

Programozás alapjai II.

(1. ea) C++

C++ kialakulása, nem OO újdonságok:

Szeberényi Imre, Somogyi Péter

BME IIT

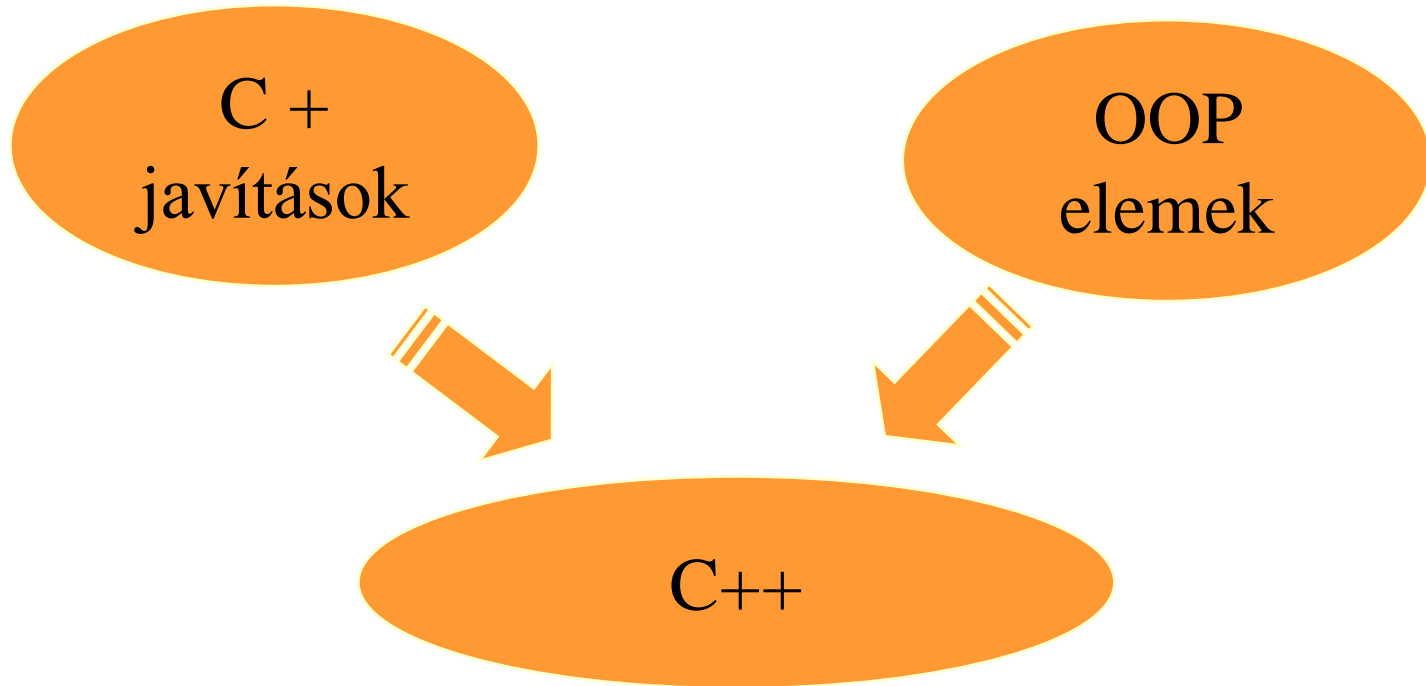
<szebi@iit.bme.hu>



MŰEGYETEM 1782

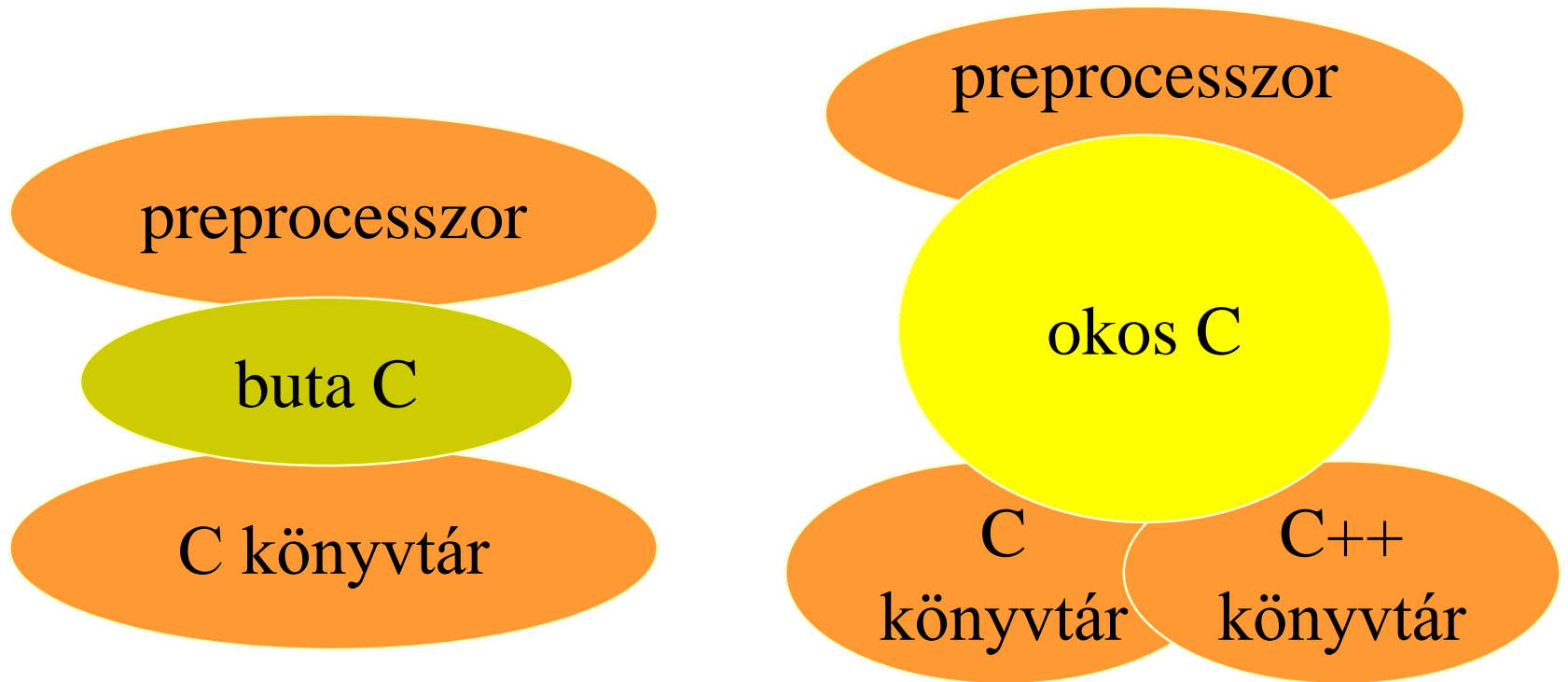
C++ kialakulása

Veszélyforrások csökkentése Objektum orientált szemlélet

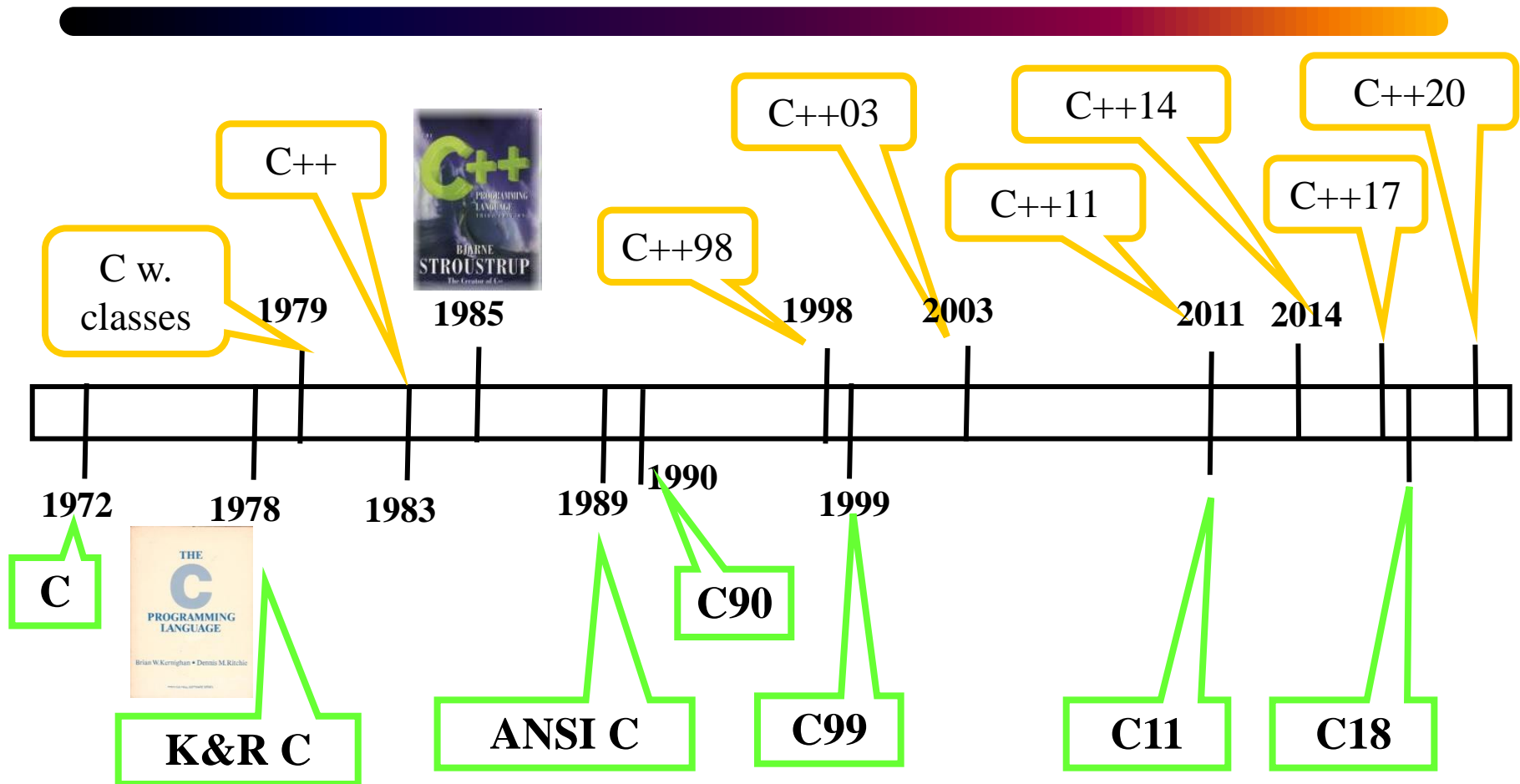


A fejlődés során jelentős kölcsönhatások voltak a C és a C++ között

C és C++ viszonya



C és C++ változatai



C++11, C++14, C++17, C++20?

- A tárgy a 2003-ban elfogadott C++ nyelvet használja az OO paradigmák bemutatásához eszközként.
- Ezt tanítja és ezt kéri számon, de lehet az újabb változatok elemeit használni háziban, zh-ban.
- A teljes C++11 azonban lényegesen bonyolultabb, amit másik tárgy keretében lehet megismerni.
- Bjarne Stroustrup:
„C++11 feels like a new language”
<http://www.stroustrup.com/C++11FAQ.html>

C99

- deklarációk és utasítások vegyesen

```
for (int i = 1; i < 12; i++) ....
```

- // comment, const, enum, inline
- változó hosszúságú tömb (függvényben)

```
void f(int b) {  
    int c[b]; // változó méretű tömb  
}
```

- új típusok (pl. long long, double _Complex)
- Pontosabb specifikáció pl: $-3/5 = 0$
 $-3\%5 = -3$ // C89-ben lehetne -1 és +2

Általános kódolási tanácsok (ism)

- Olvasható legyen a kód, ne trükkös!
- Mellékhatásoktól tartózkodni!
- Nem triviális szintaxist kerülni, akkor is, ha a C nyelv szerint az egyértelmű ($a+++b$)
- Nem feltétlenül kell haragudni a break-re és a continue-ra! Óra végén látni fogjuk, hogy C++-ban még a „goto”-t is gyakran használjuk (bár nem így hívjuk).

Mi történik, ha $x > 2.3$?

```
while (i < 12 && q != 0 && k > 87 && u < 3) {  
.....  
    if (x > 2.3) q = 0;  
.....  
}
```

```
while (i < 12 && k > 87 && u < 3) {  
.....  
    if (x > 2.3) break;  
.....  
}
```

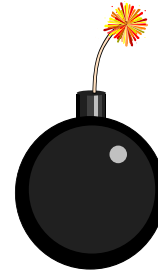

Általános kódolási tanácsok/2

- Makrókat kerüljük, ha lehet

```
#define MAX(a,b) a > b ? a : b
```

```
int a1 = 1;
```

```
int x = MAX(a1&7, 3); // x = ???
```



```
#define MAX(a,b) (a) > (b) ? a : b
```

Általános kódolási tanácsok/3

- Memória foglalás: ki foglal és ki szabadít?
- `char *valami(char *); // lefoglal? Mit csinál?`
- Ha foglal, kinek kell felszabadítani ?
- Oda kell írni kommentbe!
- Összetartozó adatok struktúrába
- Konstansok, enum
- Elfogadott kódolási stílus betartása. pl:
 - <http://google-styleguide.googlecode.com/svn/trunk/cppguide.html>
- A lényeg a következetességen van!

Deklaráció és definíció

- A deklarációs pont továbbra is legtöbbször definíció is:
 - `int a; float alma;`
 - `de: int fv(int x);` - **nem definíció**
- **A típus nem hagyható el !**
- Több deklaráció is lehet,
 - `extern int error;`
 - `extern int error;`
- **Definíció csak egy!**

Ott deklaráljuk, ahol használjuk

```
y = 12; ...
```

```
int z = 3; // és egyből inicializáljuk
```

```
for (int i = 0; i < 10; i++) {
```

```
    z += i;
```

```
    int k = i - 1;
```

```
    y *= k;
```

```
}
```

élettartam, hatókör
u.a, mint a C-ben

i és k itt már nem
léteznek !

C++ újdonságok, bővítések

- Struktúranév típusná válik
- Csak preprocesszorral megoldható dolgok nyelvi szintre emelése (const, enum, inline)
- Kötelező prototípus, névterek
- Referencia, cím szerinti paraméterátadás
- Túlterhelt/többarcú függvények (overloaded)
- Alapértelmezésű (default) argumentumok
- Dinamikus memória nyelvi szint. (new, delete)
- Változó definíció bárhol, ahol utasítás lehet

Típusok

- logikai
 - char
 - egész
 - **felsorolás**
 - valós
 - mutató
 - **referencia**
 - void
-
- The diagram shows a list of C++ types on the left, grouped into three categories on the right by curly braces. The 'integrális' category includes 'logikai', 'char', 'egész', and 'felsorolás'. The 'aritmetikai' category includes 'valós' and 'mutató'. The 'skalár' category includes 'referencia' and 'void'. Each category name is enclosed in a blue-bordered box.
- integrális
 - aritmetikai
 - skalár

- összetett adatszerkezetek (tömb struktúra) és **osztályok**

Logikai típus (új típus)

bool

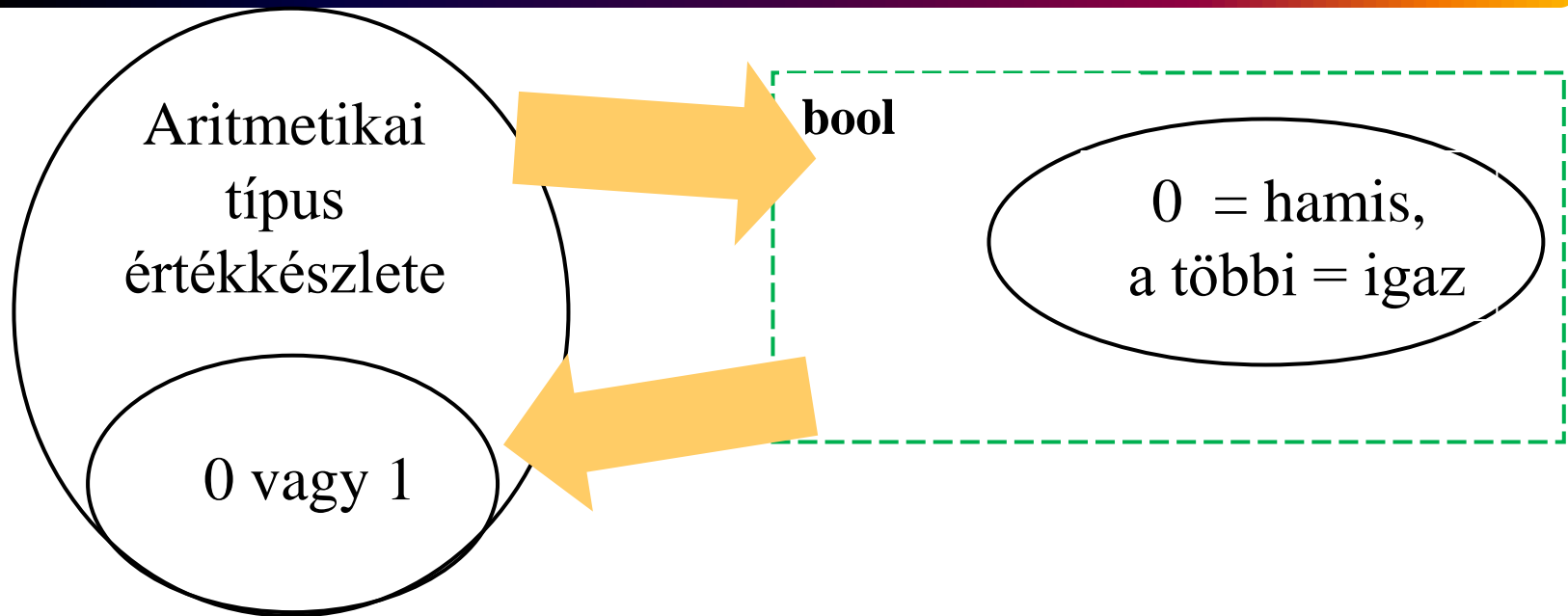
false

true

C++-ban nyelvi
elem

aritmetikai ↔ bool automatikus
típuskonverzió, ahogyan a C-ben
megszoktuk.

Aritmetikai és logikai konverzió



```
bool b1, b2, b3; int i;  
b1 = true; b2 = 3; i = b2; b3 = false;  
(b2 == true, i == 1)
```


Struktúra név típusává válik

```
struct Komplex {  
    float re;  
    float im;  
};
```

C++-ban a név
típus értékű

```
struct Lista_elem {  
    int i;  
    Lista_elem *kov;  
};
```

önhivatkozó
struktúránál kényelmes

Konstans (ism)

#define PI 3.14 helyett

const float PI = 3.14;

Típusmódosító amely megtiltja az objektum átírását
(fordító nem engedi, hogy balértékként szerepeljen)

Mutatók esetén:

const char * p; //p által **címzett terület** nem módosítható

char const * p; //ua.

char * const q; //q-t nem lehet megváltoztatni, de a
címzett **területet igen!**

const char * const q; //a mutató és a terület sem vált.

Két trükkös próbálkozás

```
const int x = 3;  
int *px = &x;  
*px = 4;
```

fordítási hiba

fordítási hiba

```
void f(int *i) { *i = 4; }  
const int x = 3;  
f(&x);
```

Felsorolás típus (szigorúbb lett)

```
enum Szinek {  
    piros, sarga, zold = 4  
};
```

típus

Szigorúbb ellenőrzés, mint az ANSI C-ben. Pl:

fordítási hiba

Szinek jelzo;

jelzo = 4;

jelzo = Szinek(8);

nincs hiba, de
meghatározatlan

érték létrehozása (konstrukció)

Prototípus kötelező



Előrehivatkozáskor kötelező

Tipikus C hiba:

```
double z = sqrt(2);
```



C feltételezi,
hogy int

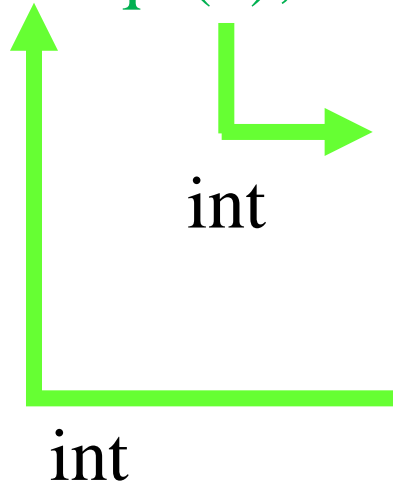


(double)
kellene

Miért baj ha elmarad?

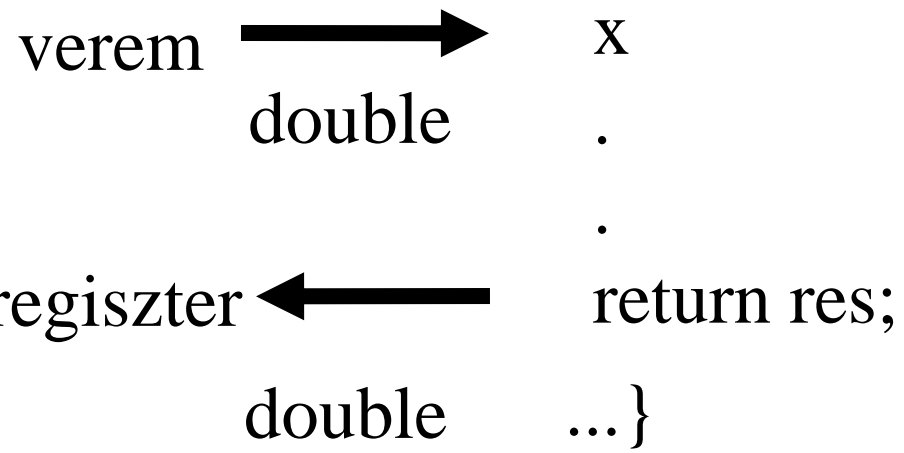
a függvényt hívó rész

```
double z=sqrt(2);
```



a hívott függvény

```
double sqrt(double x) {
```



Mutatók és címek (ism.)

- Minden változó és függvény memóriában levő helye (címe) képezhető. (pl: &valtozo)
- Ez a cím ún. pointerben vagy mutatóban tárolható.
- A pointer egy olyan típus, amelynek az értékkészlete cím, és mindig egy meghatározott típusú objektumra mutat.

int i, *ip;

float f, *fp;

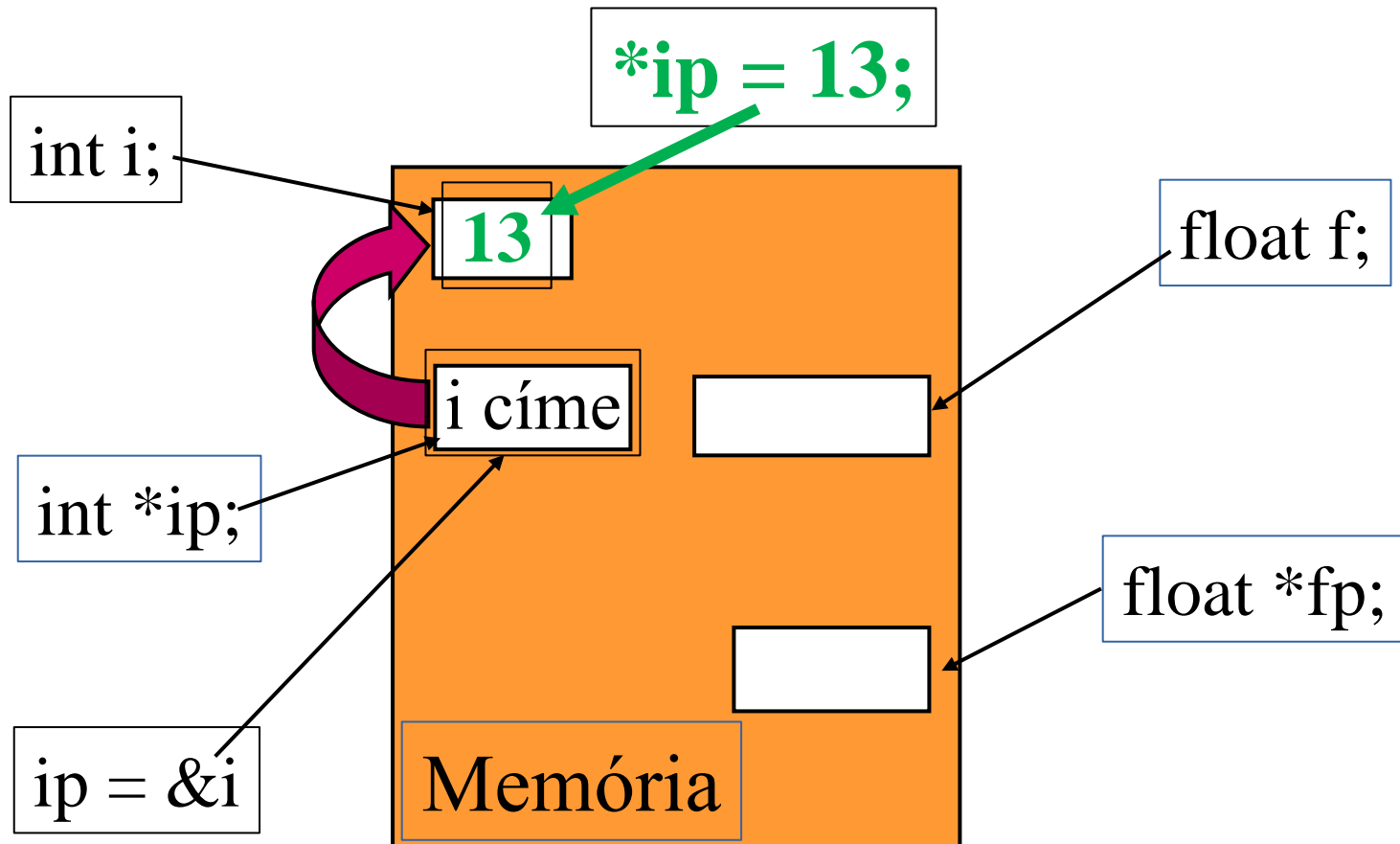
int-re mutató pointer

float-ra mutató pointer

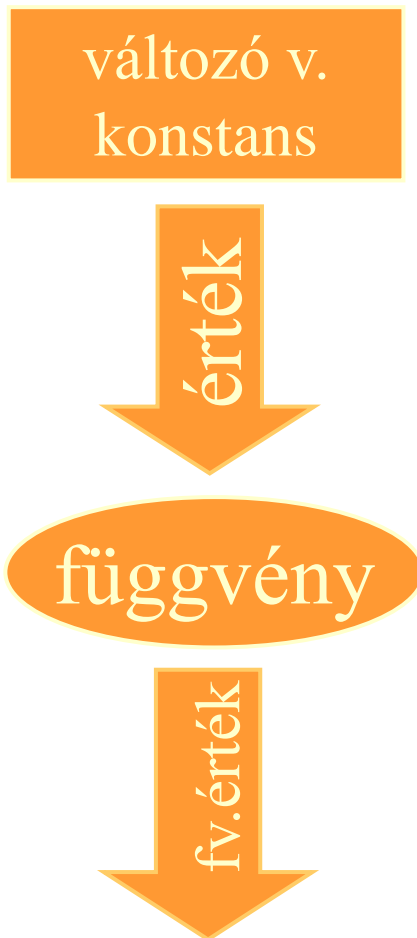
ip = &i;

fp = &f;

Indirekció (ism.)

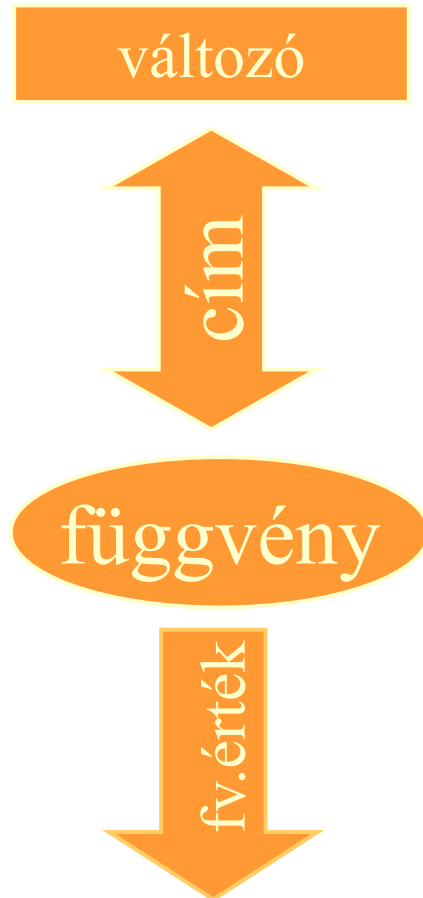


Értékparaméter (ism.)



- A paraméterek nem változhatnak meg, mivel azok értéke adódik át.
- Azok **eredeti tartalma az eredeti helyen megmarad.**
- A függvény csak a függvényértéken keresztül tud a külvilágnak eredményt szolgáltatni. (Ez sokszor kevés.)

Változó paraméter (ism.)



- A paraméter címét adjuk át, így az átadott paraméter elérhető, de meg is **változtatható**.
- A magas szintű nyelvek **elfedik ezt a trükköt**. Sem az aktuális paraméterek átadásakor, sem a formális paraméterekre való hivatkozáskor **nem kell jelölni**.
- Csupán a paraméter jellegét (változó) kell megadni.

Referencia (új típus)

Referencia: alternatív név
típus&

```
int i = 1;
```

```
int& r = i; // kötelező inicializálni, mert  
// tudni kell, hogy kinek az alternatív neve
```

```
int x = r; // x = 1;  
r = 2; // i = 2;
```

Változó paraméter referenciával

C:

```
void inc(int *a)
```

```
{
```

```
    (*a)++;
```

```
}
```

```
int x = 2;
```

```
inc(&x);
```

könnyen
lemarad

C++:

```
void inc(int &a)
```

```
{
```

```
    a++;
```

```
}
```

```
int x = 2;
```

```
inc(x);
```

nem kell
jelölni a
címképzést
híváskor

Példa pointerrel

```
// két érték felcserélése
// Változó paraméter pointerrel
void csere(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
...
int x, y;
csere(&x, &y);
```

Példa referenciával

```
// két érték felcserélése
// Változó paraméter referenciával
void csere(int& a, int& b) {
    int tmp = a;
    a = b;
    b = tmp;
}
...
int x, y;
csere(x, y);
```

Paraméterátadás összefogl.

- érték szerint
 - skalár
 - struct
- cím szerint (tömb, változtatni kell, hatékonyság)
 - típus&
 - típus*
- **Pointer paraméter és a változtatandó paraméter szétválik.**

Paraméterátadás /2

Pointer, referencia + const használatával a hozzáférés jól szabályozható:

```
struct Data { double dx[1000]; int px[2000]; } d;  
void f1(Data);           // f1(d); értékparaméter  
void f2(const Data* );  // f2(&d); nem változhat  
void f3(const Data&);   // f3(d); nem változhat  
void f4(Data*);         // f4(&d); változhat  
void f5(Data&);        // f5(d); változhat
```


Függvény argumentumok

- Konvertert írunk, ami tetszőleges számrendszerbe tud konvertálni. A fv. a számrendszer alapját paraméterként kapja.

```
char *int2Ascii(int i, int base = 10);
```

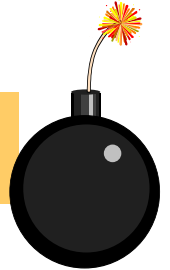
Csak az argumentumlista végén lehetnek default argumentumok, akár több is.

- f(), f(void) - nincs paraméter
- f(...) - nem kell ellenőrizni
- f(int a, **int**)- nem fogjuk használni

Inline függvények

```
#define max(a,b) (a) > (b) ? a : b
```

```
x = 8, y = 1; z = max(x++, y++); x,y,z = ?
```



```
inline int max(int a, int b) {  
    return(a > b ? a: b);  
}
```

Nincs trükk. Pontosan úgy viselkedik, mint a függvény.
A hívás helyére beilleszti a kódot (lehetőleg).

Inline fv. példa

```
inline int max(int a, int b) { return(a > b ? a: b); }
```

```
int x = 3, y = 4, z;
```

```
z = max(x++, y++);
```

// a hívás helyére beépül a kód, miközben

// fv. hívás szabályai érvényesülnek

```
z = a > b ? a : b; // a = 3, b = 4
```

Eredmény:

```
x -> 4, y -> 5, z -> 4
```

optimalizáló ezen
még optimalizálhat

Függvény név túlterhelés

```
int max(int a, int b) {  
    return(a > b ? a: b);  
}
```

```
double max(double a,  
           double b) {  
    return(a > b ? a: b);  
}
```

```
int x = max(1, 2);
```

```
double f = max(1.2, 0.2);
```

Azt a változót használja,
ami illeszkedik a hívásra

Túlterhelt (overloaded) függvények

standard I/O, iostream

- cin
 - cout
 - cerr
 - clog
- } előre definiált objektumok

```
#include <iostream>  
using namespace std;
```

```
int main() {  
    cout << "Hello C++" << endl;  
}
```

Miért iostream ?

C-ben ezt írtuk:

```
printf("i=%d j=%d\n", i, j);
```

C++-ban ezt kell:

```
cout << "i = " << i << " j=" << j << endl;
```

Kinek jó ez ?

- A `printf`, `scanf` **nem biztonságos!** Nem lehet ellenőrizni a paraméterek típusát.
- A `printf`, `scanf` **nem bővíthető** új típussal.
- Lehet vegyesen ? (`sync_with_stdio()`)

A << és a >> új operátor?

- Nem új operátorok! A már ismert << és >> operátorok **túlterhelése** (overload).
- Az operátorok a C++ -ban függvények a függvények pedig többarcúak.
- A **cout** egy **ostream** típusú objektum, amihez léteznek

```
ostream& operator<<(ostream& os, int i);  
ostream& operator<<(ostream& os, double d);  
ostream& operator<<(ostream& os, const char *p);  
... alakú függvények. (Később pontosítjuk)
```

Függvény, mint balérték?

```
int main() {  
    cout << "Hello C++" << endl;  
}
```

`ostream&` operator<<(ostream& os, const char *p);
alakra illeszkedik.

Bal oldalon van a függvény?

Referencia értékű függvény lehet bal oldalon is.

Egyszerű példa

```
int x; // ronda globális!  
int& f1( ) { return x; }  
double& f2(double& d) { return d; }
```

Referenciát, azaz alternatív
nevet szolgáltatnak

```
int main( ) {  
    f1( ) = 5;  
    f1( )++;  
    double y = 0.1;  
    f2(y) *= 100;  
    cout << "x=" << x << " y=" << y << endl;  
} // kiírás: x=6 y=10
```

Példa: nagyobb

// Fájl: nagyobb_main.cpp

```
#include <iostream>
#include "fuggvenyeim.h"
using namespace std;
```

fv prototípusok, konstansok,
típusok, egyéb deklarációk

Később finomítjuk!

```
int main() {
    cout << "Kerek ket egész számot:" << endl;
    int i, j;
    cin >> i >> j;           // i és j értékének beolvasása
    int k = max(i, j);
    cout << "A nagyobb: " << k << endl; // nagyobb kiírása
}
```

Példa: nagyobb /2

```
// Fájl: fuggvenyeim.cpp
```

```
// Ebben valósítom meg a gyakran használt függvényeket.
```

```
#include "fuggvenyeim.h"
```

```
// Két int adat felcserélése
```

```
void csere(int& a, int& b) {
```

```
    int tmp = a;
```

```
    a = b;
```

```
    b = tmp;
```

```
}
```

```
// ....
```

Saját header-t is célszerű
behúzni ellenőrzés miatt

Példa: nagyobb /3

// Fájl: fuggvenyeim.h

// Ebben találhatóak a függvények prototípusai, típusok

```
#ifndef FUGGVENYEIM_H  
#define FUGGVENYEIM_H
```

```
/* csere
```

```
* Két int adat felcserélése
```

```
* @param a - egyik adat
```

```
* @param b - másik adat
```

```
*/
```

```
void csere(int& a, int& b);
```

Egy fordítási
egységben csak egyszer

Automatikus dok.
generálás

Függvény prototípusa

Példa: nagyobb /4

```
/*  
 * max  
 * Két int adat közül a nagyobb  
 * @param a - egyik adat  
 * @param b - másik adat  
 */  
  
// Ez egy inline függvény, amit minden fordítási egységben  
// definiálni kell.  
inline int max(int a, int b) { return a > b ? a : b; }  
  
#endif // FUGGVENYEIM_H
```

Példa fordítása

Fordítás parancssorból:

```
g++ nagyobb_main.cpp fuggvenyeim.cpp -o nagyobb_main
```

Fordítás IDE segítségével:

Projektet kell készíteni, ami tartalmazza a 3 fájlt:

nagyobb_main.cpp

fuggvenyeim.cpp

fuggvenyeim.h

Példa fordítása /2

Fordítás parancssorból make segítségével:

1. Elő kell állítani a függésegeket leíró **Makefile-t** pl:

kicsit hibás, mert nem veszi figyelembe a header változását.

OBJS = nagyobb_main.o fuggvenyeim.o

CC=g++

CPPFLAGS = -Wall

nagyobb_main: \$(OBJS)

\$(CC) -o \$@ \$(OBJS)

2. le kell futtatni a make programot:

make