



# Bevezetés a programozásba 2

## 6. Előadás: A `const`

# Interface - Implementation

```
struct Particle {  
    int x,y;  
    unsigned char r,g,b;  
    void rajzol();  
};
```

```
void Particle::rajzol() {  
    gout << move_to(x, y)  
    << color (r, g, b)  
    << dot;  
}
```

# Láthatóság

```
struct Particle {  
    Particle(int X, int Y);  
    virtual void mozog( ... );  
    virtual void rajzol( ... );  
protected:  
    double x,y;  
    unsigned char r,g,b;  
};
```



# Osztály

```
class Particle {  
public:  
    Particle(int X, int Y);  
    virtual void mozog( ... );  
    virtual void rajzol( ... );  
protected:  
    double x,y;  
    unsigned char r,g,b;  
};
```

# Implementáció elrejtése

particle.hpp

```
#ifndef PARTICLE_HPP
#define PARTICLE_HPP

struct Particle {
    int x,y;
    unsigned char
r,g,b;
    void rajzol();
};

#endif
```

particle.cpp

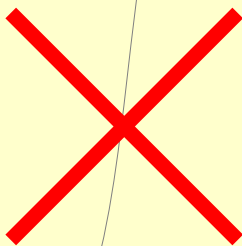
```
#include "particle.hpp"

void Particle::rajzol() {
    cout << move_to(x, y)
    << color (r, g, b)
    << dot;
}
```

main.cpp

```
#include "particle.hpp"

int main() {
    ...
    Particle p;
    p.rajzol();
}
```



# A fordítás menete

Preprocesszor

Forráskódok értelmezése, típusok feltérképezése (pl. méret miatt).

Forrásfájlonként egy tárgykód (object, .o) fájl létrehozása. Ebben gépi kódú részletek vannak, előkészítve a csatlakozási pontokat

A linker összeköti a tárgykódokat

“undefined reference”: egyik tárgykódban sincs feloldva egy csatlakozási pont, pl függvényhíváshoz nem található függvény implementáció

# Konstansok

## Literál konstansok

5, vagy „Hello world”

„literál”

## Változónak látszó, típussal, névvel ellátott konstans

```
const double pi = 3.14159;
```

„konstans”

Globális változót nem illik csinálni, globális konstanssal nincs probléma.

# Literálok

- A literáloknak is van típusa, tipikus példák:
  - 21 : int
  - 21ll : long long int
  - 21.0 : double
  - 21.0f : float
- C++11-ben lehet saját suffixeket gyártani
- Literál megadható nem csak decimálisan:
  - 0x21 : =33, hexadecimális
  - 021 : =17, oktális



# Konstansok

kizárólag kezdeti értékkel lehet deklarálni:

definiálni

Értékadás nincs értelmezve

Fordítási időben ismert az értéke

```
int a = 1;  
a = 2;  
const int c ≐ 1;  
c = 2
```

ez megy

ez nem megy

# Konstans referencia

```
const T & r;
```

A leggyakoribb paraméterezési forma

Használandó:

Másoló konstruktor

operátorok

...

Gyors, biztonságos

# Érték, vagy const referencia paraméterezés?

- Az érték szerinti paraméterátvétel explicit másolást jelent, ez néha kihasználható
- A másolás nagyméretű típusoknál lassú
- Az érték szerinti paraméter átvételnél csak statikus típus van, a referencia szerinti megőrzi a dinamikus típust
  - csonkolás: amikor egy objektumot az isa reláció szerint őszosztályként használva elhagyjuk a dinamikus típushoz tartozó adatait
- Ökölszabály: az olyan típusokat, ahol lehet öröklődés, const referenciával kezeljük

# A `const` használata

Ha valami `const`, akkor nem végezhető vele olyan művelet, ami az értéket megváltoztat*hatja*

A változatlanságra garanciát ad a fordító

paraméterként csak akkor adható át, ha az adott függvény ott konstans paramétert vár, így a lokális változó is `const`

még mutatót sem lehet rá állítani, csak olyan mutatót, ami konstansokra tud csak mutatni

Honnan tudható, hogy tagfüggvény hívható-e?

# Konstans tagfüggvények

```
int main() {  
    const string s = "Hello";  
    cout << s.length();  
}
```

Ez működik. A fordítóprogram valahonnan tudja, hogy a `length()` nem fogja megváltoztatni az értéket

# Konstans tagfüggvények

```
int main() {  
    const string s = "Hello";  
    cout << s.length();  
}
```

↓

```
size_type string::length() const  
{  
    ...  
}
```

# Konstans tagfüggvények

```
class Particle {
public:
    Particle(int X, int Y);
    virtual void mozog( ... );
    virtual void rajzol( ... ) const;
protected:
    double x,y;
    unsigned char r,g,b;
};
```

# Konstansság

Mezei változónak konstans értékül adható

úgyis másolat készül, az eredeti nincs veszélyben  
mutatót nem lehet rá állítani

Konstansnak pályafutása kezdetén mezei változó  
értékül adható

paraméterként kapott lokális változók

Ez lehetőséget ad arra, hogy a változó által tárolt  
értéknek időszakosan, egy-egy függvényhívás  
idejére védettséget biztosítsunk



# A `this`

Kizárólag tagfüggvényben használható

Mutató, ami a meghívott objektumra mutat  
más szavakkal maga az implicit paraméter

Tipikus használata

ha az objektum tagfüggvényén belül hív olyan függvényt, ahol paraméterként saját magát kell átadnia

visszatérési értéként értékadás operátorban, és egyéb olyan operátorokban, függvényekben, ahol a kifejezés hatása a visszatérési értékben is hasznos

# A this

```
struct koord {
    double x,y;
    koord operator=(const koord& masik) {
        x=masik.x; y=masik.y;
        return *this;
    }
};

int main() {
    koord a,b,c;
    c=b=a;
    return 0;
}
```

# Konstans tagfüggvények

A `this` típusa C class egy sima tagfüggvényében: `C *`

Konstans tagfüggvényben pedig `const C *`

Tehát a konstans tagfüggvényen belül a mezőkre nem hívható nem konstans művelet

Ez a technikai háttere a konstansság ellenőrzésének konstans tagfüggvényekben

# Áttekintés

A `const` használata az eddigiek alapján egy már tartalmas programra láncreakciót okoz

Ha egy tagfüggvényt megjelölünk, hogy konstans, a bennelevő meg nem jelölt tagfüggvényekre hivatkozás, illetve paraméterátadás nem `const` paraméterként nem fordul le

Ha paraméternél jelöljük a konstansságot, nem hívhatjuk meg az objektum nem megjelölt tagfüggvényeit

Érdemes idejekorán ezt elintézni

Később sem vészes, a fordító keresgél helyettünk

# Kiskapuk

Néha előfordul, hogy a szigorú konstansság követelmények bajt okoznak

Meglevő, nem módosítható könyvtár készítői nem gondoltak a konstansok helyes kezelésére

**const\_cast**

**mutable**

# A mutable

A `mutable` egy osztály mezőjének módosítója lehet

Azt jelenti, hogy ez a mező akár egy konstans objektumban is megváltozhat

Tipikus használata:

- olvasásszám nyilvántartás

- cache

Jó tervezésnél a reprezentációt nem deklaráljuk `mutable`-ként

# const\_cast

Általában nem állíthatunk mutatót konstansra, ha mégis kell\*, akkor használjuk a `const_cast`-ot

```
const T* cm = .. a konstansra mutató  
T* m = const_cast<T*>(cm) ;
```

Ezzel most `m` változtatható módon mutat bele a konstansba, és ez veszélyes

Nem definiált következmény!

\* nagyon ritkán kell. Ha kellett, akkor máris van miért újratervezni az egészet.

# Biztonságos programozás és a `const`

Fordítási időben kiderül a probléma

konstans tagfüggvények használatával a lehetséges módosítások köre szűkül

következetesség a fordítóprogram támogatásával

Ismétlődő literálok elkerülése (magic numbers)

a literálok típusai bizonytalanok ( $1/2$  vs  $0.5$ )



# Konstansok és hatékonyság

- A konstansok fordítási időben ismertek, tehát a fordítóprogram fordítás közben a műveleteket elvégezheti velük, azaz nem futásidőben telik vele az idő
- Következtetés: a szép forráskód, ahol az olvasható értelmes konstansok műveletekben állnak (pl: képernyőfelbontás-dobozméret, mint legszélső rajzolható pozíció) nem okoznak lassulást.
- C++11: `constexpr`

# Egyéb érdekességek

Mutatók is lehetnek konstansok:

```
const int * m
```

```
int const * m
```

```
int * const m
```

```
const int * const m
```

Az első két írásforma felcserélhető

mutató, ami konstans int-re mutat

Ha a \* után van a const, a mutató konstans, tehát nem mutathat később máshová

Jobbról balra érdemes olvasni