Mi a referencia?

A változódefiníció

Amikor elkezdünk programozni tanulni, az egyik legelső dolog, amit megtanulunk, hogy hogyan kell definiálni egy változót. Pl.

int szam;

De mit is jelent ez? Két dolgot:

1. Lefoglalunk egy megfelelő méretű memóriaterületet, ahol egy egész számot fogunk tárolni.
2. Létrehozunk egy azonosítót, amivel a programunkban azonosítani fogjuk azt a memóriaterületet, ahol a változót tároljuk. Vagyis ezzel az azonosítóval fogunk hivatkozni (=referálni) erre a memóriaterületre. (A lefordított program nem tartalmazza a változók nevét, hanem helyettük a változók memóriacímeit használja.) Az azonosító a definíciójakor hozzárögzül a memóriaterülethez, nem állítható át más memóriaterületre.

Ezt a két funkciót a C/C++ nyelvekben külön is tudjuk választani:

1. Dinamikusan foglalunk memóriát. A foglaló függvények/operátorok (malloc, new, stb.) a lefoglalt memória címét adják vissza.
2. Lefoglalt memóriaterülethez rendelhetünk azonosítót. Erre két mód van: a pointer és a referencia.

A pointer

A pointer memóriacím, amit általában változóban tárolunk. A pointert tároló változót is szoktuk pointernek hívni. Mivel a pointert tároló változó is változó, neki is van memóriacíme. A pointer funkciója, hogy olyan memóriaterületre hivatkozzunk vele, amit valaki már lefoglalt. Előnyös tulajdonságok:

1. A pointer változó értéke megváltoztatható.
2. [] operátor segítségével tömbként használható.

Hátránya, hogy az általa mutatott érték eléréséhez operátort kell használni, pl. **p=6;* vagy *p[1]=3*. Ha referenciával (azonosítóval) adott változóra akarjuk állítani, szintén operátor szükséges, pl. *int a,*p=&a;*

Létezik olyan pointer, ami nem mutat sehová, és ez felismerhető. Ez a NULL pointer.

A referencia

A referencia sokkal jobban hasonlít a változóhoz, mint a pointer. Ahogy egy változóhoz definíciójakor azonnal hozzárendelődik a memóriaterület (ami automatikusan kerül lefoglalásra/felszabadításra), úgy a referenciához is a definíciójakor rendelődik hozzá a memóriaterület, de ez már egy korábban lefoglalt memóriaterület kell legyen. A referencia nem egy változó, mint a pointer, hanem csak egy azonosító, ezért nincs is címe. (*int a; int &b=a; => &b* az *a* változó címét jelenti, míg *int *c=&a;* esetén *&c* a pointer címe, az *a* változó címe maga *c* értéke.) És nem is változtatható meg: amíg a referencia létezik, mindig ugyanoda mutat, ahogy egy változó neve is mindig a változó memóriaterületére mutat.

- És ha egy függvény referencia paramétert kap, akkor az mindig arra a változóra mutat, amivel először hívjuk a függvényt? Nem. A függvény paramétereit minden függvényhívásnál definiáljuk, azaz minden függvényhívásnál más-más memóriaterülethez rendelhetjük hozzá. Ez ugyanolyan, mint a változók esetében: ha egy függvényt többször meghívunk, annak lokális változói nem biztos, hogy ugyanott jönnek létre. A rekurziónál pont ezt használjuk ki.

Ez a megváltoztathatatlanság az ára annak, hogy a referencit használó kódunk „szebb”, azaz a referenciát ugyanúgy használhatjuk, mint egy változót, nem kell operátor a hozzárendeléshez meg a hivatkozott memóriaterület eléréséhez.

A referencia két leggyakoribb felhasználása:

- függvényparaméterként és
- függvény visszatérési értékként.

Mindkét esetben lehet konstans és nem konstans a referencia:

- Ha konstans a referencia, akkor azért használjuk, hogy kevesebb adatot kelljen mozgatni. Ez jellegzetesen struktúrák esetén jelent problémát: a struktúrának sok adattagja lehet, akár tömbök is. Ha referenciát használunk, akkor csak egy pointer másolódik. C++-ban azért fontos a struktúra, mert az objektumok adatait struktúrák tárolják.
- Ha nem konstans a referencia, akkor paraméter esetében a hívás helyén lévő adat módosítható. Referencia visszatérési típus esetén pedig a visszaadott változó lesz módosítható.

Pl.

```
int & randomelem(int t[], int n) {
    return t[rand() % n];
}

int tomb[20];
randomelem(tomb, 20) = 18; // függvény lehet balérték, ha referenciát ad vissza
```

A referencia mindenütt helyettesítheti a pointert, ha nem szükséges megváltoztatni a referencia által mutatott memóriaterületet, bár sok esetben célszerűbb pointert használni, mert az egyszerűbb.

Néhány példa a használatra

Ne kelljen mindig kiírni egy bonyolult adatszerkezetet:

```
for (i = 0; i < n; i++) {
    double &d = tomb[i].value;
    sum += d*d – 4 * v / d;
        // Nem kell kiírni, hogy tomb[i].value*tomb[i].value – 4*v/tomb[i].value
}
```

Tudunk NULL referenciát is csinálni:

```
struktura &keres(struktura tomb[], int n, int mit) {
    for (i = 0; i < n; i++)
        if (tomb[i].evszam == mit)
            return tomb[i];
    return *(int*)NULL;
}
```

```
struktura &talalt = keres(t, 100, 2016);
if (&talalt == NULL) {
    printf("Nem találtam meg.\n");
}
else {
    printf("Nev: %s\n", talalt.nev);
}
```

Bár ez nem tipikus felhasználás, más tanárnál írt ZH-ban vagy a nagy ZH-ban kerülendő, akárcsak a többi, ami ezután jön, mert nem biztos, hogy a javító felismeri, hogy helyes, amit írtok… (Ez a villamosmérnök hallgatóknak szólt.)

Ha a találatot így akarjuk kezelni, akkor a pointeres megoldás jobb lett volna.

Ugyanezt csinálja, de elegánsabb, nem „hekkelős” megoldás ez lenne:

```
struktura &keres(struktura tomb[], int n, int mit) {
    for (i = 0; i < n; i++)
        if (tomb[i].evszam == mit)
            return tomb[i];
    throw nem_talalt;
}

try {
    printf("Nev: %s\n", keres(t, 100, 2016).nev);
}
catch(nem_talalt)
    printf("Nem találtam meg.\n");
}
```

Dinamikus memória referenciával:

Ezt is pointerrel célszerű, de lehetne referenciával is trükközni:

```
double &d = *(double*)malloc(sizeof(double));
if (&d == NULL)…
d = 28.1;
…
free(&d);
```

Persze itt egy sima double d; lett volna az igazi megoldás.

Tömböt is lehet, csak az már tényleg nagyon ronda:

```
double &d = *(double*)malloc(sizeof(double)*10);
if (&d == NULL)…
(&d)[3] = 28.1;
…
free(&d);
```