Konstans / referencia argumentumok, konstans tagfüggvények

OOP és DB - OOP 2. óra

Konstans változók / argumentumok

const: “megígérem, hogy ezt az értéket nem változtatom meg”

a konstans változókat létrehozáskor inicializálni kell! Olyan nincs, hogy

```
const int a;    // (const variable “a” requires an initializer)…
const int a = 5; // igy mar OK!
```

Konstans változók / argumentumok

interfészek megvalósításánál ha egy függvény vagy metódus paramétere *const*, nem kell attól tartani, hogy a neki átadott adat módosul.

> (mivel az argumentumok átadása érték szerint történik, ennek főleg pointereknél és referenciák esetében van értelme)

```
const int x; // nem OK, konstanst muszaj inicializalni!
const int y = 17; // y nevvel ellatott konstans
int var = 17; // var nem konstans

double sum(const std::vector<double>&); // ez a fv nem modositja argumentumat!
```

Konstans kifejezések

constexpr: “értékeld ki fordításkor”

konstansok megadásakor hasznos, illetve lehetnek teljesítménybeli előnyei is

constexpr függvény nagyon egyszerű lehet csak, egyetlen return paranccsal (különben nehéz lenne fordításidőben kiértékelni). Példa:

```
constexpr double square(double x) { return x * x; }
```

constexpr függvény megadásakor csak olyan függvény hívható meg, amely maga is constexpr

constexpr függvény meghívható nem constexpr argumentummal is, de olyankor az eredményt nem használhatjuk fel constexpr-ben

Konstans kifejezések

constexpr előnyei:

Névvel illetett konstansok könnyebben érthetőek és fenntarthatóak, mintha egy számot vagy értéket írnánk be több helyre

constexpr lokálisan scope-olható, nem úgy mint a #define - névtér szinten is OK!

```
const int y = 17;
int var = 17;
constexpr double square(double x) { return x * x; }
constexpr double max1 = 1.4*square(y);
constexpr double max2 = 1.4*square(var); // hiba - value of var cannot be used as constant
const double max3 = 1.4*square(var); // OK - futasidoben ertekeljuk ki
double sum(const std::vector<double>&); // ez a fv nem modositja argumentumat!
const double s1 = sum(v); // OK, futasidoben - ha v erteke addigra ismert!
constexpr double s2 = sum(v); //hiba: sum(v) nem konstans kifejezes
```

Konstans mutatók

Mutatók esetén nem mindegy, hogy:

a.) a mutató konstans (más címre később már nem mutathat)

b.) az adat, amire a mutató mutat, konstans

Sokan használják a hátulról olvasás módszerét:

char* const cp; //cp egy konstans mutató egy karakterre

char const * pc; //pc egy mutató konstans karakterre

const char * pc2; //pc2 egy mutató egy karakterre, ami konstans

Konstans mutatók

Konstansra mutató pointernek adható olyan cím, amely nem konstans változót tartalmaz

> *ebből baj nem származhat, legfeljebb a pointeren keresztül nem módosítható a változó*

Konstans változó címét azonban nem adhatjuk nem konstansra mutató pointernek, mert ezáltal a változó értéke módosítható lenne!

```
int a = 1;
const int c = 2;
const int* p1 = &c; // OK
const int* p2 = &a; // OK, max. *p2 = 5; nem engedelyezett
int* p3 = &c; // hiba! c konstans...
*p3 = c; // ez gond lenne...
int* const p4 = &c; // hiba! itt csak a pointer const!
```

Konstans tagváltozók

Osztályon belül természetesen létrehozhatunk konstans tagváltozót - azt azonban ugyanúgy létrehozáskor inicializálni kell!

Ha a konstruktor törzsén belül adnánk neki értéket, az már inicializálás + külön értékeadást jelentene, amit a C++ nem enged meg

A megoldás: inicializáló lista!

```
class Node {
    const int value;
    Node* next;
public:
    Node(int val) : value(val) {}
};
```

Konstans tagváltozók

Vesd össze:

```
class Node {
    const int value;
    Node* next;
public:
    Node(int val) : value(val) {}
};
```

```
class Node {
    const int value;
    Node* next;
public:
    Node(int val) {
        value = val;
    }
};
```

Konstans tagváltozók

Az inicializáló lista elemei a törzs meghívása előtt, inicializáláskor kerülnek feldolgozásra abban a sorrendben, ahogy megjelennek.

(Az osztály destruálásakor a destruktor pedig fordított sorrendben szabadítja fel azok memóriaterületét).

Ez akkor lényeges, ha egy később inicializált tagváltozónak szüksége van egy korábban inicializált tagváltozó értékére.

Ugyanezt fontos tudni öröklésnél is. A szülő inicializálására nincs ilyen a C++-ben, hogy *super()*... cska inicializáló lista van.

Konstans (és referencia) tagváltozók inicializáló listán kívül más módon nem is hozhatóak létre.

Konstans tagfüggvények

Egy osztály tagfüggvénye lehet konstans - ilyenkor megígérjük, hogy a függvény nem módosít az osztály állapotán (ha mégis, a fordító hibát jelez)

Konstans objektumra nem hívható meg az osztály nem konstans metódusa (érthető okokból)!

```
class Node {
    int value;
    Node* next;
public:
    Node(int val) : value(val) {}
    int getValue() const { return value; }
    void setValue(int x) { value = x; }
};


int main()
{
    Node a(5);
    const Node b(10);
    std::cout << a.getValue() << std::endl;
    std::cout << b.getValue() << std::endl;
    a.setValue(6);
    b.setValue(11); // nem OK! b const!!
```

Konstans argumentumok ugyanúgy működnek...

```
class Node {
    int value;
    Node* next;
public:
    Node(int val) : value(val) {}
    int getValue() const { return value; }
    void setValue(int x) { value = x; }
};

void f(Node& a, const Node& b, int val) {
    a.setValue(val);
    b.setValue(val); // hiba!
}
```

Konstans tagfüggvények - mutable

Kivétel, ha egy tagváltozót a **mutable** módosítóval deklarálunk. Ilyenkor az adott tagváltozót még egy konstans tagfüggvény is felülírhatja.

Ez akkor hasznos, ha egy adott tagfüggvény lényegében nem módosítja az osztály állapotát, de mégis szüksége van arra, hogy egy értéket pl. cache-elni tudjon.

Konstans tagfüggvények - mutable

```
class Date {
    mutable bool cache_valid;
    mutable std::string cache;
    void compute_cache_value() const { // frissitjuk a cache erteket
        // ...
    }
public:
    std::string string_rep() const { // lekerdezzuk a string reprezentaciot
        if (!cache_valid) {
            compute_cache_value(); // meghivhatja!
            cache_valid = true; // allithatja!
        }
        return cache;
    }
};
```

Konstans tagfüggvények - mutable

```
class Node {
    int value;
    Node* next;
public:
    Node(int val) : value(val) {}
    int getValue() const { return value; }
    void setValue(int x) { value = x; }
};

void f(Node& a, const Node& b, int val) {
    a.setValue(val);
    b.setValue(val); // hiba!
}
```

```
class Node {
    mutable int value;
    Node* next;
public:
    Node(int val) : value(val) {}
    int getValue() const { return value; }
    void setValue(int x) const { value = x; }
};

void f(Node& a, const Node& b, int val) {
    a.setValue(val);
    b.setValue(val); // mar nem hiba!
}
```

Konstans tagfüggvények - *const_cast*

Egy másik lehetőség (nem elegáns, de működik), hogy a változó “konstansságát” elkasztoljuk (“cast away constness”).

```
class Node {
    int value;
    Node* next;
public:
    Node(int val) : value(val) {}
    int getValue() const { return value; }
    void setValue(int x) { value = x; }
};

void f(Node& a, const Node& b, int val) {
    a.setValue(val);
    const_cast<Node&>(b).setValue(val); // megsem hiba!
}
```

```
int main()
{
    Node a(5);
    const Node b(10);
    std::cout << a.getValue() << std::endl;
    std::cout << b.getValue() << std::endl;
    a.setValue(6);
    const_cast<Node&>(b).setValue(11); // nem OK! b const!!
    std::cout << b.getValue() << std::endl; // mar 11!
```

Konstans tagfüggvények - indirekt struktúrák

A harmadik lehetőség, hogy egy pointeren keresztül hivatkozunk a módosítandó adatstruktúrára

Ilyenkor a fordító nem vizsgálja tranzitívan, hogy mit módosítottunk

```
struct NodeValue {
    int value;
    NodeValue(int val) : value(val) {}
    void setValue(int val) { value = val; }
};

class Node {
    NodeValue* node_value;
    Node* next;
public:
    Node(int val) { node_value = new NodeValue(val); }
    ~Node() { delete node_value; } // new eseten biztosan kell destruktor is!
    int getValue() const { return node_value->value; }
    void setValue(int x) const { node_value->value = x; } // lehet const :-O
};
```

Referenciák

A referenciák sok szempontból hasonlítanak a pointerekhez, csak egyszerűbben kezelhetőek.

Fő problémák (vagy “nehézségek”) a pointerekkel:

Változhat (hacsak nem konstans pointer), hogy milyen címre mutat

Pointer értéke lehet *nullptr*, amire mindig tesztelni kell...

Máshogyan kell dereferenciálni, ha egy pointer által mutatott objektumra, vagy egy pointer által mutatott objektum tagváltozójára hivatkozunk

**p* versus p->tagvaltozo

Referenciák

Egy referencia (sima ún. “lvalue” referencia) egy változóhoz társított másik név.

- Létrehozáskor inicializálni kell (mint egy konstans)
- Később nem hivatkozhat más változóra (mint egy konstans)

Felfogható úgy is, mint egy konstans pointer, ami mindig dereferenciálódik.

```
int main()
{
    int x = 5;
    int& b = x;
    int& c; // nem OK, muszaj incializalni!
```

Referenciák

A referenciákat előszeretettel használjuk metódusok argumentumainál, illetve visszatérési értékeinél

- Ha egy argumentum egy referencia, akkor az adott változó értékét nem kell a stack-en lemásolni…
- Ha a visszatérési érték az adott osztály példányára hivatkozó referencia, akkor a hívásokat “láncolni” lehet (ld. példa)

Referenciák osztályokban

```
class Node {
    int value;
    Node* next;
public:
    Node(int val) : value(val), next(nullptr) {}
    // other-t nem kell ertek szerint masolni a stacken:
    // (ez olyan neven is ismert, hogy copy constructor)
    Node(Node& other) { value = other.value; next = other.next; }
    Node& addOne() { value++; return *this; }
    int getValue() const { return value; }
    void setValue(int x) { value = x; }
};

int main()
{
    Node a(5);
    Node b(a);
    b.addOne().addOne(); // lancolhato, mivel Node& a visszateresi tipus
    std::cout << b.getValue() << std::endl; // 7
```