# C++ referencia

Izsó Tamás

2017. február 17.

## 1. Bevezetés

A C++ nyelvben nagyon sok félreértés van a referenciával kapcsolatban. A Legyakoribb hibák:

- Sokan összetévesztik a pointerrel.
- Keveset alkalmazzák a programokban.

A második pontban történt elmozdulás, ebben a cikkben én inkább az első ponttal foglalkoznék.

## 2. Referencia a c++-ban

**Referencia**

A referencia egy létező objektum alternatív neve.

- Definiálásánál meg kell adni azt az objektumot is, amelyet alternatív névvel látunk el. Későbbiek során az alternatív nevet nem ruházhatjuk másra.
- Kifejezésekben úgy viselkedik mint amelyik objektumnak a *keresztnevét* viseli.

A második pontban direkt kerültem, azt a kifejezést, hogy *akire hivatkozik*, mivel az új névvel is ellátott objektum elérésénél nem mindig van indirekció.

A tankönyvekben a következő egyszerű mintapéldával szokták a referencia felhasználását szemléltetni.

```
void swap(int* a, int* b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

int foo() {
   int a=2;
   int b=6;
   swap(&a,&b);
   ............
}
```

```
void swap(int& a, int& b) {
  int temp = a;
  a = b;
  b = temp;
}

int foo() {
   int a=2;
   int b=6;
   swap(a,b);
   ............
}
```

Az első oszlopban pointer segítségével vesszük át a paramétereket, mivel a megváltoztatott értékeket vissza is szeretnénk kapni. Ez a módszer egyaránt alkalmazható a C és a C++ nyelvben is. A második példában pedig a C++-ban bevezetett referencia segítségével oldjuk meg a paraméter átadását. De vajon miért kellett Bjarne Stroustrup-nak egy új nyelvi elemet bevezetni? Netán azért, hogy pár rosszul képzet programozó, aki hadilábon áll a pointer jelöléssel, elkerülje a számára nehéz részeket?

Egy új nyelvi elem bevezetése hátrányokkal is járhat. Egyrészt a C programozó megszokta, hogy-ha a skalár függvényparaméter előtt nem szerepel a címoperátor, akkor az átadott paramétert a függvény nem tudja megváltoztatni. Másrészt az & szimbólum egyrészt címképző operátor, másrészt a referencia jelölésére is felhasználjuk, így esetleg aki először találkozik vele, az összekeverheti a két fogalmat.

A C++-ban nem csak a függvényeket, de az operátorokat is felül lehet definiálni. Tegyük fel, hogy van egy **class** Date {...}; nevű osztályunk, és az inkrementál operátort úgy szeretnénk átdefiniálni, hogy a következő napra állítsa a dátumot. Mivel az operátorfüggvény megváltoztatja a paraméter értékét, ezért a paraméter címét kellene átadni.

```
void operator++( Date* data )
  date->day++;
  // if( date->day > aktuális hónap napjai )
  ...
}

int foo() {
  Day ma(2017,02,17);

  ++&ma;
  ............
}
```

A ++&ma hibás, hiszen a &ma egy cím, és nem az objektum, hanem a objektum címe fog megváltozni. Egyébként már a **void operator**++( Date* today ) definícióval is baj van, mivel a pointer viselkedését nem változtathatjuk meg. Referencia esetében ez az eset nem fordul elő, mivel nem kell használni a címképző operátort.

## Jó tudni a referenciáról

```
int foo() {
  int a = 2;
  int& ra = a;
  cout << ra << endl; // eredmény 2
  ra = 10;
  cout << a << endl;  // eredmény 10

  int& harom=3;
  const int& negy = 4;
}
```

Amíg az **int**& harom=3 hibás, mivel a jobb oldalon nem balérték szerepel, addig a **const int**& negy=4 helyes. Az utóbbi esetben létrejön egy temporális név nélküli egész változó, aminek az alternatív neve negy lesz. Ezt nagyon fontos szemelőt tartani függvényeknél a temporális értékek átadásakor.

### Temporális paraméter átvétele referenciával

```
class Rational {
   int numerator;
   int denominator;
public:
   Rational(int n, int d=1) : numerator(n), denominator(d) {}
   ...
};

Rational add_1(Rational& a, Rational& b) {...}
Rational add_2(const Rational& a, const Rational& b) {...}

void foo() {
   Rational a(2,5);
   Rational b(7,13);
   Rational c;
   c = add_1(a,b);
   c = add_2(a,b);


   c = add_1(4,b);
   c = add_2(4,b);
}
```

A c = add_1(4,b) hibás mert a paraméter nem konstans referencia, ellenben a c = add_2(4,b) helyes. Habár a fordító nem talál olyan add_2 nevű függvényt, amelynek az egyik paramétere egész, a másik Rational, de a 4-et a Rational egyparaméteres konstruktorával át tudja alakítani, létrehoz egy temporális objektumot, amit már a konstans referencia fogadni tud. (Szerencsére semmi újat nem kellett megtanulnunk az előző példához képest, csak az ott megszerzett tudásunkat kellett alkalmazni.)

# 3. Tömb

### Tömbök

A tömbökkel csak két műveletet tudunk végezni:

1. megállapíthatjuk a `sizeof` operátorral a méretét
2. lekérdezhetjük az első elemének a címét

Kifejezésben a tömb neve a tömb első elemének a címét adja vissza. Ennek a típusa azonos a tömb egy elemére mutató pointer típusával.

A C nyelvben csak egydimenziós tömbök vannak, de a tömb elemei lehetnek tömbök.

```
double array[50];
double* pa;
pa=array;
```

Ha a tömb neve a függvényhívásánál a paraméterlistában szerepel sum(array,50), akkor a fentiek értelmében a tömb első elemének a címét fogja a függvény megkapni. Ezért a függvényeknél a következő definíciók azonosak.

### Tömb(?) paraméter

```
int sum(int* t, int n);
int sum(int t[], int n);
int sum(int t[100], int n);
```

Általánosan úgy fogalmazhatnánk, hogyha a tömb bármely elemének a címét és az előtte és utána lévő elemek darabszámát átadjuk[1], akkor a pointer aritmetika segítségével bejárhatjuk az össze elemet[2]. A fenti függvénydeklarációkban az első deklaráció fejezi ki a tömb átadásának a szemantikáját a többi jelölelés csak syntactic sugar `https://en.wikipedia.org/wiki/Syntactic_sugar`. A syntactic sugar a nyelv oktatásában nehézségeket vet fel.

### Tömb(?) paraméter

```
int sum1(int* t, int n);
int sum2(int t[], int n);
int sum3(int t[100], int n);
int foo() {
   char* s1 = "alma";
   char s2[] = "alma";
   char s3[100] = "alma";

   char s4[];
   int t1[] = { 1,2,3 };
   int* t2 = t1;

   int t3[] = t1;
```

Kérdések amelyeket meg kell válaszolnunk:

1. Ha sum2(**int** t[], **int** n) deklarációban az **int** t[] azonos jelentésű minta az **int*** t, akkor máshol is használhatom ezt a pointerek jelölésére (például **int** t3[] = t1) ?
2. Amint azt az előbb láttuk az **int** t[] egy pointer, akkor a **char** s2[] = "alma" definíció azonos-e a **char*** s1 = "alma" definícióval, és ugyanakkor miért hibás a **int** t3[] = t1?
3. Ha a **char** s2[] = "alma"-nál és a sum2(**int** t[], **int** n)-nél nem kellett a tömb méretét megadni, akkor definiálhatók méret nélküli tömbök is (például **char** s4[]) ?
4. Ha **int** sum3(**int** t[100], **int** n) deklarációban megadtuk a tömb méretét, akkor ez a függvény csak száz elemű tömböt képes átvenni?

A kérdések megválaszolását az olvasóra bízom ☺.

[1] Természetesen az első elemet és a tömb méretét szoktuk átadni

[2] Kihasználjuk, hogy az elemek a memóriában címfolytonosan helyezkednek el.

# 4. Referencia tömbökre

### Tömb(?) paraméter

```
int meret(int t[] ){
   cout << sizeof(t) << endl;
}

int foo() {
   int t[100];
   cout << sizeof(t) / sizeof(int) << endl;
   meret(t);
}
```

A tömb jelentésének megadása alapján a foo függvényben a **sizeof**(t) a tömb tényleges méretét adja vissza, és ha azt elosztjuk egy elem méretével, akkor a tömb elemeinek számát, jelen esetben 100-at fogunk kapni. Ugyanakkor mivel kifejezésben a tömb neve a tömb első elemére mutató pointerre konvertálódik, ezért a meret() függvény a pointer méretét írja ki. Mi van ha a tömböt referenciaként vesszük át?

### Tömb átvétele referenciával

```
void rfoo(int (&t)[100]) {
   cout << sizeof(t) / sizeof(int) << endl;
}

int main() {
   int a[100];
   int b[100];
   int c[101];
   rfoo(a);
   rfoo(b);
   rfoo(c);
   ...
}
```

Az rfoo(**int** (&t)[100]) függvény egy 100 elemű tömböt referenciaként vesz át. A referenciánál elmondtuk, hogy kifejezésekben a referencia hasonlóképpen viselkedik, mint az eredeti objektum. Ezért az rfoo() függvényben a **sizeof**(t) operator a tömb tényleges méretét adja vissza, ebből kifolyólag a program az outputra a tömb elemeinek a számát fogja kiírni. Ugyanakkor a rfoo(c) függvényhívás szintaktikailag hibás, mert referenciaként csak azonos típusú adatokat vehetünk át. C++-ban ezt az akadályt azonban könnyen áthidalhatjuk.

```
template <int N>
void rfoo(int (&t)[N]) {
   cout << N << endl;
}
```

Ezért nem szabad a referenciát a pointerrel összetéveszteni!