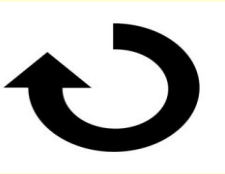
Bevezetés a programozásba

9. Előadás
Rekordok

https://wiki.itk.ppke.hu/twiki/bin/view/PPKE/BevezetesAProgramozasba201718

Több eredmény visszaadása



- Egyszerűen több paramétert veszünk át referencia szerint
  - Ezek értékével egyáltalán nem foglalkozunk, csak felülírjuk azokat
  - Így a paraméter jelentése az lesz, hogy „ide meg ide kérem az eredményt”
- Lesz még másfajta megoldás is erre a problémára
- ~~Jövő héten~~ A mai órán megnézzük

Típuskonstrukció

- Típuskonstrukció: meglévő típusokból új típus létrehozása
- Eddigi példák:
  - **T v[10]**: 10 darab T típusú változó alkot egy primitív vektort (tömböt)
  - **T m[10][10]**: 10x10 méretű (primitív) mátrix, T típusú elemekkel
  - **vector<T> v(10, e)**: 10 darab T típusú változó alkot egy STL vektort ‘e’ kezdeti értékkel
  - **vector< vector<T> > m(10, vector<T>(10,e))**: 10x10 méretű mátrix, T típusú elemekkel, ‘e’ kezdeti értékekkel

Típuskonstrukció

- Rekord: vegyes típusokból álló új típus
- C++ -ban: **`struct`**
- Az elv: az összetartozó adatok összetartozása jelenjen meg, mint nyelvi elem
- Példa: kétdimenziós koordinátákat eddig két tömbben tároltuk:

```
double x[100];
```
és
```
double y[100];
```

- Mostantól írhatjuk majd úgy, hogy:

```
koord k[100];
```

2D-s koordináták ábrázolása eddig

```
#include <iostream>
using namespace std;




int main()
{
    double x, y;
      x = 1.0;    y = 1.0;
    cout << "[" <<    x << "," <<    y << "]" << endl;
    return 0;
}
```

struct

```
#include <iostream>
using namespace std;

struct koord {
    double x, y;
};

int main()
{
    koord k;
    k.x = 1.0; k.y = 1.0;
    cout << "[" << k.x << "," << k.y << "]" << endl;
    return 0;
}
```

struct

- A **`struct`** kulcsszó jelentése: most egy új típust fogok leírni

```
struct név {
  T1 mező1, mező2, …;
  T2 mezőX, …;
  …
};
```

- A típus neve bármi lehet, ami nem foglalt még
- T1, T2, … típusoknak már ismert típusoknak kell lenniük
- A mezőnevek különbözőek legyenek

A mezők

- A rekord mezőkből áll („tag”, angolul „member”)
- Minden mező egy már ismert típusba sorolható, és változóként használható: kaphat értéket, olvasható, referálható, stb.
- A mezők használatakor a **`struct`** típusú változók után ‘**.**’ karakterrel elválasztva kell a mezőnevet írni
  - Azaz a ‘**.**’ egy operátor, amely segítségével kiválaszthatjuk a rekord egy mezőjét
- Tehát az előző példában a ‘**`k.x`**’ jelentése: a **`k koord`** típusú változónak az **`x`** mezője, amely egy **`double`** típusú érték lesz

Mező vs. változó

- Változónak szokás nevezni mindent, amit külön deklaráltunk a programkódban
  - Mezőt a rekord többi mezője nélkül nem lehet deklarálni
- Egy **`struct`** egy mezőjét azonban hasonlóan használjuk, mint egy változót: adunk neki értéket, kiolvassuk, stb.
- A szóhasználat tehát nem a képességeket, hanem a szerepet fedi: önállóan használandó (változó), vagy egy nagyobb adatcsoport része (mező)

Mi legyen mező, és mi ne?

- A rekord szerepét mindig töltse ki teljesen a mezők összessége, és ne legyen benne felesleges mező!
- **Reprezentáció**: egy absztrakt fogalom ábrázolása meglévő típusokkal
  - 2D koordináta két valós számmal
  - racionális szám két egész számmal, mint számláló és nevező
- A ciklusváltozó például nincs a mezők között: nem tartozik a fogalmat leíró, fontos adatokhoz
- Az viszont nem baj, ha a teljes értékkészlet nincsen kihasználva
  - pl.: tanuló jegye: `int mező`. (ÉK: 1, 2, 3, 4, 5)

struct és a típusok

- Az adott **`struct`** leírása után a megadott név már egy kész típus, használható deklarációkban, paraméterlistákban
  - Akár egy következő **`struct`** mezőjénél is, vektorban, primitív tömbben, stb.
- A rekord mezője is lehet vektor, vagy primitív tömb
- Fontos, hogy a forráskód fentről lefelé olvasva ezt a sorrendet betartsa!
  - (Csak arra hivatkozhatunk a kódban, amit előtte már deklaráltunk, különben a fordító nem fogja felismerni!)

struct structban I.

```
struct ember {
    string nev;
    string lakcim;
    int szuletesi_ev;
};

struct diak {
    ember e;
    vector<int> jegyek;
};
```

struct structban II.

```
struct pixel {
   char r, g, b;
};

struct kep {
     int x, y;
     vector< vector<pixel> > p;
};
```

struct és függvények

- Az új típusaink használhatóak függvény paraméterekként is

Fontos megemlíteni, hogy itt egyre jobban kezd számítani a hatékonyság, egy **`vector<double>`** mezővel is rendelkező struct érték szerint átadva lemásolódik, ami lassú (és feleslegesen nagy a memóriaigénye)

- Visszatérési értékként is használható, vagyis így lehet több eredményt egyszerre visszaadni: több, összetartozó és ezért egy típusba összefogható adatként

struct és függvények: előtte

```
struct pont{
    double x, y;
    string nev;
};

void kiir( double x, double y, string nev ){
    cout << "[" <<   x << "," <<   y << "," <<
      nev << "]" << endl;
}

int main(){
    pont p;
    ...
    kiir( p.x, p.y, p.nev );
    return 0;
}
```

struct és függvények: utána

```
struct pont{
    double x, y;
    string nev;
};

void kiir( pont p ){
    cout << "[" << p.x << "," << p.y << "," <<
    p.nev << "]" << endl;
}

int main(){
    pont p;
    ...
    kiir( p );
    return 0;
}
```

struct visszatérési típusként

```
struct koord{
    double x, y;
};

koord olvas( istream& be ){
    koord a;
    be >> a.x >> a.y;
    return a;
}

int main(){
    koord a;
    a = olvas( cin );
    cout << a.x << "," << a.y << endl;
    return 0;
}
```

Intermezzo pesszimistáknak

Az elégséges jegyhez szükséges anyag immár teljes mértékben elhangzott*.

*a hirdetés nem minősül ajánlatnak

Elődeklaráció – forward declaration

```
struct pont{
    double x, y;
    string nev;
};
void kiir( pont p ); // függvény forward deklaráció

int main(){
    …
    kiir( p );
    return 0;
}
void kiir( pont p ){ //függvény implementáció
    cout << "[" << p.x << "," << p.y << "," <<
    p.nev << "]" << endl;
}
```

Elődeklaráció – forward declaration

```
struct pont;          // struct forward deklaráció
void kiir( pont p ); // függvény forward deklaráció

struct pont{
    double x, y;
    string nev;
};
int main(){
    …
    kiir( p );
    return 0;
}
void kiir( pont p ){ //függvény implementáció
    cout << "[" << p.x << "," << p.y << "," <<
    p.nev << "]" << endl;
}
```

Elődeklaráció – forward declaration

```
struct pont;  // struct forward deklaráció

void kiir( pont p ){
    cout << "[" << p.x << "," << p.y << "," <<  //hiba: a mezőket még
    p.nev << "]" << endl;                       //nem ismerjük
}
struct pont{
    double x, y;
    string nev;
};

int main(){
    …
    kiir( p );
    return 0;
}
```

Inicializálás

- Figyelem! Az értékadás és az inicializálás két különböző dolog, csak mindkettő jele a „**=**”
- `int a = 0;` **inicializálás:** deklarálással együtt adunk kezdeti értéket
- `int a;`
  `a = 0;` **értékadás:** a deklaráció után, bármikor máskor adunk új értéket
- A `struct` esetében értéket adni mezőnként lehet, vagy egy másik `struct`-tal
- Inicializálni viszont lehet egyben is:
  koord k = { 0.0, 0.0 }; // a mezők sorrendjében!

struct inicializálás

```
struct pont{
    double x, y;
    string nev;
};

void kiir( pont p ){
    cout << "[" << p.x << "," << p.y << "," <<
    p.nev << "]" << endl;
}

int main(){
    pont p = { 1.0, 1.0, "a" };
    kiir( p );
    return 0;
}
```

struct a structban inicializálás

```
struct ember {
    string nev;
    string lakcim;
    int szuletesi_ev;
};

struct diak {
    ember e;
    vector<int> jegyek;
};

int main(){
    diak d = {{"Sanyi", " Bp", 1974 }, vector<int>(10,5) };
    cout << d.e.nev << " " << d.e.lakcim;
    return 0;
}
```

struct inicializálása

- Ez egy extra lehetőség, érdemes a használatát azonban minimálisra csökkenteni
- Hátrányok:
  - Ha változik a **`struct`** összetétele, mert bekerül egy új mező, akkor az összes inicializálás sérül, ki kell javítani őket
  - Ha változik a mezők sorrendje, akár csendes hiba is keletkezhet, pl. összekeverhető a magasság a születési évvel, mert mindkettő int
- Ugyanakkor egyszerű esetekben hatékony

struct...

- Folytatás a jövő héten