Programozási tételek osztály-sablon könyvtárának leírása

Az itt bemutatott osztály-sablon könyvtár a visszavezetéssel tervezett programok C++-beli megvalósítását támogatja. Ahhoz a programozási módszerhez illeszkedik, amely programozási tételekre vezeti vissza a megoldandó feladatot, és a megoldó programkódhoz a könyvtár elemeinek újrahasznosításával jut el. Ehhez egyrészt objektum-orientált technikákra (objektum összetétel, származtatás, virtuális metódusok felüldefiniálása), másrészt osztály-sablonok példányosítására van szükség.

A könyvtárban alapvetően kétféle osztályt találunk. Egyfelől a különféle programozási tételeket általánosan leíró osztály-sablonokat, másfelől a nevezetes felsorolásokat definiáló osztály-sablonokat.

Egy tipikus felhasználása a könyvtárnak a következő:

1. Egy konkrét feladat megoldásához származtatunk egy osztályt a feladat megoldására alkalmas programozási tétel osztály-sablonjából,
    a. megadva ezen osztály-sablon sablon-paramétereit (köztük a megoldáshoz felsorolandó elemek típusát: `Item`),
    b. felüldefiniálva az osztály-sablon bizonyos virtuális metódusait.
2. Példányosítjuk a fenti osztályt, és ezzel létrehozunk egy tevékenység objektumot.
3. Példányosítunk egy alkalmas felsoroló objektumot. Ennek osztályát vagy közvetlenül a könyvtárból vesszük, vagy magunk implementáljuk a könyvtár Enumerator interfészét megvalósítva a `first()`, `next()`, `current()`, `end()` metódusokat. Ügyelni kell arra, hogy a felsorolt elemek típusa egyezzen meg a programozási tétel által feldolgozott elemek típusával. (Ez az `Item` sablon-paraméter helyébe írt típus).
4. Hozzákapcsoljuk a tevékenység objektumhoz (`addEnumerator()`) a felsoroló objektumot.
5. A tevékenység objektumnak meghívjuk a `run()` metódusát, majd különféle getter-ekkel lekérdezzük a tevékenység eredményét.

Nemcsak a könyvtár felhasználása épül objektum-orientált technológiára, hanem maga a könyvtár is ennek szellemében készült. Például azt a feldolgozási stratégiát, amelyet mindegyik nevezetes programozási tétel követ: nevezetesen, hogy végig kell menni egy felsoroló (legyen ennek a neve mondjuk `enor`) által előállított elemeken és azokat kell feldolgozni, a `Procedure` ősosztály-sablon `run()` metódusában írtuk le általánosan:

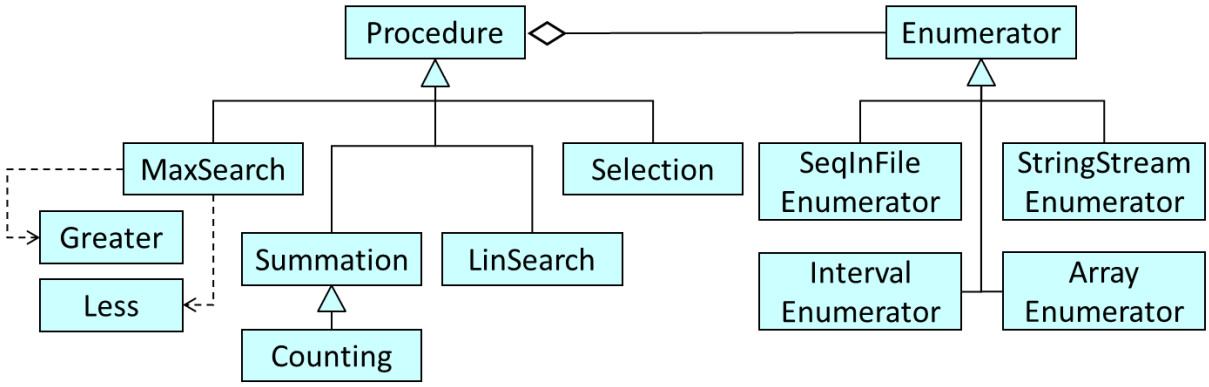
```
init()
while not enor.end() loop
    body(enor.current())
    enor.next()
endloop
```

Ezt örökölik az egyes programozási tételek osztály-sablonjai, de (a sablon-függvény tervminta szerint) felüldefiniálják az `init()` és `do()` metódusokat, ezáltal teszik egyedivé a feldolgozást. Mindeközben újabb virtuális metódusokat vezetnek be, amelyeket egy leszármazott osztályban felülírva az adott programozási tételt rá lehet hangolni a konkrét feladat megoldására.

Az osztály-sablon könyvtár nem az ipari alkalmazások számára készült. Nem hisszük, hogy a gyakorlatban való felhasználása egyszerű, illetve célszerű lenne. Egy programozási tétel (lényegében egy ciklus) implementálása ugyanis önmagában nem túl nehéz feladat, ezért sokkal könnyebb közvetlenül kódolni, mint egy összetett osztály-sablon könyvtárból származtatni, hiszen ehhez a könyvtár elemeit kell pontosan megismerni és helyesen alkalmazni. Ugyanakkor ez a könyvtár nagyon is alkalmas a különféle objektum-orientált implementációs technikák megmutatására. A könyvtár használatával igen szép megoldásokat tudunk előállítani. Programozói szemmel például nagyon érdekes, hogy az előállított megoldásokban mindössze egyetlen ciklus lesz, mégpedig a programozási tételek ősosztály-sablonjának már említett `run()` metódusában, az alkalmazás során hozzáadott kódban pedig egyáltalán nem kell majd ciklust írni.

Osztály-sablon könyvtár szerkezete

Az osztály-sablon könyvtár 14 osztály-sablont és 1 osztályt tartalmaz.



1. ábra. A gtlib osztály-sablon könyvtár szerkezete.

A könyvtárban az `IntervalEnumerator` kivételével mindegyik osztálynak van legalább egy sablon-paramétere. Ez a paraméter a `Greater` és `Less` osztályokat leszámítva a felsorolt, illetve feldolgozott elemek típusa (`Item`).

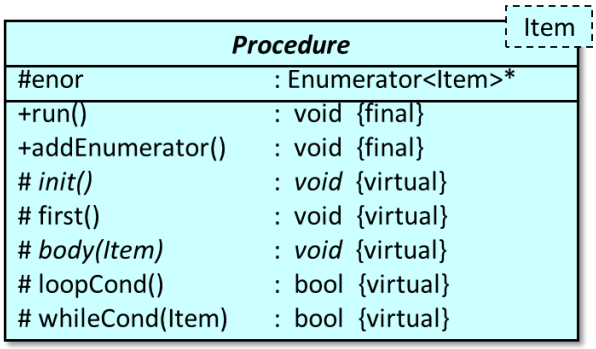
Programozási tételek osztályai	Felsorolók osztályai	Egyéb osztályok
`Procedure, MaxSearch, Summation, Counting, Selection, LinSearch`	`Enumerator, IntervalEnumerator, ArrayEnumerator, SeqInFileEnumerator, StringStreamEnumerator`	`Greater, Less`

1. táblázat. A gtlib oszályai

Tekintsük most át részleteiben az osztály-sablon könyvtár elemeit.

Procedure ősosztály-sablon

A `Procedure` osztály-sablon központi eleme a könyvtárnak, mivel minden programozási tételnek ez az őse. Ezt az osztály-sablont nem használjuk fel közvetlenül, de az itt definiált `run()` és `addEnumerator()` metódusokat mindig.



2. ábra. A programozási tételek ősosztály-sablonja

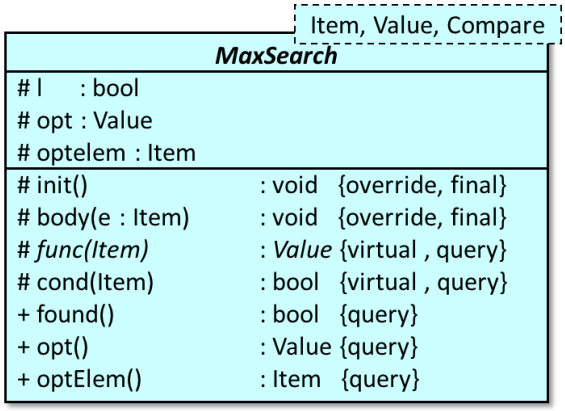
A `run()` metódus közvetve vagy közvetlenül több olyan metódust is meghív, amelyeket majd a származtatás során lehet vagy kell felüldefiniálni. Ezek között az `init()` és a `body()` absztrakt metódusok, a többi rendelkezik alapértelmezett működéssel. Az `addEnumerator()` metódussal a tevékenységhez egy felsoroló objektumot lehet kapcsolni. Ennek elmulasztásakor a `run()` metódus `MISSING_ENUMERATOR` kivételt dob.

metódus	**paraméter**	**vissza**	**tulajdonság**	**működés**
run()	-	void	final, public	init() first() **while** loopCond() **loop** body(enor.current()) enor.next() **endloop**
addEnumerator()	en : Enumerator<Item>	void	final, public	enor := en
init()	-	void	abstract, protected	-
first()	-	void	protected	enor.first()
body()	Item	void	abstract, protected	-
loopCond()	-	bool	query, protected	**not** enor.end() **and** whileCond(enor.current())
whileCond()	Item	bool	query, protected	**return** true

2. táblázat. A `Procedure` osztály-sablon metódusai

MaxSearch osztály-sablon

A maximum keresés osztály-sablonja egymagában írja le a maximum kiválasztás és a feltételes maximum keresés programozási tételeket. Pontosabban fogalmazva, ha nem adunk meg kereséshez feltételt, akkor alapértelmezés szerint minden felsorolt elemet megvizsgál, azaz ilyenkor maximum kiválasztásként működik; azzal a kellemes mellékhatással együtt, hogy üres felsoroláson sem abortál, de lekérdezhető, hogy talált-e egyáltalán megvizsgálandó elemet.



3. ábra. Az általános maximum keresés osztály-sablonja

metódus	paraméter	vissza	tulajdonság	működés
init()	-	void	override, final, protected	l := **false**

body()	Item	void	override, final, protected	**if not** cond(e) **then skip**
				elsif cond(e) **and** l **then**
				  if func(e) > opt **then**
				  opt, optelem := func(e), e
				  endif
				elsif cond(e) **and not** l **then**
				  l, opt, optelem := true, func(e), e
				endif

loopCond()	-	bool	override, final, query, protected	**not** enor.end() **and** whileCond(enor.current())
func()	Item	Value	abstract, query protected	-
cond()	Item	bool	query protected	**return true**
found()	-	bool	query, final, public	**return** l
opt()	-	Value	query, final, public	**return** opt
optElem()	-	Item	query, final, public	**return** optelem

3. táblázat. A `MaxSearch` osztály-sablon metódusai

Az osztály-sablonnak három sablon-paramétere van: a felsorolt elemek típusa (`Item`), a felsorolt elemekhez hozzárendelt értékek típusa (`Value`), amely alapján az elemeket összehasonlíthatjuk, és az elemekhez rendelt értékek (kisebb vagy nagyobb szempont szerinti) összehasonlítását definiáló típus (`Compare`). A `Value` alapértelmezettje az `Item`, a `Compare` alapértelmezettje a `Greater<Value>`, amely a „nagyobb”, azaz a maximum keresés relációját érvényesíti szemben a `Less<Value>`-val, amely a „kisebb”, azaz a minimum keresés relációját definiálja. A `Greater<Value>`, és a `Less<Value>` osztály-sablonokat a gtlib tartalmazza, de saját összehasonlító osztályokat is bevezethetünk, ha azokban felüldefiniáljuk a Value típusra felírt „nagyobb” vagy „kisebb” operátort.

Az osztály-sablon három új publikus gettert biztosít a keresés eredményének lekérdezéséhez, amelyeket az osztály adattagjai tárolnak. A `found()` a közönséges maximum kiválasztásnál is hasznos, mert a hamis értéke mutatja azt, hogy nem volt egyáltalán felsorolt elem, azaz a maximum kiválasztás értelmetlen.

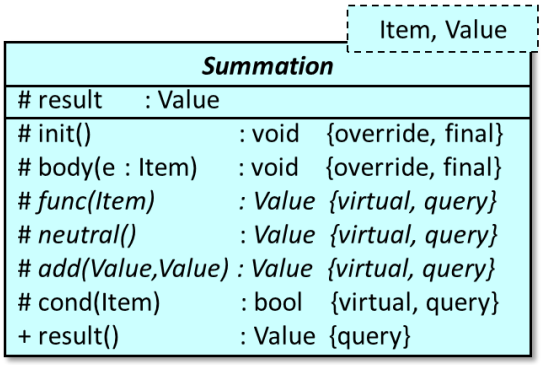
Amikor egy saját maximum (vagy minimum) keresést kell definiálnunk, és ehhez egy saját osztályt származtatunk a `MaxSearch`-ből, akkor a következőkre kell ügyelnünk. A származtatott osztályban felüldefiniálhatók a `cond()`, a `whileCond()` és a `first()` metódusok, és felül kell definiálni a `func()` metódust. Egy ilyen származtatott osztály egy objektumára meghívhatók a `MaxSearch` osztály-sablon nemcsak saját publikus metódusai (a getter-ek), hanem az örökölt (`run()`, `addEnumerator()`) publikus metódusok is.

A származtatás során az alábbiakra ügyeljünk:

1. Tisztán kell látni (és jelölni a sablon-paraméterekkel), hogy mi a feldolgozandó elemek típusa (`Item`), és mi az ezek összehasonlításához használt értékeknek típusa (`Value`).
2. A `func()` metódus absztrakt, azaz ezt minden esetben felül kell definiálni, hiszen ezzel tudjuk megadni azt, hogy egy felsorolt elemhez milyen érték tartozik, amely alapján majd összehasonlíthatjuk őket. A `func()` visszatérési típusa ezért mindig a `Value` paraméter helyébe írt típus, paraméterének típusa pedig az `Item` paraméter helyébe írt típus.
3. A keresés feltételét megadó `cond()` alapértelmezés szerint minden felsorolt elemre igazat ad (ez a maximum kiválasztásos üzemmód), ezért ha feltételes maximum keresésre van szükségünk, akkor a saját maximum keresés osztályunkban a `cond()`-ot a megfelelő feltétel megadásával felül kell írni.
4. A `whileCond()` felüldefiniálásával el tudjuk érni, hogy amikor az ebben megadott feltétel hamis lesz, akkor a keresés megálljon: hamarabb, mint hogy a felsoroló leállna.
5. A `first()` metódust üres törzzsel definiáljuk felül, ha nem akarjuk, hogy a kiválasztás elején a felsoroló first() művelete meghívódjon.
6. A harmadik sablon-paraméterrel csak akkor kell foglalkoznunk, ha minimum keresést akarunk definiálni. Ekkor ide a `Less<Value>` típust írjuk, ahol a `Value` helyén természetesen a `Value` paraméter helyébe írt típus álljon.

Summation osztály-sablon

Az összegzés tételével többféle feladat-típust meg lehet oldani. A szigorúan vett összeadás mellett ilyen lehet az összeszorzás, összefűzés, feltételes összegzés, számlálás, másolás, kiválogatás, szétválogatás. Ezt az általánosságot tükrözi a `Summation` osztály-sablon.



4. ábra. Az összegzés osztály-sablonja

Az `Item` sablonparaméter a feldolgozandó elemek típusára, a `Value` paraméter az összegzés eredményének típusára utal. A `Value` típusú `result` adattag az eredmény tárolására szolgál, amelyet majd a `result()` metódussal kérdezhetünk le.

metódus	paraméter	vissza	tulajdonság	működés
init()	-	void	override, final, protected	result := neutral()
body()	e : Item	void	override, final, protected	**if** cond(e) **then** result := add(result, func(e)) **endif**
loopCond()	-	bool	override, final, query, protected	**not** enor.end() **and** whileCond(enor.current())
func()	Item	Value	abstract, query protected	-
neutral()	-	Value	abstract, query, protected	-
add()	Value, Value	Value	abstract, query, protected	-
cond()	Item	bool	query, protected	**return true**
result()	-	Value	query, public	**return** result

4. táblázat. A `Summation` osztály-sablon metódusai

Az összegzésnek van egy általános és két speciális C++-os változata. A speciális változatokat másolások, kiválogatások, azaz összefűzések végrehajtásához használjuk. Ezek a változatok úgy

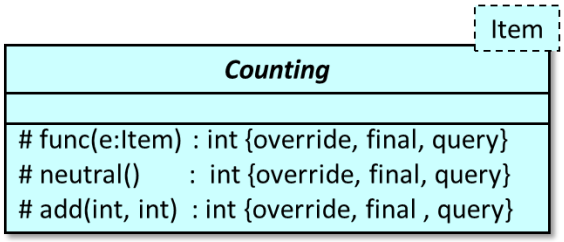
hívhatók elő, hogy a `Value` sablon-paraméter helyébe vagy egy `vector<Value>`-t (a `Value` egy konkrét típussal helyettesítendő), vagy az `ostream`-et írjuk. Előbbi esetben az eredmény egy memóriában tárolt sorozat (`vector`) lesz, utóbbi esetben egy output adatfolyam, amely lehet akár a standard kimenet (`cout`), vagy egy szöveges outputfájlra irányuló `ofstream`. A speciális változatokhoz tartozik olyan konstruktor is, amellyel megadható az output, azaz egyik esetben egy `vector<Value>`-ra történő hivatkozás, másik esetben egy `ostream`-re mutató pointer. Ezekhez fűz majd újabb elemeket a tevékenység végrehajtása. A `vector<Value>`-s változatnál az üres konstruktor is megmarad: ilyenkor a tevékenység hozza létre az eredmény-sorozatot.

Amikor egy összegzést kell definiálnunk, és ehhez egy saját osztályt származtatunk, akkor a következőkre kell ügyelnünk:

1. A `func()` metódus absztrakt, azaz ezt minden esetben felül kell definiálni, hiszen ez adja meg, hogy egy felsorolt elemhez milyen érték tartozik, és ezeket az értékeket kell „összegezni”. A `func()` visszatérési típusa általában a `Value` paraméter helyébe írt típus, kivéve, ha az `ostream`, mert ilyenkor a `func()`-nak egy `string`-et kell visszaadnia, amely majd a kimeneti adatfolyamba kerül.
2. Az összegzéshez meg kell adni az összegzés műveletét annak neutrális elemével együtt, azaz felül kell definiálni a `neutral()` és `add()` absztrakt metódusokat, kivéve, ha az összegzés valamelyik speciális változatát használjuk, mert erre az esetre ezek a metódusok már definiáltak.
3. Az összegzés feltételét megadó `cond()` alapértelmezés szerint minden felsorolt elemre igazat ad. Ezt csak akkor kell felülírnunk, ha feltételes összegzést, például egy kiválogatást kell definiálnunk.
4. A `whileCond()` felüldefiniálásával el tudjuk érni, hogy amikor az ebben megadott feltétel hamis lesz, akkor az összegzés megálljon: hamarabb, mint hogy a felsoroló leállna.
5. A `first()` metódust üres törzzsel definiáljuk felül, ha nem akarjuk, hogy a kiválasztás elején a felsoroló first() művelete meghívódjon.

Counting osztály-sablon

Speciális összegzés a számlálás, de mivel ezt mindenhol önálló programozási tételként tartják számon, a könyvtárunkban is külön osztály-sablonként szerepel, amely persze az összegzés általános változatának leszármazottja.



5. ábra. A számlálás osztály-sablonja

A számlálás az összegzésnél leírt intelmek értelmében felülírja a `func()`, `neutral()`, és `add()` metódusokat, a `Value` paramétert pedig `int`-tel helyettesíti.

metódus	paraméter	vissza	tulajdonság	működés
func()	Item	Value	override, final, query, protected	**return** 1
neutral()	-	Value	override, final, query, protected	**return** 0
add()	a, b : Value	Value	override, final, query, protected	**return** a+b

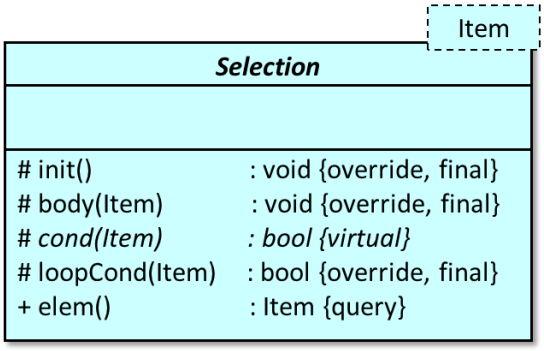
5. táblázat. A `Counting` osztály-sablon metódusai

Egy számláláshoz egy saját osztályt kell származtatunk:

1. A `cond()` metódust mindig kell felüldefiniálni. Ha ezt nem tesszük, akkor ez minden elemre igazat ad, így a `result()` metódus lényegében a számláláshoz használt felsorolás hosszát fogja visszaadni.
2. A `whileCond()` felüldefiniálásával el tudjuk érni, hogy amikor az ebben megadott feltétel hamis lesz, akkor a számlálás megálljon: hamarabb, mint hogy a felsoroló leállna.
3. A `first()` metódust üres törzzsel definiáljuk felül, ha nem akarjuk, hogy a kiválasztás elején a felsoroló first() művelete meghívódjon.

Selection osztály-sablon

A kiválasztás programozási tétele egy biztosan létező tulajdonságú elemet keres egy felsorolásban.



6. ábra. A kiválasztás osztály-sablonja

A kiválasztás nemcsak az `init()` és `body()` metódusokat definiálja felül, hanem a `loopCond()` metódust is, hiszen a feldolgozó ciklus feltétele alapvetően eltér a többi programozási tételétől. A megtalált elem lekérdezését az `elem()` getter végzi.

metódus	paraméter	vissza	tulajdonság	működés
init()	-	void	override, final, protected	**skip**
body()	Item	void	override, final, protected	**skip**
loopCond()	-	bool	override, final, query, protected	**return not** cond(enor.current())
cond()	Item	bool	abstract, query, protected	-
elem()	-	Value	query, public	**return** enor.current()

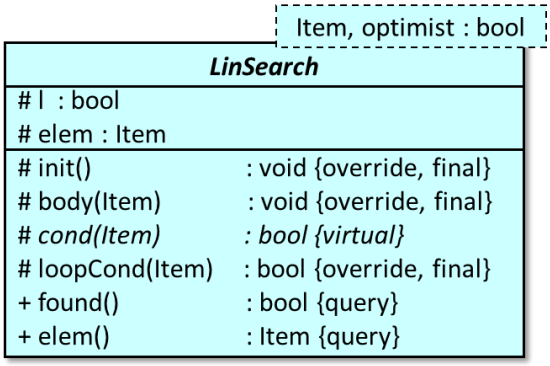
6. táblázat. A `Selection` osztály-sablon metódusai

Egy kiválasztáshoz egy saját osztályt kell származtatunk:

1. A `cond()` metódust mindig felül kell definiálni.
2. A `first()` metódust üres törzzsel definiáljuk felül, ha nem akarjuk, hogy a kiválasztás elején a felsoroló first() művelete meghívódjon.

LinSearch osztály-sablon

A lineáris keresés egy új sablon-paramétert (`optimist`) vezet be az `Item` mellé. Ennek `false`-ra állításával lehet megadni, hogy pesszimista (közönséges) lineáris keresést akarunk-e használni (ez az alapértelmezett), vagy az optimista változatot, amit a paraméter `true` értékével jelölhetünk. Ez az osztálysablon is bevezeti a `cond()` metódust, amellyel a keresési feltétel kell megadni. Ez a metódus absztrakt, tehát egy konkrét lineáris keresés bevezetésekor mindig felül kell definiálnunk.



7. ábra. A lineáris keresés osztály-sablonja

A lineáris keresés nemcsak az `init()` és `body()` metódusokat definiálja felül, hanem a `loopCond()` metódust is, hiszen a ciklus feltétele kiegészül az általános ciklusfeltételhez képest a találatot mutató logikai adattag vizsgálatával. Ennek, illetve a keresés által talált elemet tároló adattagnak az értékét getter-ek segítségével kérdezhetjük le.

metódus	paraméter	vissza	tulajdonság	működés
init()	-	void	override, final, protected	l := optimist
body()	Item	void	override, final, protected	l := cond(e) elem := e

loopCond()	-	bool	override, final, query, protected	**if** optimist **then**
				return l **and**
				Procedure::loopCond()
				else return not l **and**
				Procedure::loopCond()
				endif

cond()	Item	bool	abstract, query, protected	-
found()	-	bool	query, public	**return** l
elem()	-	Value	query, public	**return** elem

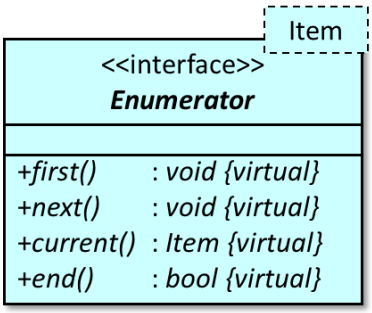
7. táblázat. A `LinSearch` osztály-sablon metódusai

Egy lineáris kereséshez egy saját osztályt kell származtatunk:

1. Be kell állítani a második sablon-paraméterrel, hogy pesszimista (közönséges), vagy optimista lineáris keresésre van-e szükség.
2. A `cond()` metódust mindig felül kell definiálni.
3. A `whileCond()` felüldefiniálásával el tudjuk érni, hogy amikor az ebben megadott feltétel hamis lesz, akkor a keresés megálljon: hamarabb, mint hogy a felsoroló leállna.
4. A `first()` metódust üres törzzsel definiáljuk felül, ha azt akarjuk, hogy a számlálás elején a felsoroló first() művelete ne hívódjon meg.

Enumerator osztály-sablon

Az `Enumerator` absztrakt osztály-sablon interfészt ad a felsoroló objektumok osztályai számára. A bejárt elemek típusát az `Item` sablonparaméter jelzi.

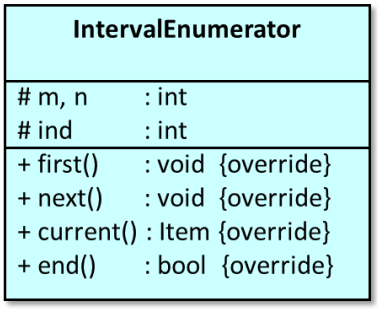


8. ábra. A felsorolók ősosztály-sablonja

Habár a megfelelő felsoroló objektum példányosításához számos esetben magunk definiálunk olyan osztályt, ami megvalósítja az `Enumerator` interfészt, a könyvtár négy speciális felsoroló osztály is bevezet. Egyet egész számok intervallumának felsorolására, egyet tömbök (C++-beli `vector`) elemeinek felsorolására, egyet szöveges állomány (C++-beli `ifstream`) elemeinek felsorolására, és egyet szöveg (C++-beli `stringstream`) elemeinek felsorolására. Ez első esetben a felsorolt elemek típusa `int`, a másik három esetben ez az `Item` sablon-paraméter segítségével adható meg.

IntervalEnumerator osztály

Ez az osztály a könyvtár egyetlen olyan eleme, amelyik nem sablon. Példányát akkor használjuk, ha az egész számok egy intervallumát kell egyesével, növekedő sorrendben bejárni.

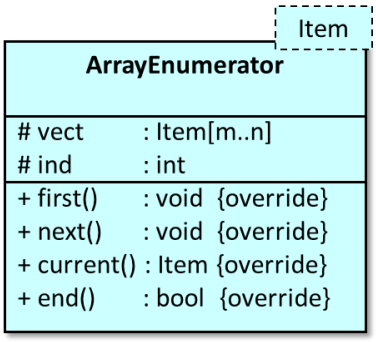


9. ábra. Intervallum felsoroló osztálya

Lényeges adattagjai az intervallum alsó- és felső határát tartalmazó változók, valamint a bejáráshoz használ aktuális index. Konstruktorának meg kell adnia bejárandó intervallum alsó- és felső határát.

ArrayEnumerator osztály-sablon

Tömbök elemeinek bejárását végző felsoroló objektum példányosításához használjuk. A tömbök felsorolása az intervallum felsorolóval is megvalósítható, így az ArrayEnumerator kevésbé fontos eleme a könyvtárnak.

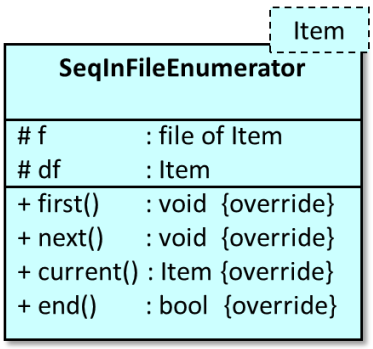


10. ábra. Tömb felsorolójának osztály-sablonja

Lényeges adattagjai a bejárandó tömb (`vector`), valamint a bejáráshoz használt aktuális index. Konstruktorának meg kell adni a bejárni kívánt tömböt, amelynek indextartományát az osztálydiagramban `m` és az `n` jelöli, de C++ esetén ez mindig `0` és `vect.size()-1`.

SeqInFileEnumerator osztály-sablon

Szöveges állomány elemeinek felsorolását végző objektum példányosításához használjuk.



11. ábra. Szekvenciális inputfájl felsorolójának osztály-sablonja

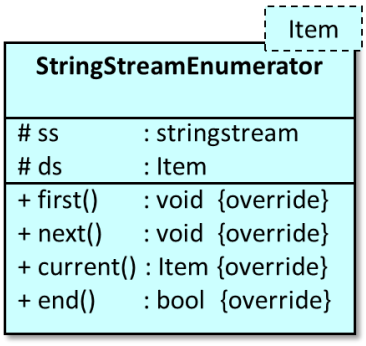
Két konstruktora is van. Az egyikkel meg lehet adni a szöveges állomány fizikai nevét, amelyhez egy C++-beli `ifstream` típusú objektumot hoz létre.; a másikkal közvetlenül az `ifstream` típusú objektumot adhatjuk meg. Adattagjai a szöveges állományra nyitott adatfolyam (`f`), és az aktuálisan beolvasott elem (`df`).

A felsorolás során automatikusan átlépi a szöveges állomány üres sorait, amely különösen akkor hasznos, ha soronkénti felsorolást valósítunk meg. Ha az `Item` paraméter a `char`, azaz karakterenkénti felsorolást végzünk, akkor kikapcsolja a white-space (szóköz, tabulátor-jel, sorvége-jel) ellenőrzést, azaz nem lépi át automatikusan ezeket.

Amennyiben az `Item` helyébe írt típusra nincs értelmezve beolvasó operátor (`operator>>()`), akkor ilyen operátort definiálnunk kell.

StringStreamEnumerator osztály-sablon

Egy szöveg elemeinek felsorolását végző objektum példányosításához használjuk.



12. ábra. Sztring adatfolyam felsorolójának osztály-sablonja

Konstruktorával kell megadni azt a C++-beli `stringstream` típusú objektumot, amelybe előzetesen elhelyeztük a felsorolni kívánt szöveget. Adattagjai a szövegest tartalmazó szövegfolyam (`ss`), és az aktuálisan beolvasott elem (`ds`).

Amennyiben az `Item` helyébe írt típusra nincs értelmezve beolvasó operátor (`operator>>()`), akkor ilyen operátort definiálnunk kell.