Programozás alapjai II.
(1. ea) **C++**

C++ kialakulása, nem OO újdonságok:

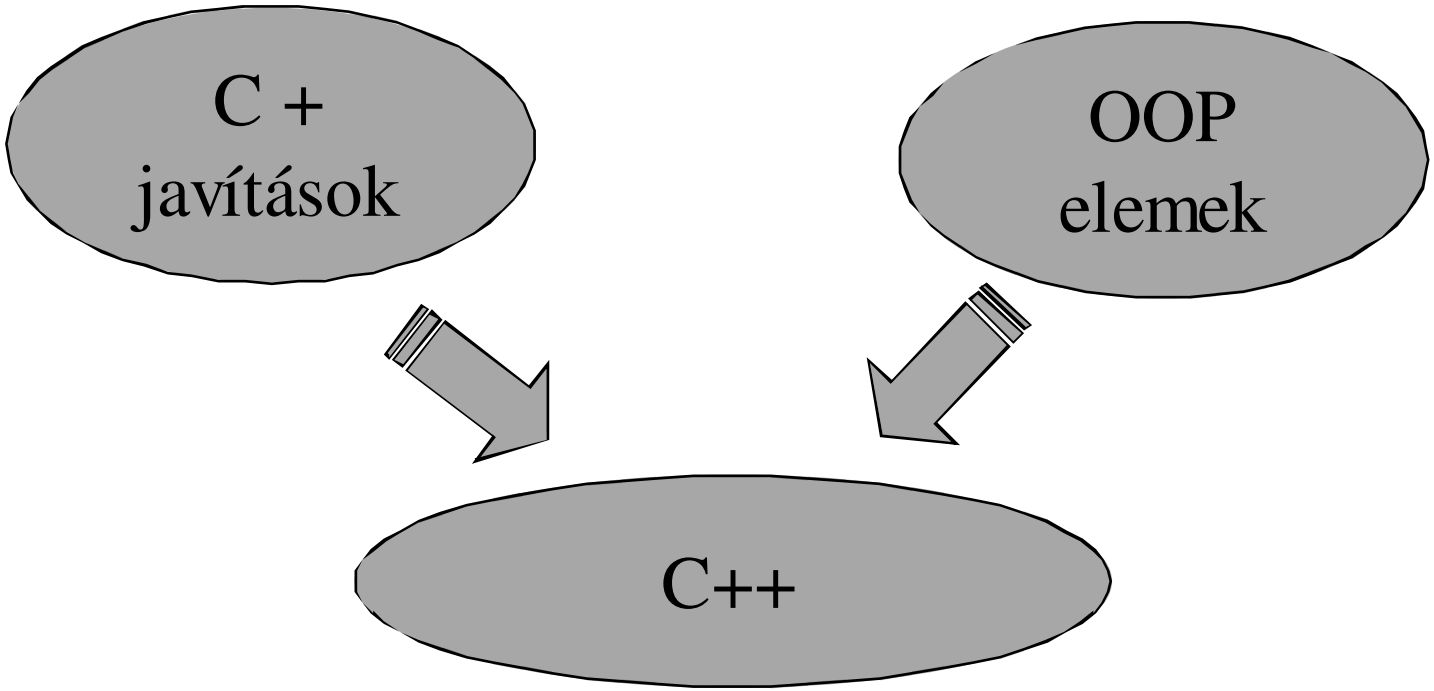
Szeberényi Imre
BME IIT
<szebi@iit.bme.hu>

MŰEGYETEM 1782

C++ kialakulása

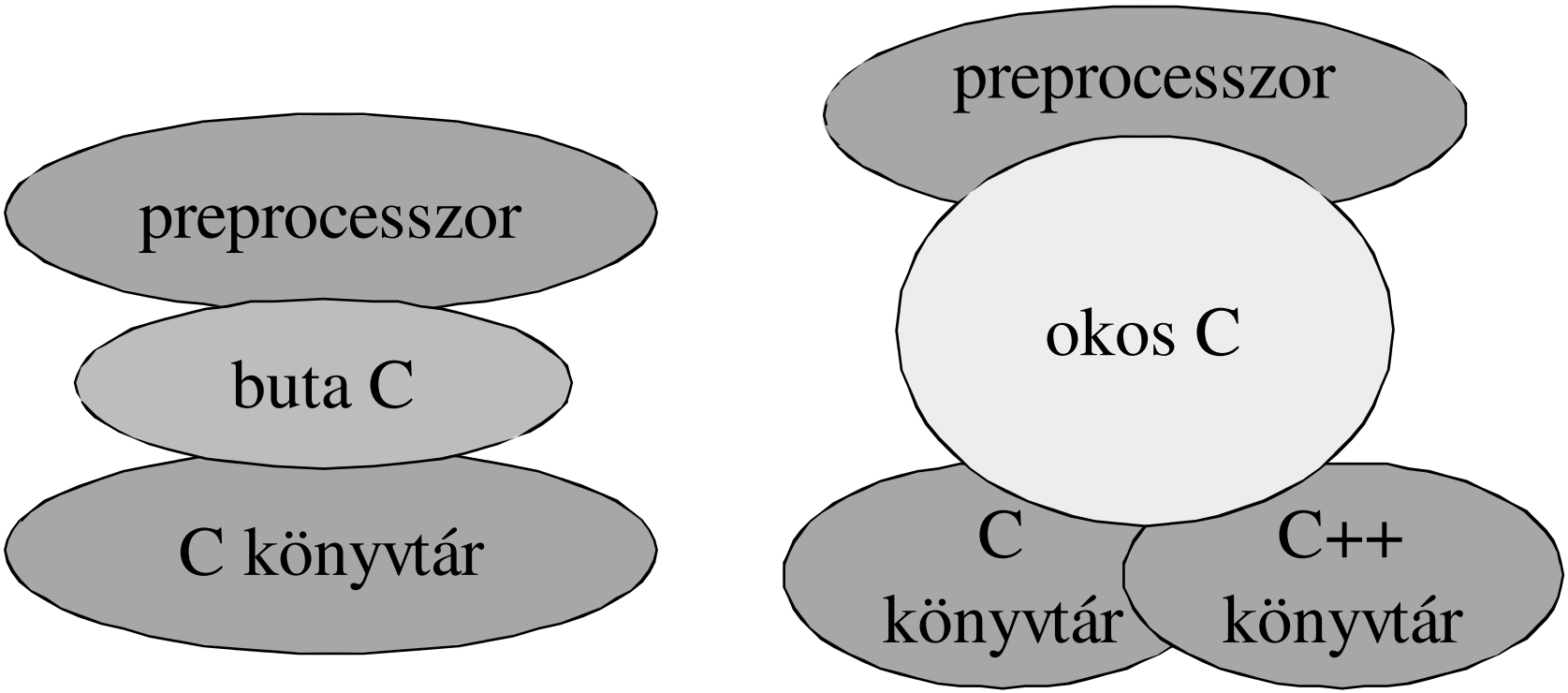
Veszélyforrások csökkentése

Objektum orientált szemlélet

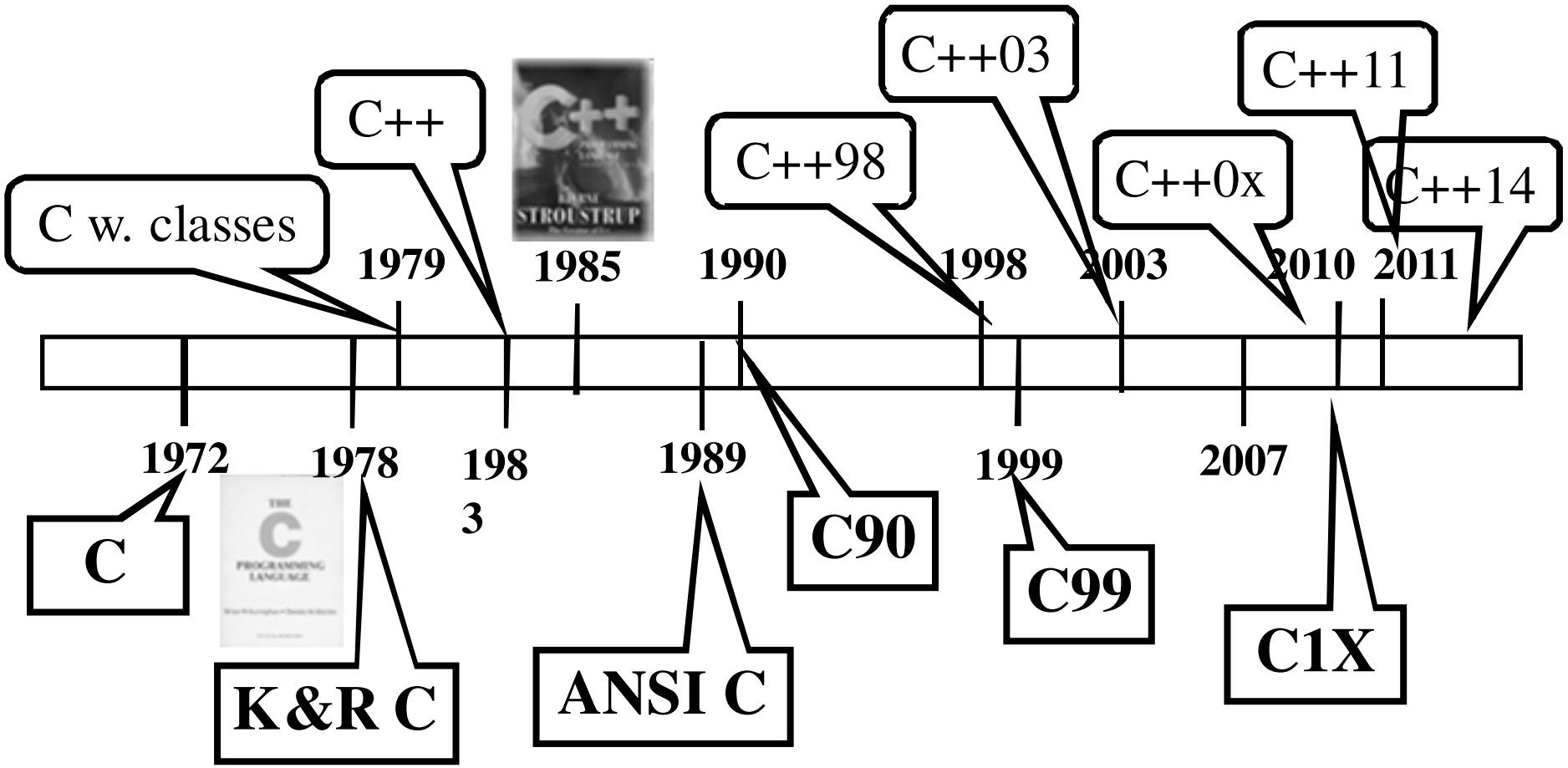


A fejlődés során jelentős kölcsönhatások voltak a C és a C++ között

C és C++ viszonya



C és C++változatai



C99

- változók és a kód keverése (for fejében is)

```
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

C99 támogatottság

Implementation [hide]	Level of support	Details
AMD x86 Open64 Compiler Suite	Mostly	Has C99 support equal to that of GCC.[10]
C++ Builder	Only in 64-bit mode, since latter is CLang fork [citation needed]	
cc65	Partial	Full C89 and C99 support is not implemented, partly due to platform limitations (MOS Technology 6502). There is no support planned for some C99 types like _Complex and 64-bit integers (long long).[11]
Ch	Partial	Supports major C99 features.[12]
Clang	Mostly	Supports all features except C99 floating-point pragmas.[13]
CompCert	Mostly	
cparser	Full	Supports C99 features.[14]
Digital Mars C/C++ Compiler	Partial	Lacks support for some features, such as tgmath.h and _Pragma.[15]
GCC	Mostly	As of June 2014 in mainline GCC, extended identifiers, standard pragmas and IEEE 754/IEC 60559 floating point support are missing. Additionally, some features (such as extended integer types and new library functions) must be provided by the C standard library and are thus out of scope for GCC.[16] GCC's 4.6 and 4.7 releases also provides the same level of compliance.[17][18] Almost complete IEEE 754 support if the hardware is compliant.[19]
Green Hills Software	Full	
IBM C for AIX, V6 [20] and XL C/C++ V11.1 for AIX [21]	Full	
IBM Rational logiscope	Full	Until Logiscope 6.3, only basic constructs of C99 were supported. C99 is officially supported in Logiscope 6.4 and later versions.[22]
Intel C++ compiler	Mostly	
Microsoft Visual C++	Partial[23]	C99 is not supported as of Visual C++ 2012,[24][25][26] while Visual C++ 2013 implements support for a limited subset of C99.[27]
Open Watcom	Partial	Implements the most-used parts of the standard. However, they are enabled only through an undocumented command-line switch.[28]
Pelles C	Full	Supports all C99 features.[29]
Portable C compiler	Partial	Working towards becoming C99-compliant.[citation needed]
Sun Studio	Full[30]	
The Amsterdam Compiler Kit	No	A C99 frontend is currently under investigation.
The Portland Group PGI C/C++	Full	
Tiny C Compiler	Partial	Does not support complex numbers.[31][32] The developers state that "TCC is heading toward full ISOC99 compliance".[33]
vbcc	Partial	

http://en.wikipedia.org/wiki/C99

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-re! 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/2

- 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
- Style guide betartása (pl. google-styleguide)

  http://google-styleguide.googlecode.com/svn/trunk/cppguide.html

A lényeg a következetességen van!

C++ újdonságok, bővítések

- Struktúranév típussá válik
- Csak preprocesszorral megoldható dolgok nyelvi szintre emelése (const, enum, inline)
- Kötelező prototípus, névterek
- Referencia, cím szerinti paraméterátadás
- Többarcú függvények (overload)
- Alapértelmezésű (default) argumentumok
- Dinamikus memória nyelvi szint. (new, delete)
- Változó definíció bárhol

Típusok

- logikai
- char
- egész
- felsorolás
- valós
- mutató
- referencia
- void

integrális (logikai, char, egész, felsorolás)

aritmetikai (egész, felsorolás, valós)

skalár (mutató, referencia)

- összetett adatszerkezetek (tömb struktúra) és osztályok

Logikai típus (új típus)

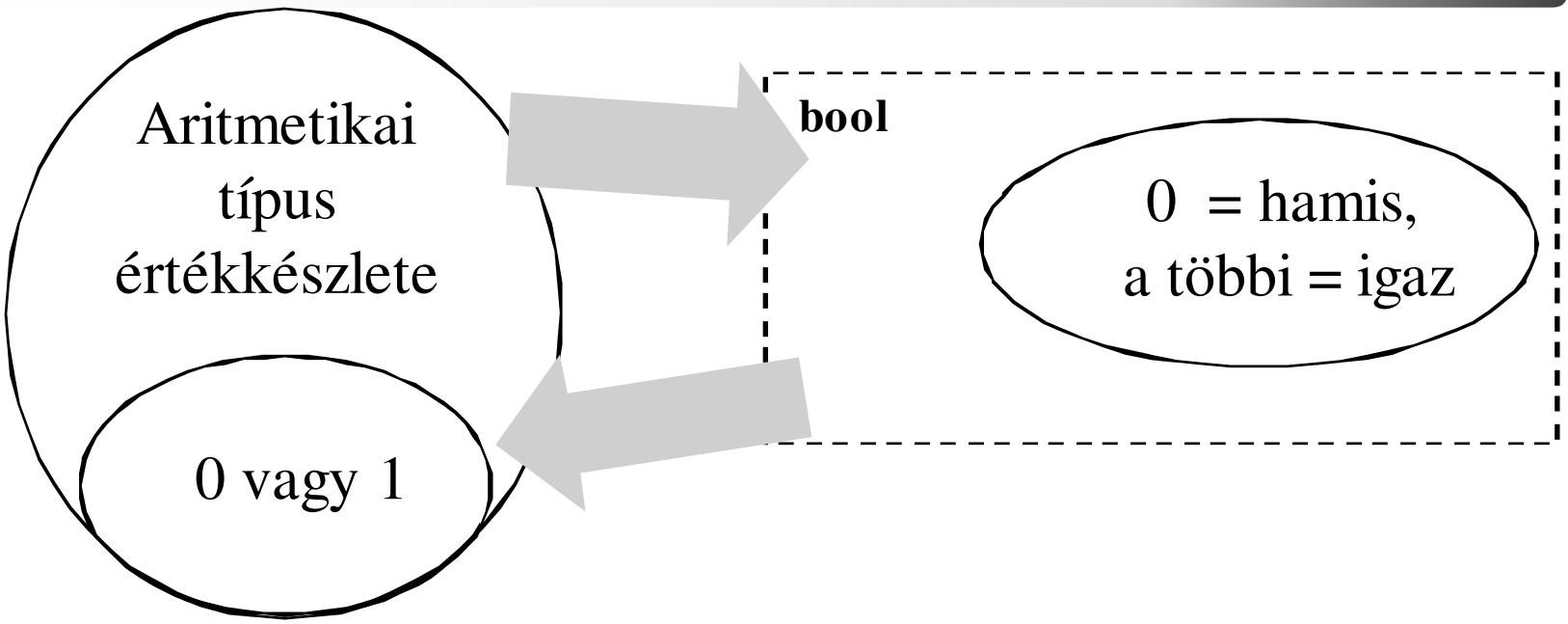
bool

false

true

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

Aritmetikai és logikai konverzió

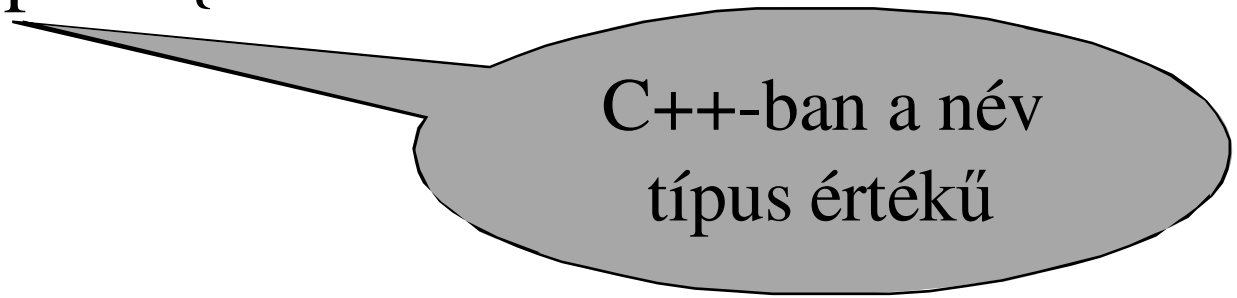


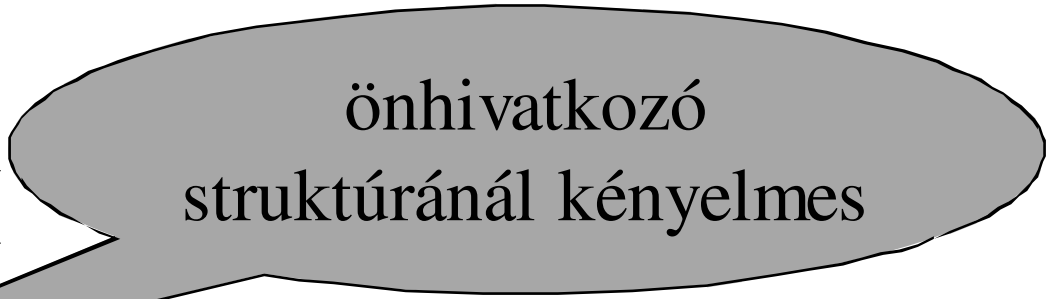

```
bool b1, b2, b3; int i;
b1 = true; b2 = 3; i = b2; b3 = false;
(b2 == true, i == 1)
```

Struktúra név típussá válik

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

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





Konstans (ism)

#define PI 3.14 helyett

const float PI = 3.14;

const:

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

Két trükkös próbálkozás

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

```
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:

```
Szinek jelzo;
jelzo = 4;
jelzo = Szinek(8);
```

fordítási hiba

nincs hiba, de meghatározatlan

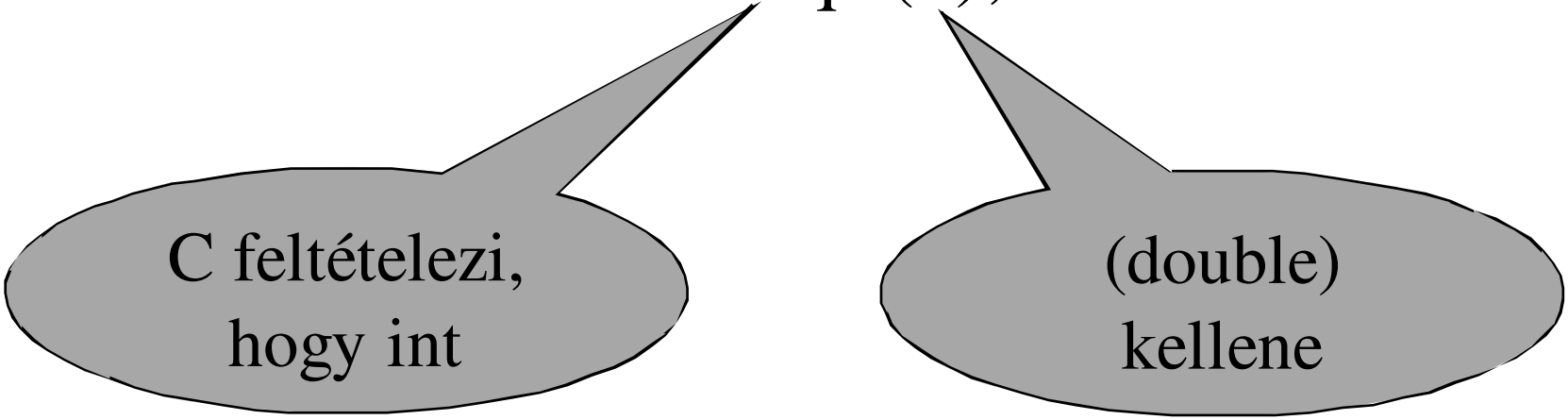
érték létrehozása

Prototípus kötelező

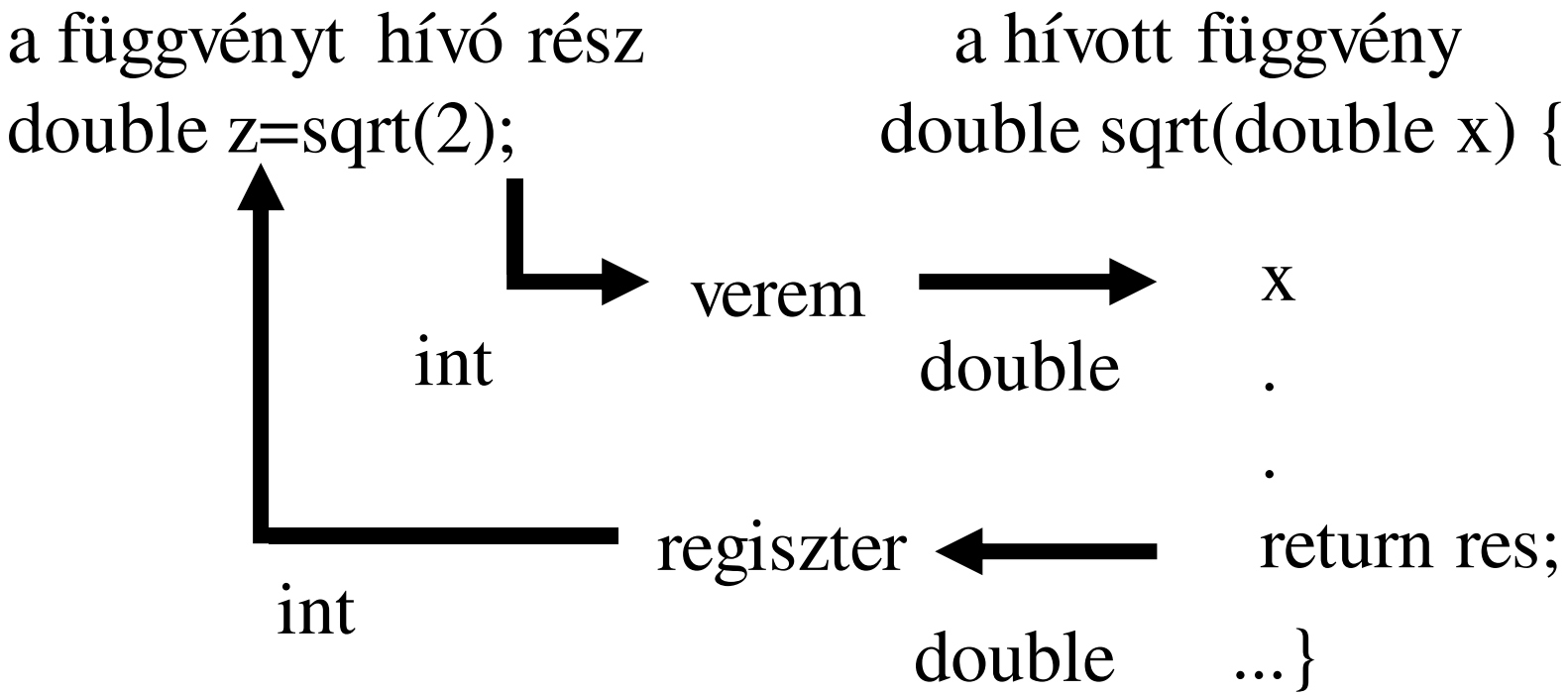
Előrehivatkozáskor kötelező

Tipikus C hiba:

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

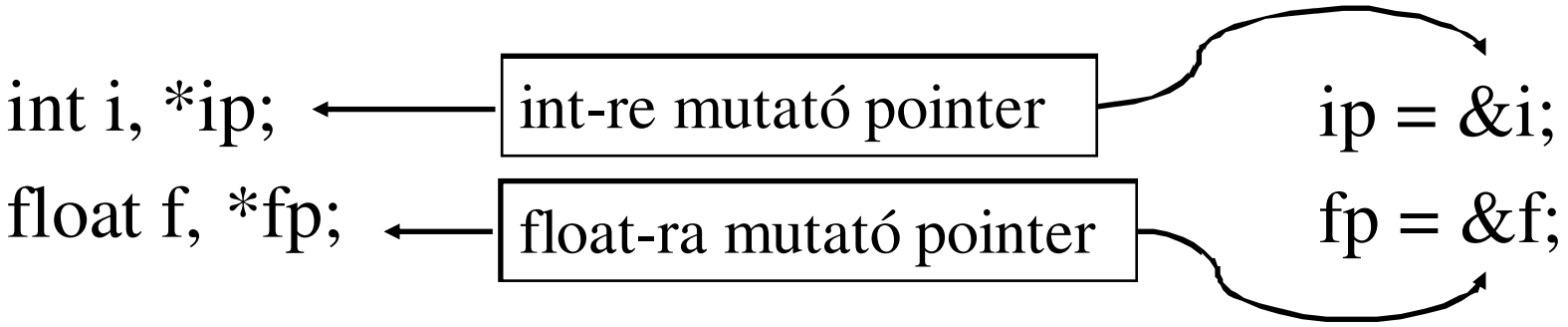


Miért baj ha elmarad?

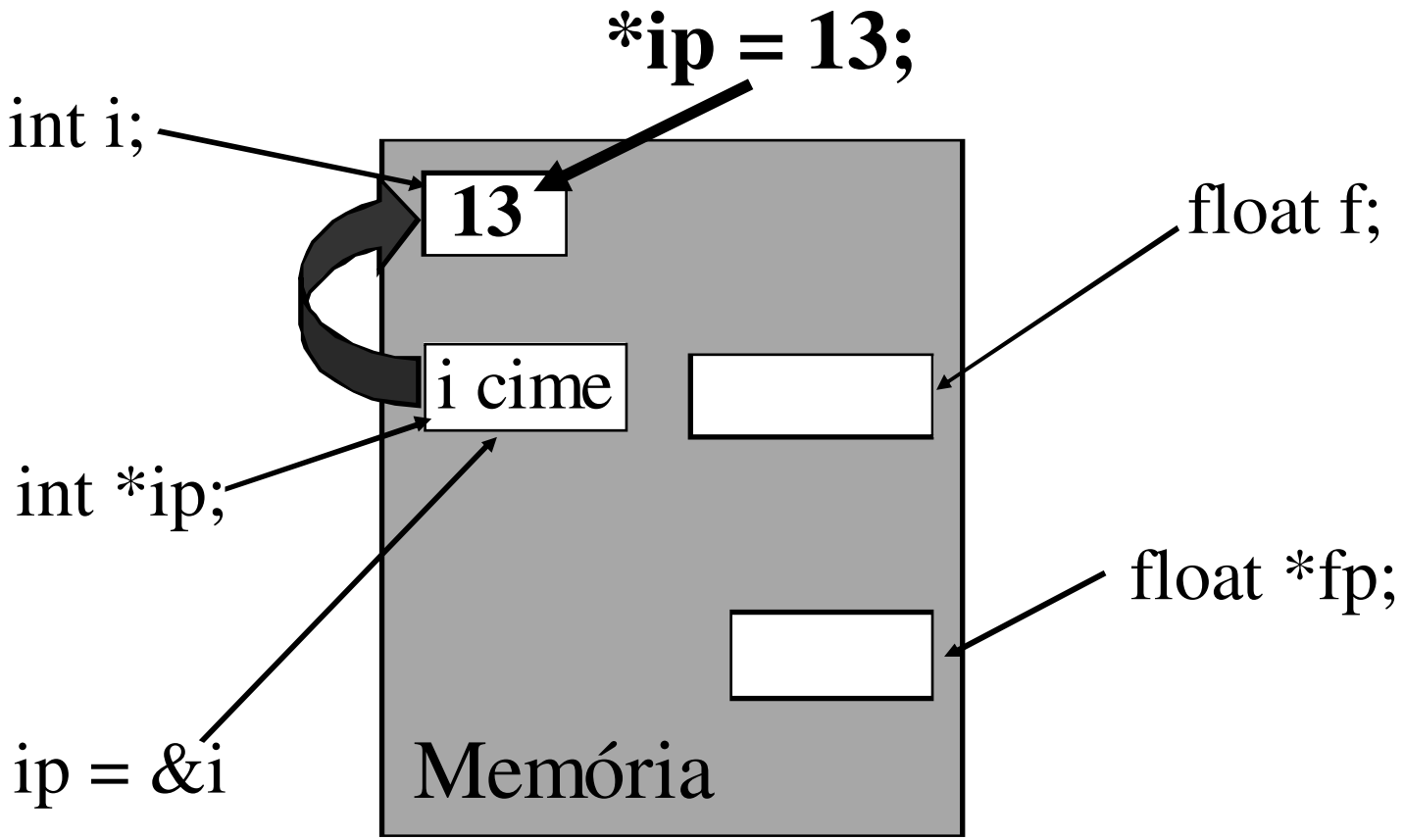


Mutatók és címek (ism.)

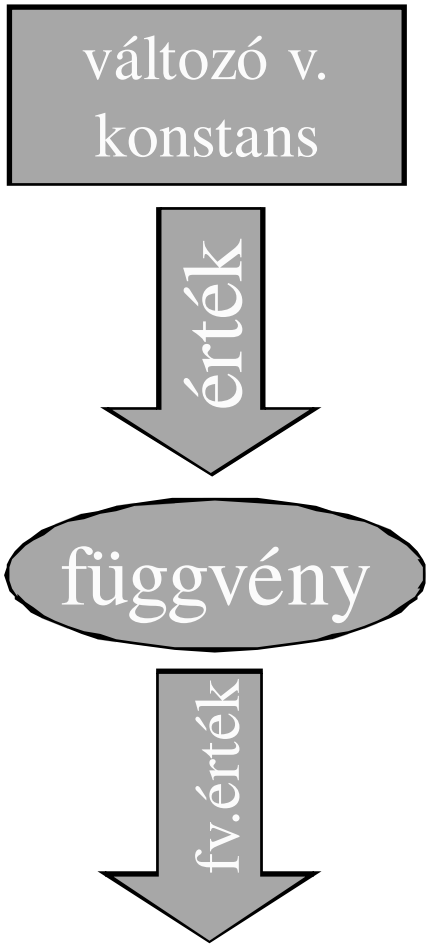
- Minden változó és függvény memóriában levő helye (címe) képezhető. (pl: &valtozo)
- Ez a cím ún. pointerben vagy mutatóban tárolható.
- A pointer egy olyan típus, amelynek az értékkészlete cím, és mindig egy meghatározott típusú objektumra mutat.



Indirekció (ism.)

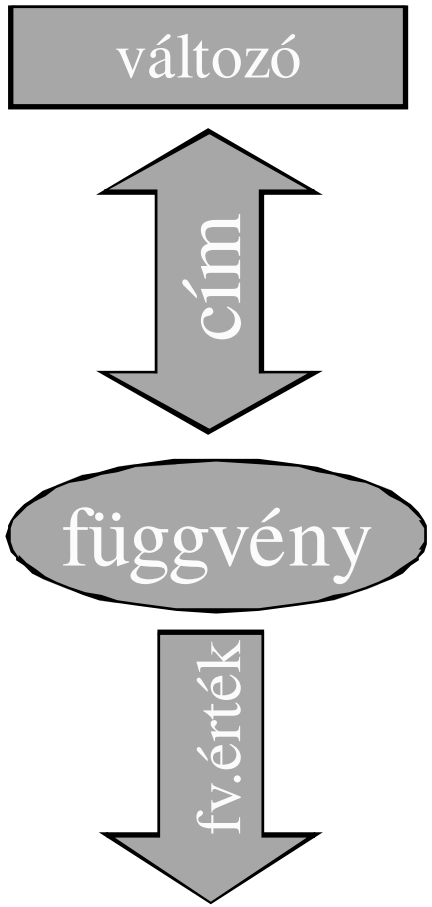


Értékparaméter (ism.)



- 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.)

Változóparaméter (ism.)



- A paraméter címe adódik át, így annak tartalma felhasználható, 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

tipus&

```
int    i = 1;
int& r = i;  // kötelező inicializálni, mert
             // valójában egy cím
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

1. gyak. példája referenciával:

```
// int ad(int a, int b, int *no)…
int ad(int a, int b, int& no) {
  no = a*a + b*b;
  return a+b;
}
int main() {
  int x, y, z;
  y = 5;
  x = ad(4, y, z);
}
```

Paraméterátadás

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

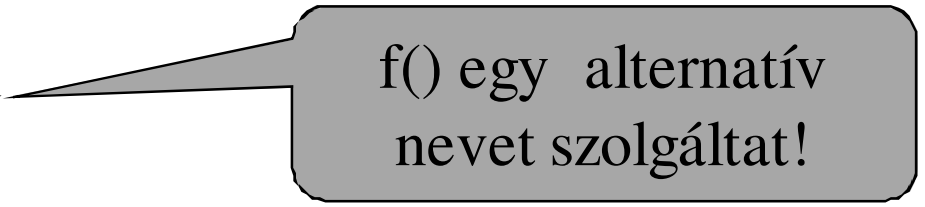
Paraméterátadás /2

- 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&);             // f2(d); változhat
```

Függvényhívás mint balérték

```
int x;
…
int& f( ) { return x; }
…
main( ) {
    f( ) = 5;
    f( )++;
    f( ) = f( ) + 2;
}
```



Inline

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

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

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

Nincs trükk. Pontosan úgy viselkedik, mint a függvény, de a hívás helyére kell beilleszteni a kódot (lehetőleg).

Függvény overload

```
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);
```

Többarcú függvények

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

Deklaráció és definíció

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

Ott deklaráljunk, 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 létezik !

Deklarációk feltételben és ciklusban

```
if (double d = fx(y)) {
   cout << d;
} else {
   d = 21.2;
   cout << d;
}
```

```
for (int i = 10; i--;)
    cout << i;


while (cin >> ch) {
    cout  << ch;
}
```

Névterek, scope operátor

- A moduláris programozás támogatására külön névterületeket definiálhatunk.
- Ez ebben levő nevekre (azonosítókra) a hatókör (scope) operátorral (::), vagy a using namespace direktívával hivatkozhatunk.

```
namespace nevterem {
  int alma;
  float fv(int i);
  char *nev;
}
```

```
nevterem::alma = 12;
float f = nevterem::fv(5);
```

```
using namespace nevterem;
alma = 8; float f = fv(3);
```

using direktíva

A using namespace direktívával a teljes névteret, vagy annak egy részét láthatóvá tehetjük:

```
using namespace nevterem;
alma = 8;
float f = fv(3);
```

```
using nevterem::alma;
using nevterem::fv;
alma = 8; float f = fv(3);
nevterem::nev = "Dr. Bubo";
```

Név nélküli névtér

Biztosítani akarjuk, hogy egy kódrészlet csak az adott fájlból legyen elérhető. Névütközés biztosan nem lesz.

```
namespace { // nincs neve
void solveTheProblem() { …. }
…
} // névtér vége
int main() {
  solveTheProblem();
  …
```

Névterek egymásba ágyazása, alias

- A névterek egymásba ágyazhatók.
- Egy létező névterhez egy újabb nevet rendelhetünk (rövidítés).

```
namespace  kis_nevterem {
  namespace belso_terem { int fontos; }
}
namespace bnt = ::kis_nevterem::belso_terem;
bnt::fontos = 8;
```

Az std névtér

- Standard függvények konstansok és objektumok névtere. Ebben van standard az I/O is.
- Az egyszerű példákban kinyitjuk az egész névteret az egyszerűbb írásmód miatt:

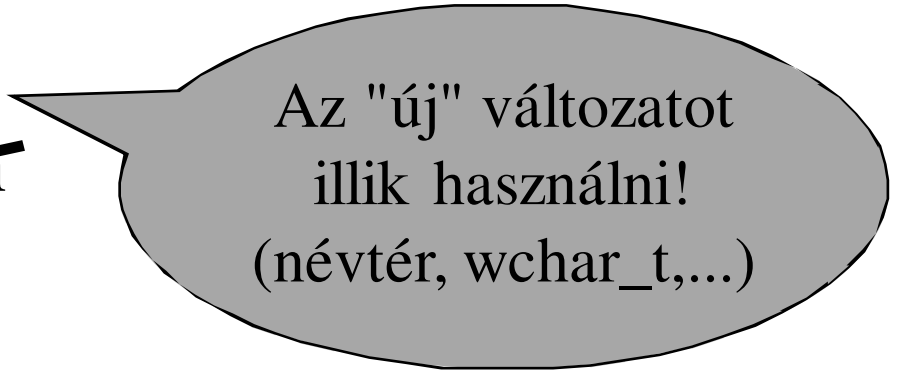
  using namespace std;

- Komoly programokban ez nem célszerű.
- Header-ben pedig soha ne tegyük!

standard I/O, iostream

- cin
- cout
- cerr

} iostream, ~~iostream.h~~




```
#include <iostream>
int main() {
   int a, b;
   std::cin >> a >> b;
   std::cout << "a+b=" << a + b << std::endl;
   return 0;
}
```

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())

Ezek új operátorok?

- Nem új operatorok! A már ismert << és >> operatorok felüldefiniálása.
- Az operatorok a C++ -ban függvények a függvények pedig többarcúak.

```
ostream& operator<<(ostream& os, int i);
ostream& operator<<(ostream& os, double d);
istream& operator>>(istream& is, int& i);
…
```

Részletek később!

Dinamikus memória

```
#include <malloc.h>
....
struct Lanc *p;
p = malloc(sizeof(Lanc));
if (p == NULL)
....
free( p );
```

```
Lanc *p;
p = new Lanc;
....
delete p;

Tömb:
int *p;
p = new int[10];
delete[] p;
```

Dinamikus memória /2

C: malloc(), free(), realloc()

- C++-ban is használható de csak nagyon körültekintően, ugyanis nem hívódik meg a megfelelő konstruktor ill. destruktor. Ezért inkább ne is használjuk.

C++ (operátor): new, delete, new[], delete[]

- Figyeljünk oda, hogy a tömböket mindig a delete[] operátorral szabadítsuk fel.

C++: nincs realloc()-nak megfelelő.

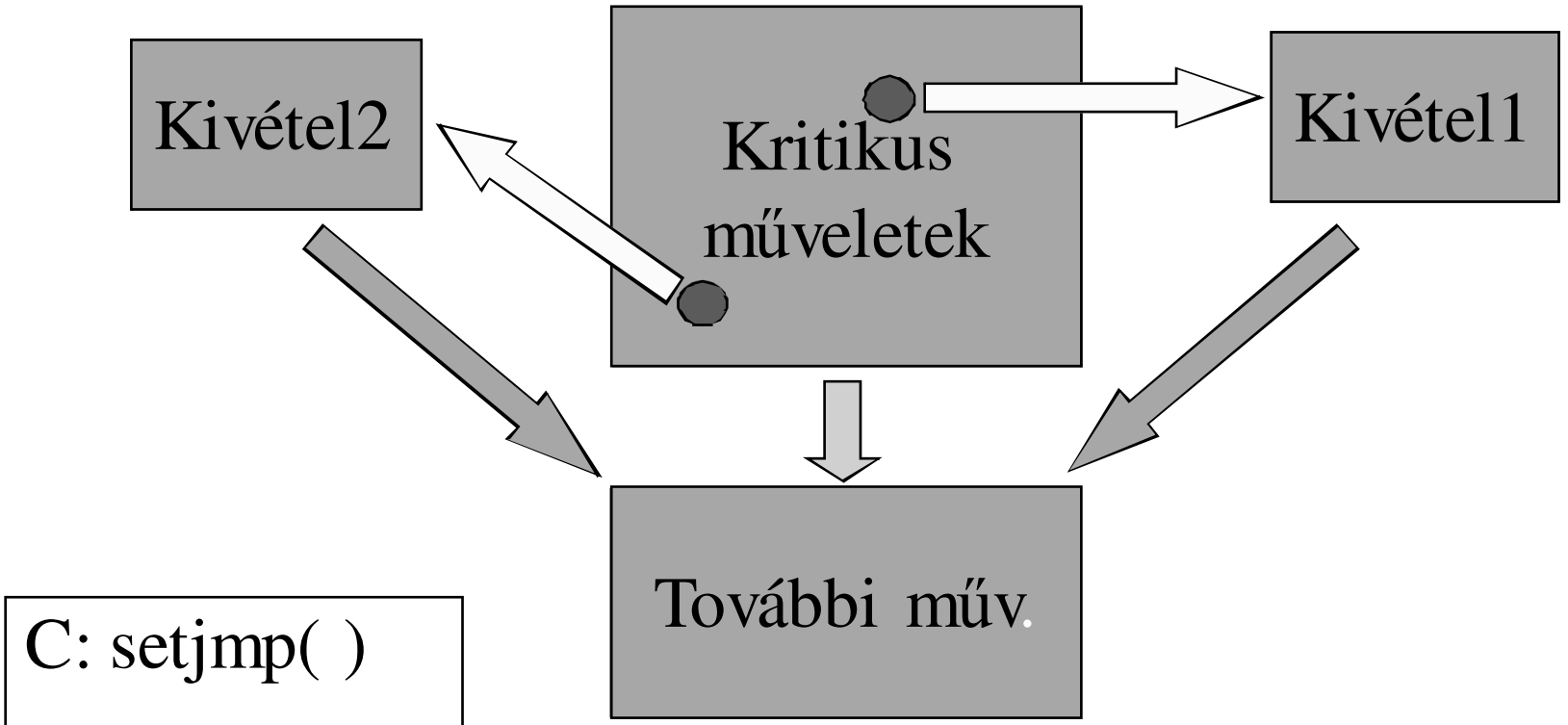
Mi van, ha elfogy a memória ?

```
void OutOfMem()
{
   cerr << "Gáz van\n";
   exit(1);
}
int main()
{
  set_new_handler(OutOfMem);
   double p* = new double;
...
```

Kivételkezelés

- Hibák kezelése gyakran nem a hiba keletkezésének helyén történik. (Legtöbbször nem tudjuk, hogy mit kell tenni. megállni, kiírni valami csúnyát, stb.)
- C++ típusorientát kivételkezelése:
  - figyelendő kódrészlet kijelölése (try)
  - kivétel továbbítása (throw)
  - esemény lekezelése (catch)

Kivételkezelés = globális goto

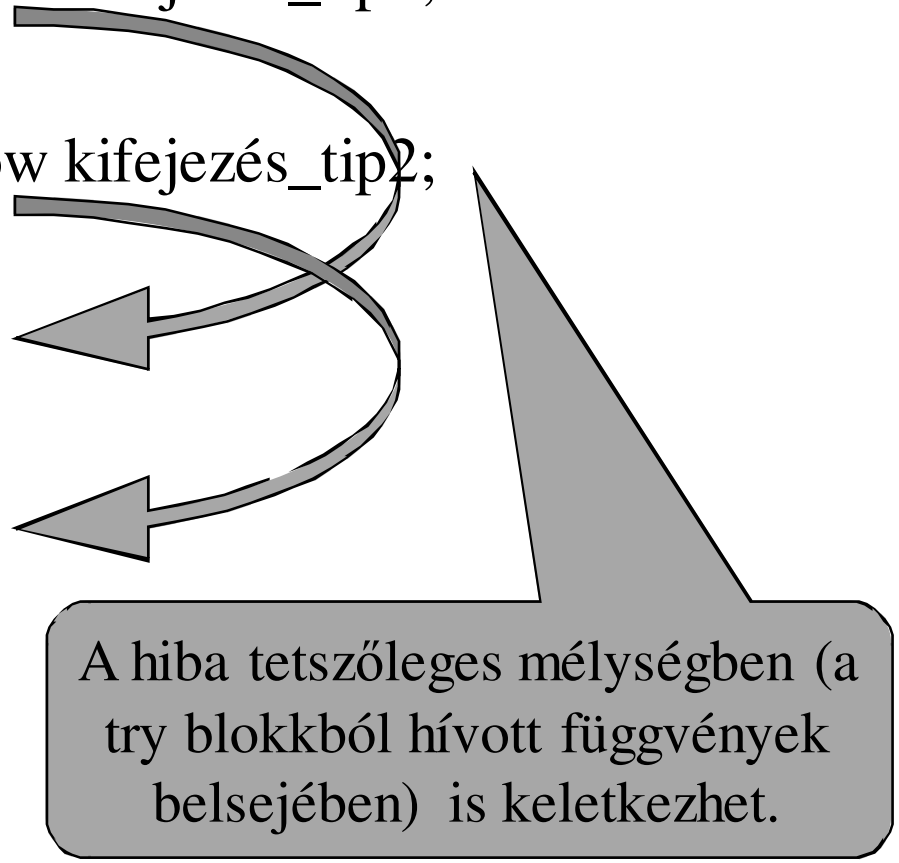


C: setjmp( )

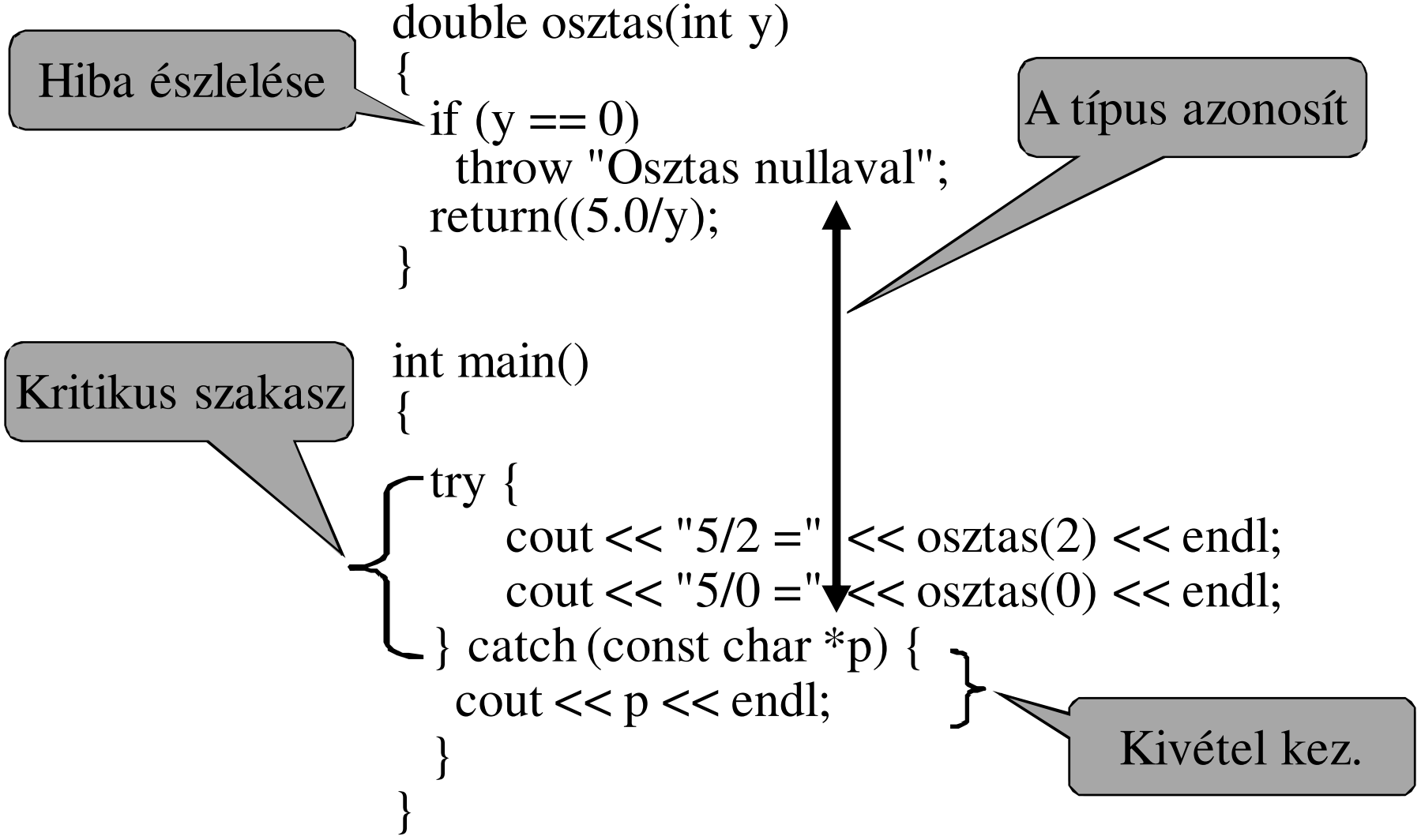
longjmp( )

Kivételkezelés/2

```
try {
  ..... Kritikus művelet1
                if (hiba) throw kifejezés_tip1;
  ..... Kritikus művelet2
                if (hiba) throw kifejezés_tip2;
} catch (típus1 param) {
  ..... Kivételkezelés1
} catch (típus2 param) {
  ..... Kivételkezelés2
}
... további utasítások
```



Kivételkezelés példa



Kivételkezelés a memóriára

```
#include <iostream>
using namespace std;    // csak az egyszerűbb írás miatt
int main() {
  long db = 0;
  try {
    while(true) {
        double *p = new double[1022];  db++;
    }
  } catch (bad_alloc) {
    cerr << "Gaz van" << endl;
  }
  cerr << "Ennyi new sikerult:" << db << endl;  return(0);
}
```

Futási példa az ural2-n

```
ural2:~$ cp ~szebi/proga2/mem_alloc.cpp  .
ural2:~$ g++ -static mem_alloc.cpp   -o mem_alloc
ural2:~$ ( ulimit -d 24; ./mem_alloc  )
```

A –static azért kell, hogy a betöltésnél ne legyen szükség
extra loader-re, ami extra memóriát használ.
A zárójel azért kell, hogy új shell induljon.

1. gyak. példája az új elemekkel

```
// createFromCharStr, createFromChar ...

void create(String& s, const char *p) {
    s.len = strlen(p);
//s->p = malloc((s->len+1)*sizeof(char));
//assert(s->p);
    s.p = new char[s-len+1];
    strcpy(s->p, p);
}
```

Polimorfizmus

Referencia

Nyelvi elem

1. gyak. példája az új elemekkel/3

```
void print(const String& s) {
//printf("%s", s->p);
    std::cout << s.p;
}
void dispose(String& s) {
//free(s->p)
    delete[] s.p;
}
char charAt(String& s, unsigned idx) {
    if (idx >= s.len)
        throw "Indexelési hiba";
    return s.p[idx];
}
```

Referencia

Memória kezelés

Kivétel dobása

1. gyak. példája az új elemekkel/3

Főprogram kivételkezeléssel

```
#include <iostream>
#include ”string0.h”

int main() {
  String a, b;
  try {
    create(a, ’x’); create(b, "alma");
    std::cout << charAt(b, 3) << std::endl;
  } catch (std::bad_alloc) {
    std::cerr << "elfogyott a mem." << std::endl;
  } catch (const char *p) {
    std::cerr << "Baj van:" << p << std::endl;
  }
  dispose(&a); dispose(&b);
  return 0;
}
```