

*Programozás alapjai 2.*  
*Hibakeresés, tesztelés*  
*madártávlatból*

Szeberényi Imre

BME IIT

<szebi@iit.bme.hu>



MUEGYETEM 1782

# *Hibakeresés módszerei*

---

## Álmódszerek:

- Erősen nézzük a kódot
- Másokat hibáztatunk
- Jó ötletnek tűnt ...

## Valódi módszerek:

- Nyomkövető kiírások
- Debugger használata
- Trace generátor (automatikus)
- Kód analízátor használata

# Nyomkövető kiírások

Tanuláskor, egyszerű programok estén:

1. Nyomkövető kiírások a kódban

```
#ifdef DEBUG
std::cout << "Kakukk << std::endl;
#endif
```

2. Saját eszközök, makrók, sablonok

```
#ifdef DEBUG
#define BAJ_VAN_HA_EZ_IGAZ(e) if ((e)) { \
std::cout << "" << #e << " Kifejezes IGAZ ebben a sorban: " \
<< __LINE__ << std::endl; }
#endif
```

# Tanuláshoz

---

Utasítások eredménye azonnal:

- C++ interpreter (pl. root, cling)
  - <https://root.cern.ch/cling-brief> Pl:  
[cling]\$ #include <iostream>  
[cling]\$ std::cout << 100 << std::endl;  
100
- Trükkös makrók, sablonok, amikkel kiíratható az utasítás és az eredménye is

## *\_\_makró és társai*

---

Nem kell megérteni. Szabad használni.

```
#define _(...) \  
std::cout << #__VA_ARGS__";" << "// "; __VA_ARGS__  
  
_(std::cout << 100 << std::endl);  
std::cout << 100 << std::endl; // 100
```

Ezt használjuk a mintapéldákban is.

# *Tesztelési követelmények*

---

- Legyen független, és megismételhető
- Legyen áttekinthető és tükrözze a tesztelt kód struktúráját.
- Legyen hordozható és újrafelhasználható.
- Segítsen a teszt írójának a problémára koncentrálni.
- Legyen gyors és automatizálható.
- Gyakran a forrás kód tárolójába (pl. git) való betétel feltétele az ún. unit teszt sikeressége.

# *Google Test*

---

- Kis méretű, forráskódban elérhető  
<http://code.google.com/p/googletest/>
- Platformfüggetlen (WinX, MAC, LINUX, Windows Mobile, MinGW, ...)
- Assertion – alapú
  - success, nonfatal, fatal
- Teszt program:
  - teszt esetek
    - tesztek

# Assertion

- Hasonlító függvényeket hívnak
  - Hiba esetén kiírják a hiba helyét, kódját
- ASSERT\_\*
  - fatális hiba – a program megáll
- EXPECT\_\*
  - nem fatális hiba – tovább fut

```
ASSERT_EQ(4, 2*2) << "2*2 hiba";  
for (int i = 0; i < 10; i++) {  
    EXPECT_LT(i-1, 2*i) << "i nem kisebb mint 2*i! i=" << i;  
}
```



# Egyszerű feltételek

Utasítás	Teljesülnie kell
EXPECT_EQ( <i>expected</i> , <i>actual</i> )	<i>expected</i> == <i>actual</i>
EXPECT_NE( <i>val1</i> , <i>val2</i> )	<i>val1</i> != <i>val2</i>
EXPECT_LT( <i>val1</i> , <i>val2</i> )	<i>val1</i> < <i>val2</i>
EXPECT_LE( <i>val1</i> , <i>val2</i> )	<i>val1</i> <= <i>val2</i>
EXPECT_GT( <i>val1</i> , <i>val2</i> )	<i>val1</i> > <i>val2</i>
EXPECT_GE( <i>val1</i> , <i>val2</i> )	<i>val1</i> >= <i>val2</i>
EXPECT_STREQ( <i>exp_str</i> , <i>act_str</i> )	<i>a két C string azonos</i>
EXPECT_STRNE( <i>str1</i> , <i>str2</i> );	<i>a két C string nem azonos</i>
EXPECT_STRCASEEQ( <i>exp</i> , <i>act</i> );	<i>a két C string azonos (kis/nagy betű az.)</i>
EXPECT_STRCASENE( <i>str1</i> , <i>str2</i> )	<i>a két C string nem azonos (kis/nagy b.)</i>

# *Egyszerű példa*

```
#include "gtest/gtest.h"
#include "sablonjaim.hpp"    // mai ora sablonjai

TEST(Sablonjaim, maxInt) {
    EXPECT_EQ(20, max(3,20)) << "maximum nem OK!";
}

TEST(Sablonjaim, maxDouble) {
    EXPECT_EQ(4.2, max(4.2, 3.1));
}

// A main automatikusan hozzáadódik
```

# *Problémák a HSZK-ban*

- VS ingyenes változatával nem fordul (tuple par.)
- CB-vel rendben, van de érzékeny a jó lib-re.
  - include
  - win32
    - gtest.lib, gtest\_main-mdd.lib (Visual Studio-hoz)
    - libgtestd.a, libgtest\_maind.a (CodeBloks-hoz)
- További beállításokra lehet szükség:
  - Projektfájlban a megfelelő útnév beállítása:
    - CB: Project -> Build options -> Linker settings, Search directories
    - VS: Project-> Properties -> C++ -> General (additional dir),  
Project-> Properties -> linker -> Input (Additional dep.)

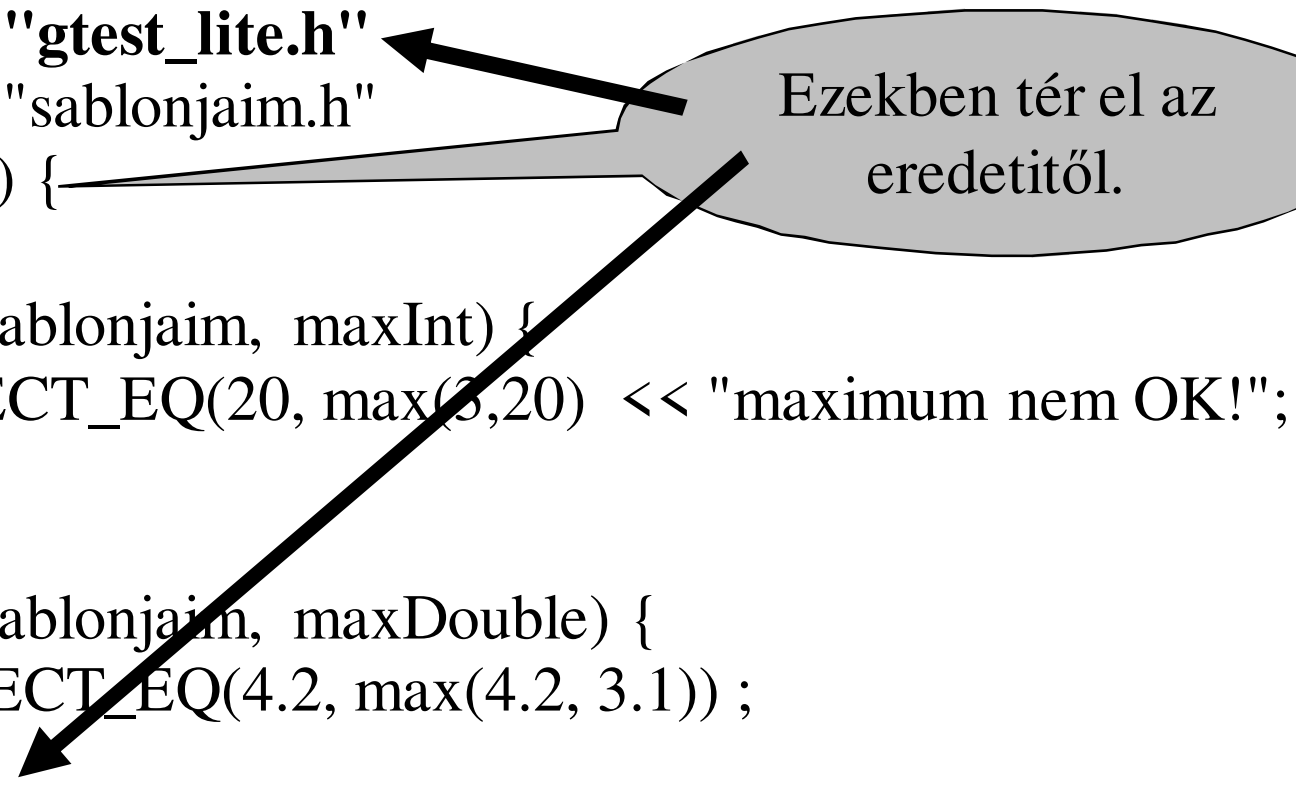
# *gtest\_lite*

---

- A prog2 tárgyhoz készült.
- Lényegében a gtest fontosabb lehetőségeit valósítja meg kompatibilis módon.
- Ronda makrókkal és statikus objektummal operál.
- HF-ben használható ill. **használendő**.
- Csak egyszerű tesztek és
- csak az EXPECT\_\* alakú ellenőrzéseket támogat.
- Kivételek ellenőrzését is lehetővé teszi.

# *gtest\_lite példa*

```
#include "gtest_lite.h"  
#include "sablonjaim.h"  
int main() {  
  
    TEST(Sablonjaim, maxInt) {  
        EXPECT_EQ(20, max(3,20)) << "maximum nem OK!";  
    } END  
  
    TEST(Sablonjaim, maxDouble) {  
        EXPECT_EQ(4.2, max(4.2, 3.1));  
    } END  
  
}
```



Ezekben tér el az eredetitől.

# További ellenőrzések

Utasítás	Teljesülnie kell
EXPECT_THROW( <i>statement</i> , <i>excep_type</i> )	adott típust <i>dobnia kell</i>
EXPECT_ANY_THROW( <i>statement</i> )	<i>bármit kell dobnia</i>
EXPECT_NO_THROW( <i>statement</i> )	<i>nem dobhat</i>
EXPECT_PRED_FORMAT1( <i>pred</i> , <i>val1</i> )	<i>pred(val1) == true</i>
EXPECT_PRED_FORMAT2( <i>pred</i> , <i>val1</i> , <i>val2</i> )	<i>pred(val1, val2) == true</i>
EXPECT_FLOAT_EQ( <i>expected</i> , <i>actual</i> )	<i>expected ~= actual</i>
EXPECT_DOUBLE_EQ( <i>expected</i> , <i>actual</i> )	<i>expected ~= actual</i>
EXPECT_NEAR( <i>val1</i> , <i>val2</i> , <i>abs_error</i> )	<i>abs(val1-val2) &lt;= abs_error</i>
SUCCEED()	siker
FAIL(); ADD_FAILURE()	végzetes; nem végzetes
ADD_FAILURE_AT("file_path", line_number);	nem végzetes


# *Memória kezelés problémái*

---

- Ritkán futó programágakon gyakran:
  - hibás pointer kezelés
  - felszabadítatlan terület
- Hosszan futó program felzabálja a memóriát
  - gép lelassul, mert a tárat diszkre írja (swap)
- Hibás inputra
  - túlcímzés, buffer overflow hiba

# Buffer overflow példa

```
...
char buff[6];
bool pass = false;
int main(void) {
    cout << "Jelszo: ";
    cin >> buff;
    if (strcmp(buff, "titok") != 0) {
        cout << "*** Hibas jelszo ***" << endl;
    } else {
        cout << "Helyes jelszo" << endl;
        pass = true;
    }
    if (pass) {
        cout <<" >> Hozzaferhet a titokhoz:" << endl;
        printsecret(); // kiírja a titkot
    }
}
```



hosszabb inputra  
felülíródik



# *Buffer overflow példa futtatása*

```
$ ./buffer
```

```
Jelszo: qwqw
```

```
*** Hibas jelszo ***
```

```
$ ./buffer
```

```
Jelszo: wqwqwqwqw
```

```
*** Hibas jelszo ***
```

>> Hozzaferhet a titokhoz:

1. Mikor jelentkezik a buffer overflow hiba?

a) reggel

b) aritmetikai túlcsordulásakor

c) ha elfogy a memória

d) egy változót (buffert) a program

hosszabban ír, mint annak a mérete

<https://git.iik.bme.hu/Prog2/>

eloadas\_peldak/ea\_teszt -> buffer

# *Mem. kezelés ellenőrzése*

- Profi eszközök
  - Fejlesztő környezet által támogatott eszközök
    - cppcheck <http://cppcheck.sourceforge.net/>
    - clang-tidy <http://clang.llvm.org/extra/clang-tidy/>
    - google sanitizers <https://github.com/google/sanitizers/wiki>
  - Külső eszközök
    - Valgrind <http://valgrind.org/>
  - Külső könyvtárak
    - duma <http://duma.sourceforge.net/>
    - memtrace

# Statikus analízis példa

```
1 // file: errors.cpp
2 #include <iostream>
3 const int L = 30;
4 char* fn1(int a, int) {
5     char *p = new char[L];
6     #ifdef ALADAR
7         p[L] = 1;
8         delete p;
9     #else
10        delete[] p;
11    #endif
12        return p;
13    }
14
15
16 int main() {
17     std::cout << *fn1(30, 4) << std::endl;
18 }
```

```
cppcheck errors.cpp
[errors.cpp:12]: (error) Returning/dereferencing 'p'
after it is deallocated / released
Checking errors.cpp: ALADAR...
[errors.cpp:7]: (error) Array 'p[30]' accessed at
index 30, which is out of bounds.
[errors.cpp:8]: (error) Mismatching allocation and
deallocation: p
```

# *Futás közbeni analízis*

```
//file: errors.cpp  
g++ -ggdb -fsanitize=address -fno-omit-frame-pointer  
errors.cpp  
./a.out
```

```
==3884==ERROR: AddressSanitizer: heap-use-after-free ...
```

```
#0 0x7f13cd01ecaa in operator delete[](void*) errors.cpp:10
```

```
...
```

```
#0 0x7f13cd01e6b2 in operator new[](unsigned long) errors.cpp:5
```

```
SUMMARY: AddressSanitizer: heap-use-after-free errors.cpp:17 main
```

# *Futás közbeni analízis /2*

**SUMMARY: AddressSanitizer: heap-use-after-free errors.cpp:17 main**

**Shadow bytes around the buggy address:**

```
0x0c067fff9da0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c067fff9db0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c067fff9dc0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c067fff9dd0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c067fff9de0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x0c067fff9df0: fa fa fa fa fa fa fa fa fa fa fa fa fa[fd]fd fd fd
0x0c067fff9e00: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c067fff9e10: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c067fff9e20: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c067fff9e30: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c067fff9e40: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
```

**Shadow byte legend (one shadow byte represents 8 application bytes):**

**Jelzi az illegális hozzáférés helyét a memóriában.**

# *memtrace*

---

- Preprocesszor segítségével lecseréli a new, és delete hívásokat.
- Így ellenőrizni tudja azok használatát.
- Kanari byte-ok elhelyezésével felszabadításkor ellenőrzi a túlcímzést.
- Program megálláskor az ellenőrzi, hogy minden fel lett-e szabadítva.

# *memtrace és C++11*

---

- C++11-től a delete kulcsszó nem csak operátorként jelenik meg a nyelvben, hanem osztály deklarációkban a tagfüggvény törzse helyett is szerepelhet. Itt azonban nem szabad lecserélni ....
- A standard header-ekben gyakran előfordul.
- Alkalmas include sorrenddel a probléma legtöbbször megkerülhető.

# *memtrace::mem\_dump*

---

- Adott címtől kezdődően megadott hosszúsággal kiírja a memória tartalmát a képernyőre.

Pl:

Dump: (addr: 0028FE7E)

0000:\* 73 73 73 7a 69 61 20 43 2b 2b 2b 2b 2b 20 2b 20 ssszia C+++++ +