Google C++ style guide

Bodai Richárd

Miskolci Egyetem

Április 3, 2013

Amiről szó lesz...

- Header állományok
- Hatókör
- Osztályok
- Elnevezések

Az útmutató célja

A Google nyílt forrású projektjeinél túlnyomórészt C++:

- hatékony szolgáltatások, összetettség.

A kód könnyen kezelhetetlenné vállhat.

- Cél: komplexitás kezelése, jól menedzselhető kód előállítása.

Az útmutató célja

A Google nyílt forrású projektjeinél túlnyomórészt C++:

- hatékony szolgáltatások, összetettség.

A kód könnyen kezelhetetlenné vállhat.

- Cél: komplexitás kezelése, jól menedzselhető kód előállítása.

Az útmutató célja

A Google nyílt forrású projektjeinél túlnyomórészt C++:

- hatékony szolgáltatások, összetettség.

A kód könnyen kezelhetetlenné vállhat.

- Cél: komplexitás kezelése, jól menedzselhető kód előállítása.

Az útmutató célja

A Google nyílt forrású projektjeinél túlnyomórészt C++:

- hatékony szolgáltatások, összetettség.

A kód könnyen kezelhetetlenné vállhat.

- Cél: komplexitás kezelése, jól menedzselhető kód előállítása.

Az útmutató célja

A Google nyílt forrású projektjeinél túlnyomórészt C++:

- hatékony szolgáltatások, összetettség.

A kód könnyen kezelhetetlenné vállhat.

- Cél: komplexitás kezelése, jól menedzselhető kód előállítása.

Hogyan?

Nagyon fontos az olvashatóság, könnyen érthetőség, amit a következő módon érhetünk el:

- konzisztencia megkövetelése,
- egységes idiómák, minták használata,
- bizonyos "extrém" nyelvi lehetőségek korlátozott használata.

Egyes esetekben indokolt lehet eltérni a megszokott mintáktól, de törekedni kell a szabályok betartására, hogy megőrizzük a kód következetességét.

Hogyan?

Nagyon fontos az olvashatóság, könnyen érthetőség, amit a következő módon érhetünk el:

- konzisztencia megkövetelése,
- egységes idiómák, minták használata,
- bizonyos "extrém" nyelvi lehetőségek korlátozott használata.

Egyes esetekben indokolt lehet eltérni a megszokott mintáktól, de törekedni kell a szabályok betartására, hogy megőrizzük a kód következetességét.

Hogyan?

Nagyon fontos az olvashatóság, könnyen érthetőség, amit a következő módon érhetünk el:

- konzisztencia megkövetelése,
- egységes idiómák, minták használata,
- bizonyos "extrém" nyelvi lehetőségek korlátozott használata.

Egyes esetekben indokolt lehet eltérni a megszokott mintáktól, de törekedni kell a szabályok betartására, hogy megőrizzük a kód következetességét.

Hogyan?

Nagyon fontos az olvashatóság, könnyen érthetőség, amit a következő módon érhetünk el:

- konzisztencia megkövetelése,
- egységes idiómák, minták használata,
- bizonyos "extrém" nyelvi lehetőségek korlátozott használata.

Egyes esetekben indokolt lehet eltérni a megszokott mintáktól, de törekedni kell a szabályok betartására, hogy megőrizzük a kód következetességét.

Hogyan?

Nagyon fontos az olvashatóság, könnyen érthetőség, amit a következő módon érhetünk el:

- konzisztencia megkövetelése,
- egységes idiómák, minták használata,
- bizonyos "extrém" nyelvi lehetőségek korlátozott használata.

Egyes esetekben indokolt lehet eltérni a megszokott mintáktól, de törekedni kell a szabályok betartására, hogy megőrizzük a kód következetességét.

Hogyan?

Nagyon fontos az olvashatóság, könnyen érthetőség, amit a következő módon érhetünk el:

- konzisztencia megkövetelése,
- egységes idiómák, minták használata,
- bizonyos "extrém" nyelvi lehetőségek korlátozott használata.

Egyes esetekben indokolt lehet eltérni a megszokott mintáktól, de törekedni kell a szabályok betartására, hogy megőrizzük a kód következetességét.

Egyéb tudnivaló

Ez az útmutató nem C++ tutorial.

Részletes leírás megtalálható interneten:
http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

Egyéb tudnivaló

Ez az útmutató nem C++ tutorial.
Részletes leírás megtalálható interneten:
http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

Header fájlok

A legtöbb .cc fájlban van hivatkozás .h fájlokra. A header-ek megfelelő használata nagyban hozzájárul a könnyen érthető és jól olvasható kód előállításhoz.

#define Guard

- Minden header fájlban el kell kerülni a többszörös beszúrás lehetőségét.
- Erre szolgálnak a #define guard-ok.
- A szimbólumok formátuma: <PROJECT>_<PATH>_<FILE>_ H_.

#define Guard példa

Példa: foo projekt, foo/src/bar/baz.h útvonal:

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_

...

#endif  // FOO_BAR_BAZ_H_
```

Inline függvények

Inline függvények

Az inline kulcsszó esetén a fordító nem hoz létre függvényt, hanem a megadott kódot minden helyre bemásolja, ahol a látszólagos függvényt meghívjuk.

Csak akkor használjunk inline függvényeket, ha azok elég kicsik (max. 10 sor).

Inline függvények

Vannak függvények, amelyeknél nagyon körültekintően kell eljárnunk:

- destruktorok,
- olyan függvények, amelyek tartalmaznak ciklust vagy switch-t

Szükség esetén a komplexebb inline függvényeket elhelyezhetjük egy -inl.h utótaggal rendelkező .h fájlban.

Inline függvények

Vannak függvények, amelyeknél nagyon körültekintően kell eljárnunk:

- destruktorok,
- olyan függvények, amelyek tartalmaznak ciklust vagy switch-t

Szükség esetén a komplexebb inline függvényeket elhelyezhetjük egy -inl.h utótaggal rendelkező .h fájlban.

Függvény paraméter sorrend

Egy függvény definiálásakor a paraméterek sorrendje: bemenetek, aztán kimenetek.
Persze kivételek lehetnek (pl. egy paraméter kiemenet is és bemenet is).

#include

A projekt header fájljait a szülő jegyzékkel elgyütt kell megadni.

- pl. google-awesome-project/src/base/logging.h ⇒ #include "base/logging.h"

#include

A projekt header fájljait a szülő jegyzékkel elgyütt kell megadni.

- pl. google-awesome-project/src/base/logging.h ⇒ #include "base/logging.h"

#include

A dir/foo.cc vagy dir/foo_test.cc fájlban, ha elsősorban a dir2/foo2.h-t szeretnénk tesztelni, vagy implementálni, akkor az include-ok sorrendje a következő:

1. dir2/foo2.h
2. C könyvtárak
3. C++ könyvtárak
4. egyéb könyvtárak
5. a projekt többi .h fájlja

#include

```
#include "foo/public/fooserver.h"  // Preferred location.

#include <sys/types.h>
#include <unistd.h>
#include <hash_map>
#include <vector>

#include "base/basictypes.h"
#include "base/commandlineflags.h"
#include "foo/public/bar.h"
```

Névterek

Névtér

A fordító a programban használt neveket különböző névtereken (namespace) tárolja. Egy névtérben lévő neveknek egyedieknek kell lenniük, azonban a különböző névtereken azonos néven is szerepelhetnek, azaz a névterek a láthatósági szabályokat teszik könnyebben alkalmazhatóvá. Egy névtérben logikailag összefüggő változókat, függvényeket, típusokat tárolunk. Egy osztály/struktúra egyben a nevével azonos nevű névteret is definiál.

Névterek

Névterek elnevezésekor vegyük alapul a projekt nevét és esetleg az útvonalát. Például két különböző projektünk is tartalmaz egy Foo osztályt a globális hatókörben. Ekkor: project1::Foo és project2::Foo különböznek, nincs ütközés.

Lehetőség szerint kerüljük a globális függvények használatát. Helyette helyezzük el őket egy névtérben, vagy használjunk statikus tagfüggvényeket.

Névterek

Névterek elnevezésekor vegyük alapul a projekt nevét és esetleg az útvonalát. Például két különböző projektünk is tartalmaz egy Foo osztályt a globális hatókörben. Ekkor: project1::Foo és project2::Foo különböznek, nincs ütközés.
Lehetőség szerint kerüljük a globális függvények használatát. Helyette helyezzük el őket egy névtérben, vagy használjunk statikus tagfüggvényeket.

Lokális változók

A függvények változóit a lehető legszűkebb hatókörben helyezzük el, és inicializáljuk őket a deklarációban.
Példa:

```
int i;
i = f();      // Bad -- initialization separate from declaration.
int j = g();  // Good -- declaration has initialization.
```

Lokális változók

Ez alól egy kivétel van: ha ugyan is egy cikluson belül hozunk létre egy objektumot, akkor annak konstruktora minden egyes belépés után lefut.
Példa:

```
// Inefficient implementation:
for (int i = 0; i < 1000000; ++i) {
  Foo f;  // My ctor and dtor get called 1000000 times each.
  f.DoSomething(i);
}

Foo f;  // My ctor and dtor get called once each.
for (int i = 0; i < 1000000; ++i) {
  f.DoSomething(i);
}
```

Konstruktor

A konstruktorban inicializálni szoktuk egy objektum adattagjait. Azonban óvakodnunk kell a komplex inicializációktól, melyek hibákat relythetnek magukban, mert a konstruktorok nagyon nehezen ellenőrizhetők.

- Konstruktorban sose hívjunk meg virtuális metódusokat.
- Ha egy objektum nem-triviális inicializációt igényel, akkor javasolt egy Init() metódus használata.

Konstruktor

A konstruktorban inicializálni szoktuk egy objektum adattagjait. Azonban óvakodnunk kell a komplex inicializációktól, melyek hibákat relythetnek magukban, mert a konstruktorok nagyon nehezen ellenőrizhetők.

- Konstruktorban sose hívjunk meg virtuális metódusokat.
- Ha egy objektum nem-triviális inicializációt igényel, akkor javasolt egy Init() metódus használata.

Konstruktor

- Mindig hozzunk létre default konstruktort, máskülönben a fordító csinálja meg helyettünk, rosszul.
- Használjuk az explicit kulcsszót az egy paraméteres konstruktorok esetén, így elkerülhetjük a nemkívánatos konverziót.

Konstruktor

- Mindig hozzunk létre default konstruktort, máskülönben a fordító csinálja meg helyettünk, rosszul.
- Használjuk az explicit kulcsszót az egy paraméteres konstruktorok esetén, így elkerülhetjük a nemkívánatos konverziót.

Másoló konstruktor

Csak ha feltétlen szükséges hozzunk létre másoló konstruktort, illetve hozzárendelő operátort. Ellenkező esetben "tiltsuk meg" a használatát (DISALLOW_COPY_AND_ASSIGN). Egy Clone(), CopyFrom() vagy hasonló metódussal biztosíthatjuk az objektumok másolhatóságát, elkerülve az implicit függvényhívásokat.

Másoló konstruktor

Példa DISALLOW_COPY_AND_ASSIGN makróra:

```
// A macro to disallow the copy constructor and operator= functions
// This should be used in the private: declarations for a class
#define DISALLOW_COPY_AND_ASSIGN(TypeName) \
  TypeName(const TypeName&);               \
  void operator=(const TypeName&)
```

Aztán a Foo osztályban:

```
class Foo {
 public:
  Foo(int f);
  ~Foo();

 private:
  DISALLOW_COPY_AND_ASSIGN(Foo);
};
```

Struktúra VS osztály

Struktúrát (struct) legfeljebb csak passzív obejktumok esetén használjunk (adatok tárolása). Minden más esetben osztályt használjunk.

Kompozíció VS öröklődés

A kompozíció használata gyakran megfelelőbb, mint az öröklődés.
Ha öröklődést használunk, akkor használjuk a public kulcsszót.

- Ne vigyük túlzásba az implementációs leszármaztatást.
- Legyen virtuális a destruktor, amennyiben szükséges. Ha már csak egy virtuális metódusunk is van, akkor mindenképp használjunk virtuális destruktort.
- Amikor felüldefiniálunk egy virtuális tagfüggvényt egy leszármazott osztályban, mindig írjuk a függvény neve elé a virtual kuécsszót, a könnyebb olvashatóság érdekében.
- Óvatosan kezeljük a többszörös öröklődést. Használatuk csak akkor megengedett, ha az ősosztályok közül legfeljebb csak egy tartalmaz implementációt, a többi csak pure interface lehet.

Kompozíció VS öröklődés

A kompozíció használata gyakran megfelelőbb, mint az öröklődés. Ha öröklődést használunk, akkor használjuk a public kulcsszót.

- Ne vigyük túlzásba az implementációs leszármaztatást.
- Legyen virtuális a destruktor, amennyiben szükséges. Ha már csak egy virtuális metódusunk is van, akkor mindenképp használjunk virtuális destruktort.
- Amikor felüldefiniálunk egy virtuális tagfüggvényt egy leszármazott osztályban, mindig írjuk a függvény neve elé a virtual kuécsszót, a könnyebb olvashatóság érdekében.
- Óvatosan kezeljük a többszörös öröklődést. Használatuk csak akkor megengedett, ha az ősosztályok közül legfeljebb csak egy tartalmaz implementációt, a többi csak pure interface lehet.

Kompozíció VS öröklődés

A kompozíció használata gyakran megfelelőbb, mint az öröklődés. Ha öröklődést használunk, akkor használjuk a public kulcsszót.

- Ne vigyük túlzásba az implementációs leszármaztatást.
- Legyen virtuális a destruktor, amennyiben szükséges. Ha már csak egy virtuális metódusunk is van, akkor mindenképp használjunk virtuális destruktort.
- Amikor felüldefiniálunk egy virtuális tagfüggvényt egy leszármazott osztályban, mindig írjuk a függvény neve elé a virtual kuécsszót, a könnyebb olvashatóság érdekében.
- Óvatosan kezeljük a többszörös öröklődést. Használatuk csak akkor megengedett, ha az ősosztályok közül legfeljebb csak egy tartalmaz implementációt, a többi csak pure interface lehet.

Kompozíció VS öröklődés

A kompozíció használata gyakran megfelelőbb, mint az öröklődés. Ha öröklődést használunk, akkor használjuk a public kulcsszót.

- Ne vigyük túlzásba az implementációs leszármaztatást.
- Legyen virtuális a destruktor, amennyiben szükséges. Ha már csak egy virtuális metódusunk is van, akkor mindenképp használjunk virtuális destruktort.
- Amikor felüldefiniálunk egy virtuális tagfüggvényt egy leszármazott osztályban, mindig írjuk a függvény neve elé a virtual kuécsszót, a könnyebb olvashatóság érdekében.
- Óvatosan kezeljük a többszörös öröklődést. Használatuk csak akkor megengedett, ha az ősosztályok közül legfeljebb csak egy tartalmaz implementációt, a többi csak pure interface lehet.

Interface

Egy osztály pure interface, ha:

- ha csak public pure virtual ("= 0") metódusai és statikus metódusai vannak,
- ha nincs nem-statikus adattagja,
- ha nincs szükség konstruktort létrehozni hozzá, de ha van is konstruktora, akkor nincs egyetlen paramétere sem,
- ha legfeljebb csak olyan szülő osztályai vannak, amelyekre ugyanezek igazak.

Interface

Egy osztály pure interface, ha:

- ha csak public pure virtual ("= 0") metódusai és statikus metódusai vannak,
- ha nincs nem-statikus adattagja,
- ha nincs szükség konstruktort létrehozni hozzá, de ha van is konstruktora, akkor nincs egyetlen paramétere sem,
- ha legfeljebb csak olyan szülő osztályai vannak, amelyekre ugyanezek igazak.

Interface

Egy osztály pure interface, ha:

- ha csak public pure virtual ("= 0") metódusai és statikus metódusai vannak,
- ha nincs nem-statikus adattagja,
- ha nincs szükség konstruktort létrehozni hozzá, de ha van is konstruktora, akkor nincs egyetlen paramétere sem,
- ha legfeljebb csak olyan szülő osztályai vannak, amelyekre ugyanezek igazak.

Interface

Egy osztály pure interface, ha:

- ha csak public pure virtual ("= 0") metódusai és statikus metódusai vannak,
- ha nincs nem-statikus adattagja,
- ha nincs szükség konstruktort létrehozni hozzá, de ha van is konstruktora, akkor nincs egyetlen paramétere sem,
- ha legfeljebb csak olyan szülő osztályai vannak, amelyekre ugyanezek igazak.

Operátorok

Ha csak lehet, kerüljük az operátor túlterhelést, különösen a hozzárendelés operátorét (operator=).

Fájlnevek

- A fájlneveknek végig kisbetűsnek kell lennie, tartalmazhat '_'-t és '-'-t.
- A C++ fájlok kiterjesztése .cc, míg a header fájloké .h.
- Ne használjunk olyan fájlneveket, amelyek már léteznek a /usr/include-ban.

Fájlnevek

- A fájlneveknek végig kisbetűsnek kell lennie, tartalmazhat '_'-t és '-'-t.
- A C++ fájlok kiterjesztése .cc, míg a header fájloké .h.
- Ne használjunk olyan fájlneveket, amelyek már léteznek a /usr/include-ban.

Fájlnevek

- A fájlneveknek végig kisbetűsnek kell lennie, tartalmazhat '_'-t és '-'-t.
- A C++ fájlok kiterjesztése .cc, míg a header fájloké .h.
- Ne használjunk olyan fájlneveket, amelyek már léteznek a /usr/include-ban.

Típusnevek

A típusok nagybetűvel kezdődnek, illetve minden egyes új szó is nagy betúvel kezdődik. Nincsenek '-' vagy '_'-ok.

```
class UrlTable { ...
class UrlTableTester { ...
struct UrlTableProperties { ...

// typedefs
typedef hash_map<UrlTableProperties *, string> PropertiesMap;

// enums
enum UrlTableErrors { ...
```

Változónevek

A változók végig kisbetűsek, a szavakat '_'-ok kapcsolják össze. Az osztály tagváltozói mindig '_'-sal végződnek.

```
string table_name_;  // OK - underscore at end.
string tablename_;   // OK.
```

Konstansoknál az első betű: k. pl. const int kDaysInAWeek

Thank you for your attention!