dobunk, arról mindig másolat készül, még ha referenciával kapjuk is el a kivétel objektumot. Ennek előnye, hogy nem egy lokális objektumra kapunk referenciát. Viszont ha pointert dobunk egy lokális változóra, akkor az egy rossz megoldás. Ha esetleg new-val hoznánk létre a kivétel objektumot, akkor abba a problémába futhatunk, hogy mikor és ki fogja felszabadítani. Elkapáskor használjunk referenciát vagy konstans referenciát.

A stack kezelése

Amennyiben egy hosszabb hívási láncban keletkezik kivétel, akkor megkezdődik a kivételt elkapó catch keresése. A hívási láncban visszafele lépkedve, ha nem találunk az adott függvényben catch blokkot, ami elkapná a hibát, még egy szintet visszalépünk. Visszalépés előtt viszont a függvény minden lokális változója felszabadul. Természetesen mindnek meghívódik a konstruktora is. Figyelem, a dinamikusan foglalt memóriaterületek nem lesznek felszabadítva!

Destruktorban soha ne dobjunk olyan kivételt, amit nem kapunk el rögtön a destruktorban. Ha egy kivétel közben a destruktorban újabb kivételt dobnánk (amit nem a destruktorban kapunk el), akkor az alkalmazás kilép.

13. előadás

Miért használjunk kivételkezelést?

A program futása során keletkező problémák, hibák kezelésére egyszerű és tiszta lehetőségeket biztosít, és használatukkal kevésbé valószínű, hogy egy hibát nem kezelünk le. Hiba esetén a vezérlés a hibakezelő részre ugrik.

A régi módszer a hibakódok visszaadása és hibaüzenetek kiírása több szempontból is előnytelen. Ekkor keverednek a programunk funkcionalitását megvalósító kódrészek a hibakezelő kódrészekkel. Ennek következtében a kód zavaros lesz, nehéz lesz karbantartani és nagy a valószínűsége az elfelejtett hibakezelésnek. Konstruktorban például más módon lehetetlen is hibát jelezni, hiszen nem tud hibakódot visszaadni.

A hibakódok kezelése nem jól skálázódik, az exception-használat viszont igen. Ez egy 10 soros kódban nem fog látszódni, de 10000 sornál már látszódik a különbség.

Gyakori, hogy a hibát nem abban a függvényben kell kezelni, ahol ténylegesen keletkezett, hanem egy hosszú hívás lánc másik végén. Ezt exceptionökkel könnyű kezelni, viszont a hibakódok vizsgálatát minden egyes sziten el kell végezni, ami ráadásul teljesen felesleges. A különbséget az alsó táblázatban látható példakódok szemléltetik:

Exception	Error code

`void f1() {`
`try {`
`// ...`
`f2();`
`// ...`
`} catch (const some_exception& e) {`
`// ...hiba kezelése...`
`}`
`}`
`void f2() { ...; f3(); ...; }`
`void f3() { ...; f4(); ...; }`
`void f4() { ...; f5(); ...; }`
`void f5() {`
`// ...`
`if (/`
`...hiba feltétel... /)`
`throw some_exception();`
`// ...`
`}`
	`int f1() {`
	`// ...`
	`int rc = f2();`
	`if (rc == 0) {`
	`// ...`
	`} else {`
	`// ...hiba kezelése...`
	`}`
	`}`
	`int f2() {`
	`// ...`
	`int rc = f3();`
	`if (rc != 0)`
	`return rc;`
	`// ...`
	`return 0;`
	`}`
	`int f3() {`
	`// ...`
	`int rc = f4();`
	`if (rc != 0)`
	`return rc;`
	`// ...`
	`return 0;`
	`}`
	`int f4() {`
	`// ...`
	`int rc = f5();`
	`if (rc != 0)`
	`return rc;`
	`// ...`
	`return 0;`
	`}`
	`int f5() {`
	`// ...`
	`if (/`
	`...hiba feltétel... /)`
	`return some_nonzero_error_code;`
	`// ...`
	`return 0;`
	`}`

Viszont soha ne használjunk exceptiont visszatérési érték átadására!

Az exception használata

A throw kulcsszóval tudunk exceptiont dobni bárhol a kódban, amikor egy probléma felmerül. A catch-csel jelölt blokkban kaphatjuk el és kezelhetjük a dobott exceptiont. A try egy védett blokkot jelent. Az itt keletkező hibákat a try blokkot követő catch blokk(ok)ban kezeljük.

Ha a hívási fában találunk egy hibafeltételt, throw-val hibát dobunk. A throw-nak paramétert kell adni, ami lehet beépített típusú változó vagy tetszőleges osztálybeli objektum. A throw olyan, mint egy return, addig ugrál felfelé a hívási fában, amíg egy megfelelő catch el nem kapja (aminek a paramétere kompatibilis a dobott típussal).

A catch blokkban definiált kódrész csak akkor fut le, ha történt hiba. Ha azt szeretnénk, hogy minden hibát elkapjon, akkor ezt a catch(...) szintaxissal tehetjük meg. Ennek a használata viszont a hibakezelési elveket betartva ritka, mivel információt kell szolgáltatnunk a felhasználónak a hiba keletkezésének okáról és lehetséges megoldásának módjáról. Ezt viszont nehéz úgy megtenni, hogy nem tudjuk, milyen hibát kaptunk el.

Ha nem keletkezik hiba a try blokkban, vagy hiba esetén lefut egy catch blokk, a végrehajtás a try-catch blokk utántól folytatódik. Try-catch blokkot helyezzünk minden olyan kódrész köré, ami hibát dobhat, és az adott helyen azt szeretnénk kezelni.

Példa

```
try {
    // védett kód
    func1();
    func2();
} catch( const ExceptionName1& e1 ) {
    // catch block 1 ...
} catch( const ExceptionName2& e2 ) {
        // catch block 2 ...
} catch( const ExceptionName3& eN ) {
// catch block N ...
}
func3();
```

A fenti kódrészletben ha a try blokk függvényhívásai során nem keletkezik hiba, a catch blokkok lefutása nélkül a program végrehajtása a func3() hívásra ugrik. Tehát lefut a func1(), func2(), func3().

Tegyük fel, hogy most a func1() függvényben exception dobódik. Ekkor a func1() azonnal visszatér, és a func2() lefutása nélkül a megfelelő catch blokkra ugrik. Ha viszont egy ExceptionName2 típusú hiba keletkezett, a második catch blokk fog lefutni, ahol pl. kiírjuk a megfelelő hibaüzenetet a felhasználónak. Ezt követően a végrehajtás a func3() hívással folytatódik. Tehát lefut a func1() valamennyi része (a hibáig), a 2. catch blokk és a func3().

Egymásba ágyazás és hiba újradobása

Lehetőségünk van try-catch blokkok egymásba ágyazására, tehát egy try blokkon belül lehet másik try blokk is. A belső catch blokkból tovább tudunk dobni egy hibát a throw; szintaxissal. Ekkor nem kell paramétert megadni a throw-nak. Ilyet csak catch blokkon belül tehetünk meg. Catch blokkon belül is lehet akár új hibát is dobni, de ha egy try blokkhoz tartozó valamelyik catch blokkba már beléptünk, akkor az ugyanahhoz a try blokkhoz tartozó másik catch blokkba már nem fog belépni.
Ennek legfőképp akkor van gyakorlati haszna, ha a belső try-catch blokkban bekövetkező hiba esetén is szeretnénk lefuttatni egy kódrészletet, de azt is védett try-catch blokkban.

Például:

```
try {
    try {
        // kód
    }
    catch (int n) {
        throw;
    }
    // kód
}
catch (int e) {
```

```
    // hiba kezelés
}
```

Típusosság

A C++ kivételek típusosak, a típusuk alapján tudjuk elkapni őket. Hiba esetén a try blokk utáni catch blokkok sorrendben lesznek ellenőrizve, hogy a paraméter típusa kompatibilis-e a hiba típusával.

Megáll a végrehajtás és belép a catch blokkba, ha catch(...)-ot talál, ami mindent megfog. Nagy hátránya, hogy nem kapjuk meg paraméterben az exception objektumot.

A catch paraméter típusa egyezhet pontosan a kivétel objektum típusával (int, double, Person). Ez úgy viselkedik, mint egy paraméterátadás függvénynek, másolat készül az objektumról.

A catch paramétere lehet referencia vagy konstans referencia ugyanarra a típusra vagy az osztály ősére. Ezt a megoldást preferáljuk általában, mert például 20 catch blokkot le tudunk fedni egy catch blokkal, ami egy olyan paraméterű hibaüzenetet vár, amiből az előző 20 catch blokk paramétere öröklődik.

Ennek megfelelően van egy beépített osztályunk (std::exception), amiből sok más, hasznos exception osztály öröklődik. Ezeket itt találjátok, érdemes jó alaposan megnézni a listát: http://www.cplusplus.com/reference/exception/exception/

Vegyük például az std::out_of_range hibaosztályt. Akkor szokás használni, ha valamilyen adatokat tároló struktúrát (például vektort) túl, illetve alul indexelünk. Laboron többször volt ilyen szituáció, amit így kellett volna megoldani.

Fontos, hogy ha saját hibaosztályt szeretnénk létrehozni, akkor érdemes az std::exceptionből örököltetni, hiszen ha valaki olyan try-catchet ír, ami std::exceptiont kap el, jó okkal feltételezhetjük, hogy a mi custom exceptionünket is el szeretné kapni. Egyébként nem érdemes konkrétan std::exceptiont elkapni, mert az olyan, mintha a szőnyeg alá söpörnénk a problémát.

Triviális, de a félreértés elkerülése végett gondoljuk ezt át: mivel a hibaosztályok is osztályok, nem csak egy szintes, de akár többszintes öröklési fát is készíthetünk velük. Ekkor a fa teteje az std::exception, majd ahogy haladunk le a fában, egyre

hibaspecifikusabb hibaosztályokkal találkozhatunk., például: std::exception → network_error → packet_error → invalid_padding_format.

Általános gyakorlat, hogy adat osztályokat (például Person, Vehicle stb.) nem dobunk és nem kapunk el, mert nem hibát képviselnek. Emellett számokkal se szokás dobálózni (pl. throw 5;), mert éppen az volt a célunk az egésszel, hogy megszabaduljunk a semmitmondó hibakódoktól. Ezek helyett érdemes például egy mappát készíteni a projektünkön belül és oda megírni a különböző hibákat reprezentáló hiba osztályainkat (network_error, packet_error stb.), amik persze tartalmazhatnak extra változókat, mint például egy hibaüzenetet vagy érintett változónevet stb. Ezáltal a kód is sokkal olvashatóbbá válik és mindig tudni fogjuk, hogy mit tudunk egyáltalán elkapni és mit nem (nem kell felkészülnünk a 40 féle adat osztályunk repkedésére).

Alapvető szabály, hogy amit dobunk, arról mindig másolat készül, még ha referenciával kapjuk is el a kivétel objektumot. Ennek előnye, hogy nem egy lokális objektumra kapunk referenciát. Viszont ha pointert dobunk egy lokális változóra, akkor az egy rossz megoldás. Ha esetleg new-val hoznánk létre a kivétel objektumot, akkor abba a problémába futhatunk, hogy mikor és ki fogja felszabadítani. Elkapáskor használjunk referenciát vagy konstans referenciát.

A stack kezelése

Amennyiben egy hosszabb hívási láncban keletkezik kivétel, akkor megkezdődik a kivételt elkapó catch keresése. A hívási láncban visszafele lépkedve, ha nem találunk az adott függvényben catch blokkot, ami elkapná a hibát, még egy szintet visszalépünk. Visszalépés előtt viszont a függvény minden lokális változója felszabadul. Természetesen mindnek meghívódik a konstruktora is. Figyelem, a dinamikusan foglalt memóriaterületek nem lesznek felszabadítva!

Destruktorban soha ne dobjunk olyan kivételt, amit nem kapunk el rögtön a destruktorban. Ha egy kivétel közben a destruktorban újabb kivételt dobnánk (amit nem a destruktorban kapunk el), akkor az alkalmazás kilép.

Hasznos tudnivalók amik nem voltak benne az előadásban

Osztály fogalma

- Az osztály egy objektum szerkezetének és viselkedésének a mintáját adja meg, azaz
  - felsorolja az objektum adattagjait a nevük és típusuk megadásával
  - megadja az objektumra meghívható metódusokat (tagfüggvény, művelet) nevük, paraméterlistájuk, visszatérési értékük típusa, és törzsük megadásával
- Az osztály lényegében az objektum típusa: az objektumot az osztálya alapján hozzuk létre, azaz példányosítjuk.
- Egy osztályhoz több objektum is példányosítható: minden objektum rendelkezik az osztályleírás által leírt adattagokkal és metódusokkal.

vec : int[0..m], max : int

erase() : void
putIn(int) : void
mostFrequent() : int

Osztály UML jelölése

- Egy osztály rendelkezik
  - névvel,
  - adattagokkal, (tulajdonság, attribútum, mező)
  - metódusokkal
- Az adattagok és metódusok láthatósága egyenként beállítható
  - kívülről is látható, azaz publikus (*public* +)
  - külvilág elől rejtett: privát (*private* -) vagy védett (*protected* #)

<osztálynév>
<+|-|#> <adattagnév> : <típus> ...
<+|-|#> <metódusnév>(<paraméterek>) : <típus> ...

Bag
- vec : int[0..m] - max : int
+ erase() : void + putIn(e:int) : void + mostFrequent() : int

static kulcsszóról bővebben (angol)