

# Resource Allocation Is Initialization

PV173 Programování v C++11

Vladimír Štill, Jiří Weiser

Fakulta Informatiky, Masarykova Univerzita

13. října 2014

# Resource Allocation Is Initialization (RAII)

O co jde?

- způsob jak přistupovat ke správě zdrojů
- relativně bezpečné z hlediska výjimek
- komfortnější než manuální řešení
- lepší náhrada za `finally` block (který C++ nemá)
- princip využívaný smart pointery

# Resource Allocation Is Initialization (RAII)

Jak to funguje?

Pro správu zdroje se používá objekt alokovaný na zásobníku

- zdroj se alokuje v konstruktoru a dealokuje v destruktoru
- alokace na zásobníku garantuje, že destruktork proběhne při „standardním“ opuštění kontextu (podrobnosti dále)

```
struct Resource {  
    Resource() { /* allocate resource */ }  
    ~Resource() { /* release resource */ }  
};
```

# Resource Allocation Is Initialization (RAII)

Opakování – pořadí volání destruktorků

```
#include <cstdio>
struct X {
    char c;
    X( char c ) : c( c ) { std::putchar(c); }
    ~X() { std::putchar('~'); std::putchar(c); }
};
int main() {
    { X a('a');
      X b('b');
    }
    X c('c');
    { X d('d'); }
}
```

Jaký bude výstup?

# Resource Allocation Is Initialization (RAII)

Kdy to funguje? (lokální proměnné)

Určitě funguje pro opuštění bloku:

- pomocí **return**
- dojitím na konec
- vyhozením výjimky, která je výše zachycená

Určitě nefunguje při:

- volání `std::exit`, `std::quick_exit`, `std::abort`,...
- volání `std::longjmp`
- ukončení na základě obdržení signálu
- vypnutí elektřiny

Nemusí fungovat při:

- vyhození výjimky, která není nikde odchycená (!)

# Resource Allocation Is Initialization (RAII)

## Výhody

- nemusíme se zabývat explicitní správou zdrojů
- snadná kompozice
  - pokud máme více zdrojů stačí je alokovat postupně
  - při opuštění kontextu se dealokují v opačném pořadí deklarace
  - funguje i pro data ve třídách → často nemusíme psát destruktory (defaultní zajistí volání destruktorků dat)
- silně podporované C++ knihovnou
  - práce s pamětí: `shared_ptr`, `weak_ptr`, `unique_ptr`
  - práce se soubory: `fstream`
  - paralelismus: `lock_guard`, `unique_lock`
- oproti `finally` bloku méně zbytečného kódu

# Resource Allocation Is Initialization (RAII)

Výhody: RAII vs. finally

```
void foo() {  
    std::fstream f1("file1.txt");  
    // pracuj se souborem ...  
} // soubor se automaticky zavře
```

oproti tomu finally (java,  $\pm$  ekvivalentní chování k výjimkám):

```
public void foo() {  
    FileReader f1 = null;  
    try { f1 = new FileReader("file1.txt");  
        // pracuj se souborem ...  
    } finally {  
        try { if (f1 != null) f1.close(); }  
        catch (IOException io) { }  
    }  
}
```

# Obecné RAII utility

RAII je pěkné, někdy se nám ale nemusí chtít definovat objekt kvůli několika málo použitím. Chtělo by to něco obecnějšího.

Inspirujeme se u jazyka GO

- ten má `defer` statement
- `defer <výraz>` zavolá `<výraz>` při opuštění bloku
- GO nemá RAII ani `finally`, proto je `defer` žádoucí
- `defer` se dá simulovat v C++11 pomocí RAII a lambda funkcí
- v C++ je použití RAII čistší řešení
- ale `defer` je vhodný pro ad-hoc použití



Implementujte třídu `Defer`, která:

- v konstruktoru bere funkci (typu `void ()`)
- zavolá tuto funkci v destruktoru

Rozšíření:

- přidejte metodu `run()`, která spustí asociovanou funkci okamžitě
- zajistěte, že funkce bude spuštěna jen jednou (nezávisle na způsobu)

```
#include <functional>
struct Defer {
    Defer( std::function< void() > fn ) :
        _fn( fn )
    { }
    ~Defer() { _fn(); }
private:
    std::function< void() > _fn;
};
```

# Řešení rozšíření

```
#include <functional>
struct Defer {
    Defer( std::function< void() > fn ) :
        _fn( fn )
    { }
    ~Defer() { run(); }
    void run() {
        if ( _fn ) {
            _fn();
            _fn = nullptr;
        }
    }
private:
    std::function< void() > _fn;
};
```

Proč `std::function`?

- `Defer` neměla být šablonovaná třída
- šablonovat třídu lambda funkcí je problém, protože její typ je nespecifikovatelný
- třída by se musela obalit pomocnou metodou díky níž by se typ odvodil, to by s použitím `auto` umožnilo vytvořit hodnotu
- to by ale mohlo vytvořit dočasnou hodnotu, která by se destruovala ihned po přiřazení
  - to v praxi nenastává, ale jen díky optimalizacím v kompilátoru (copy elision)
  - nelze se na to spolehnout

# (Ne)praktický Defer

```
void foo( std::string path ) {  
    int fd = open( path.c_str(), O_RDONLY );  
    assert( fd != -1 );  
    Defer d( [&]() { close( fd ); } );  
    // ...  
}
```

Každý Defer statement musí být pojmenovaný, jinak je to dočasná hodnota a bude zavolán okamžitě!

# Obecné RAI 2

Defer je pěkný, ale:

- zbytečně musíme vytvořit další proměnnou
- dealokace není viditelně svázaná s alokací

Další možnost: obecná třída pro správu zdrojů s alokací a dealokací

- bere v konstruktoru dva callbacky, jeden pro alokaci a druhý pro dealokaci
- destrukturu provede dealokaci
- bude vhodné jako ad-hoc obal pro správu zdrojů přístupných jen přes C API

# Cvičení – Resource

## Příklad

```
void foo() {  
    Resource< int > fd( []{  
        int f = ::open("file.txt", O_RDONLY);  
        if ( f == -1 ) throw ...;  
        return f;  
    }, []( int f ) { ::close( f ); } );  
    // ...  
  
    struct stat st;  
    int r = fstat( fd, &st ); // cast fd na int  
    if ( r == -1 ) throw ...;  
    // ...  
} // zavolá se close
```

# Cvičení – Resource

Implementujte třídu `Resource< R >`, která implementuje přístup popsáný na předchozím slidu. `R` je handle zdroje (file descriptor, pointer, ...) a třída má následující veřejné rozhraní:

```
template< typename R > struct Resource {  
    template< typename Allocate >  
    Resource( Allocate alloc,  
              std::function< void( R & ) > dealloc );  
    // přístup k alokovanému zdroji  
    // vyhodí výjimku pokud není zdroj alokován  
    R get() const;  
    operator R() const;  
    void free(); // okamžitě uvolní zdroj  
    void disown(); // zajistí, že dealokace  
                   // neproběhne  
};
```