1. Tagfüggvények és konstruktorok

Ez az anyag segít a tagfüggvények, nem tagfüggvények, konstruktorok (röviden ctor-ok), operátorok helyes megírásában, és alkalmazásában.

Operátorok megadása tag és nem tagfüggvényként (@ tetszőleges operátor).

kifejezés	tag operátor hívása	nem tag operátor hívása
@a	(a).operator@()	operator@(a)
a@b	(a).operator@(b)	operator@(a,b)
a = b	(a).operator=()	
a[b]	(a).operator[](b)	
a->	(a).operator->()	
a@	(a).operator@(0)	operator@(a,0)

1.1. Taginicializáló lista

Konstruktorban a tagok inicializálását megoldhatjuk értékadással vagy taginicializáló listán keresztül.

```
class NamePtr {
   String name;
   int* p;
public:
   NamePtr(const String& n, int *ptr ) {  name = n; p = ptr; }
};
```

vagy

```
class NamePtr {
   String name;
   int* p;
public:
   NamePtr(const String& n, int *ptr ): name(n), p(ptr) {}
};
```

Ha lehet, akkor a taginicializáló listát kell használni, mivel ez hatékonyabb. Referencia vagy konstans tagváltozókat csak taginicializáló listán keresztül lehet inicializálni. Ha elhagyjuk a taginicializáló listát, akkor a fordító az alapértelmezett konstruktorral létrehozza a tagváltozókat.

Készítsünk egy 0-tól indexelhető tömbből egy Pascal vagy Ada programozók által kedvelt, megadható alsó és felső indexszel rendelkező tömböt. Ezt C++-ban az index operátor felülírásával könnyen megtehetjük, jelenleg azonban csak a ctor-ra koncentráljunk.

```
class Array {
public:
   Array(int meret);
 ......
};

class PascalArray {
    Array data;
    int   size;
    int   lb, hb;
public:
   PascalArray(int lowBound, int highBound) :
      size(highBound-lowBound+1),
      lb(lowBound), hb(highBound),
      data(size) {}      // HIBA !!!
 ......
};
```

A fent leírt `PascalArray` ctor hibás, mivel az inicializálás a változók osztálybeli deklarációjuk sorrendjében, és nem a taginicializáló lista sorrendjében történik.

A változók megszüntetése az osztálybeli deklaráció sorrendjével ellentétes irányban zajlik.

1.2. Értékadó operátor

Általános szabály, hogyha lehet az operátorokat úgy kell megadni, hogy hasonlóan működjön, mint C-ben.

```
class C {
public:
   C(int i=0) {}
   C(const C& x) {}
   C& operator=(const C& x ) {};
  ......
};
C a, b, c;
a = b = c;       // Ezért nem void a visszatérési érték,
                 //  hogy az értékadást láncolhassuk.
a.operator=( b.operator=( c ) ); // Előző sorral azonos.
( a = b ) = c; // Ezért referencia és nem konst a viszatérési
a = b = 1;       // Ezért konstans a paraméter, mert minden
                 // temporális objektum konstans érték.
```

Mit adjon vissza az értékadó operátor?

```
class C {
public:
   C& operator=(const C& x ) {   .....   return *this;    };
  ......
};
```

vagy

```
class C {
public:
   C& operator=(const C& x ) {    .....     return x;   };
  ......
};
```

Természetesen az első (`return *this`) megoldás a jó, mivel constans referenciát nem adhatunk vissza nem konstansként.

Értékadásnál ne feledkezzünk el az önértékadás ellenőrzéséről.

```
class String {
   int len;
   char* p;
public:
   String(const String& s) {
       len = s.len;  p = new char[len+1]; strcpy(p, s.p);
   }
   ~String() { delete [] p; }
   String& operator=(const String& s ) {
      if( this != &s ) {
         delete [] p; // konstruktorban nincs
         len = s.len;  p = new char[len+1]; strcpy(p, s.p);
      }
      return *this;
   };
   ......
};
```

Egyszerűbben megírhatjuk az értékadó operátort, ha van olyan függvényünk, ami felcserél két objektumot.

```
class String {
   char* p;
   int len;
public:
   String(const String& s) { .... }
   ~String() {.... }
   void swap( String& s ) {
      char* tmp_p = s.p;
      int tmp_len = s.len;
      s.p = p; s.len = len;
      p=tmp_p; len= tmp_len;
   }
   String& operator=(const String& x ) {
      String tmp(x);
      swap(tmp );
      return *this;
   };
 ......
};
```

vagy még egyszerűbben:

```
class String {
   char* p;
   size_t meret;
public:
   String(const String& s) { .... }
   ~String() {.... }
   void swap( String& s ) { ... }

   String& operator=( String x ) {
      swap(x);
      return *this;
   };
 ......
};
```

1.3. Mikor használunk tagfüggvényt, és mikor nem tagfüggvényt

```
class Tort {
   int n; // szamlalo
   int m; // nevező
public:
  Tort(int n=0, int m=1 ) {this->n =n; this->m =m; }
  const Tort operator*( const Tort& rhs  ) const   {
     return Tort( n * rhs.n, m * rhs.m );
  }
  int szamlalo() const { return n; }
  int nevezo() const { return m; }
 ......
};
Tort a(1,8), b(1,2);
Tort c = a * b;    // OK.
c = a * 2; // OK. c.operator=( a.operator*( Tort(2) ) );
c = 2 * a; // HIBA. c.operator=( 2.operator*( a ) );
```

Szorzás megvalósítása nem tagfüggvényként.

```
class Tort {
   int n; // szamlalo
   int m; // nevező
public:
  Tort(int n=0, int m=1 ) {this->n =n; this->m =m; }
  int szamlalo() const { return n; }
  int nevezo() const { return m; }
 ......
};

const Tort operator*(const Tort& lhs, const Tort& rhs  ) {
   return Tort( lhs.n * rhs.n, lhs.m * rhs.m );
}

Tort a(1,8);
Tort b(1,2);
Tort c = a * b;    // OK.
c = a * 2; // OK. c.operator=( operator*(a, Tort(2) ) );
c = 2 * a; // OK. c.operator=( operator*( Tort(2), a ) );
```

- Virtuális függvény csak tagfüggvény lehet.
- Olvasáshoz és íráshoz az (`operator»`, `operator«`) operatorokat nem szabad tagfüggvényként létrehozni! Miért?
- Bal szélső paraméteren, ha típuskonverziót akarunk, nem tagfüggvényt kell használni.
- Minden más esetben célszerűbbnek látszik a tagfüggvény használata.

Azért használtunk visszatérési értéknek konstanst, hogy a következő kifejezést ne lehessen megadni.

```
(a*b)=c;
```

1.4. Kerüljük az adattagok átadását nyilvános felületen

```
class String {
   char*p; int len;
public:
   const String operator+( const String& s) {} //összefűzés
   const char * c_str() const {return p;}
   .....
}
String s1("alma"); String s2("korte");
const char*p = (s1+s2).c_str();
cout << p;  // BAJ VAN !!!
```