OO Programozás

Konstansok

Dr. Hatwagner F. Miklós

Széchenyi István Egyetem, Győr

https://github.com/wajzy/GKxB_INTM085

2023. október 9.

A `const` típusmódosító változókkal használva

- *Konstans változó* (paradoxon! → *elnevezett konstans*)
- Memóriában helyezik el,
- azonnali inicializálást igényel,
- értéke megjelenik a nyomkövető programokban (debugger),
- csak olvasható (fordító biztosítja),
- van típusa (vö. `#define`).
- Láthatóságuk, hatókörük jobban szabályozható.

Tömbök definiálása, méret megadása

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

constMain.cpp

```
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
```

constMain.cpp

```
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  }
```

constHeader.h

```
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
```

constSource.cpp

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

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

- „Értékeld 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 $\rightarrow$ *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 pcfn(const int* pi) {
12   // *pi *= 2; // assignment of read-only location '* pi'
13 }
14
15 void rcfn(const int& ri) {
16   // ri *= 2; // assignment of read-only reference 'ri'
17 }

39 int main() {
40   int i = 1;
41   pfn(&i);
42   rfn(i);
43   std::cout << i << std::endl;
44   const int ci = 5;
45   // rfn(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   // fncr() = 41; // assignment of read-only location 'fncr()'
```

Példány is lehet konstans. Az adattagok többnyire eleve rejtettek, ezért kívülről 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 };
```

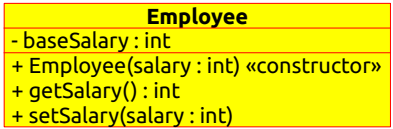
Rectangle13.cpp

```
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 }
```

Feladat: készítsünk osztályokat egy átlagos alkalmazott, és egy programozó bérének kiszámítására!



A program UML osztálydiagramja.

Employee

```
3  class Employee {
4      int baseSalary;
5    public:
6      Employee(int salary) {
7        baseSalary = salary;
8      }
9      int getSalary() const {
10       return baseSalary;
11     }
12     void setSalary(int salary) {
13       baseSalary = salary;
14     }
15 };
```

Programmer

```
class Programmer {                                                  17
    int baseSalary;                                                 18
    double bonusMult;                                               19
  public:                                                           20
    Programmer(int salary, double bonus) {                          21
      baseSalary = salary;                                          22
      bonusMult = bonus;                                            23
    }                                                               24
    int getSalary() const {                                         25
      return (int)(baseSalary * bonusMult);                         26
    }                                                               27
    void setSalary(int salary) {                                    28
      baseSalary = salary;                                          29
    }                                                               30
    void setBonus(double bonus) {                                   31
      bonusMult = bonus;                                            32
    }                                                               33
};                                                                  34
```

main()

```
36 int main() {
37   Employee e(300000);
38   Programmer p(300000, 2.5);
39   std::cout << e.getSalary() << std::endl;
40   std::cout << p.getSalary() << std::endl;
41 }
```

Kimenet

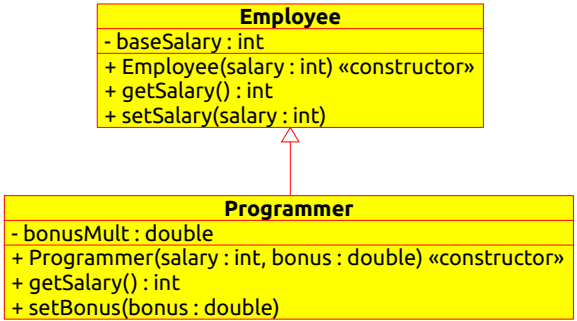
```
300000
750000
```

Probléma:

- Hasonló feladatok következménye: kódismétlés (a programozó az alkalmazott egy speciális esete).

Megoldás:

- Származtatás/öröklés (inheritance)
- Ősosztály/szülő osztály (superclass, base class) → `Employee`
- Leszármazott/származtatott/gyerek osztály (subclass, derived class) → `Programmer`


Employee
- baseSalary : int
+ Employee(salary : int) «constructor»
+ getSalary() : int
+ setSalary(salary : int)
Programmer
- bonusMult : double
+ Programmer(salary : int, bonus : double) «constructor»
+ getSalary() : int
+ setBonus(bonus : double)

- A leszármazottakban (felül)definiáljuk azokat a függvényeket, amik másképpen fognak viselkedni (`getSalary()`), mint az ősben vagy teljesen hiányoztak (`setBonus()`).
- Minden más átöröklődik, kivéve a konstruktort. (Ettől még nem feltétlenül elérhetők ezek a leszármazottban! $\rightarrow$ `private`)
- Ősosztálytól örökölt tagok inicializálása a leszármazott dolga, pl. az ős konstruktorának felhasználásával a taginicializáló listán. Ennek hiányában a fordító az ős alapértelmezett konstruktorát hívja.

Employee

```
3  class Employee {
4      int baseSalary;
5    public:
6      Employee(int salary) {
7        baseSalary = salary;
8      }
9      int getSalary() const {
10       return baseSalary;
11     }
12     void setSalary(int salary) {
13       baseSalary = salary;
14     }
15 };
```

Programmer

```
class Programmer : public Employee {                    17
    double bonusMult;                                   18
  public:                                               19
    Programmer(int salary, double bonus)                20
    : Employee(salary) {                                21
      bonusMult = bonus;                                22
    }                                                   23
    int getSalary() const {                             24
      // 'int Employee::baseSalary' is private          25
      // within this context                            26
      // return (int)(baseSalary * bonusMult);          27
      return (int)(Employee::getSalary()                28
        * bonusMult);                                   29
    }                                                   30
    void setBonus(double bonus) {                       31
      bonusMult = bonus;                                32
    }                                                   33
};                                                      34
```

main()

```
36 int main() {
37   Employee e(300000);
38   Programmer p(300000, 2.5);
39   std::cout << e.getSalary() << std::endl;
40   std::cout << p.getSalary() << std::endl;
```

Kimenet

```
300000
750000
```

main()

```
42    // Using base class is OK: Programmer inherites from Employee
43    Employee* eArray[] = { &e, &p };
44    for(unsigned i=0; i<sizeof(eArray)/sizeof(eArray[0]); i++) {
45      // Ooops! Early / static binding!!!
46      std::cout << eArray[i]->getSalary() << std::endl;
47    }
48  }
```

Kimenet

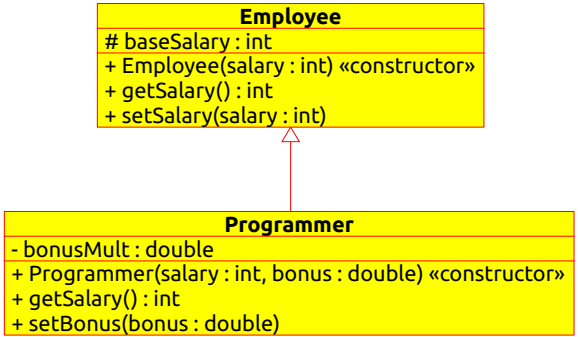
```
300000
300000
```

Probléma:

- Az ős privát tagjainak elérése körülményes a leszármazottból.

Megoldás:

- A védett (`protected`) tagok elérhetők a saját osztályukban és minden leszármazottban.
- Származtatásnál is használható ez és a `private` kulcsszó, bár szinte mindig `public`-ot használunk, azaz nem korlátozzuk tovább az öröklött láthatósági kategóriákat.


Employee
baseSalary : int
+ Employee(salary : int) «constructor»
+ getSalary() : int
+ setSalary(salary : int)
Programmer
- bonusMult : double
+ Programmer(salary : int, bonus : double) «constructor»
+ getSalary() : int
+ setBonus(bonus : double)

Probléma:

- Az ősre utaló típusú, de a leszármazott egyik példányát címző mutatóval csak az ős függvényét tudjuk hívni.

Megoldás:

- Amikor egy objektum tagfüggvényét hívjuk, mindig eldönthető, hogy az ős vagy a leszármazott felüldefiniált függvényéről van-e szó, és fordítási időben meghatározható annak címe (*early/static bindig*, *korai/statikus kötés*). Pl. `Programmer p(300000, 2.5); p.getSalary();`
- Viszont egy objektum elérhető az ős típust címző mutatóval is (biztonságos, mert a leszármazott mindennel rendelkezik, amivel az őse). A korai kötés miatt az ősben definiált függvényt tudjuk csak elérni.
- A *késői/dinamikus kötés* (late/dynamic binding) hatására a fordító *virtuális metódustáblát* (VMT, vtable) hoz létre, melyben a leszármazottak felüldefiniált függvényeinek címei is benne vannak, melynek segítségével az aktuális objektum típusának megfelelő változat is hívható, típuskényszerítés nélkül. Pl.: `Employee* pe = new Programmer(); pe->getSalary();`
- C++-ban késői kötést *virtuális tagfüggvényekkel* (`virtual`) hozhatunk létre.

Employee

```
3  class Employee {
4    protected: // private --> protected
5      int baseSalary;
6    public:
7      Employee(int salary) {
8        baseSalary = salary;
9      }
10     virtual int getSalary() const {
11       return baseSalary;
12     }
13     void setSalary(int salary) {
14       baseSalary = salary;
15     }
16 };
```

Programmer

```
class Programmer : public Employee {
    double bonusMult;
  public:
    Programmer(int salary, double bonus)
    : Employee(salary) {
      bonusMult = bonus;
    }
    // implicitely virtual
    /* virtual */ int getSalary() const {
    // OK to access protected member
      return (int)(baseSalary * bonusMult);
    }
    void setBonus(double bonus) {
      bonusMult = bonus;
    }
};
```

main()

```
35  int main() {

41    Employee* eArray[] = { &e, &p };
42    for(unsigned i=0; i<sizeof(eArray)/sizeof(eArray[0]); i++) {
43      // OK! Late / dynamic binding
44      std::cout << eArray[i]->getSalary() << std::endl;
45    }
46  }
```

Kimenet

```
300000
750000
```

Induljunk ki Rectangle12-ből, majd

- a kerület/terület lekérdezést (minden síkidomnál hasonlóan elvégezhető tevékenység) válasszuk külön a számítások elvégzésétől,
- a méretek legyenek módosíthatóak, és
- készítsünk háromszög (`Triangle`) osztályt is!

Rectangle14.h

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

Triangle14.h

```
class Triangle {
    mutable bool areaCached;
    mutable double area;
    mutable bool perimeterCached;
    mutable double perimeter;
    double mA, mB, mC;
```

Rectangle14.h

```
11    public:
12      Rectangle(
13        double width, double height) {
14        mWidth = width;
15        mHeight = height;
16        invalidateCache();
17      }
18
19      void setWidth(double width);
20      void setHeight(double height);
21      double getArea() const;
22      double getPerimeter() const;
```

Triangle14.h

```
12    public:
13      Triangle(
14        double a, double b, double c) {
15        mA = a; mB = b; mC = c;
16        invalidateCache();
17      }
18
19      void setA(double a);
20      void setB(double b);
21      void setC(double c);
22      double getArea() const;
23      double getPerimeter() const;
```

Rectangle14.h

```
23    private:
24      // called from getArea(),
25      // must be const
26      void calcArea() const;
27      void calcPerimeter() const;
28
29      void invalidateCache() {
30        areaCached = perimeterCached
31          = false;
32      }
33  };
```

Triangle14.h

```
24    private:
25      // called from getArea(),
26      // must be const
27      void calcArea() const;
28      void calcPerimeter() const;
29
30      void invalidateCache() {
31        areaCached = perimeterCached
32          = false;
33      }
34  };
```

Rectangle14.cpp

```
3  void Rectangle::setWidth(double width) {
4    mWidth = width;
5    invalidateCache();
6  }

13 double Rectangle::getArea() const {
14   if(not areaCached) {
15     calcArea();
16     areaCached = true;
17   }
18   return area;
19 }

29 void Rectangle::calcArea() const {
30   area = mWidth * mHeight;
31 }
```

Triangle14.cpp

```
void Triangle::setA(double a) {
  mA = a;
  invalidateCache();
}

double Triangle::getArea() const {
  if(not areaCached) {
    calcArea();
    areaCached = true;
  }
  return area;
}

void Triangle::calcArea() const {
  double s = (mA + mB + mC) / 2.;
  area = sqrt(s * (s - mA) * (s - mB)
    * (s - mC));
}
```

main14.cpp

```
2  #include "Rectangle14.h"
3  #include "Triangle14.h"
4
5  int main() {
6    Rectangle rArray[] = {
7      Rectangle(1., 2.), Rectangle(2., 3.), Rectangle(3., 4.)
8    };
9    const int n = sizeof(rArray)/sizeof(rArray[0]);
10   for(int i=0; i<n; i++) {
11     std::cout << "Rectangle #" << (i+1)
12               << " Area: " << rArray[i].getArea()
13               << " Perimeter: " << rArray[i].getPerimeter()
14               << std::endl;
15   }
```

Kimenet

```
Rectangle #1 Area: 2 Perimeter: 6
Rectangle #2 Area: 6 Perimeter: 10
Rectangle #3 Area: 12 Perimeter: 14
```

main14.cpp

```
16    const Triangle tArray[] = {
17      Triangle(3., 4., 5.), Triangle(5., 12., 13.), Triangle(7., 24., 25.)
18    };
19    const int m = sizeof(tArray)/sizeof(tArray[0]);
20    for(int i=0; i<m; i++) {
21      std::cout << "Triangle #" << (i+1)
22                << " Area: " << tArray[i].getArea()
23                << " Perimeter: " << tArray[i].getPerimeter()
24                << std::endl;
25    }
26    // const Shape sArray[] = { ... }; // ?!
27  }
```

Kimenet

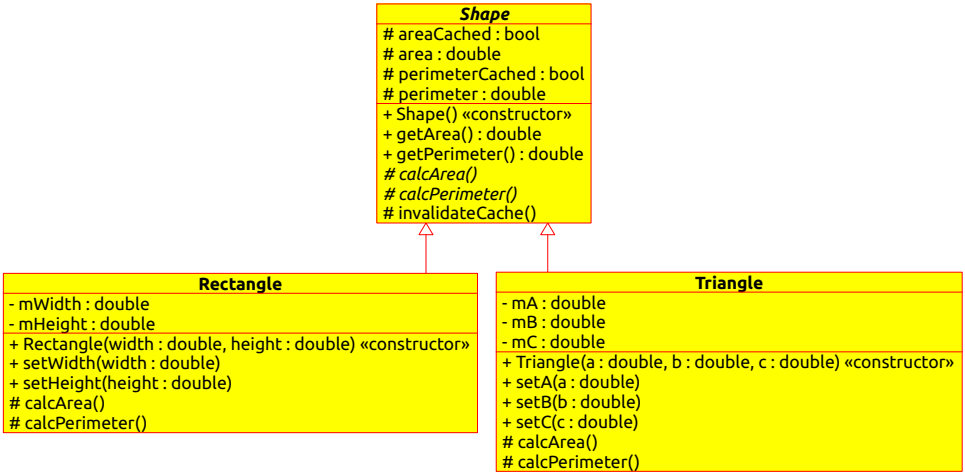
```
Triangle #1 Area: 6 Perimeter: 12
Triangle #2 Area: 30 Perimeter: 30
Triangle #3 Area: 84 Perimeter: 56
```

Probléma:

- Sok ismétlődő kódrészlet $\rightarrow$ szükség lenne egy általános síkidomra a származtatáshoz, de az milyen adatokkal írható le? Hogyan számolható ki pl. a kerülete?

Megoldás:

- *Absztrakt* (abstract) osztály: csak öröklési céllal hozzák létre, nem példányosítható a nem implementált, ún. *pure virtual* tagfüggvények (absztrakt metódus) miatt
- Ha egy absztrakt osztály kizárólag pure virtual tagfüggvényeket tartalmaz: *interfész* (interface) $\rightarrow$ elvárt szolgáltatáscsomag előírására


Shape
areaCached : bool
area : double
perimeterCached : bool
perimeter : double
+ Shape() «constructor»
+ getArea() : double
+ getPerimeter() : double
# calcArea()
# calcPerimeter()
# invalidateCache()
Rectangle
- mWidth : double
- mHeight : double
+ Rectangle(width : double, height : double) «constructor»
+ setWidth(width : double)
+ setHeight(height : double)
# calcArea()
# calcPerimeter()
Triangle
- mA : double
- mB : double
- mC : double
+ Triangle(a : double, b : double, c : double) «constructor»
+ setA(a : double)
+ setB(b : double)
+ setC(c : double)
# calcArea()
# calcPerimeter()

Shape15.h

```
4  class Shape {
5    protected: // private --> protected
6      mutable bool areaCached;
7      mutable double area;
8      mutable bool perimeterCached;
9      mutable double perimeter;
10   public:
11     Shape() {
12       invalidateCache();
13     }
```

Shape15.h

```
15      double getArea() const;
16      double getPerimeter() const;
17    protected: // private --> protected
18      virtual void calcArea() const = 0;       // pure
19      virtual void calcPerimeter() const = 0; // virtual
20      void invalidateCache() {
21        areaCached = perimeterCached = false;
22      }
23  };
```

Shape15.cpp

```
1 #include "Shape15.h"
2
3 double Shape::getArea() const {
4   if(not areaCached) {
5     calcArea();
6     areaCached = true;
7   }
8   return area;
9 }
```

Rectangle15.h

```
4  #include "Shape15.h"
5
6  // inheritance; base class -> subclass / derived class
7  class Rectangle : public Shape {
8      double mWidth;
9      double mHeight;
10   public:
11     // implicit call of base class's default constructor
12     Rectangle(double width, double height) {
13       mWidth = width;
14       mHeight = height;
15     }
```

Rectangle15.h

```
17      void setWidth(double width);
18      void setHeight(double height);
19    protected:
20      virtual void calcArea() const;
21      virtual void calcPerimeter() const;
22  };
```

Rectangle15.cpp

```
1  #include "Rectangle15.h"
2
3  void Rectangle::setWidth(double width) {
4    mWidth = width;
5    invalidateCache();
6  }


13 void Rectangle::calcArea() const {
14   area = mWidth * mHeight;
15 }
```

Triangle15.cpp

```
#include "Triangle15.h"                                  1
                                                         2
void Triangle::setA(double a) {                          3
  mA = a;                                                4
  invalidateCache();                                     5
}                                                        6


void Triangle::calcArea() const {                        18
  double s = (mA + mB + mC) / 2.;                        19
  area = sqrt(s * (s - mA)                               20
              * (s - mB)                                 21
              * (s - mC));                               22
}                                                        23
```

main15.cpp

```
1  #include <iostream>
2  #include <typeinfo> // typeid() uses RTTI
3  #include "Shape15.h"
4  #include "Rectangle15.h"
5  #include "Triangle15.h"
6
7  int main() {
8    // cannot create array of abstract objects
9    Shape* sArray[] = {
10     // new calls constructor
11     new Rectangle(1., 2.), new Rectangle(2., 3.),
12     new Triangle(3., 4., 5.), new Triangle(5., 12., 13.)
13   };
14   const int n = sizeof(sArray)/sizeof(sArray[0]);
```

main15.cpp

```
15    for(int i=0; i<n; i++) {
16      // name mangling
17      std::cout << typeid(*sArray[i]).name() << " #" << (i+1)
18                << " Area: " << sArray[i]->getArea()
19                << " Perimeter: " << sArray[i]->getPerimeter()
20                << std::endl;
21    }
```

Kimenet

```
9Rectangle #1 Area: 2 Perimeter: 6
9Rectangle #2 Area: 6 Perimeter: 10
8Triangle #3 Area: 6 Perimeter: 12
8Triangle #4 Area: 30 Perimeter: 30
```

main15.cpp

```
22    std::cout << (std::is_abstract<Shape>()
23        ? "Shape is abstract."
24        : "Shape is NOT abstract.")
25      << std::endl;
26    std::cout << (std::is_abstract<Rectangle>()
27        ? "Rectangle is abstract."
28        : "Rectangle is NOT abstract.")
29      << std::endl;
30  }
```

Kimenet

```
Shape is abstract.
Rectangle is NOT abstract.
```