

# Správa zdrojů a paměť

## PV173 Programování v C++11

Vladimír Štill, Jiří Weiser

Fakulta Informatiky, Masarykova Univerzita

6. října 2014

# Správa zdrojů

Jak něco bezpečně vlastnit a neztratit to.

## Obsah

- Co to je RAI.
- Koncept RAI v STL C++11.
- Užitečné tipy s pamětí
  - The Rule Of Three
  - *Move sémantika*
  - The Rule Of Four (copy&swap idiom v C++11)
- Samostatné programování

# Správa zdrojů

Co je to RAI

- Resource Acquisition is Initialization
- Každý zdroj spravuje právě jedna třída.
- Třída spravuje maximálně jeden zdroj.

# Správa zdrojů

Co je to RAI

- Resource Acquisition is Initialization
- Každý zdroj spravuje právě jedna třída.
- Třída spravuje maximálně jeden zdroj.
- Podrobněji příští hodinu.

# Správa zdrojů

Co je to RAI

- Idea

# Správa zdrojů

Co je to RAII

- Idea
- Třída v konstruktoru alokuje zdroj.
  - Paměť, soubor, hardware, socket, mutex, ...
  - Pouze jeden zdroj z důvodu případné konzistence při výjimce.
- V destruktoru se zdroj uvolní.
  - Destruktory jsou téměř vždy zavolány.

# Správa zdrojů

Koncept RAII v STL C++11.

Pár příkladů RAII v STL

- `std::(i/o)fstream`
- **smart pointry**
- kontejnery (svým způsobem)
- `std::lock_guard`

# Správa zdrojů – smart pointry

## Vlastnictví paměti

```
{  
std::shared_ptr< SmallObject > smallObject =  
    new SmallObject( 'a', 10 );  
std::shared_ptr< BigObject > bigObject =  
    std::make_shared< BigObject >(  
        param1,  
        param2  
    );  
bigObject->foo();  
}
```



# Správa zdrojů – smart pointry

## Ukazatel na paměť

```
std::weak_ptr< A > wp;
{
    std::shared_ptr< A > sp = new A();
    wp = sp;

    // auto == std::shared_ptr< A >
    if ( auto locked = wp.lock() )
        locked->foo();
}
if ( wp.expired() )
    std::cout << "wp has expired" << std::endl;
```

# Správa zdrojů – smart pointry

## Unikátní paměť

```
{  
    std::unique_ptr< SmallObject > so =  
        new SmallObject( param1, param2 );  
}
```

Funkce `std::make_unique` je k dispozici až v C++14.

“Pokud chcete implementovat jednu z těchto metod, pravděpodobně chcete implementovat všechny.”

- Kopírovací konstruktor
- Přiřazovací operátor
- Destruktor

- L-value reference

- L-value reference
  - “To, co může být na levé straně přiřazení.”
  - Proměnná, reference, bitfield, dereferencovaný ukazatel,  
...
  - Zapisuje se **type &variable**

- L-value reference
  - “To, co může být na levé straně přiřazení.”
  - Proměnná, reference, bitfield, dereferencovaný ukazatel,  
...
  - Zapisuje se **type &variable**
- R-value reference
  - “To ostatní.”
  - Dočasné objekty, čísla, řetězce
  - Zapisuje se **type &&variable**

- Hlavní využití – konstruktory
  - Běžný kopírovací konstruktor zkopíruje všechna data.

- Hlavní využití – konstruktory
  - Běžný kopírovací konstruktor zkopíruje všechna data.
  - Někdy stačí data pouze převzít.



- Hlavní využití – konstruktory
  - Běžný kopírovací konstruktor zkopíruje všechna data.
  - Někdy stačí data pouze převzít.
    - Návrat z funkce (ve funkci již není třeba).
    - Poslání do funkce (pokud už nebude objekt potřeba).

- Hlavní využití – konstruktory
  - Běžný kopírovací konstruktor zkopíruje všechna data.
  - Někdy stačí data pouze převzít.
    - Návrat z funkce (ve funkci již není třeba).
    - Poslání do funkce (pokud už nebude objekt potřeba).
- Funkce/metody mohou brát parametry jako R-value reference.

# Paměť

## Move sémantika – použití

```
struct A {  
    int *_d;  
    A() : _d( new int[ 10 ] ) {}  
    A( const A &o ) : _d( new int[ 10 ] ) {  
        std::copy( o._d, o._d + 10, _d );  
    }  
    A( A &&o ) : _d( o._d ) { // o is L-value  
        o._d = nullptr; // serious robbery here  
    }  
    ~A() { delete[] _d; }  
};  
A first;  
A second( std::move( first ) );  
A third( second );
```

- Na objektu po vykradení musí jít zavolat
  - Destruktor.
  - Přiřazovací operátor.
  - (Nic dalšího)

- Na objektu po vykradení musí jít zavolat
  - Destruktor.
  - Přiřazovací operátor.
  - (Nic dalšího)
- Překladač sám vybere, který ctor se zavolá (podle typu).

- Na objektu po vykradení musí jít zavolat
  - Destruktor.
  - Přiřazovací operátor.
  - (Nic dalšího)
- Překladač sám vybere, který ctor se zavolá (podle typu).
- Pokud chcete vynutit volání *Move konstrukturu*, použijte **std::move**.

- Na objektu po vykradení musí jít zavolat
  - Destruktor.
  - Přiřazovací operátor.
  - (Nic dalšího)
- Překladač sám vybere, který ctor se zavolá (podle typu).
- Pokud chcete vynutit volání *Move konstrukturu*, použijte **std::move**.
- Všechny kontejnery z STL podporují *Move sémantiku*.
  - A mnohé další třídy takéž.

“Pokud chcete implementovat jednu z těchto metod, pravděpodobně chcete implementovat všechny.”

- Kopírovací konstruktor
- **Move konstruktor**
- Přiřazovací operátor
- Destruktor



# Paměť

Copy & swap idiom – silnější verze *The Rule Of Three/Four*

```
struct A {  
    int *_d;  
    A() : _d( new int[ 10 ] ) {}  
    A( const A &o ) : _d( new int[ 10 ] ) {  
        std::copy( o._d, o._d + 10, _d );  
    }  
    A( A &&o ) : _d( o._d ) { o._d = nullptr; }  
    ~A() { delete[] _d; }  
    A &operator=( A o ) {// Předání hodnotou!  
        swap( o );  
        return *this;  
    }  
    void swap( A &o ) { std::swap( _d, o._d ); }  
};
```

# Samostatné programování

Implementace kontejneru:

- N-ární strom.
- Typem prvku bude šablona.
- Ukazatele realizujte pomocí **std::shared\_ptr**/**std::weak\_ptr**.
- Implementujte metody **begin** a **end** a iterátor.
- Možnost zavěšení prvku.
- Možnost odebrání podstromu.

# Zajímavé čtení

- Copy & swap idiom

`http://stackoverflow.com/questions/3279543/what-is-the-copy-and-swap-idiom`

- První odpověď.

- Move semantics

`http://stackoverflow.com/questions/3106110/what-are-move-semantics`

- První odpověď – základ.
- Druhá odpověď – detaily.