

GÁBOR DÉNES
FŐISKOLA

ELŐADÁS VÁZLATOK

A C++ nyelv

Tervezés alatt álló tárgy!

Budapest
1998.

92 lap



**EZ AZ ANYAG A "C NYELV" TÁRGY KIEGÉSZÍTÉSE,
NEM KÉPEZI A SZÁMONKÉRÉS RÉSZÉT!**

Bevezetés

A C++ nyelv kialakulása:

- Objektumorientált (OO) programozási szemlélet; OO programozási nyelvek (Simula-67, Smalltalk).
- Az OO nyelvek lassúak voltak. A 80-as években célul tűzték ki a C nyelv OO bővítését, a hatékonyság megtartása mellett.
- Bjarne Stroustrup a 80-as évek közepén megalkotta a C++ nyelvet.

(A C nyelv egy másik OO bővítése: Objective C.)



- **A C++ felülről kompatibilis a C-vel, minden szabályos C program egyben szabályos C++ program is, amely ugyanazt csinálja (ez csak 99,9%-ig igaz, de a 0,1% ellenpélda elég mesterkéltten adható meg).**
- **Tehát a C++ :**
 - ◆ **C nyelv**
 - ◆ **+ objektumorientáltság**
 - ◆ **+ néhány "szintaktikai cukor"**
- **A C++ előnyei:**
 - ◆ **C előnyei**
 - ◆ **az OO programozás támogatásával nagy programok könnyebben írhatók**



- ◆ **minden gépre van C++-fordító**
- ◆ **széles körben használják és sok feladatcsoporthoz készítették C++-ban felhasználható programkönyvtárakat, így a programozónak nem kell mindig a kályhától elindulnia**
- **A WWW térhódításával járt a Java nyelv robbanásszerű elterjedése, amellyel a WWW-oldalakra nemcsak statikus szöveget és képeket jeleníthetünk meg, hanem bármit, amit egy programozási nyelv megenged (játékok, animációk, adatbázis-kezelés stb.). A Java nyelv a C++-ból származik, annak leegyszerűsítése és "letisztítása".**



A C++ nyelv új lehetőségei

Némelyik bővítés az 1989-es ANSI C-szabványba is belekerült; ezeket a mai C (nem C++)-fordítók is ismerik. A továbbiakban ezekre a bővítésekre "ANSI C" jelöléssel utalunk.

Megjegyzések a C++ programszövegben

// jellel kezdődnek és a sor végéig tartanak.
A régi /* . . . */ alakú megjegyzések is használhatók.



A // előnye, hogy a megjegyzést nem felejtjük el a végén bezárni.

Érdekesség: mesterkélt ellenpélda a C és C++ közötti kompatibilitásra:

`i=i // *megjegyzés* / 2 ;`

C-ben `"i=i/2;"` -t jelent, míg C++-ban `"i=i;"` -t!



Új típusmódosító kulcsszavak

Konstans változó megadása: const (ANSI C)

```
int i=3; i=i+4; // rendben
```

de

```
const int i=3; i=i+4; // fordítási hiba!
```

Hasonlóan const float f, const char *s stb.



volatile változó (ANSI C)

```
volatile int i;
```

Azt jelenti, hogy az `i` változót nemcsak a program, hanem külső hatás (pl. megszakítás) is megváltoztathatja. Ezt az információt a fordító optimalizálója használja fel; mi nem foglalkozunk vele.



Hatáskör-operátor: "::"

```
int n;  
void main() {  
    int n;  
    n=4; }  

```

A fenti programban egy globális és egy lokális `n` egész változó szerepel. Ez szabályos; az `n=4` értékadás a *lokális* `n`-re érvényes, a lokális változó "elfedi" az azonos nevű globális.

És ha mi mégis a globális `n`-et akarjuk használni?



A C++-ban a `::`-t a változó elé írva a globális változót kapjuk:

```
int n;  
void main()  
{  
    int n;  
    ::n=4;    // a globális n lesz 4  
}
```



Az *iostream* könyvtár

Kimenet-bemenet: a C-ben a `stdio` könyvtár `scanf`, `printf` függvénycsaládot használhattuk. Emlékeztetőként az `i` egész szám kiírása sosemeléssel:

```
printf("%d\n", i);
```

Hátrány: a kiírandó illetve beolvasandó változóknak megfelelő típusokat nekünk kellett a formátumszövegbe %-jellel megadni. Ha nem a megfelelő számú vagy típusú formátumot adtuk meg, a program akár el is szállhatott.



C++-ban a barátságosabb `iostream` könyvtár is használható.

(A program elején `#include <iostream.h>` szükséges.)

A standard kimenet neve: `cout`

A standard bemenet neve: `cin`

Kiírás (`i`, `j` változók):

```
cout << i << "," << j << "\n";
```

`i`, `j` beolvasása:

```
cin >> i >> j ;
```



A relációjelek mint nyilak irányának logikája: az adat-áramlás iránya.

Fontos! Az új lehetőségek sosem jelentik azt, hogy a régi C-megoldások nem használhatók (legfeljebb nem célszerű a használatuk)! A ki-bemenet programozására tehát a `stdio` könyvtár a C++-ban is rendelkezésre áll, akár egy programon belül az `iostream`-mel keverve is.



Szabadabb változódeklarációk

A változóknak deklarációkor rögtön kezdőérték is adható (ANSI C):

```
float r=4.51;
```

A deklarációk nem csak blokk elején helyezkedhetnek el:

```
{  
    int i;  
    cin >> i;  
    int j=i*i;    ///!!!  
    cout << i << " négyzete " << j << "\n";  
}
```

A megjelölt sor C-ben hibát adna, mert a j változót a blokk elején kellene deklarálni, nem a cin... sor után.



Típusdeklarációk (felsorolás, struktúra)

C-ben:

```
typedef enum {...} etype;  
etype e;                /* 'e' etype típusú változó */
```

vagy

```
enum etype {...};  
enum etype e;
```

C++-ban így is lehet:

```
enum etype {...};  
etype e;
```

Struktúrák definiálása hasonlóan:

```
struct styp { ... };  
styp s;
```



Dinamikus memóriakezelés

Emlékeztető: tömb mérete nem lehet változó.

```
int a=5;  
char t[a];    // hibás!
```

Dinamikusan lefoglalt memória: new operátor

```
int a=5;  
char *t=new char[a];    // helyes
```

Általában: new típus egy 'típus' típusú objektumnak helyet foglal a memóriában és egy erre mutató pointert (vagyis az objektum címét) adja vissza értékül.



A `new` típus[meret] ugyanezt csinálja, csak meret darab objektumnak foglal helyet a memóriában.

Az így, "kézileg" lefoglalt memóriaterületeket kézileg is kell felszabadítani, amikor már nincs rájuk szükségünk. Erre jó a `delete` (tömb esetén a `delete[]`) operátor.

```
{  
    int i;  
    // i-t használjuk  
    ...  
} // a blokk végén i élete véget ér, a me-  
  // móriaterülete automatikusan felszabadul
```



Ugyanez dinamikusan:

```
{  
    int *ip;  
    ip=new int;  
    ...           // *ip-t használjuk  
    delete ip;    // mi magunk szabadítjuk fel  
                  // a memóriát  
}
```

Többel:

```
{  
    int *ip; int n=15;  
    ip=new int[n];  
    ...  
    delete[] ip; }
```



Referencia típusú változók

Emlékeztető: ("A C nyelv" Előadás vázlatok)

```
void duplaz(int i)
{
    i*=2;
}
```

Ekkor `int i=6; duplaz(a);` hatására a értéke 6 marad!



Működő megoldás mutatókkal:

```
void duplaz(int *i)
{
    *i *= 2;
}
```

Hívása pl. `int a=6; duplaz(&a);`

Természetesebb megoldás referencia típusú változóval:

```
void duplaz(int& i)
{
    i *= 2;
}
```

(Hasonló a Pascal cím szerinti paraméterátadásához.)



Alapértelmezett paraméterek

A függvények fejében felsorolt paramétereknek konstans alapértelmezést adhatunk. Híváskor - ha az alapértelmezés megfelelő - a paramétert nem kell megadnunk. Például:

```
int novel(int mit, int mennyivel=1)
{
    return mit+mennyivel;
}
```

Hívása: `novel(6,3) → 9`, `novel(6) → 7`

Ha egy paraméternek alapértelmezést adunk, akkor az összes utána következővel is ezt kell tennünk.

Rossz: `void fv(int a, int b=1, char c);`



Függvénytúlterhelés (overloading)

Például

```
void fv(int a, char b);  
void fv(char *s);
```

Ez C-ben hiba, mert az `fv` függvényt két különböző módon adtuk meg. C++-ban ez szabályos.

```
fv(5, 'q'); // az első definíciót használja  
           // a fordítóprogram  
fv("abc"); // a másodikat használja
```



Objektum-orientált programozás C++-ban

- **Procedurális programozás:** a program eljárásokból, függvényekből áll; ezek "mellesleg" adatokat várnak paraméterként és adatokat adnak eredményül.
- **Objektum-orientált programozás:** a program adatokból, objektumokból áll; ezeken "mellesleg" műveleteket lehet végezni.
- Mindkét szemléletben adatok és műveletek vannak, csak a hangsúly máshova helyeződik.
- Rögzített, "kötéltáblába vésett" feladatot mindkét módon kb. ugyanolyan könnyen lehet megoldani.
- Az OO előnye a feladat megváltoztatásakor jelenik meg: a feladat kis módosítását a megoldó program kis módosításával követni lehet (folytonosság).



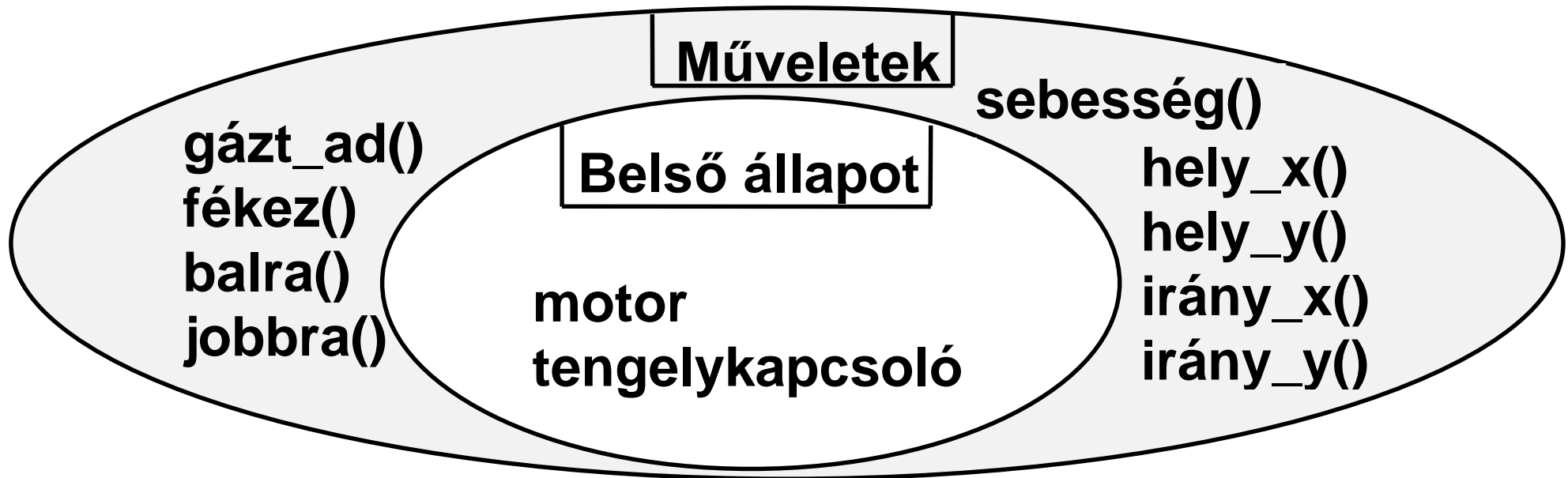
Objektumok

Mi az autó?

A műszaki szakember szempontjából: egy csomó alkatrész.

A vezető szempontjából: olyan eszköz, amellyel menni lehet: a gázpedállal gyorsíthatunk, a fékkel lassíthatunk, a kormányval kormányozhatunk.

Nagy előnye az autónak, hogy a vezetőnek nem kell tisztában lennie az autó működésével.



A belső állapotot csak a megadott műveleteken keresztül módosíthatjuk illetve csak azokon keresztül tudhatjuk meg az autó belső állapotát.

Az ábra az autók *osztályát* jelképezi; ennek megfelelő autópéldányok létezhetnek.



Egy a nevű autó felgyorsítása 80 km/h-ra:

```
Auto a;  
while(a.sebesseg() < 80) a.gazt_ad();
```

A fenti rövid programot az is meg tudja írni, aki nem ismeri az Auto osztály megvalósítását.



Osztályok

A C struktúrákhoz hasonló szintaxissal adhatók meg a C++ osztályok.

```
struct rek          // rek típus
{
    int a;
    float f;
};
rek r;              // r egy rek típusú változó
r.a=3;
```



Ugyanígy:

```
class rec {           // rec típus
    int a;
    float f;
};
rec p;                // p egy rec típusú változó
```

A fenti rek és rec típus majdnem ugyanolyan. A különbség:

p.a=3; hibás, mert a rec osztály a és f változójához nem férhetünk hozzá! (Akkor mire jók? Később meglátjuk.)



A rek struktúrának valóban pontosan megfelelő rec osztály:

```
class rec {    // rec típus
public:
    int a;
    float f;
};
```

A `public:` után felsorolt változók "publikusak", azaz egy `rec` típusú objektum `a` és `f` mezőjét szabadon felhasználhatom. A `public:` párja a `private:`. A `private:` után következő adatok nem hozzáférhetőek (az eddig megis-



mert módokon). Az osztálydefinícióban a `private:` az alapértelmezés, ezért kellett a `public:` kulcsszót kiírnunk.

```
class cl
{
    int a;
public:
    int b;
private:
    int c;
};
cl v;
v.a=1; v.b=2; v.c=3;
```

Csak a középső értékadás szabályos, a többi hibát ad!



Egyszerű példa: ember-osztály és két ember-objektum.

```
#include <iostream.h>
class ember
{
    public:
        char *nev;
        int kor;
};
void main()
{
    ember Kovacs, Szabo;
    Kovacs.nev="Kovács Ilona";
```

```
Kovacs.kor=43;
Szabo.nev="Szabó János";
Szabo.kor=25;
cout << "Kovacs neve: "
    << Kovacs.nev
    << ", kora: " <<
    Kovacs.kor << "\n";
cout << "Szabo neve: "
    << Szabo.nev
    << ", kora: " <<
    Szabo.kor << "\n";
}
```



Metódusok

Az osztályban nemcsak adatokat, hanem függvényeket (ún. metódusokat) is megadhatunk.

```
class ember {  
public:  
    char *nev;  
    int kor;  
    void kiir();  
};  
void ember::kiir()  
{
```

```
    cout << "Neve: " << nev <<  
        ", kora: " << kor << "\n";  
}  
...  
ember e;  
e.nev="Laci";  
e.kor=31;  
e.kiir();
```




Megjegyzések:

- **Az osztálydefinícióban a függvénynek csak a fejét adtuk meg.**
- **A metódus (függvény) tényleges megvalósításakor a neve elé írjuk az osztálynevet és egy " :: "-ot.**
- **A metódus megvalósításában az osztály változóira szabadon hivatkozhatunk.**
- **A metódust `objektum.metodus(param...)` alakban hívjuk meg.**



- A metódusok a privát adattagokra is hivatkozhatnak!

Pl. Autó osztály:

```
class Auto {  
public:  
    void gazt_ad();  
    void fekez();  
    float sebesseg();  
private:  
    float seb;  
};  
void Auto::gazit_ad()  
{  
    seb=seb+5;
```

```
        if(seb>120) seb=120;  
}  
void Auto::fekez()  
{  
    seb=seb-10;  
    if(seb<0) seb=0;  
}  
float Auto::sebesseg()  
{  
    return seb;  
}
```



Magyarázat: csak a `gazt_ad` és a `fekez` metódusokkal módosíthatjuk az autó sebességét.

Pl. `Auto a; a.seb=50;` szabálytalan. Ez a valóságban is így van; a sebességet nem állíthatjuk tetszés szerint. Tehát az adattagok privátként való definiálásával utánozhatjuk a valóság kötöttségeit.

Megjegyzés: a gázadás elég egyszerűen lett megvalósítva; a valódi autó nem egyenletesen gyorsul. A modellt lehet finomítani, miközben az `Auto` osztály használója ugyanazt látja (ti. a gázt és a féket).



Konstruktorok

```
Auto a;  
cout << a.sebesség( );
```

A fenti sor véletlenszerű eredményt fog kiírni, mivel a sebesség nincs inicializálva.

Az osztálynak adható egy speciális metódus, az ún. konstruktor; ennek neve megegyezik az osztály nevével, visszatérési értéke nincs, és az objektum létrejöttekor automatikusan végrehajtódik.



Pl.

```
Auto::Auto(float kezdoseb)
{
    seb=kezdoseb;
}
...
Auto a(30);           // egy új "a" autó,
                      // 30 km/h sebességgel
```

Megjegyzés: a függvényterhelés lehetősége itt is fennáll, tehát egy osztályhoz több, különböző paraméterezésű konstruktor is tartozhat.



Destruktor

A konstruktor "ellentéte", az objektum megsemmisülésekor hívódik meg automatikusan. Célja a "rendrakás".

A destruktormetódus neve: egy ~ jel után az osztály neve.

Paramétere nincs.

Pl.

```
Auto::~~Auto()  
{  
    seb=0;  
}
```



Objektumok dinamikus helyfoglalása

Ugyanolyan, mint az előre definiált típusoknál (`new`, `delete`).

```
Auto *ap;  
ap=new auto(10);  
ap->gaz();  
cout << ap->sebesseg();  
delete ap;
```

A `->` operátort osztályra (vagy struktúrára) mutató pointer-eknél használjuk. Jelentése: `a->b` ugyanaz, mint `(*a).b`



Öröklődés

Az OO programnyelvek ereje akkor mutatkozik meg, amikor egy már megoldott feladathoz hasonlót kell megoldani. Az OO programozás támogatja a meglévő kód újrafelhasználását (vagyis hogy ugyanazt a kódot ne kelljen többször megírnunk).

A "hasonlóság" egy gyakori esete a specializáció.

Példa: Kutya osztály

"Definíció": A kutyának négy lába van és ugat.

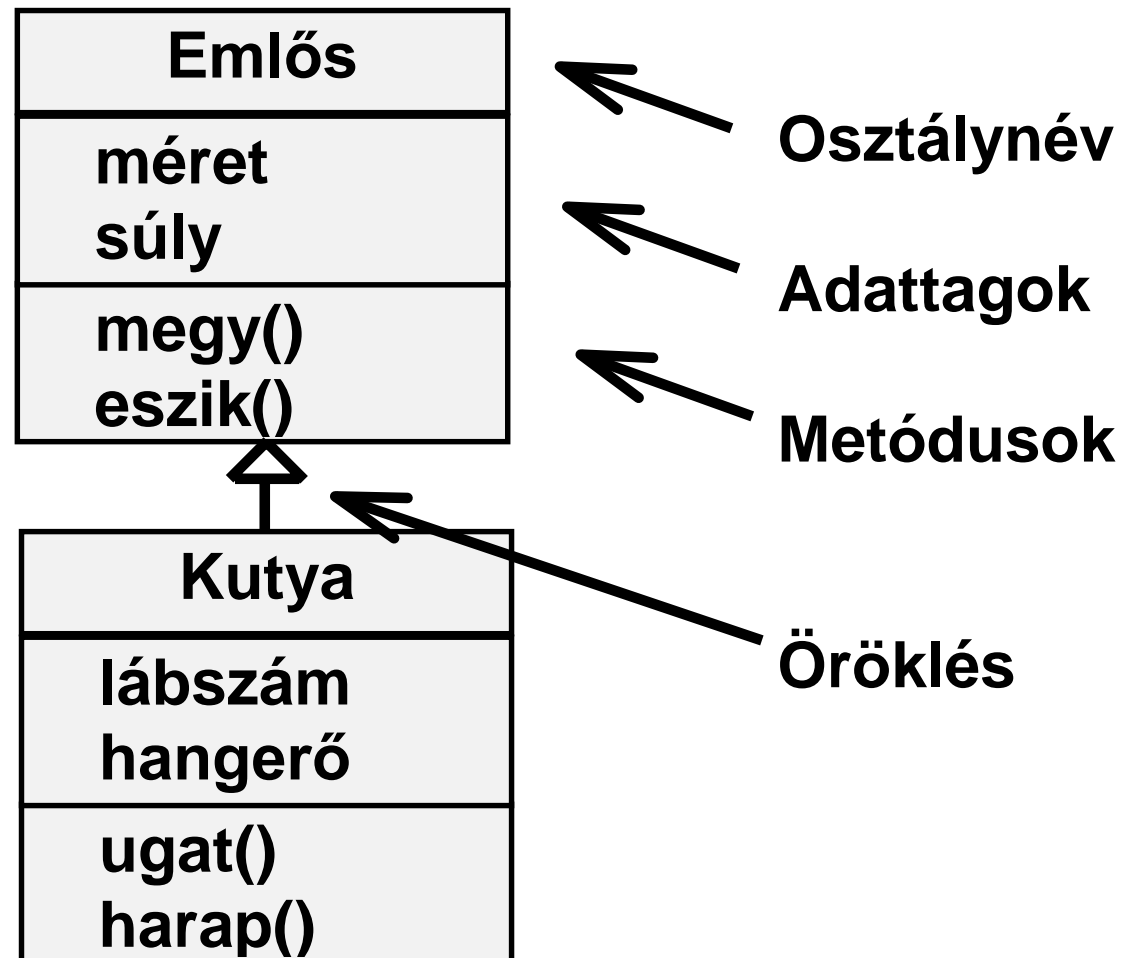
Valójában ezt úgy értjük, hogy a kutya olyan emlős (eset-



leg ragadozó), amelynek négy lába van és ugat. Ezért az emlős általános tulajdonságait külön nem soroljuk fel.

Ábrázolása:

A kutya osztály dobozában nem írjuk ki külön az emlős jellemzőit; azok megletét az öröklődést jelző nyíl fejezi ki.





A C++ definíció:

```
class Emlos {  
public:  
    Emlos(float mer, float su);  
    float merete();  
    float sulya();  
    void megy();  
    void eszik();  
private:  
    float meret;  
    float suly;  
};
```



```
class Kutya : public Emlos
{
    public:
        Kutya(float mer, float su, float hang);
        void ugat();
        void harap(Emlos& em);
        float hangereje();
    private:
        float hangero;
};

...
Kutya bodri(0.7, 12, 5);
bodri.eszik();
bodri.ugat();
```



A "class Kutya : public Emlos" sor jelentése: a Kutya osztály az Emlos tulajdonságait és metódusait örökli. A public kulcsszó jelentése itt: az osztály adatainak és metódusainak hozzáférési jogai megmaradnak (a public public marad a kutya osztályban is).

Lehetett volna emlősoosztály nélkül is definiálni a kutyát, az összes adatát és metódusát egy osztálydefinícióban megadva. A mi megoldásunk előnyei:



- Ha definiálni akarunk egy **Macska** osztályt is, akkor nem kell a semmiből indulunk; az **Emlos** osztályból kiindulva elég a csak a macskára jellemző dolgokkal foglalkoznunk.
- Ha további (emlős) állatfajokat definiálunk hasonlóan, majd eszünkbe jut egy új tulajdonság, amely minden emlősre érvényes, akkor elég az **Emlos** osztályt módosítani; a változás az öröklésen keresztül minden fajt érinteni fog.
- Később látni fogjuk, hogy ahol a programban **Emlos** típusú adatot várunk, oda nemcsak emlős, hanem kutya- és macskapéldányokat is behelyettesíthetünk.



Védett (*protected*) adatok

Az osztály `public` tagjai (adatai és metódusai) mindenhol hozzáférhetők.

Az osztály `private` tagjait csak az osztály metódusai érhetik el.

Az öröklés egy sajátos kapcsolat két osztály közt, például a `Kutya` közelebb áll az `Emlos`-höz, mint egy más, "vadidegen" osztályhoz. Az `Emlos` privát tagjait még a `Kutya` sem látja! A `protected` tagokhoz csak az osztály és annak leszármazottai férhetnek hozzá.

Példa:



```
class A {  
    public:  
        void metodus();  
    protected:  
        int adat1;  
        int get_adat2();  
    private:  
        int adat2;  
};  
int get_adat2()  
{ return adat2; }
```

```
class B : public A {  
    public:  
        void met();  
};  
void B::met()  
{  
    cout << adat1; // szabályos  
    cout << adat2; // ROSSZ!  
    cout << get_adat2(); // jó  
}  
A obj;  
cout << obj.adat1; // ROSSZ!
```



Megjegyzés: az `adat2`-t miért rejtettük el, ha a `get_adat2()` metódussal aztán úgyis hozzáférhettünk? Pl. azért, mert írni viszont nem tudja `adat2`-t senki, csak az **A** osztály metódusai.

Tehát a hozzáférési jogosultságok:

	<i>osztály- metódusban</i>	<i>leszárm. metódusában</i>	<i>máshol</i>
public	X	X	X
protected	X	X	
private	X		



Polimorfizmus

Hizlaljunk fel egy emlőst legalább 50 kilóra!

```
void hizlal(Emlos& e)
{
    while( e.sulya()<50 )
        e.eszik();
}
Emlos eml(1,30);
hizlal(eml);
cout << eml.sulya();
```



A paraméter azért `Emlos&` típusú, nem pedig sima `Emlos`, mert utóbbi esetben a függvény nem az általunk átadott paraméterrel, hanem annak másolatával dolgozna.

Általában nem emlősökkel, hanem konkrét kutyákkal és macskákkal foglalkozunk. (Vagy - hizlalásról lévén szó - disznókkal.) Tehát kellenek a

```
void hizlal(Kutya& e)
void hizlal(Macska& e)
void hizlal(Diszno& e) stb.
```

függvények.



Szerencsére nem kell minden faj osztályára külön definiálni a függvényt. A C++-ban ugyanis egy A& típusú változóba nem csak A osztályú objektumokat helyettesíthetünk be, hanem tetszőleges A-ból származó osztályút is. Hasonlóképpen egy A * pointer mutathat A-ból leszármazó objektumra is.

Példa:

```
Emlos *ep;  
ep=new Kutya(0.5, 10, 3);  
ep->megy();  
delete ep;
```



Metódusok felüldefiniálása

Az öröklés során nem csak új adat- és metódus-tagokat vehetünk fel, hanem a meglévő metódusokat is felüldefiniálhatjuk.

Például:

```
class Emlos {  
    ...  
    void eszik();  
    ...  
};
```

```
void Emlos::eszik() {  
    suly += 1; }  
class Elefant : public  
    Emlos { ...  
void eszik();  
    ... };  
void Elefant::eszik() {  
    suly += 10;  
}
```



Kérdés: Mennyit hízik Trombi?

```
void zabal(Emlos *em)
{
    em->eszik();
    em->eszik();
    em->eszik();
}
Elefant trombi;
zabal(&trombi);
```



Józan ésszel számolva 30 kilót. A C++ azonban a `zabal()` függvényben az `Emlos` osztály eredeti definícióját veszi figyelembe, ezért Trombi csak 3 kilót hízik. Ahhoz, hogy mindig az adott aktuális osztálynak megfelelő definíció aktivizálódjon, a metódust *virtuális*nak kell deklarálni:

```
class Emlos {  
    ...  
    virtual void eszik();  
    ...  
};
```



Így módosítva már a várakozásunknak megfelelően működik a program. Ökölszabály: minden olyan metódus deklarációja elé írjuk ki a `virtual` kulcsszót, amelyet várhatóan a leszármazott osztályokban felül fogunk definiálni.

Megjegyzés: a C++ a C nyelv OO bővítése. Az eleve objektum-orientáltra tervezett "tisztá" OO nyelvekben a virtuálisság az alapértelmezés, így ezt nem kell külön kulcsszóval jelezni:

```
void Elefant::eszik()  
{  
    súly += 10;  
}
```



Az elefánt tízszer annyit eszik, mint az átlag emlős. Ezt úgy is kifejezhetjük, hogy az `Emlos` evő metódusát az elefánt-tal tízszer hajtjuk végre:

```
void Elefant::eszik() {  
    int i;  
    for(i=1; i<=10; ++i) Emlos::eszik();  
}
```

Az `Emlos::` előttét fejezi ki, hogy nem az éppen most definiált elefántmetódust hívjuk, hanem az ősz osztály evőmetódusát. Gyakori, hogy felüldefiniáláskor nem írjuk újra a teljes metódust, hanem az ősz osztály metódusát meghívjuk, és még valamit csinálunk.



Konstruktorok öröklődése

```
class Emlos
{
    public:
        Emlos(float mer, float su);
        ... };
class Kutya : public Emlos
{
    public:
        Kutya(float mer, float su, float hang);
        ... };
```



A kutya konstruktora ugyanazt csinálja, mint az emlőse és még valamit. Az emlőssel közös részt kár lenne még egyszer leírni. A konstruktor definíciójakor megadható, hogy az őssztály konstruktora hogyan aktivizálódjon:

```
Emlos::Emlos(float mer, float su)
{
    meret=mer;
    súly=su;
}
Kutya::Kutya(float mer, float su,
             float hang) : Emlos(mer, su)
{ hangero=hang; }
```



A this pointer

A metódusdefiníciókban használhatjuk a `this` változót, amely mindig az aktuális objektumra mutat. Valójában implicit módon gyakran felhasználjuk:

```
adattag=ertekek;
```

ugyanaz, mint

```
this->adattag=ertekek;
```



Többszörös öröklődés

Örökléssel nemcsak egy osztályt bővíthetünk, finomíthatunk, hanem több osztály tulajdonságait is egyesíthetjük.

```
class Auto
{
    public:
        void gazt_ad();
        void fekez();
        float sebesseg();
    protected:
        float seb; };

```



```
class Raktar {
    public:
        void bepakol(Aru& a);
        void kipakol();
        int hely();
    protected:
        ...
};

class Teherauto:
    public Auto, public Raktar { };

Teherauto ifa; Aru a;
ifa.bepakol(a);
ifa.gazt_ad();
```



A Windows programozása Borland C++ -ban

Eseményvezérelt programok

Ha emberi nyelven elmondjuk, hogy mit csinál egy program, ilyen mondatokat használunk:

"Ha megnyomom a gombot, akkor ez történik."

"Ha elhúzó az egeret, az történik."

"Ha kiválasztom a menüből X-et, amaz történik."



A hagyományos programokban a programozó döntötte el, hogy hol és milyen módon kér adatokat a felhasználótól. Ennek előnye a nagyobb szabadság, hátránya viszont, hogy gyakran a program irányítja a felhasználóját, nem pedig fordítva. Másrészt szinte minden program más megoldást használ az adatbevitelre, így a felhasználónak nehezebb a dolga.



Az eseményvezérelt rendszerekben, mint amilyen a Windows, a rendszer maga megoldja, hogy a felhasználó beavatkozásainak - az eseményeknek - hatására a megfelelő programrész lefusson. A program lelke az *eseményciklus*, amelynek váza így nézhet ki:

```
Esemeny e;  
while(1) {  
    e=kovetkezo_esemeny( );  
    dolgozd_fel(e);  
}
```

Mivel ez a ciklus a rendszer része, a programjainkban nem kell külön megírni.



A Windows programozása

A Windows rendszer programozók számára nyújt egy könyvtárat, amely pl. C nyelven is elérhető. Ez azonban elég alacsony szintű; egy üres ablakot megjelenítő minimális Windows program 2-3 képernyőoldal.

Ezért erre a könyvtárra magasabb szintű, kényelmesebb könyvtárakat építettek. Az egyik ilyen a Borland OWL-je (Object Window Library), amely C++ nyelven használható és erősen objektumorientált.



Miért objektumorientált?

Az ablakozó rendszerek egyik nagy előnye az egységesség. Minden program hasonlóan néz ki és hasonlóan kezelhető. Például minden programablak mozgatható, átméretezhető. Ez arra emlékeztethet, mintha lenne egy mozgatható, méretezhető osztály, és a konkrét programok ezt használnák fel, ezt tölténék meg tartalommal.



A Windowsban nemcsak a főprogramok ablakok, hanem a nyomógombok, menüelemek, inputsorok is!

"A nyomógomb olyan ablak, amely egérekattintásra csinál valamit."

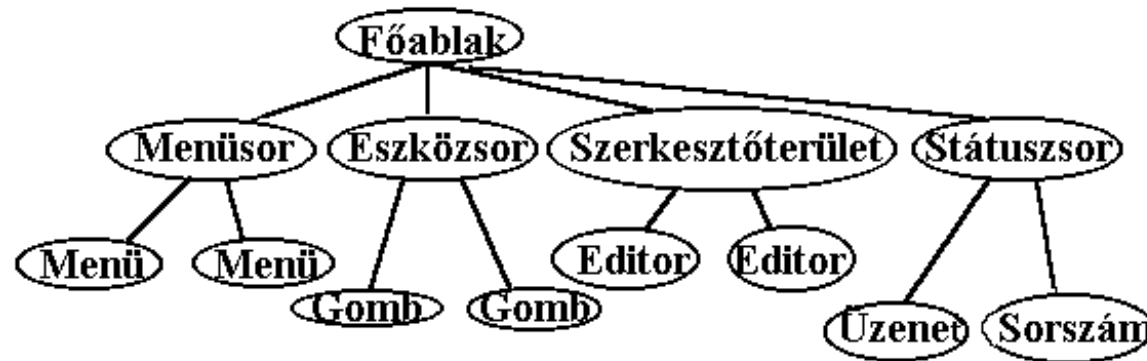
"A főablak olyan ablak, amelynek kerete van és mozgatható." stb.

A rendszer elemei így egy hatalmas osztályhierarchiát alkotnak.



Egy másik hierarchia az ablakok közt szemmel is látható: egy ablak újabb ablakokat tartalmazhat. Az ablak "szülője" a benn levő "gyerekablakoknak".

Például a Borland C++ fejlesztői környezete:



A gyerekablak sosem "lóg ki" a szülő területéből.



Minimális OWL program (egy üres ablakot jelenít meg):

```
#include <owl\applicat.h>

int OwlMain(int argc, char *argv[])
{
    TApplication app("OWL miniprogram");
    return app.Run();
}
```



- **main függvény helyett az OwlMain a program fő függvénye**
- **Egy Windows program elindítása nem egyszerű, sok mindent be kell állítani. Ezeket a TApplication osztály "tudja"; nekünk csak létre kell hozni egy ilyen típusú objektumot, és meghívni a Run() metódusát.**
- **A TApplication használatához szükség van az <owl\applicat.h> include-fájltra.**
- **A TApplication konstruktorának paraméterként megadható (nem kötelező) egy string; ez lesz az ablak fejlécére címként írva.**



Hogyan tovább?

A TApplication osztály önmagában csak egy keret; közvetlen felhasználásával csak a fentihez hasonló "üres" programok készíthetők. Valójában a TApplication tele van olyan virtuális metódusokkal, amelyek nem csinálnak semmit. Értelmes programot úgy írhatunk, ha a TApplication-ből örököltetünk egy osztályt, amelyben a megfelelő metódusokat felüldefiniáljuk.



Bevezető példa: a semmittevő program bonyolultabb, de továbbfejleszthető változata.

```
#include <owl\applicat.h>
```

```
#include <owl\framewin.h>
```

```
class SajatApp : public TApplication {  
public:  
    SajatApp();  
    void InitMainWindow();  
};
```




```
SajatApp::SajatApp() : TApplication() {}  
void SajatApp::InitMainWindow()  
{  
    SetMainWindow(new TFrameWindow(NULL,  
                                     "Itt a program neve"));  
}  
  
int OwlMain(int argc, char *argv[])  
{  
    SajatApp app;  
    return app.Run();  
}
```



- **Definiáltunk egy új osztályt (`SajatApp`). Az `InitMainWindow()` akkor hajtódik végre, amikor a program fő ablakát létre kell hozni. Ezt mi felüldefiniáltuk; a fő ablak egy keretes ablak (`TFrameWindow`). A `TFrameWindow` konstruktorának első paramétere egy szülőablakra mutató pointer; azért `NULL`, mert a fő ablakunk nem egy másik ablak része.**
- **A `SajatApp` konstruktora semmit nem csinál, csak meghívja az ősoosztály konstruktorát.**



- **Furcsa lehet, hogy olyan metódust definiálunk (`InitMainWindow()`), amelyre aztán sehol sem hivatkozunk. Persze: a `TApplication`-ból örökölt `Run()` metódus az, amely valójában meghívja az `InitMainWindow()`-t.**
- **Az első példánk azért nem csinált semmit, mert a `TApplication` csak egy üres keretprogramot adott. A mostani azért, mert a `TFrameWindow` mint fő ablak szintén csak egy semmit nem csináló alapértelmezés. A következő pld-ban egy saját ablakosztályt definiálunk, amely már reagál az egérgombnyomásra.**



```
#include <owl\applicat.h>
#include <owl\framewin.h>

class SajatWin : public TWindow {
public:
    SajatWin(TWindow *szulo=NULL);
protected:
    void EvLButtonDown(UINT u, TPoint& p);

    DECLARE_RESPONSE_TABLE(SajatWin);
};
DEFINE_RESPONSE_TABLE1(SajatWin, TWindow)
    EV_WM_LBUTTONDOWN,
END_RESPONSE_TABLE;
```



```
void SajatWin::EvLButtonDown(UINT u,TPoint& p)
{
    MessageBox("Megnyomta a bal egérgombot.",
               "Cím", MB_OK);
}
```

```
SajatWin::SajatWin(TWindow *szulo)
{ Init(szulo, 0, 0); }
```

```
class SajatApp : public TApplication {
public:
    SajatApp();
    void InitMainWindow();
};
```



```
SajatApp::SajatApp() : TApplication()  
{  
  
void SajatApp::InitMainWindow()  
{  
    SetMainWindow(new TFrameWindow(NULL,  
        "Itt a program neve", new SajatWin));  
}  
  
int OwlMain(int argc, char *argv[])  
{  
    SajatApp app;  
    return app.Run();  
}
```



- **A program a bal egér lenyomására kiír egy üzenetet.**
- **A `SajatApp` osztály annyit változott, hogy a benne létrehozott `TFrameWindow` konstruktorában egy harmadik paramétert is megadtunk. Ez a paraméter egy `TWindow`-ra (vagy leszármazottjára) mutató pointer, itt egy új, `SajatWin` osztályú objektumot adtunk meg. Így a fő keretablak tényleges tartalma ez a `SajatWin` objektum lesz.**



- **Eseménykezelő ablakok osztálydefiníciójában szükség van a `DECLARE_RESPONSE_TABLE(AblNév);` deklarációra.**

- **A**
`DEFINE_RESPONSE_TABLE1(SajatWin, TWindow)`
`EV_WM_LBUTTONDOWN,`
`END_RESPONSE_TABLE;`

jelentése: a `SajatWin` az osztálynév, a `TWindow` az ősz osztály, az `EV_WM_LBUTTONDOWN` a kezelendő esemény neve (az őt lekezelő virtuális függvény neve pedig `EvLButtonDown`).



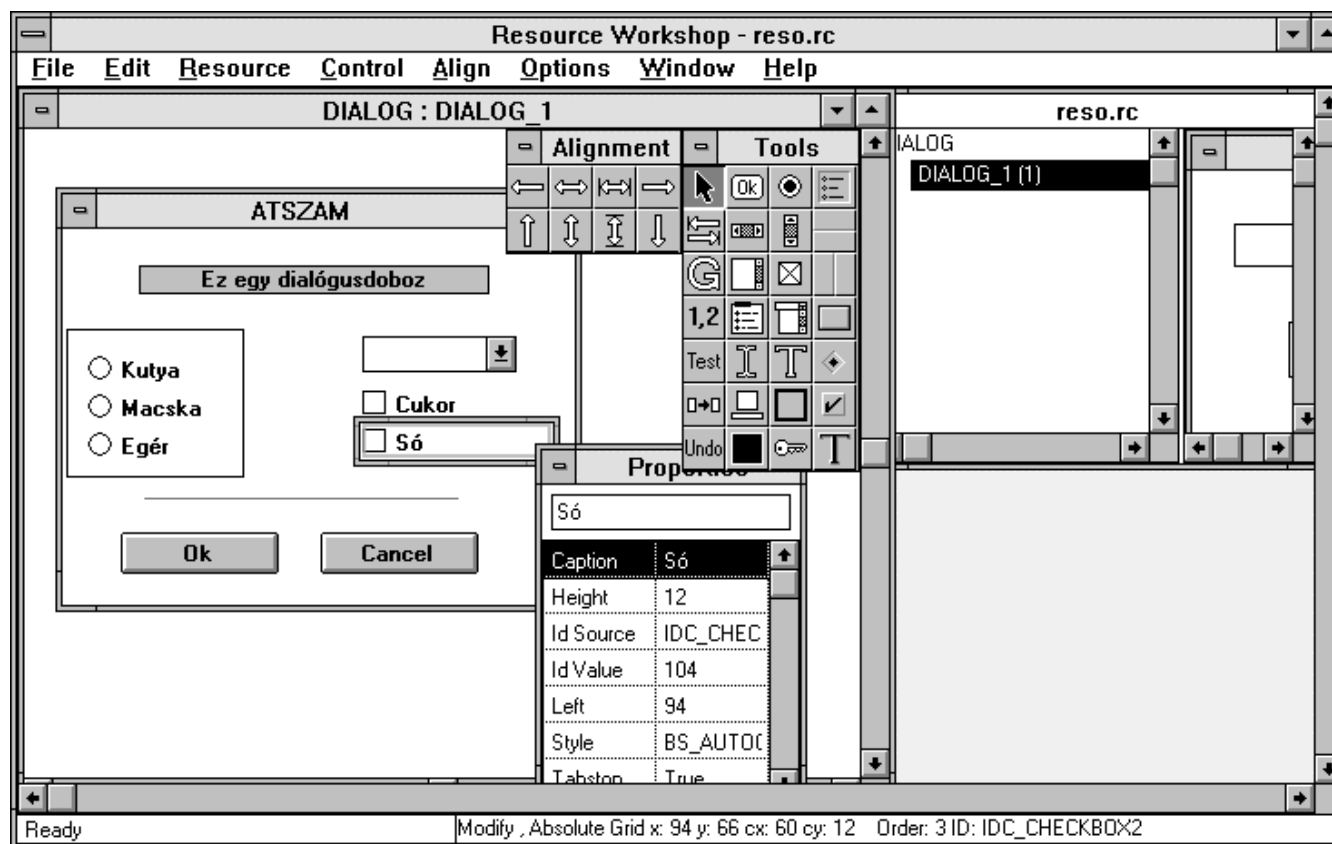
A Resource Workshop

Egy programnak jól elkülöníthetően két része van: egy belső műveletvégző rész, amelynek felépítése nem érdekli a felhasználót, és a felhasználói felület, amelyen keresztül a program használója megadja a kívánságait, és ahol az eredmény megjelenik.

A felhasználói felület elkészítése nagy mértékben automatizálható. A Resource Workshop (Erőforrás Műhely) segítségével szinte megrajzoljuk a felületet, a programszövegben nem kell a megjelenítendő mezők elhelyezkedésével



foglalkozunk; csak egy azonosítóval hivatkozunk a Workshopban létrehozott felületre.





Egy program létrehozásának lépései:

- **Az alap keretprogram megírása (ez az OwlMain függvényt és - a mi esetünkben - a SajatApp osztályt jelenti).**
- **A felhasználói felület elkészítése a Resource Workshoppal.**
- **Az eseménykezelés megírása.**



Példa: sebesség átszámítása km/h és mph között.

Az alap keretprogram:

```
class SajatApp : public TApplication {  
public:  
    SajatApp();  
    void InitMainWindow();  
};
```

```
SajatApp::SajatApp() : TApplication()  
{  
}
```



```
void SajatApp::InitMainWindow( )
{
    SetMainWindow(
        new TFrameWindow(
            NULL, "Sebesség-átszámító",
            new SajatDialog(0, DIALOGUS)));
}

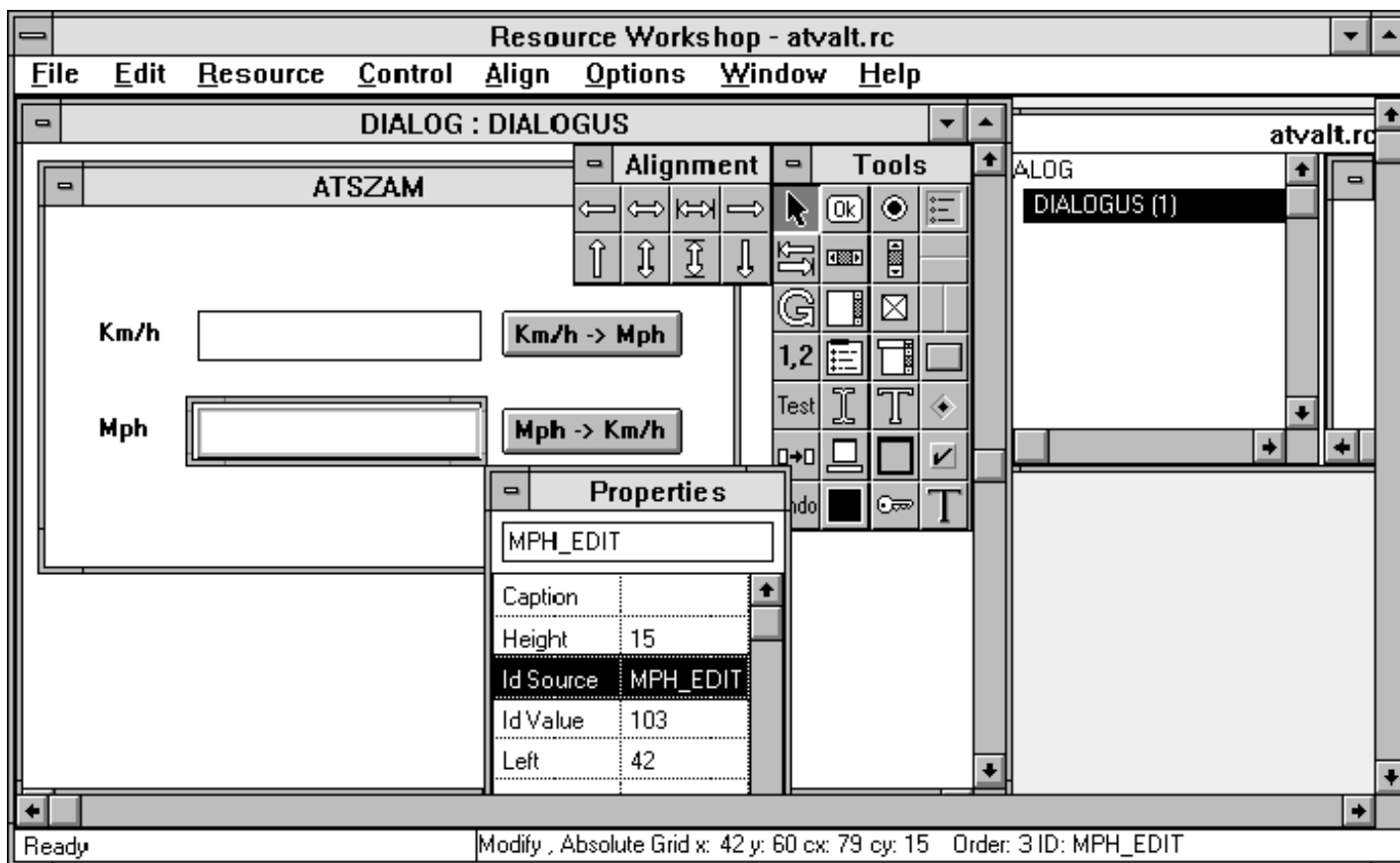
int OwlMain(int argc, char *argv[])
{
    SajatApp app;
    return app.Run( );
}
```



Az egyetlen újdonság a főablak; egy `SajatDialog` osztályú ablak. Ezt az osztályt később hozzuk létre, a hozzá tartozó felületet pedig a Resource Workshoppal készítjük el. A `DIALOGUS` paraméter egy azonosító, amellyel a RW-ban létrehozott felületet jelöljük meg.

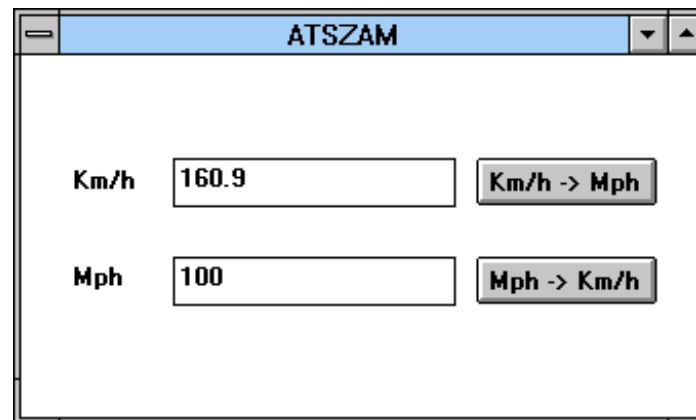


A felhasználói felület elkészítése a Resource Workshoppal:





A programunk felülete egy dialógusablak lesz, benne két inputsorral és két gombbal; a felső sorban a sebesség km/h szerint jelenik meg, az alsóban mph szerint, a két gomb pedig a kétirányú konverziót végzi el:



A resource magyarul erőforrást jelent. A Windowsban erőforrásnak nevezik a menüket, a dialógusablakokat és



elemeiket, a képeket, ikonokat stb. Ahhoz, hogy a programban használni tudjuk az erőforrásokat, hivatkozni kell tudnunk rájuk. A RW-ban minden erőforráshoz egy egész szám illetve egy azzal megegyező értékű konstansszimbólum rendelhető.

A mi programunkban az erőforrások és szimbólumaik:

Dialógusablak - DIALOGUS

Km/h inputja - KMH_EDIT

Km/h gombja - KMH_CONV

Mph inputja - MPH_EDIT

Mph gombja - MPH_CONV



A RW generál egy .rh kiterjesztésű fejlécfájlt, amelyben ezeket a konstansokat definiálja. Ezt célszerű a programunkban `#include`-dal beolvasni.

Az eseménykezelés megírása.

```
#include "atvalt.rh"
```

```
class SajatDialog : public TDialog {  
public:  
    SajatDialog(TWindow *par, int resid)  
        : TDialog(par, resid)
```



```
{  
    mph=new TEdit(this, MPH_EDIT);  
    kmh=new TEdit(this, KMH_EDIT);  
}  
protected:  
    TEdit *mph,*kmh;  
    void KmhConv();  
    void MphConv();  
  
    DECLARE_RESPONSE_TABLE(SajatDialog);  
};
```



```
DEFINE_RESPONSE_TABLE1(SajatDialog, TDialog)
    EV_COMMAND(KMH_CONV, KmhConv),
    EV_COMMAND(MPH_CONV, MphConv),
END_RESPONSE_TABLE;

void SajatDialog::KmhConv()
{
    char s[11];
    kmh->GetLine(s, 10, 0);
    float m=atof(s); sprintf(s, "%f", m/1.609);
    mph->Clear(); mph->Insert(s);
}
```



```
void SajatDialog::MphConv( )
{
    char s[11];
    mph->GetLine(s, 10, 0);
    float m=atof(s); sprintf(s, "%f", m*1.609);
    kmh->Clear(); kmh->Insert(s);
}
```

Két esemény érdekes számunkra: a két nyomógomb aktivizálása. Ezek megnyomásakor az egyik inputsor tartalmát átszámolás után betesszük a másik input-sorba.

VÉGE