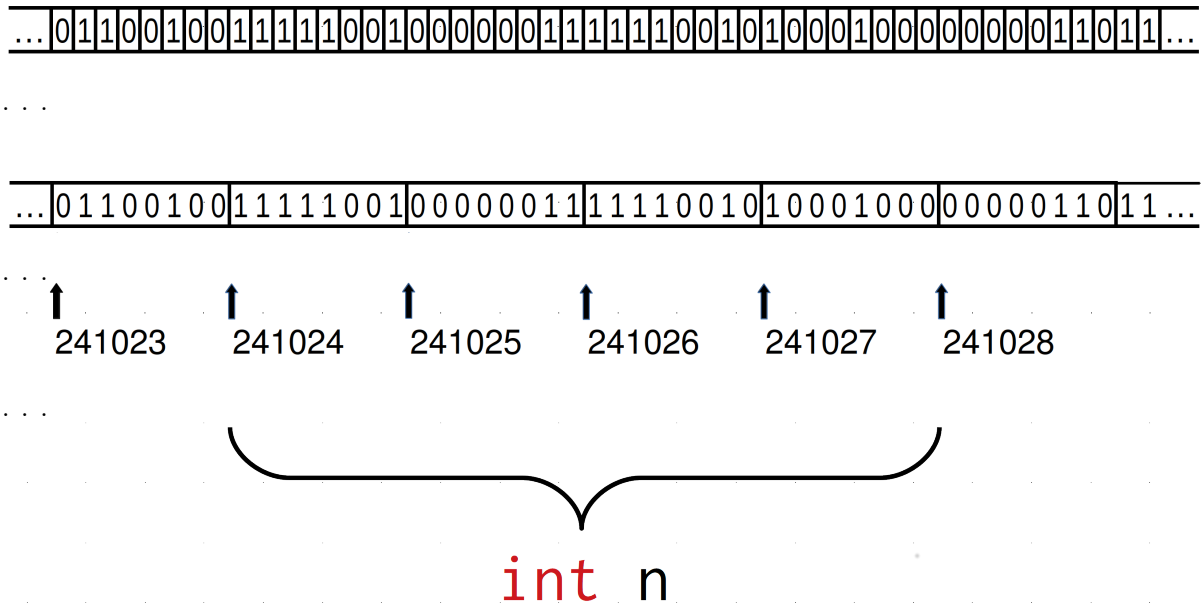
Imperatív programozás
Áttekintés

Kozsik Tamás és mások

Imperatív programozás

- Utasítások, vezérlési szerkezetek
- Memória írása, olvasása
- C programozási nyelv (link!)



1 Programok felépítése

Programok felépítése

- Kulcsszavak, literálok, operátorok, egyéb jelek, azonosítók
- Kifejezések
- Utasítások
- Alprogramok (függvények/eljárások, rutinok, metódusok)
- Modulok (könyvtárak, osztályok, csomagok)

A legtöbb programozási nyelvben a kód felépítéséhez használt főbb szerkezeti elemek a következők.

- Kifejezés – **expression**: például `v+1`
- Utasítás: értékadás, elágazás, ciklus stb.
- Alprogram: egy jól meghatározott számítási feladat (újrafelhasználható) megoldása. Különböző *paraméterekkel* elvégezhető ugyanaz a számítás a program különböző pontjain.
- Modul: egy összetettebb probléma megoldásához szükséges adatszerkezetek és algoritmusok gyűjteménye (egységbe foglalása, egységbe zárása).

Példa

Python

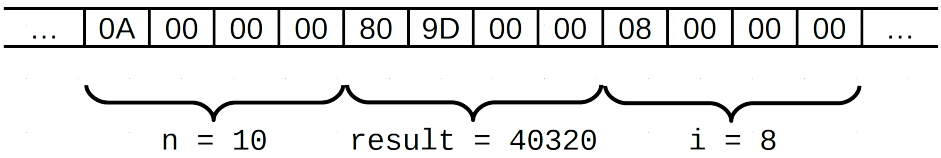
```
def factorial(n):
    result = 1
    for i in range(2,n+1):
        result *= i
    return result
```

...

C

```
int factorial( int n )
{
    int result = 1;
    int i;
    for(i=2; i<=n; ++i)
    {
        result *= i;
    }
    return result;
}
```

...



Kifejezések

`n`

`"Hello world!"`

`100`

`n+1`

`++i`

`range(2,n+1)`

`employees[factorial(3)].salary * 100`

Melyik kifejezés a kakukktojás?

Utasítások

```
result = 1;
```

```
result *= i;
```

```
return result;
```

```
for( i=2; i<=n; ++i ){ result *= i; }
```

```c
while(1) printf("Gyurrrika szép!\n");
```

Egyszerű utasítások

- értékadás
- üres utasítás
- alprogramhívás
- visszatérés függvényből

Vezérlési szerkezetek

- elágazások
- ciklusok stb.

Python

```python
def gcd(n,m):
    while n != m:
        if n > m:
            n -= m
        else:
            m -= n
    return n
```

C

```c
int gcd( int n, int m )
{
    while( n != m )
        if( n > m )
             n -= m;
        else
             m -= n;
    return n;
}
```

Nagyon hasonlít a két kód egymásra. Mik a különbségek? A C kódban szerepelnek típusok, a Python kódban nem (erről majd később). A C kódban zárójelekbe írjuk a feltételeket, a Python kódban kettőspontokat írunk utánuk. A C kódban a *függvény törzsét* kapcsos zárójelek közé írjuk, a Python kódban csak egy kettősponttal választjuk el a *függvény fejlécétől* (*specifikációjától*) – viszont Pythonban a `def` kulcsszóval kell kezdjük a *függvény definícióját*.

Ami nem látszik, de ugyanilyen fontos: Pythonban az indentálással fejezzük ki az egymásba épülést, a belső struktúrát. Mindent pontosan kell indentálni (lehetőleg szóközökkel, és nem tabulátorokkal) ahhoz, hogy értelmes programot kapjunk. Ha rosszul indentálunk, hibát kapunk. A C-ben is sokat segít a kód olvashatóságán az indentálás, és az a konvenció, hogy mindent ugyanolyan szépen, tökéletesen indentálunk, mint Pythonban, de ez nem feltétele annak, hogy helyes, értelmes legyen a kód. Rosszul indentált C program is lehet tökéletesen helyes: az egymás mellett álló *fehér szóközök* (*whitespace*) valójában csak egynek számítanak – sok helyre beírhatjuk őket, sok helyről törölhetjük őket. (Találd meg a lenti C kódban azt, hogy hol nem lehet kitörölni a szóközt.)

Például az alábbi Python kód hibás.

```python
def gcd(n,m):
    while n != m:
        if n > m:
            n -= m
      else:             # az else rossz helyre került
            m -= n
    return n
```

Az alábbi C kód viszont tökéletesen helyes.

```
int gcd( int n,
         int m ){
    while(
 n !=
               m )if(n>m)n-=m;else m

-= n;

    return n;}
```

Természetesen senki nem ír ilyen kódot, csak maximum heccből. Ugyanolyan kínosan ügyelünk a helyes indentálásra C-ben is, mint a Pythonban – mintha a program helyessége múlna ezen.

Kapcsos zárójelek vezérlési szerkezetekben

Elhagyott kapcsos zárójelek

```
int gcd( int n, int m )
{
    while( n != m )
        if( n > m )
            n -= m;
        else
            m -= n;
    return n;
}
```

Bolondbiztos megoldás

```
int gcd( int n, int m )
{
    while( n != m ){
        if( n > m ){
            n -= m;
        } else {
            m -= n;
        }
    }
    return n;
}
```

Sokan, sok helyen megkövetelik a kapcsos zárójelek redundáns, de bolondbiztos használatát. Ha egy vezérlés szerkezet törzsében (pl. ciklusmagban vagy elágazás egy ágában) csak egy utasítás szerepel, felesleges kiírni a kapcsos zárójeleket köré. Viszont segít bizonyos programozási hibák elkerülésében, ha ilyen esetben is kiírjuk azokat. Az adott környezet (munkahely, csapat) meghatározhatja ezt a biztonságra törekvő stílust (coding style) követelményként, betartandó konvencióként (coding convention).

Csellengő else (dangling else)

Ezt írtam

```
if( x > 0 )
   if( y != 0 )
      y = 0;
else
   x = 0;
```

Ezt jelenti

```
if( x > 0 )
   if( y != 0 )
      y = 0;
   else
      x = 0;
```

Ezt akartam

```
if( x > 0 ){
   if( y != 0 )
      y = 0;
} else
   x = 0;
```

Lásd még...

goto-fail (Apple) link!

Kiírás a szabványos kimenetre

Kiírunk egy egész számot és egy soremelést (*newline*)

Python

```
print( factorial(10) )
```

C

```
printf("%d\n",factorial(10));
```

Bonyolultabb kiírás

Python

```
print( "10! =", factorial(10), ", ln(10) =", log(10) )
```

C

```
printf("10! = %d, ln(10) = %f\n", factorial(10), log(10));
```

. . .

```
10! = 3628800, ln(10) = 2.302585
```

Ha kicsit összetettebb kimenetet szeretnénk előállítani, akkor lehet Pythonban a `print`, C-ben a `printf` művelettel ilyet csinálni. Pythonban ez kicsit egyszerűbb, mint a példa mutatja. A vájtfülűek észrevehetik, hogy a példában szereplő Python és C kódok nem *ekvivalensek*, nem pontosan ugyanazt írják ki. Természetesen kicsit nagyobb munkabefektetéssel lehetne tökéletesen egyforma kimenetet előállítani a két nyelvben, de itt most nem ez volt a célunk.

Típusok

- Kifejezik egy bitsorozat értelmezési módját
- Meghatározzák, milyen értéket vehet fel egy változó
- Megkötik, hogy műveleteket milyen értékekkel végezhetünk el

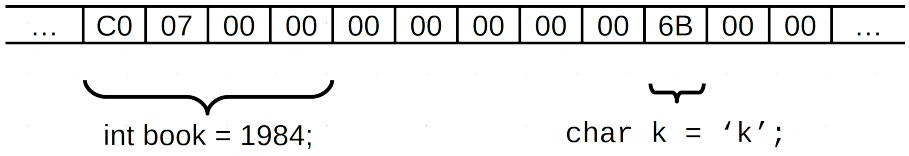
C-ben

- `int` – egész számok egy intervalluma, pl. $[(-1) * 2^{31} \,..\, 2^{31} - 1]$
- `float` – racionális számok egy részhalmaza
- `char` – karakterek
- `char[]` – szövegek, karakterek tömbje
- `int[]`– egész számok tömbje

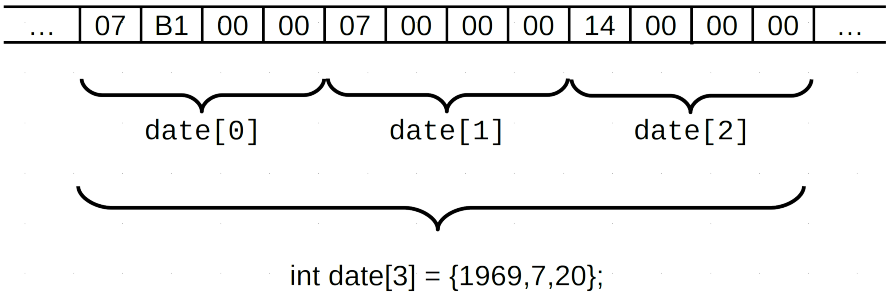
- `int*` – mutató (pointer) egy egész számra

stb.

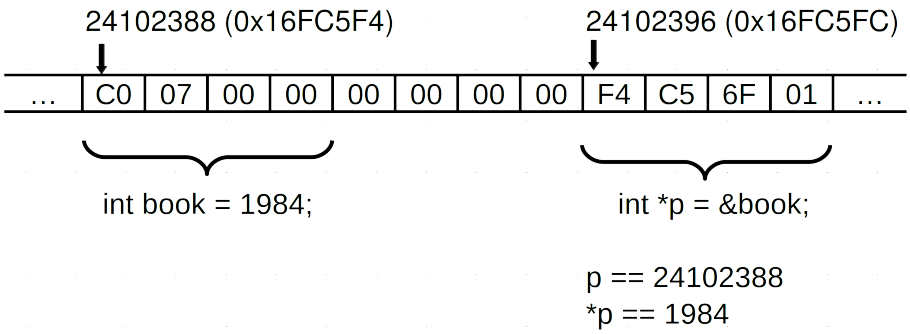
Különböző típusú értékek a memóriában



. . .



. . .



Típus szerepe

- Védelem a programozói hibákkal szemben
- Kifejezik a programozók gondolatát
- Segítik az absztrakciók kialakítását
- Segítik a hatékony kód generálását

Típusellenőrzés

- A változókat, függvényeket a típusuknak megfelelően használtuk-e
- A nem típushelyes programok értelmetlenek

Statikus és dinamikus típusrendszer

- A C fordító ellenőrzi *fordítási időben* a típushelyességet
- Pythonban *futási időben* az interpreter vizsgálja ezt

Erősen és gyengén típusos nyelv

- Gyengén típusos nyelvben automatikusan konvertálódnak értékek más típusúra, ha kell
  - Eleinte kényelmes
  - De könnyen írunk mást, mint amit szerettünk volna

- A C-ben és Pythonban viszonylag szigorúak a szabályok (erősen típusosak)

A programozási nyelvek egy részénél a fordítóprogram már a fordítási időben minden egyes részkifejezésről el tudja dönteni, hogy az milyen típusú. Ezeket a nyelveket statikus típusrendszerrel rendelkezőnek nevezzük. Ennek vannak előnyei, hiszen a nyelv alaposabb ellenőrzéseket tud végrehajtani és optimálisabb kódot is tud generálni. Ilyen nyelv a Fortran, Algol, C, Pascal, C++, Java, C#, Go.

Más nyelveknél, legtöbbször az interpretált nyelveknél, egy változó idővel más típusú értékekre is hivatkozhat. Ilyenkor a fordító futási időben kezeli a típusinformációkat. Ezt dinamikus típusrendszer-nek nevezzük. Ilyen nyelv pl. a Python.

Mindez nem jelenti, hogy a dinamikus típusrendszer nem ellenőrizheti a típusok alkalmazását, sőt helytelen alkalmazás hibát okozhat. Azokat a nyelveket, ahol ilyen hibák előfordulnak erősen típusos-nak nevezzük, szemben a gyengén típusos nyelvekkel.

A C erősen típusos statikus típusrendszerrel rendelkező nyelv, a Python erősen típusos dinamikus típusrendszerű.

Alprogramok (subprograms)

- Több lépésből álló számítás leírása
- Általános, paraméterezhető, újrafelhasználható
- A program strukturálása – komplexitás kezelése
  - egy képernyőoldalnál ne legyen hosszabb
- Különböző neveken illetik
  - rutin (routine vagy subroutine)
  - függvény (function): kiszámol egy értéket és “visszaadja’’
  - eljárás (procedure): megváltoztathatja a program állapotát
  - metódus: objektum-orientált programozási terminológia

Főprogram (main program)

Ahol a program végrehajtása elkezdődik

Python

Nincs jelölve, egy csomó utasítás egymás után

```
half = 21
print(2*half)
```

C

Egy megfelelő nevű alprogram: `main`

```
int main()
{
    int half = 21;
    printf("%d\n",2*half);
    return 0;                /* sikeres végrehajtás */
}
```

Megjegyzés

[…]

Python

[…]

```
half = 21
print(2*half)                # itt így írok megjegyzést
```

C

[…]

```
int main()
{
    int half = 21;
    printf("%d\n",2*half);
    return 0;                /* itt így írok megjegyzést */
}
```

Modul

Modularitás: egységbe zárás, függetlenség, szűk interfészek

- Újrafelhasználható programkönyvtárak
  - pl. a nyelv szabványos könyvtára (standard library)
- A program nagyobb egységei
- Absztrakciók megvalósítása

A modularitás azt jelenti, hogy a programot felbonthatjuk részekre, melyek viszonylag függetlenek egymástól, a kapcsolataik pedig szabályozottak: egy szűk interfészen keresztül történik. A modul egységbe zárja (encapsulation) a logikailag összetartozó, erősen összefüggő dolgokat, és a külvilág felé minimalizálja a függőségi kapcsolatokat. A modulok azért fontosak, mert a programkód komplexitását (bonyolultságát) tudjuk a segítségükkel kordában tartani. A rendszer megfelelő tagolása, logikus, strukturált felépítése, a szűk interfészek mind csökkentik a komplexitást.

A legszigorúbb értelemben azt a rendszerfelbontást szokták modulárisnak nevezni, amelyben a modulok egymásra épülnek, így például körkörös függés nem is alakulhat ki közöttük.

Egy másik értelmezése a szónak arra utal, hogy egy moduláris rendszerben egy modult könnyű kicserélni egy vele azonos szolgáltatásokkal rendelkező másik modulra.

Mi itt most egyelőre a lehető legáltalánosabb értelmezését vesszük a modul szónak: a program (alprogramoknál nagyobb) egységekre bontását, újrafelhasználható egységek, könyvtárak kialakítását, absztrakciók (például objektum-orientált nyelvekben osztályok) megvalósítását.

Modulokra bontás

Újrafelhasználható `factorial`

- `factorial.c` – a `factorial` függvényt
- `tenfactorial.c` – a főprogramot

tenfactorial.c

```
#include <stdio.h>

int factorial( int n );  /* deklaráljuk ezt a függvényt */

int main()
{
    printf("%d\n", factorial(10));
    return 0;
}
```

Ha a `factorial` függvényt több programban is szeretném használni, praktikus lehet szétbontani a programot több "modulra". Ezeket C-ben két külön forrásfájlba szervezem.

- A `factorial.c` tartalmazza a `factorial` függvényt.
- A `tenfactorial.c` tartalmazza a főprogramot.

A kettő közötti kapcsolat a `factorial` függvény deklarációjában testesül meg a `tenfactorial.c` állományban. Ezzel a deklarációval jelzem, hogy valahol máshol (egy másik modulban) meg van adva ennek a függvénynek a definíciója.

2 Programok fordítása és futtatása

Forráskód

- Programozási nyelven írt kód
- Számítógép: gépi kód
- Végrehajtás
  - interpretálás (Python)
  - fordítás, futtatás (C)
- Forrásfájl
  - `factorial.py`
  - `factorial.c`

Itt beszélhetünk a Javáról is, miszerint fordítás plusz interpretálás…

Forrásfájl Pythonban

factorial.py

```
def factorial(n):
    result = 1
    for i in range(2,n+1):
        result *= i
    return result

print(factorial(10))
```

Végrehajtás

```
python3 factorial.py
```

Parancsértelmező (interpreter)

- Például `python3`
- Forráskód feldolgozása utasításonként
  - Ha hibás az utasítás, hibajelzés
  - Ha rendben van, végrehajtás
- Az utasítás végrehajtása: beépített gépi kód alapján

Hátrányok

- Futási hiba, ha rossz a program (ritkán végrehajtott utasítás???)
- Lassabb programvégrehajtás

Előnyök

- Programírás és -végrehajtás integrációja
  - REPL = Read-Evaluate-Print-Loop
  - Prototípus készítése gyorsan
- Kezdők könnyebben elsajátítják

Forrásfájl C-ben

factorial.c

```
#include <stdio.h>

int factorial( int n ){
    int result = 1;
    int i;
    for(i=2; i<=n; ++i){
        result *= i;
```

```
    }
    return result;
}

int main(){
    printf("%d\n", factorial(10));
    return 0;
}
```

Fordítás és futtatás szétválasztása

- Sok programozási hiba kideríthető a program futtatása nélkül is
- Előre megvizsgáljuk a programot
- Ezt csak egyszer kell (a *fordítás* során)
- Futás közben kevesebb hiba jön elő
- Cél: hatékony és megbízható gépi kód!

"Fordítási idő" és "futási idő"

Fordítás és futtatás

Forrásfájl

`factorial.c`

Fordítás

```
gcc factorial.c
```

Lefordított program

`a.out`

Futtatás

```
./a.out
```

A gcc -o kapcsolója

Forrásfájl

`factorial.c`

Fordítás

```
gcc -o fact factorial.c
```

Lefordított program

`fact`

Futtatás

```
./fact
```

Fordítási hibák

- Nyelv szabályainak megsértése
- Fordítóprogram detektálja

factorial.c-ben

```
int factorial( int n )
{
    int result = 1;
    for(i=2; i<=n; ++i)
    {
        result *= i;
    }
    return result;
}
```

gcc factorial.c

```
factorial.c: In function ‘factorial’:
factorial.c:6:9: error: i undeclared (first use in this function)
     for(i=2; i<=n; ++i)
         ^
```

Ha a fordítási egység nem felel meg a nyelv szabályainak, és a fordítóprogram ezt kiszúrja, akkor fordítási hibát (compilation error) kapunk. Előnye: szigorú nyelvben nehéz rossz programot írni. Az itt látható fordítási egységben elfelejtettük definiálni az `i` változót, kaptunk is egy fordítási hibát. Nem mindig ilyen könnyű megfejteni a hibaüzenetet. Az a minimum, hogy tudunk angolul. Ezek a fordítási hibák és a hibaüzenetek elég általánosak, sok nyelvben hasonlítanak egymásra, de azért elég nyelvspecifikusak is tudnak lenni. Sokszor bele kell lapozni a nyelv dokumentációjába is, hogy megértsük, mi a baj.

Fordítási figyelmeztetések

- Nyelv szabályai betartva
- A fordítóprogram valamilyen furcsaságot detektál
- Valószínűleg hibát vétettünk
- **Cél: warning-mentes fordítás!**

factorial.c-ben

```
int factorial( int n )
{
    int result = 1, i;
    for(i=2; i<=n; ++i)
    {
        result *= i;
    }
}
```

gcc -W -Wall --pedantic factorial.c

```
factorial.c: In function ‘factorial’:
factorial.c:10:1: warning: control reaches end of non-void function
                                                        [-Wreturn-type]
 }
 ^
```

Előfeldolgozás

C preprocessor: (forráskódból) forráskódot állít elő

Makrók

```
#define WIDTH 80
...
char line[WIDTH];
```

Deklarációk megosztása

```
#include <stdio.h>
...
printf("Hello world!\n");
```

Feltételes fordítás

```
#ifdef FRENCH
printf("Salut!\n");
#else
printf("Hello!\n");
#endif
```