Упрезање браве вишенитног програмирања са подацима које штити

Новембар 2014

1 lockstrap

Неколико макроа за чување података и њихове браве, за Ц++11 и новије.

Код вишенитног програмирања, браве (најчешће мутекси) се доста користе. Заправо, много више него што би требало, али то је друга прича.

Проблем са бравама је то што не постоји начин да се брава повеже са неким подацима. Рецимо да имате овакву класу:

```
class User {
            int a;
            float b;
            std::string c;
            std::vector<long> x;

            std::mutex m;
            // ...
    };
```

Брзо: Које податке штити ‘m’?

Не зна се. Можда све, али можда и не. Било би још горе да постоје два мутекса у тој класи. Морате заправо да погледате код да бисте разумели “ко-шта-штити”. А код може да буде хаос. Можете прибегавати коментарима (срећно са тим) или конвенцијама (које брзо измакну контроли).

Али чак и ако остављате добре коментаре или користите разумљиве конвенције, не постоји ништа што ће спречити лоше коришћење - штићење података када не би требало и обрнуто.

Па, ови “lock-strap” макрои обезбеђују начин да се то поправи. Знам, волео бих да нису макрои, али сам поприлично сигуран да од С++11 до С++23 не постоји начин да се они избегну и да се задржи фина синтакса за кориснике.

1.1 Употреба

Користећи пример од малопре:

```
#include "lockstrap.h"
    class User {
            class Data {
                    int a;
                    float b;
                    std::string c;
                    LOCKSTRAP(Data, std::mutex, a,b,c);
            } d;
            std::vector<long> x;
            // ...
    };
```

Дакле, увели смо класу која ће да држи податке које чува мутекс (што је декларисано у 'Data' класи "LOCKSTRAP" макроа). Вектор 'x', сада очигледно, није заштићен мутексом.

С обзиром да је 'Data' класа, логично не можете приступити 'a', 'b' и 'c' споља. "LOCKSTRAP" макро дефинише две помоћне функције чланице за такве сврхе - 'access' и шаблон 'with'.

Овако бисте их користили у неким User функцијама чланице:

```
void User::f()
    {
            auto al = d.access();
            // уовомтренутку , мутексјезакључан ,
                 можетемуприступитиирадити...
            // штагод.
            al.a = 3;
            al.b = al.a / 2;
            // мутексћесеоткључатикада 'al' изађеизопсегаовде
    }


    void User::g()
    {
            // У C++14, 'auto' можедазамени 'User::locker'
            d.with([](User::locker l) {
                    // уовомтренутку , мутексјезакључан ,
                         можетемуприступитиирадити...
                    // штагод.
                    l.a = 33;
                    l.c.append(std::to_string(l.b));
                    // мутексћесеоткључатикада 'l' изађеизопсегаовде
            });
            // with() прихватабилокојиобјекаткојиможедапозове ,
                 чакипоказивачфункције .
    }
```

Можете да комбинујете и 'access' и 'with' у истој функицији, али то би било чудно.

1.2 Једноставно извођење

Једноставно извођење има сличну употребу. Укључите “lockstrap_simple.h” уместо “lockstrap.h” и користите “LCKSTRAPSIMP” уместо “LOCKSTRAP” макроа, при приступу подацима увек користите синтаксу позива функција. Ево целог примера са једноставним извођењем:

```
#include "lockstrap_simple.h"
    class User {
            class Data {
                    int a;
                    float b;
                    std::string c;
                    LCKSTRAPSIMP(Data, std::mutex, a,b,c);
            } d;
            std::vector<long> x;
            // ...
    };


    void User::f()
    {
            auto al = d.access();
            al.a() = 3;
            al.b() = al.a() / 2;
    }


    void User::g()
    {
            d.with([](User::locker l) { l.c().append(std::
                to_string(l.b())); });
    }
```

Можете користити “једноставно” извођење за неке класе а обично за друге, али радити то у истој датотеци би било мало збуњујуће за читаоца.

1.3 Зашто уопште користити једноставно извођење?

Можда више волите ову синтаксу са свим тим (), с обзиром да наговештава да ово није обичан приступ.

Чак и ако не волите заграде, брже се преводи. Колико брже, то зависи од тога како га користите. Али, у основним тестовима где је код радио мало шта друго, осим приступао овим подацима, било је око 15% брже. Са више кода који није везан за браве, релативно убрзање ће бити мање, али, са друге стране, ако имате много кода са закључавањем онда можда буде значајно апсолутно убрзање.

У теорији, “обично” извођење можда генерише лошији код, пошто декларише референцу у сенци за сваког заштићеног податак члана. У свим тестовима, поготово са оптимизацијама, генерисани код је заправо исти, пошто се све развије на лицу места.

Такође, за већину преводилаца, једноставно извођење ће дати донекле финије извештаје о грешкама.

1.4 Напомене

Брава не мора да буде мутекс. Може да буде било шта што примењује “Lockable” концепт, тј. да има “lock()” и “unlock()” функције чланице (класе).

Пошто морате да “поменете” сваки податак члан, можда направите грешку:

- Ако изоставите члана, преводилац ће вам дати грешку “class XXX::locker does not have a member …”, што је добар наговештај да треба да га додате.
- Ако дате лоше име члану, преводилац ће дати грешку као “XXX::bad_name doesn’t exist”, што је такође добар наговештај.

Прво извођење може да подржи до 9 податак чланова. Лако је додати још, погледајте коментаре у заглављима.

Логично, ово није било осмишљено да барата статичким подацима (иако ће радити, у извесној мери) или функцијама чланицама - које ће давати чудне извештаје грешака преводиоца - знате да функције чланице не могу бити заштићене мутексом, јел тако?

1.5 Извођење

Извођење није ништа специјално:

- Макро генерише “lock” податак члан у самој класи
- Онда генерише угњеждену класу по имену “locker” који ће бити коришћен за приступ подацима
- За једноставно извођење, “locker” има референцу на “прави” објекат и има гомилу функција чланица са истим именима као податак чланови “праве” класе
- За обично извођење, “locker” има референцу за сваки податак члан праве класе, са потпуно истим именом
- Било како било, “locker” ће закључати браву током конструкције и откључати током уништења (да, RAII)
- Макро генерише “access” функцију члана која ће вратити “locker” објекат
- Макро генерише “with” шаблонску функцију чланицу која ће да прихвати шаблонизовани параметар као објекат који може да се позове и направити “locker” објекат који ће пренети на тај параметар

1.6 Дискусија

1.6.1 Зашто ово не може бити урађено са шаблонима?

Добро, можете да направите нешто као “паметни показивач” ако направите ” јавне” податке у “правој” класи, али то нема сврхе радити, јер су подаци сада јавни и било ко може да их користи без браве.

Ако подаци остану приватни, онда можете користити нешто слично уређеној н-торци, али то не генерише имена симбола, тако да бисте користили другачију (и прилично ружну) синтаксу како бисте приступили подацима. Постоје други трикови којима можете да се служите, али су сви са истим “тупле” проблемом.

C++11 извођење и употребу учини много лакшим јер подржава макрое са променљивим бројем аргумената и decltype“. Заправо, са C++14, можете користити ‘auto’ уместо ‘decltype(ME::x)’ током декларисања”функција за прослеђивање” у “једноставном” извођењу.

1.6.2 C++03

Сличне ствари су могуће у C++03, али морали би поново да декларишете типове свих података (и позовете макро "верзију" са тачним бројем података):

```
class Data {
            int a;
            float b;
            std::string c;
            LCKSTRAPSIMP3(Data, std::mutex, int, a, float, b,
                 std::string, c);
    } d;
```

или пробајте са нечим попут:

```
DECL_LOCKSTRAP_CLASS(User, std::mutex)
    DECL_LOCKSTRAP_MEMBER(int, a);
    DECL_LOCKSTRAP_MEMBER(float, b);
    END_LOCKSTRAP_CLASS()
```

Херб Сатер има C++03 дизајн који је сличан овоме у једном чланку на сајту DrDobb's Journal. Последњи познати URL:

http://www.drdobbs.com/windows/associate-mutexes-with-data-to-prevent-r/224701827

Осим поприлично ружне макро синтаксе, његов дизајн заправо излаже lock() и unlock(), што га чини подложним грешкама (можете приступити подацима без закључавања). Он провери (да је брава закључана) тако што користи assert, али, верујем да је lockstrap дизајн бољи, пошто ове провере нису ту током конфигурације за испоруку, и бубе у вишенитном програмирању су озлоглашене да се појављују (само) на терену(а отпремате конфигурацију за испоруку на терен Такође, чак и у превођењу за дебубовање ова провера није довољна. То јест, можете утврдити да је брава закључана, али, док приступите подацима, брава се можда откључа (од друге нити наравно).

1.6.3 Да ли параметар од with() треба да буде пренесен по вредности или двредности (енг. rvalue)?

Па, се преноси као двредност по дизајну. Да ли ћете га декларисати као параметар по вредности, овако:

```
    d.with([](User::locker l) {
```

или двредност параметар, овако:

```
            d.with([](User::locker &&l) {
```

није претерано битно. У теорији може да буде, али у пракси, поготово ако су оптимизације укључене, ово производи исти бинарни код.