Bevezetés a programozásba 2

8. Előadás: függvénymutatók,
template bevezetés

ISMÉTLÉS

Osztály

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

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 static 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.

template

- Típussal paraméterezett függvény vagy osztály
- pl. vector<T>

```
template <typename T>
class TC {
   T mező;
   T fv(T a);
};


  ...
TC<int> tci;
TC<string> tcs;
```

template példa

```
template <typename T>
class Tomb {
   T *_m; int _s;
public:
   Tomb(int s) : _s(s) { _m=new T[_s];}
   ~Tomb() {delete[] _m;}
   T operator[](int i) const {return _m[i];}
};
...
Tomb<int> t(10); int k = t[1];
Tomb<string> t2(10); string s = t2[1];
```

template működése

- Szövegszerű helyettesítése történik fordítás közben
  - a felmerülő template deklarációknál új forráskód keletkezik a template szövegébe történő behelyettesítéssel
- Emiatt az implementáció elrejtése nem megy
- Előre fordítás sem megy
- Csak az a helyettesítés fordul le, ami deklarálva is van
  - szemantikai hibák csak a példányosítás után jönnek, előtte csak szintaktikai elemzés van

template

Nem lehet elrejteni a template implementációt, a kód a fejlécfájlba kerül

szokás .tcc fájlnevet használni

template

```
template <typename T>
class TC { ...
  void fv(T t) {
    T x=t%10;
  } ...
};
TC<int> tci; tci.fv(); //OK
TC<string> tci; tci.fv();
  // ez már nem fordul
```

template függvény

```
template <typename T>
T maxt(const T& a, const T& b)
{
  return a > b ? a : b;
}
...
char k = maxt('a','b');
int i = maxt(3,4);
```

template függvény

- függvénynél nem kötelező kiírni a template paramétert, ha az egyértelmű a függvényhívás paramétereiből
- Hasznos lehetőség meglevő típuskonverziók egységes kezelésére is

```
template <typename T>
string to_str(const T& a)
{
   stringstream ss;
   ss << a;
   return ss.str();
}
```

template proxy

- meglevő típus feldíszítése tagfüggvényekkel
- hasonlít a privát öröklésre, akkor is használható, ha nem lehet örökölni

```
template <typename T>
class matekvector {
  vector<T> v; //meglevő template
public:
  ...
  T& operator[](int i) {return v[i];}
  T max() const { ... } //extra tagfv
};
```

std::function

- még egy Gomb implementációs lehetőség, tevékenységet std::function-ben tárolni
- A szignatúra így nem beégetett, hanem fordítási idejű paraméter

```
class StdFuncButton: .. {
    std::function<void()> f;
public:
    virtual void action() { f(); }
};
void fv() {ezt kellene meghívni}
StdFuncButton <void()> *b = new StdFuncButton <void()>
(..., fv);
```

std::function

- Ha nem elég a paraméter nélküli megoldás, proxy osztály készíthető
- lambda függvény használatával kiváltható

```
template <typename T, typename Param>
class StdFuncButton: .. {
    std::function<T> f;
    Param param;
public:
    virtual void action() { f(param); }
};
...
void fv(MyApp *host) {ezt kellene meghívni}
... //MyApp konstruktorban:
StdFuncButton <void(), MyApp*> *b = new StdFuncButton
<void(), MyApp*> (..., fv, this);
```

Lehetséges nyomógomb implementációk

- üzenet alapú
- öröklődéses
- függvénypointeres
- std::function / funktor / lambda
- Szempontok:
  - szükséges kódmennyiség
  - rugalmasság
  - bonyolultság