C++

A sötét oldal

Linus Torvalds

C++ is a horrible language. It's made more horrible by the fact that a lot of substandard programmers use it, to the point where it's much much easier to generate total and utter crap with it.

Bjarne Stroustrup

C makes it easy to shoot yourself in the foot.

C++ makes it harder, but when you do, you blow away your whole leg!

Bjarne Stroustrup, The C++ reference manual, section 12.4.

The body of a destructor is executed before the destructors for member objects. Destructors for nonstatic member objects are executed before the destructors for base classes. Destructors for nonvirtual base classes are executed before destructors for virtual base classes. Destructors for nonvirtual base classes are executed in reverse order of their declaration in the derived class. Destructors for virtual base classes are executed in the reverse order of their appearance in a depth-first left-to-right traversal of the directed acyclic graph of base classes; "left-to-right" is the order of appearance of the base class names in the declaration of the derived class.

FőEllenségek

1. A varázsló

2. A lusta

3. A nyelv

1. A varázsló

"Ez benne van a c++ szabványban, ezt mindenkinek értenie kell!"

"Hogy miért írtam ilyen bonyolultra? Ez művészet!"

"Ha mindent 3-szor define-olok, akkor sokkal gyorsabb írni és kifejezőbb lesz a kód"

Valódi példa:

```
#define DROP delete
...
#define OBJDROP DROP
...
...
...
OBJDROP(valami);
```

Aztán ha hiba van lehet keresni, hogy ez mit csinál

Másik példa

"Az előző munkahelyen ezeket a makrókat használtuk. Bevezettem itt is, lehet, hogy elsőre nem értitek, de higgyétek el, hogy tök jó"

```
#define foreach(iter, container) ...
//...
foreach( user, users) {
   if (user->isValid) {
   ....
   }
}
```

#define gondok (még 1x)

- Mindent átír
- Nem veszi figyelembe a namespace-t
- Ha hiba van nehéz kidebuggolni
- Évtizedek múlva is szidhatnak érte
  - pl. GetMessage, GetUserName, GetVersion, és még sok minden MS Windowson

Speciális nyelvi elemek, amik alig ismertek

```
char* calculate (char word<::>)  <%
    while (not compl *word++)<%%>
    return word;
%>

int main() {
    char word[] = "asdads";
    calculate(word);
}
```

(Lényegtelen mit csinál a program, a szintaxis a lényeg)

Speciális nyelvi elemek, amik alig ismertek

```
BYTE* buffer = (BYTE*) calloc(1024, sizeof
BYTE);

buffer[0]='a';
buffer[1]='b';
buffer[2]='c';

strlen(buffer) ?
```

Speciális nyelvi elemek, amik alig ismertek

```
BYTE* buffer = (BYTE*) calloc(1024, sizeof BYTE);

buffer[0]='a';
buffer[1]='b';
buffer[2]='c';

strlen(buffer) ?
```

3

A calloc kinullázza a memóriát. Legalább kommenttel jelezzük!

Speciális nyelvi elemek, amik alig ismertek

```
struct B {
   int b;
   B::B() try : b(f()) {
   } catch(int e) {
   }
};
```

És ez mit csinál?

Felesleges hackelés

Pont jól jönne az a privát változó abban a libben. Néha muszáj.

```
#define class struct
#define private public
#define protected public

#include <library.h>

#undef class
#undef private
#undef protected
```

Template metaprograming

"Milyen jó, hogy fordítási időben ki tudjuk számolni X-et, ahelyett, hogy odaírnánk konstansként."

Ha sok összefüggő paraméterünk van, amiket ki lehet számolni egymásból, akkor inkább generáljunk egy headert pl. egy scripttel.

Korai optimalizálás

*"We should forget about **small** efficiencies, say about 97% of the time: premature optimization is the root of all evil"* *Donald Knuth*

1. Legyen jó az algoritmus

2. Az aprólékos gyorsítás ott fontos, ahol sokat fut a program.

Korai optimalizálás

```
void peerConnected(const QWORD& peerId)
```

???

Korai optimalizálás

"Inkább referenciaként adom át ezt a QWORD-öt, mert akkor 32 bites gépeken nyerek a stacken 4 bájtot"

```
void peerConnected(const QWORD& peerId)
```

Felesleges

CPU cache

Korai optimalizálás

"A normális new meg delete nagyon lassú, ezért írtam egy saját memory poolt placement new-val, de te erről úgysem hallottál még"

Korai optimalizálás

"A normális new meg delete nagyon lassú, ezért írtam egy saját memory poolt placement new-val, de te erről úgysem hallottál még"

```
User* newuser  = lefoglaltunk neki előre memóriát
new (newuser) User(); // constructor
// ...
newUser->~User(); // Kézzel kell destruktort hívni
```

Korai optimalizálás

"Tudom, hogy ide tehettem volna egy mutexet, de az drága és **úgysem állhat elő** olyan helyzet, hogy szükség legyen rá"

És erre refactor közben is mindenki emlékezni fog?

Korai optimalizálás

"Az ideiglenes változók drágák, úgyhogy mindent bezsúfolok egy sorba"

```
userManager.addUser( new User(userManager.
getNextUserId(prevUser)), UserManager::
createDefaultUserConfig(), getUserName() + "
Béla" );
```

Megcsinálom, mert ez így izgalmasabb

```
int main() {
  int a = 1, b = 2;
  print "this is a test";
  print "the sum of", a, "and", b, "is",a+b;
  return 0;
}
```

EZMI? Lehet ilyet?

Megcsinálom, mert ez így izgalmasabb

```
#define print __hidden__::print(),

namespace __hidden__ {
   void print() ...
}
```

Vessző operatort is át lehet írni.

Megcsinálom, mert ez így izgalmasabb

"Az igaz, hogy van rá X db kész library, de én inkább megírom újra, mert macerás átlátni a libraryket."

jó és rossz tapasztalat is van ezzel

Megcsinálom, mert ez így izgalmasabb

```
class OwnContainer<class T> {
   // Rubyban is így van és ez így tök jó,
és mindenki így fogja használni, mert nekem
tetszik
   void operator<<(T element) {
      // Add the element
   }
};
```

Megcsinálom, mert ez így izgalmasabb

```
class Codepage {
   int m_CodepageId;
   // Nem írok gettert, mert a C++ így
átalakítja nekem mikor számként szeretném
használni.
   operator int() {
      return 0x1000 + m_CodepageId;
   }
   Codepage(int codepage)
      :m_CodepageId(codepage)
};
```

2. A lusta

"Bocs, elfelejtettem"

"Az oda nem is kell, lefordul nélküle is"

"Ha úgy írnám lassabb lenne/több memóriát enne"

2. A lusta

"Nem kezdek új eljárást írni, mert..."

- Nem használom csak innen
- Az eljáráshívás drága
- Csak pár sort kell még hozzáírni, eszembe sem jutott

A lusta

"Nem kezdek új eljárást írni, mert..."

- Nem használom csak innen Attól még átláthatatlan
- Az eljáráshívás drága És tényleg olyan sokszor le fog futni?
- Csak pár sort kell még hozzáírni, eszembe sem jutott

A lusta

"Nem hozok létre új változót ennek az értéknek, hanem használok egy meglévőt, mert..."

- Már úgysem használom az első értelemben
- Drága
- Ilyen apróság miatt nem akartam még egy változót felvenni

Rövid változónevek

Konkrét pl:

```
main()
{   long int   db, i, j, k;
 float   a,b,c,dd,ee,ff;
```

Nem, nem egyértelmű

Nem, te sem fogod érteni 2 hét múlva

Nem, nincs olyan, hogy ideiglenes kód

```
try {
   //...
} catch (...) {
   // TODO: log this
}
```

Elkapok minden exceptiont. Odaírom a TODO-t, hogy el ne felejtsük.

Side effectek

```
int getVisitorCounter() {
   refreshUI();
   updateConnections();
   return m_visitorCounter;
}
```

"Amikor ezt a gettert hívják, akkor úgyis mindig ugyanazokat a műveleteket kell elvégezni, úgyhogy beleírom a getterbe"

Side effectek

```
int updateVisitors() {
   refreshUI();
   updateConnections();
   return getVisitorCounter();
}

int getVisitorCounter() {
   return m_visitorCounter;
}
```

Főnix singletonok

Kilépéskor lehet gond:

A singletonok leállító függvényei más singletonokra hivatkoznak.

Leálláskor mindig újra létrehozzák egymást.
Sose fog kilépni a program.

Logolásban művelet

```
newId = oldId + 2;
```

Logolásban művelet

Ki kéne logolni:

```
log("new id: %d", newId = oldId + 2);
```

Logolásban művelet

Már nem kell a log:

```
//log("new id: %d", newId = oldId + 2);
```

Jön a csodálkozás, hogy miért nem változik a newId

Utasítás közben ++

```
{
   doSomething(counter);
   doOtherThing(counter);
   int x = counter * 2;
   foo(counter++);
   something(x);
   thirdthing(x);
}
```

Hol nő a counter?

Utasítás közben ++

```
{
   doSomething(counter);
   doOtherThing(counter);
   int x = counter * 2;
   foo(counter++);
   something(x);
   thirdthing(x);
   oneMoreThing(counter);
}
```

Leszármazott osztályban virtual kihagyása

```
class Base {
  virtual void stuff();
}

class Derivate : public Base {
  void stuff();
}
```

Leszármazott osztályban virtual kihagyása

A Derivate::Stuff virtuális, de nem látszik.

Ha egy leszármazottban felül akarom definiálni, akkor végig kell néznem a teljes leszármazási fát

{} kihagyása

```
if (a < b*2)
   aTooSmall = true;
```

{} kihagyása

```
if (a < b*2)
   log("a is too small");
   aTooSmall = true;
```

Miért nem lesz aTooSmall sose true?

Pedig még a logot is kiírja!

implicit konverziók használata

```
int counter = 0;
//...
if (counter) {
}
```

Vagy:

```
my_bool check() {
   return memcmp(hash1, hash2, HASH_SIZE);
}
```

implicit konverziók használata

```
int counter = 0;
//...
if (counter == 0) { // Olvashatóbb
}
```

Vagy:

```
my_bool check() {
   return memcmp(hash1, hash2, HASH_SIZE) ==
0;
}                              //konkrét hiba volt
```

Kommentek kihagyása

```
void doJob(void* arg0, int type) {
   // 600 sor
}
```

"Nem írok kommentet, akit érdekel a működése megnézi a kódot"

constok kihagyása

Lásd előző alkalommal

Változónév rövidítés

IDE előtti időkből berögzülés

```
void chkCfgParms(CfgParams* pPrms)
{
}
```

Sokszor nem konzisztens a rövidítés

Esélyed nincs rákeresni

Autocomplete-et nehezíti

Osztályok funkció toldozása

```
class PeerModel {
};
```

"Mivel a peer összes információja benne van, tegyük bele még a hálózati logikát is."

"Mivel már benne van a hálózati működés is tegyük bele a UI cuccokat is"

3. A nyelv

```
User checkUser(User user) {
   User tempUser = user;
   return tempUser;
}
```

Hány függvényhívás történik?

virtuális destruktorok

```
struct Base {
   ~Base(){cout << "Hello" << endl;}
}

struct Child : public Base {
   ~Child(){cout << "Bello" <<endl;}
}
int main() {
   Child child;
}
```

Mi íródik ki?

virtuális destruktorok

```
struct Base {
   virtual ~Base(){cout << "Hello" << endl;}
}

struct Child : public Base {
   virtual ~Child(){cout << "Bello" << endl;}
}
int main() {
   Child child;
}
```

Mi íródik ki?

Függvényhívás nullptr-n

```
class Summarizer {
    void Add(int a) {
        m_sum += a;
    }
    int Add(int a, int b) { // static
        return a + b;
    }
    int m_sum;
}
Summarizer* sum1 = nullptr;

sum1->Add(3);     // Crash
sum1->Add(1, 2); // OK!
```

Referencia visszaadás

```
User& getUser() {
    return m_user;
}
```

Eddig ok

Referencia visszaadás

Az igény megváltozik, de lusta refaktorolni

```
User& getUser() {
    User copy = m_user;
    return copy;
}
```

Ha kilép a scope-ból el fog veszni az objektum

Változó inicializáció sorrend

```
class Numbers {
   int bigNumber;
   int smallNumber;
   Numbers();
};

Numbers::Numbers()
   : smallNumber(1)
   , bigNumber(1000) {
}
```

Változó inicializáció sorrend

```
class Numbers {
   int bigNumber;
   int smallNumber;
   Numbers();
};

Numbers::Numbers()
   : bigNumber(1000)
   , smallNumber(1) {
}
```

Globális változók, singletonok

- Nehéz konkrét példát hozni
- Spagetti kódot eredményez. Össze fognak nőni a komponensek, és utána macerás lesz szétváálasztani.
- Többszálúság.
- Hibakeresés. (50 helyről hívódik, melyik okozta a konkrét hibát?)
- Meddig és mikortól él egy ilyen objektum?

Hungarian notation

A változó típusát beleírjuk a változó nevébe
Van aki szereti (és rutinból használja).

Szerintem nehezen olvasható lesz tőle a kód.
Hosszúak lesznek a változónevek -> elégnek tűnik
nehezíti az IDE-k autokiegészítését
Ha változik a típus, nevet is kell változtatni.

De, ha konzisztensen van használva segíthet

Hungarian notation

A változó típusát beleírjuk a változó nevébe
Van aki szereti (és rutinból használja).

konkrét rossz pl:
m_pdwRC

7 karakter hosszú

Hungarian notation

A változó típusát beleírjuk a változó nevébe
Van aki szereti (és rutinból használja).

konkrét rossz pl:
m_pdwRC

Member

Hungarian notation

A változó típusát beleírjuk a változó nevébe
Van aki szereti (és rutinból használja).

konkrét rossz pl:
m_pdwRC

pointer valamire

Hungarian notation

A változó típusát beleírjuk a változó nevébe
Van aki szereti (és rutinból használja).

konkrét rossz pl:
m_pdwRC

Double Wordre pointer (4 bájtos unsigned int)

Hungarian notation

A változó típusát beleírjuk a változó nevébe
Van aki szereti (és rutinból használja).

konkrét rossz pl:
m_pdwRC

???
Itt kéne a lényegnek lennie!

Hungarian notation

Még példa:

bOk

Most ez bájt, vagy boolean?

uSize

Hordoz bármiféle plusz infót, hogy a méret csak pozitív lehet?

Biztonság

Buffer overflow

Leggyakoribb hiba

```
char id[4];
```

Ha sikerült az id-t külső adattal felülírni, akkor átírható a memória más része is, akár felülírható a régi instuction pointer a stacken

Oda kerül a vezérlés, ahova én akarom. Az fut le amit én akarok.

Buffer overflow

Leggyakoribb hiba

```
char id[4];
```

"Á, a szervert is mi írjuk, úgysem fog 4 karakternél hosszabb ID-t küldeni"

Buffer overflow

Leggyakoribb hiba

```
char id[4];
...
strcpy(id, idFromAnOtherString);
```

Nagyon nagyon rossz.
Egyáltalán nincs határ ellenőrzés

Buffer overflow

Leggyakoribb hiba

```
char id[4];
...
strncpy(id, idFromAnOtherString, 4);
```

Sokkal jobb. Mi van, ha a célterület méretét meg akarjuk változtatni.

Buffer overflow

Leggyakoribb hiba

```
char id[4];
...
strncpy(id, idFromAnOtherString, sizeof id);
```

Tökéletes

printf

```
string kulsoUserInputBol = ...;

printf(kulsoUserInputBol.c_str());
```

printf

string kulsoUserInputBol = ...;

printf(kulsoUserInputBol.c_str());

És mi van ha a külső input ilyen:
%d %n %ul ...

printf

string kulsoUserInputBol = ...;

printf(kulsoUserInputBol.c_str());

És mi van ha a külső input ilyen:
%d %n %ul ...

És ezek eddig csak a programozási hibák voltak.

- Kommunikáció hiánya: Lehet, hogy más megcsinálta, amin éppen dolgozol, vagy épp most csinálja
- Nem kérsz segítséget
- Folyton segítséget kérsz
- ...

Források

http://madebyevan.com/obscure-cpp-features/

http://bazaar.launchpad.net/~maria-captains/maria/5.1/view/3144/sql/password.c

http://en.wikiquote.org/wiki/Donald_Knuth

http://dinodini.wordpress.com/2011/01/12/are-global-variables-really-all-that-bad/