# 4. gyakorlat

## Pluszminusz

Volt. A két kérdés:

1. Fordítás három fő lépése.
2. Definíció vagy deklaráció (3 db, volt: , és még egy.)

## DRY elv – Don't Repeat Yourself (Újrafelhasználható képletek)

fprintf**(**stdout**,**"F = %d\tC=%f\n"**,**fahr**,**5.0**/**9**\*(**fahr**-**32**));**

Itt van egy képlet, amit újra felhasználhatóvá érdemes tenni:

double fahr2cels**(**int fahr**){**

**return** 5.0**/**9**\*(**fahr**-**32**);**

**}**

Ekkor:

fprintf**(**stdout**,**"F = %d\tC=%f\n"**,**fahr**,**fahr2cels**(**fahr**));**

De a függvényhívás költséges dolog! Meg kell jegyezni, hol tartunk a főprogramban, el kell ugrani a meghívott függvényhez, át kell adni a paramétereket, takarítani kell a függvény után, vissza kell adni az eredményt, vissza kell térni, törölni kell a megjegyzett "itt tartottunk" mutatót.

Ma már nem olyan fontos dolog ez a költség, de régen az volt. És kis chipeken ma is fontos.

## Függvény stílusú makrók

A függvényhívás költségét elkerüli, de előnyeit megtartja. (De debuggolni nem lehet)

Minden előfordulása behelyettesítődik. Paraméterezést jól kezeli.

#define F2C(f) 5.0/9\*(f-32)

Ekkor:

fprintf**(**stdout**,**"F = %d\tC=%f\n"**,**fahr**,**F2C**(**fahr**));**

Ha rossz a képlet, csak egy helyen kell módosítani. Függvényhívás nem történik.

Ezek a makrók veszélyesek.

### Buta helyettesítés

Nincs ellenőrzés.

Nincs típusozott paraméter, sőt, nem is paraméter. Nincs típusmegfeleltetés.

Mivel nem függvény,

#define NEG(x) -x

NEG**(**4**)** //=-4

NEG**(-**5**+**2**)** //=7 (!=3)

Ekkor lesz, helyett.

Megpróbálhatjuk kijavítani agyonzárójelezéssel:

#define NEG(x) -(x)

NEG**(-**5**+**2**)** //=3)

Ekkor már jól működik:

De nem tökéletes.

Például: #define MAX(x,y) x > y ? x : y

Ezt még agyonzárójelezzük: #define MAX(x,y) ((x) > (y)) ? (x) : (y)

MAX**(-**5**,**2**)** //Működik

MAX**(++**x**,**2**)** //Nem működik max(5,2)=5, de a makró 6-ot ad vissza:

**((++**x**)** **>** **(**2**))** **?** **(++**x**)** **:** **(**2**)** //Kétszer növeli x-et

## Pontosvessző fontossága a C++-ban

Ez egy utasítás. (Üres utasítás) Szekvencia pont.

Addig a pontig minden előtte levő dolog garantáltan kiértékelődik.

f**()** g**()**

Nem biztos, hogy a várt sorrendben értékelődnek ki. Ha függ által állított globális változótól, ez gondot okozhat.

Javítás: f**();** g**();**

## C++-ban a fahrenheit-es kód

#include <iostream>

**namespace** myConsts**{**

const int LOWER**=**0**;**

const int UPPER**=**200**;**

const int STEP**=**40**;**

**}**

inline double fahr2cels**(**int fahr**){**

**return** 5.0**/**9**\*(**fahr**-**32**);**

**}**

int main**(){**

**for(**int fahr**=**myConsts**::**LOWER**;**fahr**<=**myConsts**::**UPPER**;**fahr**+=**myConsts**::**STEP**){**

std**::**cout**<<**"F="**<<**fahr**<<**"\tC="**<<**fahr2cels**(**fahr**)<<**std**::**endl**;**

**}**

**return** 0**;**

**}**

Fordítása:

A konstansokat be lehet rakni **namespace** myConsts**{**...**}**-ba, hogy ne legyen névütközés.

ZH-n pedig ezt fogjuk debuggolni fél óráig:

**using** **namespace** std**;**

int max**=**200**;** (Mert már van egy max függvény std-ben.)

### Konstans

const

Operációs rendszer szinten nem módosítható változó. Ennél több jelentése is van.

A fordító garantálja, hogy a változó konstans tárterületre fog kerülni. Írásvédett memóriába.

Ha érték nélkül deklaráljuk, nem tudunk később értéket adni neki:  
const int LOWER**;**LOWER**=**0**;**

Sokkal jobb a programozóknak a fordításidei hiba, mint a futásidő-beli. Törekedjünk is arra, hogy fordítás közben legyen hiba, ha valami rossz. Nem kell konstansok elé kiírni, hogy const, de nagyon érdemes. Megvédi a változónkat. Szerződés a fordító és a programozó(k) között, hogy soha ne lehessen módosítani a változót.

### Inline kifejtés

inline

Jó fordító fel is ismeri az inline-képes függvényeket.

Ez a kulcsszó csak egy ajánlás, nem tudjuk rákényszerítani a fordítót, hogy inline helyettesítse b.

A sűrűn hívott függvényeket nem helyettesíti be, inkább benntartja memóriában. Mert ma már úgy a gyorsabb.

Egyszer hívott függvényt általában berakja inline.

# Pointerek

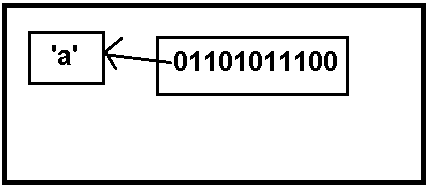
## Címképzés és

Van egy típus → az egy típusú változó címét tároló változó.

char c**=**'a'**;**

char**\*** cp**=&**c**;**

Címképző operátor: **&**



Minden bájt a memóriában be van címezve. 16-os számrendszer-beli szám.

Dereferálás (indirekció, visszafejtés) művelete: **\***cp

Poitnereknél mindig két dolgoról van szó:

* Tartalmazott érték (Pointer változó tartalma mindig egy memória cím. Ezt nem szoktuk kiiratni. Példa pointer kiírva: )
* Tartalmazott érték által meghatározott tartalom (**\***cp)

## Tömb

T típus → T típusú elemek tömbje

int v**[]={**1**,**2**,**3**,**4**}**

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |

=int v**[**4**]={**1**,**2**,**3**,**4**}**

Ha 4 a méret, 4 elemnél többet nem adhatok meg. De kevesebbet igen. Ekkor a típus (int) alap értéke (0) kerül bele a többi értékbe.

Tömb méret helyén konstans kifejezés kell szerepeljen.

Nem lehet felhasználótól beolvasott szám tömbméret. Van olyan fordító, ami lefordítja, de inkább fordítási hibát kapunk.

## Tömbök és pointerek kapcsolata

A tömb C++-ban implicit módon átalakítódik a tömb első elemére mutató pointerré.

int v**[]={**1**,**2**,**3**,**4**};**

int**\*** p**=**v**;**

Tömb elelei -tól -ig indexelődnek.

Tömböt for ciklussal be lehet járni.

Nincs ellenőrzés. 4 méretű tömb 100-adik eleme lekérhető, memóriaszemetet kapunk vissza.

### Tömb átadása paraméterként

f**(**int v**[])**

Ekkor elveszik a tömb mérete. Nem működik többé a **sizeof(**v**)/sizeof(**int**)**.

Mindig pointer fog átmenni.

Nincs más megoldás, át kell adni a függvénynek a tömb méretét.