Programozás alapjai 2.
(2. ea) **C++**

névterek, memóriakezelés

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

Hol tartunk?

- Változó definíció bárhol √
- Struktúranév típussá válik √
- Korábban csak preprocesszorral megoldható dolgok nyelvi szintre (const, bool, inline, template) √ **...**
- Kötelező a prototípus használata √
- Referencia, cím szerinti paraméterátadás √
- Többarcú fv.-ek (polimorf, túlterhelés, overload) √ **...**
- Alapértelmezésű (default) argumentumok √
- I/O stream √ **...**
- Névterek
- Dinamikus memória kez. nyelvi szint. (new, delete)

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 fordítási egységből legyen elérhető. Névütközés biztosan nem lesz.

```
#include <iostream>
namespace { // nincs neve
void solveTheProblem() { std::cout << "Solved\n";}
} // névtér vége
int main() {
  solveTheProblem();
}
```


Névterek egymásba ágyazása, alias

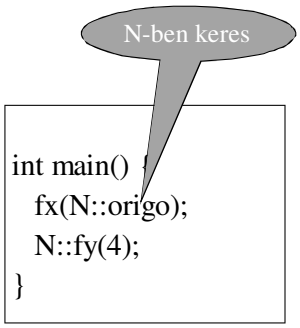
- A névterek egymásba ágyazhatók.
- Egy létező névterhez egy újabb nevet rendelhetünk (rövidítés).

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

Argument-dependent lookup (ADL)

Nem minősített függvények hívásakor a fv. argumentumainak névterében is keres.

```
namespace N {
   struct P { int x, y;};
   void fx(P p) { ... }
   void fy(int i) { ... }
   P origo;
}
```



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-be pedig soha ne tegyük! Miért?**

Standard I/O madártávlatból

```
#include <iostream>
using std::cout;
using std::endl;

int main() {
   cout << "Hello C++\n";
//→ std::operator<<(std::cout, "Hello C++\n");
   int i;
   std::cin >> i
   cout << "2 * " << i " = " << 2 * i << endl;
}
```

<iostream>

Standard I/O objektumait definiáló fejléc fájl valahogy így néz ki:

```
#include <ios>
#include <streambuf>
#include <istream>
#include <ostream>
namespace std {
    extern istream cin;
    extern ostream cout;
    extern ostream cerr;
    extern ostream clog;
....
```

Kiírás: **operator<<** egy ostream típushoz (insert)
Beolvasás: **operator>>** egy istream típushoz (extract)

A működés bonyolult. Most csak a felszínt kapargatjuk.

<iostream> /2

- Az iostream kapcsán megismert eszközök a fájlkezelésnél is használhatók.
- A stream típus logikai környezetben igazzá, vagy hamissá konvertálódik attól függően, hogy hibás állapotban van-e a stream.
- A streamek működése, belső állapota többek között manipulátorokkal befolyásolható

<iostream> /3

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
  cout << "Hexa konverer:" << endl;
  int x;
  while (cin >> x)
      cout << hex << x << endl;
}
```

logikai környezet.

manipulátorok

Manipulatorok <iomanip>

Hatás szempontjából 3 fajtájuk van:

- Azonnali hatás:
  cout << endl; // kiír egy ,'\n'-t és üríti a buffert
  cin >> ws; // eldobja az összes whitespcace-t
- csak a következő kiírásig érvényes
  cout << setw(4) << 3 << 4; // "␣␣␣34"
- a stream állapota tartósan megváltozik
  cout << setfill('0') << setw(3) << 7 // "007"

https://infocpp.iit.bme.hu/iomanip

Inline függvények (ism.)

#define max(a,b) (a) < (b) ? b : a

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

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

Úgy viselkedik, mint a függvény, de beépül.

OK, de kell double-re, long-ra, ...

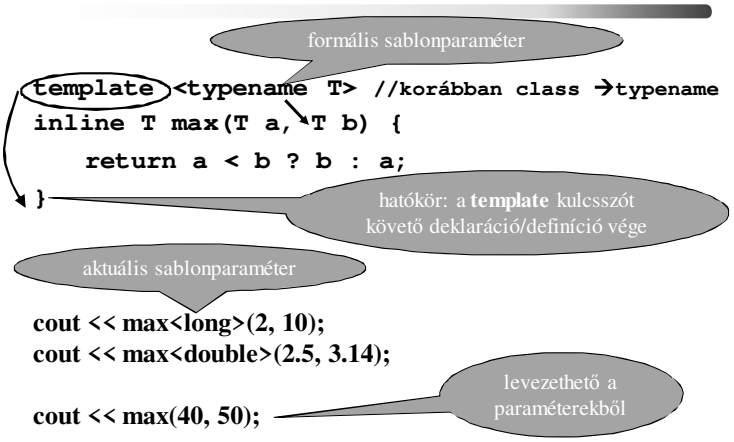
1. megoldási próbálkozás

```
inline int max(int a, int b) {
        return(a < b ? b : a); }
inline long max(long a, long b) {
        return(a < b ? b : a); }
inline double max(double a, double b) {
        return(a < b ? b : a); }
```

Az azonos nevekből nincs baj (túlterhelés), de fárasztó leírni minden típushoz.

Lehet, hogy a makró mégis jobb?

Megoldás: template – nyelvi elem



aktuális sablonparaméter

```
cout << max<long>(2, 10);
cout << max<double>(2.5, 3.14);
cout << max(40, 50);
```

levezethető a paraméterekből

Mi a sablon?

- Parametrikus polimorfizmus.
- Típus biztos nyelvi elem az általánosításhoz.
- Gyártási forma: a sablonparaméterektől függően példányosítja a fordító (megírja a programot).
- Paraméter: típus, konstans, függvény, sablon
- Feldolgozása fordítási idejű ezért a példányosítás helyének és a sablonnak egy fordítási egységben kell lennie. → gyakran header fájlba tesszük
- A példában függvénysablont láttunk, de később adatszerkezeteknél is használni fogjuk.

Mindeden típusra jó ez a max?

```
template <typename T>
T max(T a, T b) { // inline felesleges, mert...
    return a < b ? b : a;
}

cout << max(40, 50);
cout << max("alma", "korte") // "alma" → const char *
```

Valóban a címeket akartuk összehasonlítani?

strcmp kellene, de hogyan? Több megoldás van:

1. **Összehasonlító függvény (predikátum)**
2. **Specializáció**
3. **...**

1. megoldás: összehasonlító fv.

```
// max 3 paraméteres változata (overload)
template <typename T, typename C>
T max(T a, T b, C cmp) {
    return cmp(a, b) ? b : a;
}

bool strLess(const char *s1, const char *s2) {
    return strcmp(s1, s2) < 0;
}

cout << max(40, 50);  // 50
cout << max("alma", "korte", strLess); // korte
```

Predikátum

2. megoldás: Template specializáció

```
// Teljes specilaizáció T::= const char* esetre
template<>
inline
const char* max(const char* a, const char* b) {
    return strcmp(a,b) > 0 ? a : b;
}

cout << max<long>(2, 10);          // 10
cout << max<double>(1, 3.14);      // 3.14
cout << max(40, 50);               // 50
cout << max("alma", "korte");      // korte
cout << max("Ádám", "Béla");       // Ádám ??
```


Ékezettel C-ben is baj volt

Egy betű → egy karakter kódolásnál srcoll() fv.

```
template<>
const char* max(const char* a, const char* b) {
    return strcoll(a,b) < 0 ? b : a;
}
cout << max("Ádám", "Béla") << endl; // Béla
```

```
git.ik.bme.hu/Prog2/eloadas_peldak/ea_02/
 -> template
```

Egyéb kódolásnál (pl. UTF-8) a C string helyett valami mást érdemes használni, de majd később.

Függvénysablon összefoglalás

- Függvények, algoritmusok általánosítási eszköze.
- Hatékony, paraméterezhető, újrafelhasználható, általános.
- Fordítási időben generálódó függvény.
- → A példányosítás helyének és a sablonnak egy fordítási egységben kell lennie.
- → Automatikusan inline-ként fordulhat.
- Függvény overload, default paraméterek ugyanúgy mint más függvényekkel.
- Sablon specializáció → paraméter típustól függően más kód generálódik. Generikus programozás.

Predikátum

- Logikai függvény, ami egy algoritmus működését befolyásolja
- Pl.: válasszuk ki egy tömbből a leg... elemet!
- Melyik a leg? A predikátum függvény adja meg.

```
template <typename T, typename S>
T legElem(T a[], int n, S sel) {
    T tmp = a[0];
    for (int i = 1; i < n; ++i)
        if (sel(a[i], tmp)) tmp = a[i];
    return tmp;
}
```


Predikátum példa

```
template <typename T, typename S>
T legElem ...

// a predikátum is lehet template
template <typename T>
bool nagyobb_e(T a, T b) {
    return a > b;
}

int tomb[] = { 1, 3, 4, 80, -21 };
cout << legElem(tomb, 5, nagyobb_e<int>);

git.ik.bme.hu/Prog2/eloadas_peldak/ea_02/
 -> predicate
```

Dinamikus memória

C:

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

C++:

```
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.

Példa: fordit_main.cpp

```
#include <iostream>
#include "szimpla_lanc.h"
using std::cout;
using std::cin;
int main() {
   int i;
   Lanc_elem* kezdo = NULL;  // üres lánc
   while (cin >> i)  kezdo = lanc_epit(kezdo, i);
   cout << "Adatok forditott sorrendben:" <<
   std::endl;
   lanc_kiir(cout, kezdo);
   lanc_felszabadit(kezdo);
}
```

Számokat olvasunk be és fordított sorrendben kiírjuk. Láncban tárolunk.

Példa: szimpla_lanc.h

```
#ifndef SZIMPLA_LANC_H
#define SZIMPLA_LANC_H
#include <iostream>
struct Lanc_elem {
   int adat;
   Lanc_elem* kov;
};
Lanc_elem* lanc_epit(Lanc_elem* p, int i);
void lanc_kiir(std::ostream& os, const Lanc_elem*
   p);
void lanc_felszabadit(Lanc_elem* p);
#endif // SZIMPLA_LANC_H
```

Struktúra név típussá vált

Névteret nem nyitunk .h-ban!

Példa: szimpla_lanc.cpp

```
#include "szimpla_lanc.h"

Lanc_elem* lanc_epit(Lanc_elem* p, int a) {
   Lanc_elem *uj = new Lanc_elem;
   uj->adat = a;
   uj->kov = p;
   return uj;
}
```

Vigyázat ez operátor! Nem szabad a malloc-ot formálisan lecserélni, mert mást jelent!

Példa: szimpla_lanc.cpp /2

```
...
void lanc_kiir(std::ostream& os,
                          const Lanc_elem* p) {
   while (p != NULL) {
      os << p->adat << ' ';
      p = p->kov;
   }
}
```

Példa: szimpla_lanc.cpp /3

```
...
void lanc_felszabadit(Lanc_elem *p) {
   while (p != NULL) {
      Lanc_elem *tmp = p->kov;
      delete p;
      p = tmp;
   }
}
```

git.ik.bme.hu/Prog2/eloadas_peldak/ea_02/
-> fordít

Mi van, ha elfogy a memória ?

Két működési mód választható:

1. Hibakezelő mechanizmus indul be
   - meghívódik a new_handler, ha van, egyébként
   - bad_alloc kivétel generálódik
2. NULL pointerrel tér vissza

Mai implementációkban az alapeset a kivételkezelés, ami manapság a hibakezelés szabványosnak tekinthető megoldása.

new_handler

Kevésbé használt megoldás, de pl. garbage colletor-hoz használható.

```
void outOfMem() {
   cerr << "Gáz van\n";
   exit(1);
}

int main() {
  set_new_handler(outOfMem);
  double p* = new double;
....
```

A visszatérésnek (return) akkor van értelme, ha ez a rutin képes területet felszabadítani. Különben végtelen ciklus.

Kivételes esetek

- Gyakran hibakezelésnek mondjuk, de nem csak hiba lehet kivételes eset. Kivételes eset pl. amikor egy program először indul el az adott környezetben, és a további működéséhez ...
- Alapvetően két kategóriába sorolhatók:
  - végzetes,
  - nem végzetes
- Kezelésük nagyon különböző
  - nincs mit tenni, meg kell állni
  - az eset ott helyben kezelhető
  - az eset a program más részében kezelhető
  - nem tudjuk, elhalasztjuk (másra bízzuk) a döntést.

Példák kivételes esetre

- printf függvényt nem a vssszatrési értéke miatt hívjuk, pedig a visszatérési értékében jelzi, hogy sikerült-e kiírni.

  Mit kell/lehet tenni? Pl:

  assert(printf("Hello Cicus") >= 0);
- Egy erőforrás lefoglalása nem sikerül

  Mit kell/lehet tenni?

  Meg lehet próbálni később

Példák kivételes esetre /2

- Adott nyelvhez (természetes) tartózó üzenetek állománya nem található.

  Mit kell/lehet tenni? Pl:

  Meg lehet próbálni másik nyelvet
- Másodfokú egyenletnek nincs valós megoldása. Mit kell/lehet tenni?

  Jelezni kell a függvényt hívónak

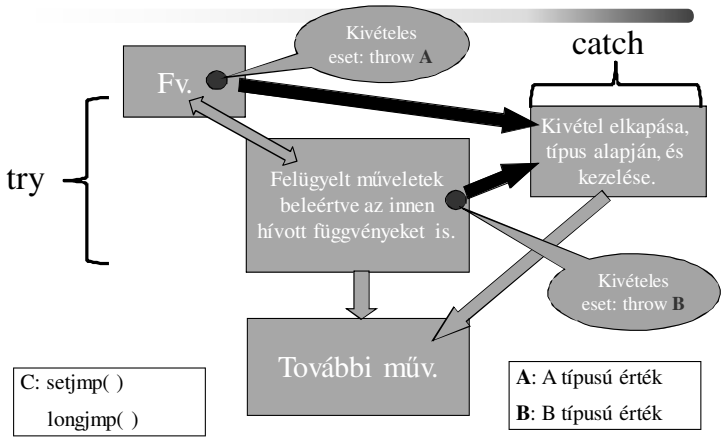
Kivételes esetek kezelése, jelzése

- Kinek kell jelezni?
  - felhasználó, másik programozó, másik program
  - saját magunknak
- A kivételes eset kezelése gyakran nem annak keletkezési helyén történik. (Legtöbbször nem tudjuk, hogy mit kell tenni. Megállni, kiírni valami csúnyát, stb.)

Kivétel kezelés

- C++ típus orientált kivételkezelést támogat, amivel a kivételes esetek kezelésének szinte minden formája megvalósítható.
- A kivételkezeléshez tartozó tevékenységek:
  - figyelendő kódrészlet kijelölése (try)
  - kivétel továbbítása (throw)
  - esemény lekezelése (catch)

Kivételkezelés = globális goto



Kivételkezelés/2

```
try {
  ..... Kritikus művelet1 pl. egy függvény hívása, aminek a
    belsejében: { if (hiba) throw kifejezés_tipus1; }
  ..... Kritikus művelet2 pl. egy másik függvény hívása,
    aminek a belsejében: { if (hiba) throw kifejezés_tipus2; }
} catch (típus1 param) {
  ..... Kivételkezelés1
} catch (típus2 param) {
  ..... Kivételkezelés2
}
```

A hiba tetszőleges mélységben keletkezhet.
Közvetlenül a **try-catch** blokkban a **throw**-nak nincs sok értelme, hiszen akkor már kezelni is tudnánk a hibát.

Kivételkezelés példa

```
double osztas(int y)
{
  if (y == 0)
    throw "Osztas nullaval";
  return((5.0/y);
}

int main()
{
  try {
      cout << "5/2 =" << osztas(2) << endl;
      cout << "5/0 =" << osztas(0) << endl;
  } catch (const char *p) {
     cout << p << endl;
    }
}
```

Hiba/kivétel észlelése

A kivételt azonosító érték eldobása.

Típus azonosít (köt össze).

Felügyelt szakasz. Ennek a működése során fordulhat elő a kivételes eset.

Kivétel elkapása és kezelése.

Kivételkezelés a memóriára

```
#include <iostream>
using std::cerr;
int main() {
  long db = 0;
  try {
    while(true) {
        double *p = new double[1022];  db++;
    }
  } catch (std::bad_alloc) {
    cerr << "Gaz van" << endl;
    // Fel kellene szabadítani, de .... ?
  }
  cerr << "Ennyi new sikerult:"  << db << endl;
}
```

A kivételes eset itt keletkezhet.

Kivétel elkapása és kezelése.

Futtatás egy Linux VM-ben

```
~$ git clone \
    https://git.ik.bme.hu/Prog2/eloadas_peldak/ea_02
~$ cd ea_02/mem_alloc
~$ g++ -static mem_alloc.cpp  -o mem_alloc
~$ ( ulimit -d  500; ./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.
# A sorvégi \ folytatósort jelöl.
```


Futtatás egy Linux VM-ben /2

Az kóddal a new_handler működése is tesztelhető. Ehhez a fordításkor definiálni kell a HANDLER azonosítót:

```
~$ g++ -static mem_alloc.cpp  \
    –DHANDLER  -o mem_alloc
~$ ( ulimit -d 500; ./mem_alloc )
```