

# OO Programozás

## Konstansok

Dr. Hatwagner F. Miklós

Széchenyi István Egyetem, Győr

[https://github.com/wajzy/GKxB\\_INTM085](https://github.com/wajzy/GKxB_INTM085)

2023. szeptember 21.

Széchenyi István Egyetem, Győr

- konstans kifejezéssel,
- konstans kifejezéssel inicializált elnevezett konstanssal,
- vagy tetszőleges kifejezéssel C99 / C++14-től

## Tömbök definiálása, méret megadása

\_\_\_\_\_

```
1 #include <iostream>
2 #include "constHeader.h"
3 #define MEANING_M 42 // no type, cumbersome debugging
4
5 void array(int);
6
7 int main() {
8     const int MEANING = 42;
9     int meaningCopy = MEANING; // read
10    // MEANING++; // error
11    // MEANING = -42; // error
```

```

12 // size expressed with constant expression
13 int oldArray1[MEANING_M * sizeof(double)]; // OK
14 int oldArray2[MEANING * sizeof(double)]; // OK
15 array(21);
16 // std::cout << i << std::endl; // error
17 std::cout << ci << std::endl; // OK
18 std::cout << si << std::endl; // OK
19 }
20
21 void array(int size) {
22     // C99+ / C++14+ -> variable size is OK, allocated on stack
23     int newArray[size * sizeof(double)];
24 }

```

## Széchenyi István Egyetem, Győr

```
1 // int i = 1; // error, multiple definition of 'i'
2 const int ci = 2; // OK, visible only in defining file scope
3 static int si = 3; // OK, file scope
```

## Széchenyi István Egyetem, Győr

```
1 #include "constHeader.h"
```

A constexpr módosító (C++11/14) hatása a függvényekre:

- „Értékel ki fordításkor!” → gyorsabb programok, lassabb fordítás
- Viszonylag egyszerű függvények készíthetők csak vele (C++11: csak 1 utasítás)
- Csak olyan globális változókra hivatkozhat, melyek konstansok
- Csak constexpr függvényt hívhat, akár önmagát is
- Értékkel kell visszatérnie
- Nem csak konstansokkal „hívható”, de akkor az eredmény nem használható konstans inicializálására
- C++11-ben még tiltott volt a prefix ++ operátor használata

## constexpr.cpp

```
3  constexpr double PI = 3.14159;
4  constexpr double deg2rad(double degree) {
5      return degree * PI / 180.;
6  }
7
8  int main() {
9      constexpr double rightAngle = deg2rad(90.);
10     std::cout << rightAngle << std::endl;
11     int deg180 = 180;
12     // the value of 'deg180' is not usable in a constant expression
13     // constexpr double rad180 = deg2rad(deg180);
14     double rad180 = deg2rad(deg180); // OK
15     std::cout << rad180 << std::endl;
16     return 0;
17 }
```

## fibonacci1.cpp

```
3  constexpr int fibonacci1(int n) {  
4      return (n <= 1) ? n : fibonacci1(n-1) + fibonacci1(n-2);  
5  }  
6  
7  int main() {  
8      constexpr int i = fibonacci1(32);  
9      std::cout << i << std::endl;  
10 }
```

## Mérés

```
$ time ./fibonacci1  
2178309  
  
real 0m0,005s  
user 0m0,005s  
sys 0m0,000s
```



## fibonacci2.cpp

```
3  int fibonacci2(int n) {
4      return (n <= 1) ? n : fibonacci2(n-1) + fibonacci2(n-2);
5  }
6
7  int main() {
8      int i = fibonacci2(32);
9      std::cout << i << std::endl;
10 }
```

## Mérés

```
$ time ./fibonacci2
2178309

real 0m0,036s
user 0m0,036s
sys 0m0,000s
```

Osztályok tagfüggvényei, sőt, konstruktor is jelölhető constexpr-nek, de

- Az adattagok kvázi konstansok lesznek, amik inicializálása túl későn van a konstruktorban → *taginicializáló lista*
- Konstans adattagok és referenciák csak taginicializáló listával hozhatók létre.
- Ez egyébként használható lett volna a nem konstans adattagok inicializálására is.
- Példányosításkor a paramétereknek konstansnak kell lenniük.
- A tagfüggvények implicit inline-ok lesznek.

## Rectangle10.cpp

```
3  class Rectangle {
4      double mWidth;
5      double mHeight;
6  public:
7      constexpr Rectangle(double width, double height) :
8          mWidth(width), mHeight(height) {}
9
10     constexpr double getArea() const {
11         return mWidth * mHeight;
12     }
13
14     constexpr double getPerimeter() const {
15         return 2. * (mWidth + mHeight);
16     };
17 };
```

## Rectangle10.cpp

```
19  int main() {  
20      const int width = 5.;  
21      constexpr Rectangle r1(width, 3.);  
22      std::cout << "Area: " << r1.getArea() << std::endl;  
23      std::cout << "Perimeter: " << r1.getPerimeter() << std::endl;  
24  }
```

Mutató konstans változóra:

- Maga a mutató nem konstans, nem *kell* inicializálni.
- Mutathat változóra, csak olvashatóvá téve azt.

### constptr.cpp

```
5  const int ci = 1; // must be initialized
6  int i = 2;
7  const int* pci; // the pointer is NOT constant
8  pci = &ci;
9  // *pci = 3; // assignment of read-only location '* pci'
10 pci = &i;
11 // *pci = 3; // read-only access to 'i'
```

Mutató változóra:

- Nem tartalmazhatja elnevezett állandó címét, mert a védelem nem kerülhető meg.

### constptr.cpp

```
14  int* pi = &i; // ok
15  // pi = &ci; // invalid conversion from 'const int*' to 'int*'
```

Konstans mutató egy változóra:

- Mivel a mutató konstans, inicializálni kell.
- A mutatott változó módosítható, de a mutató nem mutathat máshova.

### constptr.cpp

```
18  int* const cpi = &i; // must be initialized
19  *cpi = 3; // OK
20  int i2 = 4;
21  // cpi = &i2; // assignment of read-only variable 'cpi'
```

Konstans mutató konstansra:

- Mindent csak olvasni lehet.
- Kiolvasás hátulról előre:  
*cpci* egy *const* mutató (\*), ami olyan *int*-et címez, ami *const*.

### constptr.cpp

```
24  const int* const cpci = &ci;  
25  // *cpci = 4; // error  
26  // cpci = &ci; // error
```



## Referencia konstansra:

- Minden referenciát inicializálni kell.
- Ha változót rendelünk hozzá, akkor az érték ezen keresztül csak olvasható lesz.
- Inicializálható konstans kifejezéssel!

### constref.cpp

```
5  const int ci = 1;
6  int i = 2;
7  const int& rci = ci; // references must be initialized
8  // rci = 3; // assignment of read-only reference 'rci'
9  const int& rci2 = i; // read-only acces to 'i'
10 // rci2 = 3; // assignment of read-only reference 'rci2'
11 const int& rci3 = 3; // OK
```

A referenciák mindig konstansok.

constref.cpp

```
14 // int& const cri = i; // 'const' qualifiers cannot be applied to 'int&'
```

Nem kerülhető meg a védelem konstansot címző nem konstans referenciával.

constref.cpp

```
17 // int& ri = ci; // binding reference of type 'int&' to 'const int' discards qualifiers
```

Ha a függvény paramétere konstans, akkor a fv.-en belül sem változtatható meg az értéke, de a hívót ez nem érdekli (érték szerinti paraméterátadás).

De ha a paraméter mutató vagy referencia, akkor a fv. megváltoztathatná a változó értékét!

### constfn.cpp

```
3 void pfn(int* pi) {  
4     *pi *= 2;  
5 }  
6  
7 void rfn(int& ri) {  
8     ri *= 2;  
9 }
```

## constfn.cpp

```
11 void pcfnc(const int* pi) {  
12     // *pi *= 2; // assignment of read-only location '* pi'  
13 }  
14  
15 void rcfnc(const int& ri) {  
16     // ri *= 2; // assignment of read-only reference 'ri'  
17 }  
  
39 int main() {  
40     int i = 1;  
41     pcfnc(&i);  
42     rcfnc(i);  
43     std::cout << i << std::endl;  
44     const int ci = 5;  
45     // rcfnc(ci); // binding reference of type 'int&' to 'const int' discards ...
```

Ha a visszatérési érték típusa alaptípus, nincs haszna a `const`-nak (nem balérték).  
De ha mutató vagy referencia, a visszatérési érték megváltoztatása tiltható `const`-tal.

### constfn.cpp

```
19 int* fnp() {  
20     static int si = 10;  
21     return &si;  
22 }  
23  
24 const int* fncp() {  
25     static int si = 20;  
26     return &si;  
27 }
```

### constfn.cpp

```
29 int& fnr() {  
30     static int si = 30;  
31     return si;  
32 }  
33  
34 const int& fncr() {  
35     static int si = 40;  
36     return si;  
37 }
```

## constfn.cpp

```
39  int main() {  
  
46      rcfn(ci); rcfn(5); pcfn(&i); pcfn(&ci); // ok  
47      *fnp() = 11;  
48      // *fncp() = 21; // assignment of read-only location '* fncp()'  
49      fnr() = 31;  
50      // fnr() = 41; // assignment of read-only location 'fnr()'
```

Példány is lehet konstans. Az adattagok többnyire eleve rejtettek, ezért nem írhatók.

Nyilvános, konstans tag: getter elhagyható.

Konstans tagfüggvény konstans objektumon is hívható!

### Rectangle11.cpp

```
3 class Rectangle {  
4     public:  
5         double mWidth; // bad idea  
6         const double mHeight;  
7  
8         Rectangle(double width, double height) : mHeight(height) {  
9             mWidth = width;  
10        }
```

## Rectangle11.cpp

```
12     double getArea() const {
13         return mWidth * mHeight;
14     }
15
16     double getPerimeter() /* const */ {
17         return 2. * (mWidth + mHeight);
18     };
19 };
20
21 int main() {
22     const Rectangle r1(5., 3.);
23     // r1.mWidth = 55.; // assignment of member 'Rectangle::mWidth' in
24     std::cout << "Area: " << r1.getArea() << std::endl; // read-only object
25     // std::cout << "Perimeter: " << r1.getPerimeter() << std::endl;
26     // passing 'const Rectangle' as 'this' argument discards qualifiers
27 }
```



Konstans tagfüggvény is módosíthat egy adattagot, ha az `mutable`. Cél: érdemi állapotváltozást nem jelentő változások, pl. gyorsítótárak menedzselése.

### Rectangle12.cpp

```
3 class Rectangle {  
4     mutable bool areaCached;  
5     mutable double area;  
6     mutable bool perimeterCached;  
7     mutable double perimeter;  
8 public:  
9     const double mWidth;  
10    const double mHeight;
```

## Rectangle12.cpp

```
12     Rectangle(double width, double height) : mWidth(width), mHeight(height) {  
13         areaCached = perimeterCached = false;  
14     }  
15  
16     double getArea() const {  
17         if(not areaCached) {  
18             area = mWidth * mHeight;  
19             areaCached = true;  
20         }  
21         return area;  
22     }
```

(Sajnos) néhány további kivételes helyzetben is módosíthatja a `const` tagfüggvény a példány állapotát, pl. a tag struktúrát nem lehet lecserélni, de annak tagját már lehet módosítani (tranzitívan).

Egy objektumnak lehet beágyazott objektuma, ami szintén a taginicializáló listán keresztül inicializálható. Ha ezt nem tesszük meg → alapértelmezett konstruktor hívása, ha van ilyen.

### Rectangle13.cpp

```
4 class Point {  
5     public:  
6         const double x;  
7         const double y;  
8  
9         Point(double x, double y) : x(x), y(y) {}  
10 };
```

## Rectangle13.cpp

```
12  class Rectangle {
13      Point ul;
14      Point br;
15
16  public:
17      Rectangle(Point ul, Point br) : ul(ul), br(br) {}
18
19      double getArea() const {
20          return abs(ul.x - br.x) * abs(ul.y - br.y);
21      }
22
23      double getPerimeter() const {
24          return 2. * (abs(ul.x - br.x) + abs(ul.y - br.y));
25      };
26  };
```

## Rectangle13.cpp

```
28  int main() {
29      Rectangle r1(Point(0, 3), Point(5, 0));
30      std::cout << "Area:_" << r1.getArea() << std::endl;
31      std::cout << "Perimeter:_" << r1.getPerimeter() << std::endl;
32  }
```