Programozás
(GKxB_INTM114)

Dr. Hatwagner F. Miklós

Széchenyi István Egyetem, Győr

https://github.com/wajzy/GKxB_INTM114.git

2024. március 11.

Mi az a függvény (function)?

Programkód egy konkrét, azonosítható, paraméterezhető, újrahasznosítható blokkja

Miért használunk függvényeket?

- Hosszú forrásszöveg áttekinthető, kisebb részekre tördelése (modularitás)
- Többszöri felhasználás lehetősége
  - egy programon belül, kódismétlés nélkül
  - több programban, gyakran használt részeket nem kell újra megírni (ld. `sqrt`, `pow`)

Függvény definíció (definition)

- Teljes formai információ a függvényről
  - visszatérési érték típusa (return type)
  - azonosító (name)
  - *formális* paraméterek (parameters)
  - függvénytest (body, ld. `main`)
- Pontosan egy létezhet belőle
- Forrásfájlokban vagy előfordított könyvtárakban tárolják

abszolut3.cpp Abszolútérték számítás függvénnyel

```
4 double abszolut(double szam) {
5   return szam<0. ? -szam : szam;
6 }
```

Függvényhívás (call, invoke)

- A függvénynek a híváskor ismertnek kell lennie
- Vezérlés + *aktuális* paraméterek átadása
- *Érték szerinti* paraméterátadás (pass by value)
- Vezérlés visszaadása + visszatérési érték szolgáltatása: `return`

abszolut3.cpp

```
 8 int main() {
 9   double v;
10   cout << "Szam: "; cin >> v;
11   cout << "Abszolut erteke: " << abszolut(v)
12        << "\nabszolut(-3) == " << abszolut(-3)
13        << "\nabszolut(v*3) == " << abszolut(v * 3)
14        << "\nabszolut(abszolut(-3)) == "
15        << abszolut(abszolut(-3)) << endl;
16   return 0;
17 }
```

Visszatérési érték

- Vt. típusa nem lehet tömb
- `return` utáni kifejezés: *hozzárendelési* konverzió szükséges lehet
- `void` típus: valaminek a hiányát jelzi („eljárás")

Formális paraméterlista

- Nincsenek paraméterek: `int main() {...}`
- Egy paraméter: `double abszolut(double szam) {...}`
- Két paraméter: `double hatvany(double alap, double kitevo) {...}`
- Aktuális paraméterek (arguments) → hozzárendelési konverzió → formális paraméterek
- Tömb átadása speciális eset

A függvény teste tartalmazhat mindent, ami a `main`-ben is megengedett volt, azaz

- Változók deklarációit
- A blokkon kívül deklarált tételekre történő hivatkozásokat
- Tevékenységet meghatározó utasításokat

Visszatérés a függvényből

- a függvény végén
- `return` utasítással (a fv. tartalmazhat több `return`-t is)

keres.cpp Karakter első előfordulásának keresése `string`-ben

```
4  int keres(string miben, char mit) {
5    for(size_t i=0; i<miben.length(); i++) {
6      if(miben[i] == mit) return i;
7    }
8    return -1;
9  }
```

Függvények definíciói nem ágyazhatóak egymásba!

beagyazas.cpp

```
1  int main() {
2    double abszolut(double szam) {
3      return szam<0. ? -szam : szam;
4    }
5    cout << abszolut(-1) << endl;
6    return 0;
7  }
```

Fordítási hiba

```
beagyazas.cpp: In function 'int main()':
beagyazas.cpp:2:32: error: a function-definition is not allowed here before '{' token
double abszolut(double szam) {
```

Implicit típuskonverzióra szükség lehet amikor egy változóhoz új értéket rendelnek, pl. fv. visszatérési értékének kialakításakor.

keres.cpp size_t → signed int

```
4 int keres(string miben, char mit) {
5   for(size_t i=0; i<miben.length(); i++) {
6     if(miben[i] == mit) return i;
7   }
8   return -1;
9 }
```

Hasonlóan, pl. függvény aktuális paraméterének konverziójánál.

abszolut3.cpp `int` → `double`

```
4  double abszolut(double szam) {
5    return szam<0. ? -szam : szam;
6  }

12         << "\nabszolut(-3) == " << abszolut(-3)
```

Az implicit típuskonverzió részletei
Néhány példa:

Miről?	Mire?	Kimenetel
signed+	unsigned	✓
signed−	unsigned	előjel funkcióvesztése
long int	int	értékvesztés veszélye
int	double	értékvesztés veszélye
float	double	✓
double	float	pontosságvesztés veszélye
double	int	törtrész levágás

Minta alkalmazás megvalósítandó szolgáltatásai (függvények):

Kombináció Adott n különböző elem. Ha n elem közül $k(0 < k \leq n)$ elemet úgy választunk ki, hogy mindegyik csak egyszer kerül sorra, és a kiválasztás sorrendje nem számít, akkor az n elem egy k-ad osztályú ismétlés nélküli kombinációját kapjuk. Jele: C_n^k

$$C_n^k = \frac{n!}{(n-k)!k!} = \binom{n}{k}$$

Példa: hányféleképpen tudunk *három* gyümölcs (mondjuk alma, körte, barack) közül *kettőt* kiválasztani?

1. alma, körte
2. alma, barack
3. körte, barack

Faktoriális Egy n nemnegatív egész szám faktoriálisa az n-nél kisebb vagy egyenlő pozitív egész számok szorzata. Jele: $n!$

$n! = \prod_{k=1}^{n} k$ minden $n \geq 0$ számra.

Megállapodás szerint $0! = 1$

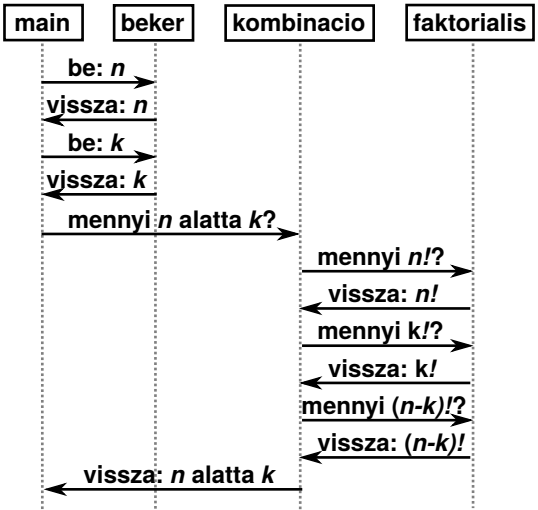
n elemet $n!$ sorrendbe lehet állítani (permutációk)

Példa: hányféleképpen tudunk *három* gyümölcsöt (mondjuk alma, körte, barack) sorba állítani?

1. alma, körte, barack
2. alma, barack, körte
3. körte, alma, barack
4. körte, barack, alma
5. barack, alma, körte
6. barack, körte, alma

Beolvasás Beolvasandó n és k értéke

Főprogram Adatok beolvasása, $\binom{n}{k}$ megjelenítése


main
beker
kombinacio
faktorialis
be: n
vissza: n
be: k
vissza: k
mennyi n alatta k?
mennyi n!?
vissza: n!
mennyi k!?
vissza: k!
mennyi (n-k)!?
vissza: (n-k)!
vissza: n alatta k

nk1.cpp

```
1  #include <iostream>
2  #include <climits>
3  using namespace std;
4
5  int beker(int max) {
6    int szam;
7    bool hibas;
8    do {
9      cout << "Szam: ";
10     cin >> szam;
11     hibas = szam<1 or szam>max;
12     if(hibas) cout << "Hibas adat!\n";
13   } while(hibas);
14   return szam;
15 }
```

nk1.cpp

```
17 unsigned long faktorialis(int n) {
18   if(n < 2) return 1;
19   unsigned long f = 1ul;
20   for(int i=1; i<=n; i++) {
21     f *= i;
22   }
23   return f;
24 }
25
26 unsigned long kombinacio(unsigned long n, unsigned long k) {
27   return faktorialis(n) / (faktorialis(k)*faktorialis(n-k));
28 }
29
30 int main() {
31   int n = beker(INT_MAX);
32   int k = beker(n);
33   cout << kombinacio(n, k);
34   return 0;
35 }
```

A függvényt a neve és paramétereinek száma, típusa együttesen azonosítja. A felültöltött (overload) változatok közül híváskor az aktuális paraméterek alapján választ.

nk2.cpp – beker()

```
4  int beker() {
5    int szam;
6    bool hibas;
7    do {
8      cout << "Szam: ";
9      cin >> szam;
10     hibas = szam<1;
11     if(hibas)
12       cout << "Hibas adat!\n";
13   } while(hibas);
14   return szam;
15 }
```

nk2.cpp – beker(int)

```
int beker(int max) {                                  17
  int szam;                                           18
  bool hibas;                                         19
  do {                                                20
    cout << "Szam: ";                                 21
    cin >> szam;                                      22
    hibas = szam<1 or szam>max;                       23
    if(hibas)                                         24
      cout << "Hibas adat!\n";                        25
  } while(hibas);                                     26
  return szam;                                        27
}                                                     28
```

Alapértelmezett függvényparaméterek:

- Egy paraméternek alapértelmezett értéket adunk → tőle jobbra lévőknek is adni kell!
- Híváskor nem adunk át értéket egy alapértelmezett paraméternek → ettől jobbra lévők se kaphatnak!

nk3.cpp

```
5  int beker(int max=INT_MAX) {
6    int szam;
7    bool hibas;
8    do {
9      cout << "Szam: ";
10     cin >> szam;
11     hibas = szam<1 or szam>max;
12     if(hibas) cout << "Hibas adat!\n";
13   } while(hibas);
14   return szam;
15 }
```

Élettartam (lifetime, duration): az a *periódus a futásidő alatt*, amíg a változó/függvény létezik, memóriát foglal. Részletek. Típusai:

- Statikus
  - Futás kezdetétől végéig foglal memóriát
  - Összes függvény, és *globális* (függvényeken kívül deklarált) változó ilyen
  - Globális változók implicit inicializálása: minden bit zérus értékű
  - Lehetőleg kerülni kell a globális változók használatát
    - \+ Paraméter-átadás költsége megtakarítható
    - – Nehézkes újrahasznosíthatóság, rugalmatlan, környezetfüggő kód, névütközések veszélye, ...

- Lokális
  - blokkba belépéstől, vagy a definíció sorát elérve, a blokk elhagyásáig rendelnek memóriát hozzájuk
  - függvényparaméterek, vezérlési szerkezetek blokkjai is ilyenek
  - csak explicit inicializáció történhet

nk1.cpp

```
unsigned long faktorialis(int n) {                17
  if(n < 2) return 1;                             18
  unsigned long f = 1ul;                          19
  for(int i=1; i<=n; i++) {                       20
    f *= i;                                       21
  }                                               22
  return f;                                       23
}                                                 24
```

Élettartamok:

faktorialis program teljes futásideje alatt létezik

n 3 fv. hívás miatt 3x létrejön a fv. hívásakor/megszűnik visszatéréskor (17-23. sor)

f létrejöhet pl. a függvény hívását követően és megszűnik amikor a végrehajtás a 23. sorhoz ér

i a 20. sor elérése pillanatában jön létre, és a ciklusból kilépéskor felszabadul

Hatáskör (érvényességi tartomány, hatókör, scope): meghatározza, hogy valamit a program *mely részén* lehet elérni. Részletek. Deklarációtól és annak helyétől függően:

- Blokk (lokális, belső)
  - Deklarációtól a tartalmazó blokk végéig, beleértve a beágyazott blokkokat is
  - A beágyazott blokkban deklarált azonosító ideiglenesen elrejtheti a befoglaló blokkban lévő azonosítót → rossz programozói gyakorlat, kerülendő!
  - Pl. függvény formális paraméterei, lokális változói
- Fájl (globális, külső)
  - minden függvény testén kívül deklarált azonosítók; deklarációs ponttól a fájl végéig
  - Pl. függvények, globális változók

Rekurzív függvényhívás

- Minden fv. hívhatja magát közvetlenül vagy közvetve (ön- és kölcsönös rekurzió)
- Minden hívásnál új területet foglalnak a formális paramétereknek, lokális változóknak
- A globális változók mindig ugyanazon a területen maradnak!
- El kell kerülni a végtelen mélységű rekurziót!

nk4.cpp

```
17 unsigned long faktorialis(int n) {
18   if(n < 2) return 1;
19   return n * faktorialis(n-1);
20 }
```

hatvany1.cpp Hatványozás szorzásokra visszavezetve

```
4  long hatvany(int alap, unsigned kitevo) {
5    long eredmeny = 1;
6    unsigned i;
7    for(i=0; i<kitevo; i++) {
8      eredmeny *= alap; }
9    return eredmeny; }
```

hatvany2.cpp Rekurzív hatványozás, pl. $-3^5 = -3^{2^2} \times -3^1 = -243$

```
4  long hatvany(int alap, unsigned kitevo) {
5    long eredmeny;
6    if(kitevo == 0) return 1;
7    if(kitevo == 1) return alap;
8    eredmeny = hatvany(alap, kitevo/2);
9    eredmeny *= eredmeny; // nem hivjuk 2x!
10   if(kitevo%2 == 1) eredmeny *= alap;
11   return eredmeny; }
```

Fibonacci-sorozat: másodrendben rekurzív sorozat. Képzeletbeli nyúlcsalád növekedése: hány pár nyúl lesz n hónap múlva, ha

- az első hónapban csak egyetlen újszülött nyúl-pár van,
- az újszülött nyúl-párok két hónap alatt válnak termékennyé,
- minden termékeny nyúl-pár minden hónapban egy újabb párt szül,
- és a nyulak örökké élnek.

$$F_n = \begin{cases} 0, & \text{ha } n = 0 \\ 1, & \text{ha } n = 1 \\ F_{n-1} + F_{n-2} & \text{ha } n > 1 \end{cases}$$

fibonacci1.cpp Iteratív változat

```
4  unsigned long fibonacci(unsigned ho) {
5    unsigned long i=0, j=1, k;
6    if(ho < 2) return ho;
7    for(unsigned n=1; n<ho; n++) {
8      k = i+j;
9      i = j;
10     j = k;
11   }
12   return k;
13 }
```

fibonacci2.cpp Rekurzív változat

```
4  unsigned long fibonacci(unsigned ho) {
5    if(ho < 2) return ho;
6    return fibonacci(ho-1)+fibonacci(ho-2);
7  }
```