C++
Gyakorlat jegyzet
6. óra.

A jegyzetet Umann Kristóf készítette Horváth Gábor gyakorlatán. (2017. január 6.)

0.1. Értékadó operátor

Az előző órán elkészült másoló konstruktor megoldotta a probléma egy részét, de az értékadással hasonló problémák merülnek fel. Hasonlóan a fordító által generált másoló konstruktorhoz, az alapértelmezetten az értékadás operátor (*assignment operator*, vagy röviden =) is meghívja az egyes tagok értékadó operátorait, amik a primitív típusok esetén bitről bitre másolnak.

```
struct List
{
    // ...

    // copy constructor
    List(const List &other) : data(other.data), next(0)
    {
        if (other.next != 0)
            next = new List(*other.next);
    }

    // assignment operator
    List& operator=(const List &other)
    {
        delete next;
        data = other.data;
        if (other.next) // if(other.next) == if(other.next != 0)
            next = new List(*other.next);
        else
            next = 0;
        return *this;
    }

    //...
};
```

Az új értékadás operátor először kitörli az aktuális lista elemeit, majd rekurzív módon lemásolja az `other` elemeit.
Kicsit részletesebben: az első lépés a `delete next;`, a fejelemet leszámítva felszabadítja az összes többi listaelemhez tartozó memóriaterületet, eztán lemásoljuk az `other` fejelemében lévő adatot. Végül a másoló konstruktorhoz hasonlóan rekurzív módon lemásoljuk az `other` farokrészét (amennyiben az létezik).
Azáltal, hogy egy listaelem másolásakor az összes listaelem által birtokolt (heapen lévő) objektum is másolásra kerül, elértük azt, hogy a másolaton végzett módosításoknak ne legyen hatása az eredeti objektumra.

Figyeljük meg, hogy az értékadás operátorban egy az adott objektumra mutató referenciával tértünk vissza. Ennek oka az, hogy szeretnénk, ha lehetséges lenne az értékadások láncolása:

```
int a = b = c = d = 0; // a = (b = (c = (d = 0)))
```

Itt rendre (az operátorok kiértékelési sorrendjét tekintve) `d`-re, `c`-re, `b`-re, `a`-ra mutató referenciát ad vissza az értékadás operátor, így a végeredmény az lesz, hogy mindegyik változót 0-ra inicializáltuk.

0.1.1. Megjegyzés. Miért nem konstans referenciával térünk vissza? Akkor nem csinálhatnánk hasonlót: (legyen `f(int &)`) `f(a = 0)`.

Egyszerűsítsük a lista kiírását is!

```
struct List
{
    //...
```

```
    void print ()
    {
        std::cout << data << ' ';
        if (next)
            next->print();
    }

    //...
};
```

Ellenőrizzük, hogy az eredeti elemek nem változtak a másolás következtében!

```
int main ()
{
    List head(7);
    head.add(8);
    head.add(2);
    {
        List cHead = head;
    }
    head.print();
}
```

Kimenet: `7 8 2`
Elértük, hogy biztonságos legyen a lista használata? Most már látszólag a másolás és az értékadás sem okozhat memóriakezeléssel kapcsolatos problémát. Van egy eset, amire a kód még nincs felkészítve! Mi történik, ha önmagának adjuk értékül a listát? Az első lépésben törlésre fog kerülni a fejelemet leszámítva minden elem. Eztán a következő lépésekben már felszabadított memóriaterületetről próbáljuk átmásolni a megfelelő adatokat. A `cHead = cHead` tehát use-after-free hibát fog okozni. Mennyire valós a félelem ettől a hibától? Hiszen nem gyakran adunk értékül egy objektumot önmagának! Tekintsük a következő függvényt:

```
void f(List &l1, List &l2)
{
    //...
    l1 = l2;
    //...
}
```

Ha valaki ugyanazt a listát adja meg mindkét paraméternek, akkor máris megvan a baj. Látható tehát, hogy ilyen jellegű hibát nem feltétlen olyan nehéz elkövetni. Módosítsuk az értékadás operátorunkat oly módon, hogy ne okozzon problémát az önértékadás:

```
List& operator=(const List &other)
{
    if (this == &other) return *this;
    delete next;
    if (other.next)
        next = new List(*other.next);
    else
        next = 0;
    return *this;
}
```

Emlékezzünk, a `this` kulcsszó segítségével tudunk rámutatni tagfüggvényen belül az adott objektumra, amin a tagfüggvény meghívásra került. Tulajdonképpen arról van szó, hogy például a `head.print()` esetében enélkül a kulcsszó nélkül nem tudnánk `print()`-en belül `head`-re hivatkozni. Az objektum neve tagfüggvényen belül nem elérhető. A `this` egy pointerként is felfogható, mely a `head` objektumra mutat.

Sikeresen létrehoztunk egy félig *reguláris típust.*

Egy `T` típus **reguláris típus**, ha van neki:

1. default konstruktora

2. destruktora
3. értékadás operátora, továbbá

```
T b;
T a;
a = b;
```

   Ekkor `a` ekvivalens `b`-vel, és ha `a`-t módosítjuk, akkor `b` nem változik, és viszont.

4. másoló konstruktora, továbbá

```
T b;
T a = b;
```

   Ekkor `a` ekvivalens `b`-vel, és ha `a`-t módosítjuk, akkor `b` nem változik, és viszont.

5. egyenlőségvizsgálat (`operator==`) (Ha nincs, akkor az adott típust félig regulárisnak nevezzük)

0.2. Adattagok védettsége

Biztonságossá sikerült tenni a lista használatát? A felhasználó most már a megfelelő metódusok, valamint a másolás és értékadás használatával nem tud elkövetni memóriakezeléssel kapcsolatos hibát. Az adattagokhoz azonban továbbra is hozzá fér. Így, a következő módon továbbra is érhetik meglepetések:

```
int main()
{
    List head(8);
    head.add(7);
    head.add(2);
    head.next = 0;
}
```

Itt az első listaelem utáni elemeket lecsatoljuk, és azoknak a memóriája elszivárog. A probléma forrása az, hogy a **felhasználó hozzáfér az adattagokhoz.** A felhasználó így olyan állapotba állíthatja az adott objektumot, amire az objektum tagfüggvényei nincsenek felkészülve. Másik probléma, hogy a felhasználó azáltal, hogy hozzáfér az adattagokhoz, ki tudja használni a lista belső reprezentációját. Emiatt nehezebben fenntarthatóvá válik a kód, mivel a lista reprezentációját érintő változtatások esetén a felhasználói kódokat is módosítani kell. Példaképp, ha a `List next` adattagjának új nevet adunk, akkor át kéne írni minden olyan kódot ami `next`re hivatkozik.
Jó lenne, ha hiába változna a belső reprezentáció, a kódnak nem kéne a felhasználói kódnak változnia.
Rejtsük el az adattagokat a felhasználó elől!

```
class List // lehetne struct is
{
public:
    List(int data, List *next = 0) : //...
    void add(int data) {//...}
    List(const List &other) //...
    List& operator=(const List &other) {//...}
private:
    int data;
    List *next;
};
```

Az osztály minden tagja, mely a `public` kulcsszó után jön, elérhető bárki számára. A `private` kulcsszó után következő adatok csak is kizárólag az adott osztályon belül érhető el (leszámítva a `friend`eket, de erről később). A `class` és a `struct` abban tér el, hogy alapértelmezetten a `struct` minden tagja publikus, míg `class`-nak privát. A `public` ill. `private` kulcsszóval mind a kettőnél explicit megadhatjuk a láthatóságot.

Van egy súlyos következménye az imént bevezetett adatrejtésnek. Például, így nem tudunk hozzáférni a második elemhez kívülről. Vagy bármelyik másikhoz. Ilyenkor a naív megoldás az, ha módosítjuk a listát, és minden olyan függvényt, aminek hozzá kéne férnie az elemekhez, metódussá tesszük. Hiszen a lista metódusai hozzáférnek a privát adattagokhoz is.

Ezt ugyanakkor nem akarjuk minden egyes függvénnyel megtenni. A végén nagyra nőne a lista osztály, ami nagyban rontja az kód érthetőségét és fenntarthatóságát. Emellett a reprezentáció változtatásával megint sok függvényt kellene módosítani.
Írjunk felsorolót, amivel hozzáférhetünk az elemekhez kívülről is.

```
class List
{
    //...

    void First()
    {
        cursor = this;
    }
    int& Current()
    {
        return cursor->data;
    }
    void Next()
    {
        cursor = cursor->next;
    }
    bool End()
    {
        return cursor == 0;
    }

private:
    int data;
    List *next;
    List *cursor;
};
```

Figyeljük meg, hogy `Current()` referenciával tér vissza, hogy ne `data` másolatát, hanem a `data`-ra hivatkozó referenciát kapjunk meg, így tudjuk a lista elemeinek az értékeit módosítani.

Első látásra a problémát orvosoltuk, tudunk a listákkal tárolt adatokkal dolgozni (pl. összeadni őket, stb). De egy rendezésnél már problémásabb. Ha csak egy kurzorunk van, nem tudunk egyszerre 2 elemhez hozzáférni, hogy összehasonlítsuk őket vagy megcseréljük őket. Egy lehetséges megoldás:

```
class List
{
    //...

    void First()
    {
        cursor = this;
    }
    int& Current()
    {
        return cursor->data;
    }
    void Next()
    {
        cursor = cursor->next;
    }
    bool End()
    {
        return cursor == 0;
    }

    void First2()
```

```
    {
        cursor2 = this;
    }
    int& Current2()
    {
        return cursor2->data;
    }
    void Next2()
    {
        cursor2 = cursor2->next;
    }
    bool End2()
    {
        return cursor2 == 0;
    }

private:
    int data;
    List *next;
    List *cursor;
    List *cursor2;
};
```

Ezzel már egy rendezést meg tudunk oldani. De talán érezhető, hogy ez nem a legszebb megoldás. Ha három `cursor` kéne egy algoritmushoz, akkor megint bajban lennénk. Ennél létezik sokkal szofisztikáltabb megoldás.

0.3. Iterátorok

Az iterátorok a pointerek általánosításai, segítségükel tudunk végigiterálni egy konténer elemein (azaz velük tudjuk lekérdezni a konténer elemeit). Ehhez szükségünk van arra, hogy tudjuk hol kezdődik és végződik a konténerünk. E láncolt lista esetében hozzá tudunk férni az első elemhez (az lesz ugye `head` a fenti példákban) és tudjuk hogy mindig nullpointerrel az utolsó elemben lévő mutató értéke. Legyen az iterátunk neve `Iterator`!

```
class List
{
    //...
public:
    Iterator begin()
    {
        return Iterator(this);
    }
    Iterator end()
    {
        return Iterator(0);
    }
};
```

E két metódus segítségével már meg tudjuk adni a lista elejét és végét! A `head.begin()` vissza fog adni egy iterátort, ami a `head` a legelső elemére mutat, `head.end()` az utolsó utáni elem. Már csak magát az iterátort kell megírni, mely egy un. *forward iterator* lesz.

Egy `T` típus **forward iterator**, ha rendelkezik:

1. `++` operátorral
2. egyenlőség vizsgáló operátorral `==`
3. egyenlőtlenség vizsgáló operátorral `!=`
4. dereferáló operátorral `*`

De hova írjuk az `Iterator` osztályt? Hiszen ha a lista elé tesszük, az `Iterator` nem fogja tudni, hogy mi az a `List`. Ha utána tesszük, nem fogja tudni a `List` hogy mi az az `Iterator`. Erre szükség van a Listában az `end` és a `begin` metódus megírásához. A körkörös függőség feloldása érdekében **forward deklarálni** fogunk.

```
class List; //forward declaration

class Iterator
{
public:
    Iterator(List *_p) : p(_p) {}
    bool operator==(Iterator &other)
    {
        return p == other.p;
    }
    bool operator!=(Iterator &other)
    {
        return !(*this == other);
    }
    //...
private:
    List *p; //(*)
};

class List {//...};
```

A fordító ezek után a csillaggal jelölt sorban nem fog arra panaszkodni, hogy a `List` egy ismeretlen azonosító. A forward deklaráció azt mondja a fordítónak, hogy a `List` később, akár egy másik fordítási egységben definiálva lesz. Amennyiben nem akarjuk dereferálni `p`-t, elég csupán a forward deklarálni!

Még hiányzik a `++` és a `*` operátor.

```
class List;

class Iterator
{
public:
    Iterator(List *_p) : p(_p) {}
    bool operator==(Iterator &other)
    {
        return p == other.p;
    }
    bool operator!=(Iterator &other)
    {
        return !(*this == other);
    }
    Iterator operator++()
    {
        p = p->next;
        return *this;
    }
    int& operator*()
    {
        return p->data;
    }
private:
    List *p;
};

class List {//...};
```

Amennyiben az előbb említett operátorokat az `Iterátor`on belül fejtjük ki, a fordító dob egy hibaüzenetet, miszerint a `List` egy un. *incomplete type*. A forward deklaráció miatt a fordító tudja már, hogy a `List` az egy osztály. De nem tudja ellenőrizni, hogy `next` adattaggal rendelkezik-e. Ezért ezeket a tagfüggvényeket a `List` kifejtése után kell írni.

```
class List;

class Iterator
{
public:
    Iterator(List *_p) : p(_p) {}
    bool operator==(Iterator &other) const
    {
        return p == other.p;
    }
    bool operator!=(Iterator &other) const
    {
        return !(*this == other);
    }
    Iterator operator++();
    int& operator*();
private:
    List *p;
};

class List {//...};

Iterator Iterator::operator++()
{
    p = p->next;
    return *this;
}
int& Iterator::operator*()
{
    return p->data;
}
```

Felmerül még egy probléma: a `next` és `data` private adattagok. Az `Iterator` nem fér hozzá! Erre megoldás, ha **barát** (*friend*) osztállya lesz az Iterátor a `List`nek.

```
class List
{
    //...
    friend class Iterator;
    //...
}
```

A barátként deklarált osztályok és függvények hozzá tudnak férni az osztály privát adattagjaihoz is.

Az így kapott iterátorunkkal be tudjuk járni a lista elemeit, hozzá tudunk férni a listában tárolt adatokhoz. Így már a `print()`-et tagfüggvény helyett szabad függvényként is megírhatjuk. Amennyiben lehetőségünk van egy függvényt szabad (osztályon kívüli) függvényként megírni, érdemes élni a lehetőséggel. Ezáltal az osztályaink kisebbek és könnyebben érthetőek lesznek.

```
void print(List &l)
{
    for(Iterator it = l.begin(); it != l.end(); ++it)
    {
        std::cout << *it << ' ';
    }
    std::cout << std::endl;
}
```

Ez a függvény, csak úgy mint az *STL* függvények is, balról zárt, jobbról nyitott [ ) intervallummal dolgozik. Azaz az `end()` már nem eleme a listának, az az utolsó utáni elem (*past-the-end iterator*).

Nézzük meg, hogy miért jobb az iterátor, mint a felsoroló. Annyi `Iterator`-t hozunk létre, amennyit csak akarunk, nem vagyunk korlátozva a `cursor`ok darabszáma által. Az iterátorok emellett tetszőlegesen tárolhatóak,

átadhatóak függvényeknek. Nincs elrejtve az objektum belsejébe. Továbbá, tudjuk módosítani a listában található elemeket anélkül, hogy a lista reprezentációját ismernünk kellene. Egy iterátorokat használó kód módosítása nélkül megváltoztatható a lista belső reprezentációja. Ez lehetővé teszi azt, hogy több programozó csapatban dolgozzon, egymástól független kódrészleteket úgy tudjon módosítani, hogy ne kelljen egymás kódját átírni.

Megfigyelhető, hogy `print` nem módosítja a paraméterként kapott listát, így átvehetnénk konstansk referenciával is. Ennek az eredménye fordítási hiba: az `end` és `begin` nem konstans metódusok, hiszen ami iterátort visszaad, azokon keresztül tudjuk módosítani az objektumot.

0.4. Konstans iterátorok

Erre a megoldás, ha konstans iterátort is írunk, ami egy konstansra mutató mutató általánosítása.

```
class List;

class Iterator
{
    //...
};

class ConstIterator
{
public:
    Const Iterator(const List *_p) : p(_p) {}
    bool operator==(ConstIterator &other) const
    {
        return p == other.p;
    }
    bool operator!=(ConstIterator &other) const
    {
        return !(*this == other);
    }
    ConstIterator operator++();
    int operator*() const; //nem referenciával tér vissza!
private:
    const List *p; //konstanra mutató pointer!
};

class List
{
    //...
    ConstIterator begin() const
    {
        return ConstIterator(this);
    }
    ConstIterator end() const
    {
        return ConstIterator(0);
    }

    friend class ConstIterator;
    //...
};

Iterator Iterator::operator++() {//...}
int& Iterator::operator*() {//...}

ConstIterator ConstIterator::operator++()
{
    p = p->next;
```

```
    return *this;
}
int ConstIterator::operator*() const
{
    return p->data;
}
```

Azon metódusok, amik után `const` kulcsszó szerepel (ld. fenti példa) nem tudják megváltoztatni az adott osztály adattagjait. Ezeket a metódusokat **konstans metódusok**nak (*const method*) hívják. Egy konstans objektumon csak akkor lehet meghívni egy metódust, ha az a metódus konstans.

0.4.1. Megjegyzés. Ami lehet `const`, az **legyen** `const`!

Egy `const List` típusú objektumon csak a konstans metódusok hívhatóak meg, így a meghívott `begin()` visszatérési értékének típusa `ConstIterator` lesz. Egy nem konstans `List` objektumnál viszont `Iterator` lesz.

```
int main()
{
    List head(8);
    head.add(7);
    head.add(2);
    {
        ConstIterator cit = head.begin(); //hiba
    }
}
```

Két különböző típust nem tudunk egymásnak értékül adni, ha nem létezik köztük konverzió. Mivel nem konstansra mutató mutató konvertálódhat konstansra mutató mutatóvá, ezért természetes lenne hasonló konverziót az iterátorok közt is bevezetni. Ezt egy új konstruktor segítségével tehetjük meg:

```
class ConstIterator
{
    //...
public:
    ConstIterator(Iterator &other) : p(other.p) {}
    //...
}
```

Emlékezzünk, hogy az `Iterator`-nak a `p` adattagja privát. A forduló kódhoz egy friend deklaráció bevezetése szükséges!
Meglepő lehet, hogy az értékadás egy új konstruktor bevezetését követően le fog fordulni. A `head.begin()` a fenti esetben egy `Iterator`-t ad vissza, mivel `head` nem konstans! Eztán a fordító az imént megírt konstruktor segítségével az `Iterator` típusú változóból készít egy `ConstIterator` típusút. Ezzel *implicit kódon átkonvertálja* `Iterator`-t `ConstIterator`-rá, és utána hívja meg a másoló konstruktort.

0.5. Konverziós operátor

Más útja is létezik annak, hogy át lehessen konvertálni egy `Iterator` típusú objektumot `ConstIterator` típusúvá, ha létrehozunk egy un. **konverziós operátor**t (*conversion operator* vagy *user defined conversion*).

```
// ...
class ConstIterator; //forward deklaráció itt szükséges

class Iterator
{
public:
    // ...
    operator ConstIterator() const;
    // ...
};

class ConstIterator{//...}
```

```
Iterator::operator ConstIterator() const
{
    return ConstIterator(p);
}
// ...
```

A fenti függvény segítségével a fordító végre tudja majd hajtani a konverziót implicit módon.

```
ConstIterator cit2 = head.begin(); //ok
```

Amennyiben azt szeretnénk, hogy a konverzió létezzen, de implicit módon ne lehessen végrehajtani, használhatjuk az `explicit` kulcsszót:

```
class Iterator
{
public:
    // ...
    explicit operator ConstIterator() const;
    // ...
};
```

Ennek következtében explicit módon jeleznünk kell, hogy `ConstIterator` típusra szeretnénk konvertálni.

```
int main()
{
    List head(8);
    ConstIterator cit1 = head.begin(); // hiba
    ConstIterator cit2 = ConstIterator(head.begin()); // ok
    ConstIterator cit3 = static_cast<ConstIterator>(head.begin()); // ok
}
```

0.5.1. Megjegyzés. A fent látható `static_cast`-ról később lesz részletesen szó.

0.5.2. Megjegyzés. A fenti módosítások ismét csak gyakorlás célját képezték, az elkészítedő listának nem lesz része a konverziós operátor.

0.6. Explicit konstruktorok

Azt, hogy az egy paraméteres konstruktorokat a fordító implicit konverzióra felhasználja a fenti módon, megtilthatjuk itt is, ha `explicit` kulcsszót írunk a konstruktor elé.

```
class ConstIterator
{
    //...
public:
    explicit ConstIterator(Iterator &other) : p(other.p) {}
    //...
}
```

Jelen esetben nem akarjuk a konverziót megtiltani, lévén az `Iterator` és a `ConstIterator` hasonló feladatot lát el. Nézzük meg az eddigi kódjainket. Okozhat valahol meglepetést egy konverzió? A `List` egyik konstruktra gyanús lehet. Ha egy függvény listát vár paraméterül, egy darab `int`-et is elfogad, hisz a `List` konstruktorát felhasználva tudott volna csinálni abból az `int`-ből egy egy elemű listát! Ezt elkerülendő, tegyük a `List` konstruktorát `explicit`-té.

```
class List
{
    //...
    explicit List(const int _data, List *_next = 0) : data(_data), next(
        _next) {}
    //...
}
```