

A következő feladatokat azért dolgoztam ki, hogy legyen néhány olyan feladat, ami hasonlít a laborokon végigvittekre, és amelyeknek a kidolgozása magyarázattal együtt szerepel, hogy önállóan tudjatok velük tanulni. A megoldások magyarázatokat is tartalmaznak, majd a végén egy kérdezz-felelek ad mélyebb betekintést a téma körbe.

Javaslom, hogy a feladatokat megértés után először egyedül próbáljátok megoldani. Ha elakadtok, tudtok a részfeladatokhoz „puskázni”. A leghatékonyabb akkor lesz a feldolgozás, ha addig vizsgáld a kódot, amíg a végén önállóan tudod reprodukálni a teljes megoldásokat. Ez esetben a ZH-n sem lesezel elveszve, ha valami hasonló feladat érkezne.

Jó munkát kívánok!

Bertalan

1. Feladat: Varázstárgyak és bájitalok interfésszel

Egy fantasy játékban szeretnénk kezelní **varázstárgyakat** és **bájitalokat**. A játék motorja (amit most egyszerűsítsünk le egy külső függvényre) képes megjeleníteni egy tetszőleges tárgy vagy bájital rövid adatait: például egy nevet vagy egy rövid leírást, illetve egy értéket kiszámolni, például az erejét vagy a gyógyító hatását.

A feladatod:

- Készíts két különböző osztályt: egy **Potion** (bájital) és egy **MagicItem** (varázstárgy) nevűt. Ezek teljesen különálló osztályok legyenek, semmi közös nincs bennük.
- A játék motorja (külső függvény) viszont azt várja el, hogy minden rendelkezzen két funkcióval:
 - Egy, ami visszaad valamilyen **szöveges leírást** (Potionnál pl. egy "leírás", MagicItemnél pl. "név").
 - Egy, ami **számítást végez**, például egy bájitalnál kiszámolja a hatás intenzitását, vagy egy varázstárgynál az erejét.
- A "külső rendszer" előírja, hogy ezeknek a funkcióknak fix szignatúrával elérhetőknek kell lenniük, hogy az objektumokat közös módon lehessen kezelní. Hogyan oldanád ezt meg?

A cél: úgy készítsd el az osztályokat, hogy a "külső rendszer" tudja őket egységesen használni anélkül, hogy tudná, hogy éppen bájittal vagy varázstárggyal van dolga.

1. lépés: Megtervezni az osztályokat

Elsőként létrehozzuk a két teljesen független osztályt: Potion és MagicItem. Egyelőre csak privát adattagokat adunk nekik.

```
#include <string>
```

```
class Potion {
```

```
private:  
    std::string description;  
    int healingPower; // gyógyító hatás mértéke  
  
public:  
    Potion(const std::string& desc, int power)  
        : description(desc), healingPower(power) {}  
};  
  
class MagicItem {  
private:  
    std::string name;  
    int magicStrength; // varázserő mértéke  
  
public:  
    MagicItem(const std::string& n, int strength)  
        : name(n), magicStrength(strength) {}  
};
```

Miért így?

Mert a feladat szerint két teljesen önálló, különböző szemantikájú dolgot kell kezelni. Nem öröklök egymástól, nem is közösek. Ha ezt nem így csinálnánk, elveszne a feladat lényege: különböző típusok egységes kezelése.

2. lépés: Megtervezni a külső rendszer függvényét

Képzeljük el, hogy a játék motorjában van egy függvény, ami bármilyen varázstárgyat vagy bájitalt kiír a képernyőre.

Mivel a tárgyak fajtája változhat, azt akarjuk, hogy a függvény ne tudja előre, pontosan milyen objektumot kezel.

A függvény szignatúrája így nézne ki (egy "absztrakt" ősosztályon keresztül):

```
void displayItemDetails(const IGameEntity& entity);
```

Itt IGameEntity egy absztrakt ősosztály lesz — de ezt még nem csináltuk meg!

Miért van szükség erre?

Azért, mert ha konkrét típusokat (pl. Potion, MagicItem) várna el a függvény, nem tudnánk közös módon kezelní a különböző dolgokat. Úgy kell kialakítani a rendszerünket, hogy elég legyen az "általános viselkedést" előírni.

3. lépés: Megtervezni az absztrakt ősosztályt, amit interfész öröklésre fogunk használni

Most létrehozzuk az absztrakt ősosztályt, IGameEntity néven.

```
class IGameEntity {  
public:  
    virtual ~IGameEntity() = default;  
  
    virtual std::string getDescription() const = 0;  
    virtual int calculatePower() const = 0;  
};
```

Fontos dolgok:

- A = 0 a függvények végén azt jelenti, hogy ezek **tisztán virtuális függvények**, tehát az osztály absztrakt. Nem lehet belőle példányt létrehozni.
- A virtual kulcsszó biztosítja, hogy dinamikus (futásidőbeli) polimorfizmus történjen: tehát a helyes leszármazott osztály megfelelő függvénye fog lefutni.
- A destruktur is virtuális, hogy biztos legyen a helyes objektum-létbontás.

Miért van így?

Ha nem lenne virtual, akkor híváskor nem a konkrét objektum (Potion, MagicItem) saját implementációja futna, hanem az alapértelmezett (ami nincs is). És ha nem lenne tisztán virtuális, akkor az egész interfész használhatatlanná válna erre a célra.

4. lépés: Leszármazni az absztrakt ősosztályból és megvalósítani a virtuális függvényeket

Most kiterjesztjük a Potion és MagicItem osztályt, hogy megvalósítsák az interfészt.

```
class Potion : public IGameEntity {  
private:  
    std::string description;  
    int healingPower;  
  
public:  
    Potion(const std::string& desc, int power)  
        : description(desc), healingPower(power) {}  
  
    std::string getDescription() const override {  
        return description;  
    }  
  
    int calculatePower() const override {  
        // Egyszerű példaként: healingPower kétszerese  
    }
```

```
        return healingPower * 2;  
    }  
};  
  
class MagicItem : public IGameEntity {  
private:  
    std::string name;  
    int magicStrength;  
  
public:  
    MagicItem(const std::string& n, int strength)  
        : name(n), magicStrength(strength) {}  
  
    std::string getDescription() const override {  
        return "Magic Item: " + name;  
    }  
  
    int calculatePower() const override {  
        // Mondjuk itt egy négyzetes skálázást alkalmazunk  
        return magicStrength * magicStrength;  
    }  
};
```

Fontos részletek:

- Bár erről nem beszéltünk órán, de hasznos lehet (ZH-n nem kérem számon): az override kulcsszóval jelezzük, hogy ezek a függvények az ősosztályban definiált tisztán virtuális függvényeket valósítják meg. Így a fordító figyelmeztet, ha elírtuk a szignatúrát.
- A calculatePower minden esetben nem egyszerű tagváltozó visszaadás, hanem kiszámítást végez.

Miért fontos?

Azért, mert így biztosítjuk, hogy bármelyik tárgy más módon tudja megvalósítani az ereje kiszámítását. Ez későbbi bővítéskor (új típusok hozzáadásakor) is nagyon hasznos lesz.

5. lépés: Használat bemutatása

Most írunk egy példát arra, hogyan hozzuk létre az objektumokat, és hogyan használjuk őket a külső függvényben:

```
#include <iostream>
```

```
void displayItemDetails(const IGameEntity& entity) {  
    std::cout << "Description: " << entity.getDescription() << std::endl;  
    std::cout << "Power: " << entity.calculatePower() << std::endl;  
}  
  
int main() {  
    Potion healingPotion("Healing Potion", 25);  
    MagicItem magicSword("Sword of Flames", 10);  
  
    displayItemDetails(healingPotion);  
    displayItemDetails(magicSword);  
  
    return 0;  
}
```

Miért így?

- A displayItemDetails függvény csak egy IGameEntity típusú referenciát vár. Ezért teljesen mindegy neki, hogy valójában egy Potiót vagy egy MagicItemet kapott.
- A referencia (const IGameEntity&) biztosítja, hogy ne legyen fölösleges másolás, és a polimorfizmus működjön.
- Így tudjuk igazán kihasználni az objektum-orientált programozás egyik alapelveit: a **polimorfizmust**.

Most egy második feladatot is adok, ami a **heterogén kollekciók** kezelésére épít. Ezúttal az interfészen keresztül két különböző típusú objektumot (bájitalt és varázstárgyat) szeretnénk egy **kollekcióban** tárolni és kezelní. Ehhez saját, egyszerű tárolási struktúrát építünk, amely mutatókat tartalmaz.

2. Feladat: Áruház létrehozása heterogén kollekcióval

Képzeld el, hogy egy **fantasy áruházat** kell modellezni, ahol az előző példabeli különböző típusú varázstárgyak és bájitalok kerülnek tárolásra. Az áruház feladata, hogy tárolja az egyes termékeket egy **heterogén kollekcióban**, és később a vásárlók számára ki tudja nyújtani az árukat egy-egy leírás és érték alapján.

A feladatot:

1. Hozd létre az **áruházat**, ami egy heterogén kollekcióban tárolja a különböző varázstárgyakat és bájitalokat (max 10-et).
2. A kollekcióban lévő elemeket egy **ős típusú** mutatón keresztül tároljuk, ami az **IGameEntity** interfészre mutat.

3. Az áruház legyen képes:

- Visszaadni minden termék leírását.
 - Kiszámolni és megjeleníteni a termékek erejét vagy gyógyító hatását.
-

1. lépés: Áruház osztály tervezése

Azt szeretnénk, hogy az áruház egy tárolót tartalmazzon, amelyben az **IGameEntity** típusú mutatók lesznek, így többféle objektumot is tárolhatunk benne (Potion és MagicItem).

Az alapvető struktúra tehát így néz ki:

```
class Store {  
  
private:  
  
    IGameEntity* inventory[10]; // Egy egyszerű tároló, ami max 10 elemnek ad helyet  
  
public:  
  
    Store() {  
  
        // Inicializálás  
  
        for (int i = 0; i < 10; ++i) {  
  
            inventory[i] = nullptr; // Kezdetben nincs benne semmi  
        }  
    }  
  
    void addItem(IGameEntity* item, int index) {  
  
        if (index >= 0 && index < 10) {  
  
            inventory[index] = item; // Hozzáadás a kollekcióhoz  
        }  
    }  
  
    void displayItems() const {  
  
        for (int i = 0; i < 10; ++i) {  
  
            if (inventory[i]) {  
  
                std::cout << "Item " << i + 1 << " - Description: " <<  
inventory[i]->getDescription() << "\n";  
  
                std::cout << "Power: " << inventory[i]->calculatePower() << "\n";  
            }  
        }  
    }  
};
```

Miért így?

- A Store osztályban egy egyszerű, **fix méretű tömböt** használunk (**IGameEntity*** inventory[10]). Ez az áruházunk "kollekciója", amely maximum 10 terméket tud tárolni.
- Az addItem függvény hozzáad egy új terméket az áruházba, ha még van hely.
- A displayItems pedig végigmegy az áruházban tárolt termékeken, és mindegyikről kiírja a leírást és a kiszámított erőt.
- Észrevetted? az áruház nem veszi át a termékek tulajdonjogát ebben a példában, mivel az nem is volt előírás. Így a destrukturálásban sem szabadítja fel őket.

A nullptr biztosítja, hogy a nem használt helyek üresek legyenek, és ne okozzanak problémát a kiíráskor.

2. lépés: Használat bemutatása

Most hozzunk létre néhány bájitalt és varázstárgyat, majd töltük fel őket az áruházba. Ezen keresztül bemutatjuk, hogy hogyan kezelhetjük a heterogén kollekciót, ahol a különböző típusú objektumok ugyanazon interfészen keresztül érhetők el.

```
int main() {  
    // Két termék létrehozása  
    Potion healingPotion("Healing Potion", 25);  
    MagicItem magicSword("Sword of Flames", 10);  
  
    // Áruház létrehozása  
    Store store;  
  
    // Termékek hozzáadása az áruházi kollekcióhoz  
    store.addItem(&healingPotion, 0); // Bájital a 0. helyre  
    store.addItem(&magicSword, 1); // Varázstárgy az 1. helyre  
  
    // Az áruház kiíratása  
    store.displayItems();  
  
    return 0;  
}
```

Miért így?

- Mivel az áruház **az IGameEntity interfészre mutató mutatókat** tárol, **nem kell tudnia a konkrét típusokról** (Potion vagy MagicItem). Ez biztosítja a **polimorfizmust**, így minden típus ugyanazon interfészen keresztül használható.

- Az objektumokat **referenciaként** (`&healingPotion`, `&magicSword`) adjuk át az áruháznak, hogy elkerüljük a másolást, és a tényleges objektumok módosítását közvetlenül befolyásoljuk.
-

3. lépés: Miért működik a heterogén kollekció?

A kulcselem a **polimorfizmus** és a **dinamikus kötés (late binding)**. Most részletesebben elmagyarázom, hogy miért működik a heterogén kollekció, és hogyan segíti a programunk működését:

- A Store osztályban lévő kollekció csak egy típusú mutatókat tárol: `IGameEntity*` típusú mutatókat. Ez lehetővé teszi, hogy különböző típusú objektumokat tároljunk benne (pl. `Potion` vagy `MagicItem`).
- Az interfész (`IGameEntity`) miatt a **tagfüggvények** (pl. `getDescription`, `calculatePower`) minden egyik típusnál másképp lesznek implementálva. Tehát, amikor az áruház végigmegy a kollekcion, minden egyes elemhez a megfelelő tagfüggvényeket hívja meg.
- Az **ősosztály pointerek** használata és a virtual kulcsszó biztosítja, hogy a helyes implementáció hívódjon meg futásidőben (nem a tárolás pillanatában), tehát a `Potion` osztály `calculatePower` metódusa fog lefutni, míg a `MagicItem` metódusa másként.

Mi történne, ha nem így csinálnánk?

- Ha nem használnánk interfészt (`IGameEntity`) és az objektumokat konkrét típusokkal (pl. `Potion` vagy `MagicItem`) tárolnánk, akkor az áruház nem lenne képes heterogén típusú objektumokat kezelni. minden egyes típusnak külön tárolóban kellene lennie.
 - Ha nem lenne virtuális a függvény, akkor **nem a megfelelő függvények futnának le**. Ez azt jelentené, hogy az objektum típusától függetlenül minden objektum ugyanazokat a metódusokat futtatná, nem pedig a saját típusához tartozó implementációt.
-

Most a feladatot úgy bővítjük, hogy **többszörös öröklés** segítségével a varázstárgyakat több interfész és ősosztály is meghatározza. Az új követelmény az, hogy a **raktár** képes legyen tárolni azokat a varázstárgyakat, amelyek különböző ősosztályokból, illetve interfészektől származnak, és további információt is tudjon nyújtani, például azt, hogy az adott tárgy hány légköbmétert foglal el.

4. Feladat: Többszörös öröklés és raktár kiegészítése

A feladat célja:

- Több ősosztály segítségével le kell származtatni a **varázstárgyakat** (pl. egy varázsital és egy mágikus fegyver).
- Az új interfész az ūrtartalomra vonatkozó adatokat adjon meg, amiket a **raktár** képes legyen kiírni.

- Az áruház (store) képes legyen ezeket az adatokat kezelní, és a termékekről minden leírást, minden az ūrtartalmat ki tudja írni.
-

1. lépés: Többszörös öröklés kialakítása

Képzeld el, hogy a **varázstárgyak** több jellemzővel rendelkeznek, mint például egy **fizikai méret** (űrtartalom). Ehhez egy újabb interfészöt hozunk létre, amely meghatározza, hogy a varázstárgyak rendelkezzenek egy metódussal, amely visszaadja az ūrtartalmukat.

Az új interfész: ISize

Ez az absztrakt ősosztály olyan osztályok számára készült, amelyek fizikai mérettel rendelkeznek (térfogat, súly stb.). Az interfész meghatározza az alábbi metódust:

```
class ISize {  
public:  
    virtual double getVolume() const = 0; // Függvény, ami visszaadja az  
    // ūrtartalmat  
    virtual ~ISize() {}  
};
```

Miért szükséges ez az interfész?

- Az új interfész segít abban, hogy a varázstárgyak pontosan meghatározhassák, hogy **milyen fizikai teret foglalnak el**. Ez egy fontos adat, amit a raktárnak kezelnie kell.
-

2. lépés: Többszörös öröklés a varázstárgyaknál

Most már van egy **ISize** interfészünk, amit a varázstárgyaknak örökölniük kell. A varázstárgyak osztályai ezen interfészen keresztül nyújtanak információt az ūrtartalmukról, miközben továbbra is megmaradnak az alap **IGameEntity** interfész használatában.

MagicItem és Potion osztályok módosítása

```
class MagicItem : public IGameEntity, public ISize {  
private:  
    std::string name;  
    double size; // Lékgöbméterben  
  
public:  
    MagicItem(std::string name, double size)  
        : name(name), size(size) {}  
  
    std::string getDescription() const override {  
        return "Magic Item: " + name;  
    }  
}
```

```
double calculatePower() const override {
    return size * 5; // Erő: az ūrtartalom szorozva 5-tel
}

double getVolume() const override {
    return size;
}

};

class Potion : public IGameEntity, public ISize {
private:
    std::string name;
    double volume; // Literben

public:
    Potion(std::string name, double volume)
        : name(name), volume(volume) {}

    std::string getDescription() const override {
        return "Potion: " + name;
    }

    double calculatePower() const override {
        return volume * 3; // Gyógyító erő: az ūrtartalom szorozva 3-mal
    }

    double getVolume() const override {
        return volume;
    }
};
```

Miért így?

- A MagicItem és a Potion osztályok most **két interfészöt örökölnek**: IGameEntity (amely a leírást és erőt határozza meg) és ISize (amely a termék ūrtartalmát adja meg).
- A getVolume() metódust minden osztály implementálja, hogy a raktár pontosan megtudja, hogy hány légköbmétert foglal el az adott tárgy.

3. lépés: A raktár frissítése

Most frissítenünk kell a **raktárat** (Store osztály), hogy kezelje az új adatot, az űrtartalmat. Az áruháznak képesnek kell lennie arra, hogy ne csak a leírást és az erőt, hanem az űrtartalmat is kiírja.

```
class Store {  
  
private:  
    IGameEntity* inventory[10]; // Tároló változó  
  
public:  
    Store() {  
        for (int i = 0; i < 10; ++i) {  
            inventory[i] = nullptr; // Kezdetben nincs semmi  
        }  
    }  
  
    void addItem(IGameEntity* item, int index) {  
        if (index >= 0 && index < 10) {  
            inventory[index] = item; // Elem hozzáadása  
        }  
    }  
  
    void displayItems() const {  
        for (int i = 0; i < 10; ++i) {  
            if (inventory[i]) {  
                std::cout << "Item " << i + 1 << " - Description: " <<  
inventory[i]->getDescription() << "\n";  
                std::cout << "Power: " << inventory[i]->calculatePower() << "\n";  
  
                // Az új metódus hívása, ha az elem implementálja az ISize  
                interfészet  
                if (const ISize* sizeItem = dynamic_cast<const  
ISize*>(inventory[i])) {  
                    std::cout << "Volume: " << sizeItem->getVolume() << " cubic  
                    meters\n";  
                }  
            }  
        }  
    }  
};
```

Miért így?

- Az áruház a **dynamic_cast** segítségével ellenőrzi, hogy az adott elem leszármazik-e az ISize absztrakt ősosztályból. Ha igen, akkor meghívja a `getVolume()` metódust, hogy kiírja az adott tárgy ūrtartalmát.
 - A **dynamic_cast** lehetővé teszi, hogy biztonságosan ellenőrizzük, hogy az objektum tényleg olyan típusú, amit kezelní akarunk, és ez csak akkor fog működni, ha a metódus virtual és az osztály örökölte a megfelelő interfész. Erről még nem biztos, hogy hallottál, a konverzióról szóló órán van (lesz) róla szó.
-

4. lépés: Bemutatás és tesztelés

Most hozzunk létre néhány **varázstárgyat** és töltük fel őket a raktárba. A raktár kiírja a termékek leírását, erejét és ūrtartalmát is.

```
int main() {  
    // Két termék létrehozása  
    Potion healingPotion("Healing Potion", 2.5); // 2.5 literes  
    MagicItem magicSword("Sword of Flames", 1.2); // 1.2 m³  
  
    // Áruház létrehozása  
    Store store;  
  
    // Termékek hozzáadása  
    store.addItem(&healingPotion, 0);  
    store.addItem(&magicSword, 1);  
  
    // Az áruház kiíratása  
    store.displayItems();  
  
    return 0;  
}
```

Miért így?

- A `Potion` és a `MagicItem` osztályok minden tartalmazza az ūrtartalomra vonatkozó adatokat, és az áruház megfelelően kezeli azokat.
 - Az `ISize` interfész biztosítja, hogy minden varázstárgy, ami implementálja ezt az interfészt, képes legyen megadni az ūrtartalmát.
-

Végül itt egy kérdés-válasz lista, amely a kódban szereplő fontos tulajdonságokra és alapvető C++-os programozási kérdésekre épít, és segíthet (reményeim szerint 😊) jobban megérteni a program működését és a fontos koncepciókat.

Kérdések-válaszok

1. Miért van szükség virtuális destruktora az interfészknél (interfész öröklésre használt absztrakt ősosztálynál)?

Kérdés: Miért van szükség virtuális destruktora, amikor interfész implementáló osztályok példányosítása történik?

Válasz: A virtuális destruktur szükséges, mert ha egy osztály példányosítását interfészen keresztül történik, akkor a helyes destruktur hívása biztosítja, hogy az összes erőforrás (pl. dinamikusan foglalt memória) megfelelően felszabaduljon. Ha nem használunk virtuális destruktort, akkor csak az interfész destruktora kerül meghívásra, ami nem elég a leszármazott osztályok erőforrásainak felszabadításához. Ez memóriaszivárgást eredményezhet.

2. Miért használtunk dynamic_cast-ot a kódban?

Kérdés: Miért alkalmaztuk a dynamic_cast operátort a raktár osztályban, és mikor van szükség erre?

Válasz: A dynamic_cast operátort arra használtuk, hogy biztonságosan ellenőrizzük, hogy az IEntity típusú objektum tényleg egy ISize típusú objektum is-e. Ez akkor szükséges, amikor polimorfikus típusokat kezelünk, és biztosítani akarjuk, hogy a kódban használt objektumok valóban a kívánt típusok. Ha nem a dynamic_cast-ot használnánk, előfordulhatna, hogy egy nem megfelelő típusú objektumot próbálunk meg kezelní, ami hibához vezethet.

3. Miért fontos a statikus és konstans tagváltozók megfelelő használata?

Kérdés: Miért nem célszerű statikus vagy konstans tagváltozókat használni az interfészekben?

Válasz: Interfészben nem célszerű statikus vagy konstans tagváltozókat használni, mert az interféseknek csak azokat a metódusokat kell tartalmazniuk, amelyeket az osztályok implementálniognak. A statikus tagváltozók minden példányban osztoznak, míg a konstans tagváltozók csak egyetlen értékkel rendelkeznek, ami nem változtatható meg. Az interfész célja, hogy egy közös szerződést biztosítson a különböző osztályok számára, nem pedig egy globális vagy állandó értékeket.

4. Mi történik, ha egy interfész (interfész öröklésre használt absztrakt ősosztály) tagváltozót is tartalmaz?

Kérdés: Mi történik, ha egy interfészben tagváltozókat definiálunk, és miért nem szabad ezt tenni?

Válasz: Ha egy interfészben tagváltozókat definiálunk, akkor az interfész nem lesz igazán interfész, hanem egy szokásos osztály lesz, mivel az interfész célja, hogy csak metódusokat tartalmazzon, nem pedig állapotot. A tagváltozók az osztályok saját adatainak kezelésére valók, nem pedig egy közös szerződésre. Az interfésznek tisztán csak a metódusok deklarációt kell tartalmaznia, nem pedig állapotot.

5. Miért van szükség destruktorkra a ISize interfészen belül?

Kérdés: Miért kell a ISize interfészben is deklarálni egy destruktort, ha az interfészknél nincs szükség erőforrások kezelésére?

Válasz: Bár az interfészknél nem szükségesek erőforrások kezelése, fontos, hogy a destruktort **virtualissá** tegyük, ha az interfészhez kapcsolódó osztályok rendelkeznek destruktoral. Ez biztosítja, hogy a megfelelő destruktur hívódjon meg, amikor az interfész típusú mutatót törlünk, amely a leszármazott osztály példányára mutat. A virtuális destruktur segít megelőzni a memóriaszivárgást.

6. Miért érdemes használni a const kulcsszót a tagfüggvényeknél?

Kérdés: Miért van szükség a const kulcsszóra a tagfüggvényeknél, és mikor használjuk?

Válasz: A const kulcsszót akkor használjuk, ha a függvény nem módosítja az objektum állapotát. Ez biztosítja a kód biztonságát, mivel a const metódusok garantálják, hogy az objektum állapotát nem változtatják meg, tehát az objektum "olvasható" marad. Ha például a getDescription() metódus nem módosítja az objektum adatokat, célszerű const-t alkalmazni.

7. Miért célszerű interfész öröklésre használt absztrakt ősosztályokat használni a programozás során?

Kérdés: Miért célszerű interfészeket használni az objektum-orientált programozásban, különösen ebben a feladatban?

Válasz: Az interfész öröklésre használt absztrakt ősosztályok lehetővé teszik a polimorfizmust és a kód újrahasznosítását. Az interfések segítségével különböző osztályok implementálhatják ugyanazokat a metódusokat, lehetővé téve az egységes működést anélkül, hogy az osztályok közvetlenül kapcsolónának egymáshoz. Ez a **függetlenséget** és a **kiterjeszthetőséget** biztosítja a kód számára.

8. Miért fontos a "const-correctness" alkalmazása?

Kérdés: Miért fontos, hogy a metódusok const-ot alkalmazzanak, amikor az nem változtatja meg az objektumot?

Válasz: A "const-correctness" alkalmazása biztosítja, hogy a kód könnyebben olvasható és karbantartatható legyen, mivel garantálja, hogy egy adott metódus nem módosítja az objektumot. Ezen kívül a fordítónak is segít, hogy hibákat találjon, amikor véletlenül módosítjuk a nem módosítható adatokat.

9. Miért használjuk a nullptr-t, és mi a különbség a NULL-al szemben?

Kérdés: Miért használjuk a nullptr-t a kódban a NULL helyett?

Válasz: A nullptr egy típusbiztos null pointer, amely garantálja, hogy nem keveredik más típusokkal, ellentétben a régebbi NULL makróval. A nullptr használata segít a típusellenőrzésben, és megelőzi a hibákat, amelyeket a NULL okozhat, mivel a NULL nem garantálja, hogy csak pointer típusokkal működjön.

10. Miért van szükség az osztályok virtual kulcsszavára?

Kérdés: Miért van szükség a virtual kulcsszóra az ősosztályokban?

Válasz: A virtual kulcsszó lehetővé teszi a **polimorfizmust**, azaz azt, hogy egy osztály példányosításától függően a megfelelő metódus hívódjon meg, még akkor is, ha a metódust egy ősosztály mutatóján keresztül hívjuk. A virtual segít abban, hogy a program futás közben dinamikusan válassza ki a megfelelő metódust a származtatott osztályokban.