Imperatív programozás
Paraméterátadás

Kozsik Tamás és mások

Függvénydeklarációk és -definíciók C-ben

```
int f( int n );
int g( int n ){ return n+1; }

int h();
int i(void);

int j(void){ return h(1); }

int h( int p, int q ){ return p+q; }

extern int k(int,int);

int printf( const char *format, ... );
```

Az `f`-et csak deklaráltuk, nem definiáltuk. A `g`-t definiáltuk is.

A `h`-t úgy deklaráltuk, hogy nem adtuk meg, hogy milyen paramétere(i) van(nak). Az ANSI C előtt még így deklaráltunk függvényeket, és visszafelé kompatibilitási okokból ez a forma még mindig legális – de kerülendő! Ha azt akarjuk kifejezni, hogy egy függvénynek nincsen paramétere, akkor írjuk úgy, mint az `i` esetében: legyen egy `void` a paraméterlistában. (Szokás ezt úgy mondani, hogy a `h`-hoz egy *nem prototípusos* deklarációt adtunk.)

A C++-ban a `h` deklarációja mást jelent, mint a C-ben: paraméter nélküli függvényt deklarál. A C++-ban már nincs meg az a lehetőség, hogy elhagyjuk a deklarációban a paraméterlistát!

A `j` esetén egy paraméter nélküli függvényt definiáltunk. A definícióban a `h`-t egy paraméterrel hívjuk – bár rögtön ez után két paraméterrel definiáljuk a `h`-t. Ebből persze baj lesz: fordítási hibát nem kapunk, de a `h` hibásan, definiálatlanul fog működni a `j`-ből meghívva (definiálatlan memóriatartalmat fog használni).

A `k` deklarációja azt mutatja be, hogy egyrészt a függvénydeklarációk esetén kiírhatjuk az `extern` kulcsszót, másrészt, hogy a prototípusból a paraméterek nevét nyugodtan elhagyhatjuk – azoknak legfeljebb csak dokumentációs szerepük van.

Az utolsó példa a jól ismert `printf` függvény deklarációja. A három pont (angolul *ellipsis*) a paraméterlistában azt jelzi, hogy a függvénynek az első, explicit paramétere utan még akárhály további paramétere lehet. Nem könnyű ilyen jellegű függvényeket írni. A trükk itt az, hogy a `format` paraméter tartalmazza azt az információt, hogy mik lesznek a további paraméterek: hány paraméter lesz, és milyen típusúak lesznek.

Paraméterátadási technikák

- **Érték szerinti** (pass-by-value, call-by-value)
- Érték-eredmény szerinti (call-by-value-result) – Ada
- Eredmény szerinti (call-by-result) – Ada
- Cím szerinti (call-by-reference) – Pascal, C++
- **Megosztás szerinti** (call-by-sharing)
- Igény szerinti (call-by-need) – Haskell

- Név szerinti (call-by-name) – Scala
- Szövegszerű helyettesítés – C-makró

Érték szerinti paraméterátadás

```c
int lnko( int a, int b )
{
    int c;
    while( b != 0 ){
        c = a % b;
        a = b;
        b = c;
    }
    return a;
}
int main()
{
    int n = 1984, m = 365;
    int r = lnko(n,m);
    printf("%d %d %d\n",n,m,r);
}
```

```python
def lnko(a,b):
    while b != 0:
        a, b = b, a%b
    return a

n = 1984
m = 365
r = lnko(n,m)
print(n,m,r)
```

Bemenő szemantika

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

int main()
{
    int n = 1984, m = 365;
    swap(n,m);
    printf("%d %d\n",n,m);
}
```

```python
def swap(a,b):
    c = a
    a = b
    b = c
```

```
n = 1984
m = 365
swap(n,m)
print(n,m)
```

Mutató átadása érték szerint

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



int main(){
   int *n, *m;
   n = (int*) malloc(sizeof(int));
   m = (int*) malloc(sizeof(int));
   if( n != NULL ){
      if( m != NULL ){
         *n = 1984; *m = 365;
         swap(n,m);
         printf("%d %d\n",*n,*m);
         free(n); free(m);
         return 0;      // success
      } else free(n);
   }
   return 1; // allocation failed
}
```

Cím szerinti paraméterátadás emulációja

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



int main(){
   int n = 1984, m = 365;
   swap(&n,&m);
   printf("%d %d\n",n,m);
}
```

Cím szerinti paraméterátadás – Pascal

```
program swapping;

   procedure swap( var a, b: integer );   (* var: cím szerint *)
   var
      c: integer;
   begin
      c := a; a := b; b := c
```

```
   end;

   var n, m: integer;

begin
   n := 1984; m := 365;
   swap(n,m);
   writeln(n,' ',m)     (* 365 1984 *)
end.
```

Cím szerinti paraméterátadás – C++

```
#include <cstdio>

void swap( int &a, int &b )   /* &: cím szerint */
{
    int c = a;
    a = b;
    b = c;
}

int main()
{
    int n = 1984, m = 365;
    swap(n,m);
    printf("%d %d\n",n,m);
}
```

Érték-eredmény szerinti paraméterátadás: Ada

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure Swapping is

   procedure Swap( A, B: in out Integer ) is  -- be- és kimenő
      C: Integer := A;
   begin
      A := B; B := C;
   end Swap;

   N: Integer := 1984;
   M: Integer := 365;

begin
   Swap(N,M);
   Put(N); Put(M);  -- 365 1984
end Swapping;
```

Az érték-eredmény szerinti paraméterátadásról szót ejtettünk már az előző előadáson. Mechanizmusa hasonlít az érték szerinti paraméterátadáshoz, de a hívás végén a formális paraméternek megfelelő lokális változó tartalma visszaíródik az aktuális paraméterbe. Tehát nem csak a híváskor történik információátadás a hívóból a hívottba (bemenő paraméter), hanem a hívás befejeződésekor is a hívottból a hívóba (kimenő paraméter). Az Adában az `in out` kulcsszavak jelzik, hogy be- és kimenő paraméterekről van szó, és az `Integer` típusú ilyen paraméterek esetén érték-eredmény szerinti paraméterátadás történik.

Természetesen ennek a működésnek szükséges feltétele az, hogy az aktuális paraméter értéket tudjon kapni: ne mondjuk egy literál legyen, hanem egy úgynevezett *balérték*-kifejezés, egy olyan kifejezés, amely állhat egy értékadás baloldalán. Nyilván értelmetlen lenne a `Swap(1984,365)` hívás, de az `N` és `M` változók megfelelő aktuális paraméterek.

Megosztás szerinti paraméterátadás

```c
void swap( int t[] )
{
    int c = t[0];
    t[0] = t[1];
    t[1] = c;
}
int main()
{
    int arr[] = {1,2};
    swap(arr);
    printf("%d %d\n",arr[0],arr[1]);
}
```

```python
def swap(t):
   t[0], t[1] = t[1], t[0]

arr = [1,2]
print(arr)
swap(arr)
print(arr)
```

Ez nem cím szerinti paraméterátadás

```c
void twoone( int t[] )
{
    int arr[] = {2,1};
    t = arr;
}

int main()
{
    int arr[] = {1,2};
    twoone(arr);
    printf("%d %d\n",arr[0],arr[1]);
}
```

```python
def twoone(t):
    t = [2,1]

arr = [1,2]
print(arr)
twoone(arr)
print(arr)
```

Automatikus változó visszaadása?

C: hibás

```c
int *twoone()
{
    int arr[] = {2,1};
```

```
    return arr;
}
```

Python: ok

Nem automatikus, hanem dinamikus tárolású változót ad vissza!

```
def twoone():
    arr = [2,1]
    return arr

print(twoone())
```

Igény szerinti paraméterátadás

```
f True  a _ = a
f False _ b = b + b

main = print result
  where result = f False (fact 20) (fact 10)

        fact 0 = 1
        fact n = n * fact (n-1)
```

Ebben a Haskell programban az `f False (fact 20) (fact 10)` kifejezés értékét írjuk ki a képernyőre. Az `f` függvény ezen meghívása során csak azok az aktuális paraméterek értékelődnek ki, amelyekre az eredmény meghatározásához szükség van. Mivel az első aktuális paraméter `False`, a második ág kerül kiértékelésre a definícióban, így a második aktuális paraméterre nincs szükség: a program végrehajtása során `fact 20` nem kerül kiszámításra. A harmadik aktuális paramétert viszont ki kell értékelni, ki kell számolni `fact 10` értékét, hogy a kétszeresét visszaadhassuk.

A név szerinti paraméterátadás egészen hasonló az igény szerintihez, de az `f` második ágában a `b+b` kifejezés értékének meghatározásához a harmadik aktuális paramétert, a `fact 10` értéket kétszer is kiszámítja. Ezt figyelhetjük meg az alábbi Scala kódban. (Scalában érték szerinti és megosztás szerinti paraméterátadás van, hacsak nem helyezzük el a formális paraméter deklarációjában a `=>` jelet.)

```
def f( condition: Boolean, a: => Int, b: => Int ): Int =
   if (condition) a else b + b

def fact(n: Int): Int = n match {
  case 0 => 1
  case _ => n * fact(n-1)
}

f(false, {print("a"); fact(20)}, {print("b"); fact(10)})
```

Ez a Scala szkript kétszer is kiírja a `b` betűt a végeredmény kiírása előtt, viszont az `a` betűt egyszer sem. Ebből látszik, hogy a második aktuális paramétert egyszer sem, míg a harmadik aktuális paramétert kétszer is kiértékeli.

Szövegszerű helyettesítés

```
#define DOUBLE(n) 2*n
#define MAX(a,b) a>b?a:b

int main()
{
    printf("%d %d\n", MAX(10,100), DOUBLE(10));
    {
       int n = 5;
       printf("%d\n", DOUBLE(n+1));
       printf("%d\n", MAX(5,++n));
```

```
    }
}
```

A C preprocesszor a fordító előtt fut le, és a forráskódot átalakítja. (Kimenete: fordítási egység.) A preprocesszálás legismertebb lépései az `#include` és a `#define` direktívák végrehajtása, melyek hatására a forráskódba bemásolódnak fájlok (jellemzően *header*-állományok), illetve makrók kifejtésre kerülnek. A `#define` direktívákkal tehát makrók definiálhatók. Korábban említést tettünk paraméter nélküli makrókról: gyakran arra használjuk ezeket, hogy literálokhoz nevet vezessünk be, és ezzel a kód olvashatóságát, karbantarthatóságát növeljük.

Definiálhatunk azonban paraméterrel rendelkező makrókat is, mint azt a fenti példán láthatjuk. Ezek olyan szerepet tölthetnek be, mint a függvények: valamilyen számítás nevesített absztrakcióját adhatjuk meg általuk. A fenti `DOUBLE` makró kiszámítja paramétere kétszeresét, a `MAX` makró pedig két érték közül a nagyobbikat (vagy egy nem kisebbiket) választja ki.

Első ránézésre sok a hasonlóság a függvények és a makrók között. Ez azonban nem szabad, hogy megtévesszen bennünket. Könnyen okozhatnak fura hibákat a programunkban a makrók – oda kell figyelni, hogy jól használjuk őket. Nézzük, hogy mi is történik a fenti példában. Az első két makróhívás nem okoz problémát, itt az történik, amire mindenki gondol, aki rápillant a kódra. A blokk utasításban szereplő makróhívásokról ez már nem mondható el.

Szövegszerű helyettesítés – becsapós

```
#define DOUBLE(n) 2*n
#define MAX(a,b) a>b?a:b

int main()
{
    printf("%d %d\n", MAX(10,100), DOUBLE(10));
    {
       int n = 5;
       printf("%d\n", DOUBLE(n+1)); /* printf("%d\n", 2*n+1); */
       printf("%d\n", MAX(5,++n));
    }
}
```

Makrók esetében nem olyan paraméterátadás történik, mint a függvények esetén. A makrók definíciója szövegszerűen kifejtésre kerül az alkalmazás helyén, és a makrók aktuális paramétere(i) szövegesen behelyettesítésre kerül(nek) ennek során. Például a blokk utasításban szereplő `DOUBLE(n+1)` kifejtődik a `2*n+1` kifejezésre, ami persze nem ekvivalens azzal, amit elsőre gondoltunk volna: `2*(n+1)`. Az ilyen jellegű, a precedenciák miatt bekövetkező bakikkal szemben úgy védekezhetünk, hogy a makrók definíciójában minden „formális paramétert'' bezárójelezünk. Sőt, magát a makrótörzset is érdemes bezárójelezni, mert precedenciával kapcsolatos problémát a törzs kifejtésénél is felléphetnek. Például az `INC(x)` makrót nem egyszerűen `(x)+1` formában érdemes megadni, mert a `3*INC(1)` a szándékaink ellenére a `3*(1)+1` kifejezésre fejtődik ki, nem a `3*(1+1)`-re.

Szövegszerű helyettesítés – zárójelezés

```
#define DOUBLE(n) (2*(n))
#define MAX(a,b) ((a)>(b)?(a):(b))

int main()
{
    printf("%d %d\n", MAX(10,100), DOUBLE(10));
    {
       int n = 5;
       printf("%d\n", DOUBLE(n+1)); /* (2*((n)+1)) */
       printf("%d\n", MAX(5,++n));
    }
}
```

Tehát a makró definíciójában elég sok zárójelet elhelyezve a `DOUBLE` makrót sikerült úgy elkészítenünk, hogy az különféle hívási környezetben is kettővel való szorzás értelemben működjön.

Vannak azonban olyan makródefiníciók is, amelyek esetén még ez az óvintézkedés sem elegendő. Nézzük az alaposan bezárójelezett `MAX(a,b)` makrót.

Szövegszerű helyettesítés – még így is veszélyes

```
#define DOUBLE(n) (2*(n))
#define MAX(a,b) ((a)>(b)?(a):(b))

int main()
{
    printf("%d %d\n", MAX(10,100), DOUBLE(10));
    {
       int n = 5;
       printf("%d\n", DOUBLE(n+1)); /* (2*((n)+1)) */
       printf("%d\n", MAX(5,++n));  /* ((5)>(++n)?(5):(++n)) */
    }
}
```

A kifejtés hatását jól láthatjuk a kommentben. A precedenciák nem okoznak gondot, de az aktuális paraméterként használt kifejezések mellékhatásai elsőre meglepő eredményt hozhatnak. A blokk utasításban olyan kifejezés jön létre a makró alkalmazásának hatására, amely kiértékelése során a `++n` részkifejezés kétszer is kiértékelődik, így nem 6, hanem 7 végeredményt kapunk. A `MAX(a,b)` makrót csak mellékhatásmentes aktuális paraméterekkel célszerű meghívni.

Annak ellenére, hogy a makrók veszélyesek, mert könnyen megtévesztik a programozókat, sokszor találkozunk velük: a programozók előszeretettel írnak (jellemzően egyszerű) számításokat függvénydefiníció helyett makróval. Ennek az az oka, hogy a makrók kifejtése során olyan kód keletkezik, ami közvetlenül tartalmazza a számítást, a függvényhívás nélkül: így a függvényhívás költsége megspórolható. Persze egy jobb fordítóprogram optimalizációval képes arra, hogy függvények törzsét *inline-osítsa*, ezért az az érv, hogy a makró használatából származó hatékonyságnövekedés ellensúlyozza a makrók veszélyességét, egyszerűen nem állja meg a helyét.

Változó számú paraméter

```
int printf( const char *format, ... );
```

```
def sum( *args ):
   s = 0
   for n in args:
      s += n
   return s

sum()
sum(3)
sum(3,2)
sum(3,2,7,6,1,8)
```

A C-beli `...` jelöléshez hasonló hatást érhetünk el a Pythonban azzal, hogy `*`-gal jelölt paramétert használunk. Ez a paraméter valójában akárhány paramétert jelöl, melyeken például egy `for` ciklussal iterálhatunk végig. Fontos, hogy a közönséges paraméterek meg kell előzzék az ilyen *variadikusságot* biztosító paramétert (szokás ezt a paramétert *vararg*nak is nevezni).

Névvel jelölt paramétermegfeleltetés

```
def copy( src, dst ):
   for item in src:
      dst += [item]

a = [1,2,3]
b = [4,5]
copy( dst=b, src=a )
```

A legtöbbször *pozícionálisan* feleltetjük meg az aktuális paramétereket a formálisoknak, azaz az első aktuális felel meg az első formálisnak stb. Van, amikor kényelmesebb a *névvel jelölt* megfeleltetés. A `copy` függvénynél az ember hajlamos elfelejteni, hogy az első paraméter tartalmát kell bemásolni a második paraméterként megadott listába. Hogy elkerüljük a félreértést, ki is írhatjuk, hogy melyik nevű formális paraméternek milyen aktuális paramétert szeretnénk megfeleltetni. Ebben az esetben az aktuális paramétereket tetszőleges sorrendben megadhatjuk.

A Python még egy érdekes lehetőséget rejt: a kulcsszavas paraméterek (keyword-only arguments) használatát, melyre itt most nem térünk ki.

Paraméter alapértelmezett értéke

```
def unwords( words, separator=' ' ):
   length = len(words)
   if length == 0:
      return ''
   else:
      result = words[0]
      for i in range(1,length):
         result += separator
         result += words[i]
      return result

unwords(["alma","a","fa","alatt"])
unwords(["alma","a","fa","alatt"], separator='\n')
unwords(["alma","a","fa","alatt"], '\n')
unwords(separator='\n', words=["alma","a","fa","alatt"])
```

Azt is megtehetjük, hogy a formális paraméterlistában néhány paraméterhez alapértelmezett értéket (default value) rendelünk, így ezeket a paramétereket opcionálissá tesszük. Amikor a függvényt meghívjuk, az alapértelmezett értéket fel tudják venni azok a formális paraméterek, amelyeknek van olyan, és amelyeknek nem feleltettünk meg aktuális paramétert.

Ha több alapértelmezett értékkel rendelkező paraméter is van, akkor igen jó szolgálatot tehet nekünk a névvel jelölt paramétermegfeleltetés.

Vájtfülűeknek: Pythonban az alapértelmezett érték kiszámítása nem akkor történik, amikor a függvényt meghívják a megfelelő aktuális paraméter megadása nélkül, hanem már akkor, amikor a függvénydefiníció (a def-szerkezet) kiértékelésre kerül. Egy globális függvény esetében ez egyetlen egyszer történik meg a program végrehajtása során, lokális függvény esetén persze minden esetben, amikor végrehajtjuk a tartalmazó blokkot, és így kiértékeljük a def-szerkezetet. Ennek lehetnek furcsa következményei is. Például elég megtévesztő lehet az, ha egy alapértelmezett paraméterérték egy módosítható (mutable) adat, például egy lista. A lista ugyanis akkor jön létre, amikor a def-szerkezet kiértékelődik, és az egymást követő olyan hívásai a szóban forgó függvénynek, amelyek az alapértelmezett értéket használják, az időközben esetleg módosult adatot látják majd. Még furcsább, ha a függvény maga módosítja ezt az alapértelmezett értéket…

Változó számú paraméter után névvel jelölt paraméter(ek)

```
def unwords( *words, separator=' ' ):
   length = len(words)
   if length == 0:
```

```
        return ''
    else:
        result = words[0]
        for i in range(1,length):
            result += separator
            result += words[i]
        return result

unwords("alma","a","fa","alatt")
unwords("alma","a","fa","alatt", separator='\n')
```

A változó hosszú paraméterlistát kombinálhatjuk névvel megfeleltetett paraméterekkel is, mint a fenti példa mutatja. Ebben az esetben a pozícionálisan megfeleltetett *vararg*-elemeket követhetik a névvel megfeleltetett paraméterek – amelyek persze rendelkezhetnek alapértelmezett értékkel is, így ilyen esetben akár el is hagyhatók a hívásból.