Bevezetés a programozásba 2

9. Előadás: A static

ISMÉTLÉS

Osztály

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

ISMÉTLÉS

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;
}
```

A static kulcsszó

- Sok jelentését a kontextus egyértelműsíti
  - lokális változó deklarációja előtt
  - globális változó deklarációja előtt
  - tagfüggvény szignatúrája előtt
  - mező deklarációja előtt
- Általánosságban körülbelül a „maradandó, egyedüli” jelentést érdemes elképzelni

static lokális változó

- Ha egy függvényben lokális változót deklarálunk, annak minden híváskor újra lefoglalódik a memória (a stack-en), kivéve, ha static
- A static változó érvényessége szerint olyan, mint egy globális változó: nem szűnik meg a függvény végén, és értékét megtartja a program teljes futásán át
- Azonban a static változó lokális láthatóság szempontjából.

static lokális változó

```
void fv() {
  static int count = 0;
  cout << ++counter << endl;
}

int main() {
  for( int i = 0; i < 10; ++i ) {
    fv();
  }
}
```

static lokális változó

- Lokális láthatóságú globális érvényességű változó
- Érdekessége, hogy kivételesen kap kezdeti értéket: nullát
- Kitérő: egy változó a memóriában három helyen lehet
  - a stack-en: lokális változó
  - a heap-en: dinamikusan lefoglalt változó
  - a globális változók között (fix méretű hely)

static globális változó

- ..és függvény: ez az extern ellentéte, a fordítási egységből kifelé nem látszik a változó/függvény
- Deprecated (nem javasolt) feature
  - név nélküli namespace van helyette C++-ban

```
static ofstream flog((string(__FILE__)
                                +".log").c_str());

void fv() {
   flog << "log: " << változó;
}
```

Osztályok és a static

- Objektumszintű – osztályszintű
- A static jelentése: osztályszintű
- Ha mező: egyetlen példány van, közös az egész típusnak (vs minden példány saját mezővel rendelkezik)
- Ha tagfüggvény: objektum nélkül hívható metódus

A statikus metódus objektum nélkül is hívható

```
class A {
public:
  A();
  static void sfv();
};

int main() {
  A::sfv();
  A a;
  a.sfv();
}
```

Statikus mező

```
class A {
public:
   A() {++count;}
   int getCount() const {return count;}
protected:
   static int count;
};

int A::count;

int main() {
   A a,b,c;
   cout << a.getCount();
}
```

Statikus mező

```
class A {
public:
   A() ;
   int getCount();
protected:
   static int count;
};
```

interface

```
int A::count;
A::A() {
   ++count;
}
int A::getCount() {
   return count;
}
```

implementation

```
int main() {
   A a,b,c;
   cout << a.getCount();
}
```

Statikus mezők és tagfüggvények

- Statikus tagfüggvény nem hivatkozhat mezőre, kivéve ha az statikus mező
- A statikus tagfüggvénynek nincs implicit paramétere, más szavakkal nincs benne this
- Ezért olyasmit illik osztályszintű tagfüggvényként megfogalmazni, ami kihasználja, hogy nem kell hozzá objektum, pl graphicslib groutput::instance()

Kitérő

Függvénypointerek

Függvénymutatók

- Amikor függvényt hívunk, a szignatúra alapján fix a működés (paraméterek, visszatérési érték)
- Hívásnál a processzor utasításolvasó regisztere a függvény elejére ugrik
  - Ez egy mutató
- Eszerint van értelme annak, hogy ezt a mutatót változóként kezeljük, és megmutatjuk vele, hogy melyik függvényt kell hívni
  - természetesen az adott szignatúrájúak közül

Függvénymutatók

```
void fv() {
 …
}

int main() {
   void (*f)();    //deklaráció, f nevű változó
                   //ami void típusú, üres
                   //paraméterlistájú függvényt
                   //mutathat

   f=fv;           //mutassa fv()-t

   f();            //hívjuk meg amit éppen mutat
}
```

Függvénymutatók

```
void fv1(int a) { …
}

void fv2(int a) { …
}

int main() {
   int a;
   void (*f)(int);
   f=fv1;
   f(a);
   f=fv2;
   f(a);
}
```

Függvénymutatók

```
typedef void(*Fvint)(int);

void fv1(int a) { …
}

int main() {
  int a;
  Fvint f;
  f=fv1;
  f(a);
}
```

Függvénymutatók

- Sajnos metódus függvénymutatóval nem hívható: nincs meg az implicit paraméter
  - Van olyan is, hogy metódusmutató, ahol nem csak a szignatúra, de az osztály is rögzített
- De statikus (osztályszintű) metódus meghívható
- Hogyan lehet ebből objektumszintű metódushívás? Valahonnan kell keríteni egy implicit paramétert.
  - legyen a statikus függvény explicit paramétere

Workaround metódus híváshoz

```
class C {
public:
   static void sFv(C *x) {x->Fv();}
   void Fv() { … }
};

typedef void(*FvC)(C*);

class Hivo {
public:
   Hivo(C *Pobj, FvC Pfvc) {
      obj=Pobj; fvc=Pfvc;
   }
   void call() {fvc(obj);}
protected:
   FvC fvc;
   C *obj;
};
```

```
int main() {
   C *c = new C;
   Hivo h(c,C::sFv);

   h.call();

}
```

Függvénymutatók általában

- Callback függvény
  - Olyan esetekben, amikor a kontroll nem nálunk van, például realtime eszközhasználatnál
    - példa: hangkártya programozása, puffer megtelik, a callback függvény dolga a feldolgozás
- Érdemes tudni
  - hogy a virtuális függvények a háttérben metódusmutatókból készült táblázatokat használnak („vtable”). Minden virtuális függvényt tartalmazó objektumnak van egy (rejtett) mezője, ami erre a táblára mutat.

Lehetséges nyomógomb-widget stratégia: függvénymutatók

- Az összes widget, ami a programban van, az eseményciklus, és a viselkedés enkapszulálva egy Application osztályba
  - Konstruktorban a widgetek létrejönnek, bekerülnek a vector<Widget *>-ba
  - Egy run() függvényben van az eseményciklus
  - Egy-egy tagfüggvényben az egyes események lekezelése
  - Ilyenkor a tagfüggvényekre látott workaround alkalmazásával a nyomógomb widget megkaphatja az „és mi történjen ha megnyomják”-ot

Más nyomógomb megvalósítások

- Üzenet típus bővítése (akár genv::event-ből örökléssel)
  - saját eseményciklust kell hozzá írni
  - cserébe az események közé be lehet tenni „megnyomták a 'Bezár' gombot” eseményt is, amit így csak egy helyen kell lekezelni
  - a windows API körülbelül így működik, és a legtöbb ablakozó eseménytípusa is bővíthető

Más nyomógomb megvalósítások

- ŐsGomb és LeszármazottGomb
  - Az ŐsGomb lekezeli az eseményeket, és eldönti, hogy megnyomták-e. Ha igen, meghív egy virtuális „megnyomtak” tagfüggvényt
  - A LeszármazottGomb pedig megvalósítja az adott nyomógombhoz tartozó funkciót, például tartalmaz mutatót az Application objektumra, és annak tetszőleges tagfüggvényét meghívja.
  - Ezt a megoldást használja a Visual Studio, a Borland C++ Builder, a Qt, a wxWindow, és még sokan mások

Más nyomógomb megvalósítások

- ŐsGomb és LeszármazottGomb
  - Előnyei:
    - világos, egyszerű
    - minden könnyen megoldható (pl nem kell fix szignatúra, mint a függvénymutatós megoldásnál)
    - nem csak (egy) függvényt hívhat, rugalmas
  - Hátrányai:
    - Sokat kell gépelni (szokás a kódot generálni)
    - Forward deklaráció kell hozzá, mert az Application-nak kell lennie LeszármazottGomb mutató mezőjének, de annak meg kell lennie Application mutató mezőjének.