

# Modern C++ Programming

## 18. ADVANCED TOPICS I

---

*Federico Busato*

2024-02-09

## 1 Move Semantic

- lvalues and rvalues references
- Move Semantic
- `std::move`
- Class Declaration Semantic

## 2 Universal Reference and Perfect Forwarding

- Universal Reference
- Reference Collapsing Rules
- Perfect Forwarding

## 3 Value Categories

## 4 `&`, `&&` Ref-qualifiers and `volatile` Overloading

- `&`, `&&` Ref-qualifiers Overloading
- `volatile` Overloading

## 5 Copy Elision and RVO

## 6 Type Deduction

- Pass-by-Reference
- Pass-by-Pointer
- Pass-by-Value
- auto Deduction

## 7 const Correctness

# Move Semantic

---

***Move semantics refers in transferring ownership of resources from one object to another***

Differently from *copy semantic*, *move semantic* does not duplicate the original resource

In C++ every expression is either an **rvalue** or an **lvalue**

- a **lvalue** (left) represents an expression that occupies some identifiable location in memory
- a **rvalue** (right) is an expression that does not represent an object occupying some identifiable location in memory

```
int x = 5;           // "x" is an lvalue, "5" is an rvalue  
int y = 10;         // "y" is an lvalue  
  
int z = (x * y);    // "z" is an lvalue, (x * y) is an rvalue
```

C++11 introduces a new kind of *reference* called **rvalue reference** `X&&`

- An **rvalue reference** only binds to an **rvalue**, that is a temporary
- An **lvalue reference** only binds to an **lvalue**
- A **const lvalue reference** binds to both **lvalue** and **rvalue**

```
int      x = 5;          // "x" is an lvalue
int&     r1 = x;        // "r1" is an lvalue reference
// int&  r2 = 5;        // compile error, "5" is an rvalue
const int& cr = (x * y); // "cr" is an const lvalue reference

int&&     rv = (x * y); // "rv" is an rvalue
// int&&  rv1 = x;      // compile error, "x" is NOT an rvalue
```



```
struct A {};  
  
void f(A& a) {}           // lvalue reference  
  
void g(const A& a) {}    // const lvalue reference  
  
void h(A&& a) {}         // rvalue reference  
  
A a;  
f(a);           // ok, f() can modify "a"  
g(a);           // ok, f() cannot modify "a"  
// h(a);        // compile error f() does not accept lvalues  
  
// f(A{});      // compile error f() does not accept rvalues  
g(A{});         // ok, f() cannot modify the object A{}  
h(A{});         // ok, f() can modify the object A{}
```

```
#include <algorithm>
class Array { // Array Wrapper
public:
    Array() = default;

    Array(int size) : _size{size}, _array{new int[size]} {}

    Array(const Array& obj) : _size{obj._size}, _array{new int[obj._size]} {
        // EXPENSIVE COPY (deep copy)
        std::copy(obj._array, obj._array + _size, _array);
    }

    ~Array() { delete[] _array; }
private:
    int _size;
    int* _array;
};
```

```
#include <vector>

int main() {
    std::vector<Array> vector;
    vector.push_back( Array{1000} ); // call push_back(const Array&)
}                                     // expensive copy
```

**Before C++11:** `Array{1000}` is created, passed by const-reference, copied, and then destroyed

Note: `Array{1000}` is no more used outside `push_back`

**After C++11:** `Array{1000}` is created, and moved to `vector` (fast!)

**Class prototype** with support for *move semantic*:

```
class X {  
public:  
    X();                // default constructor  
  
    X(const X& obj);    // copy constructor  
  
    X(X&& obj);         // move constructor  
  
    X& operator=(const X& obj); // copy assign operator  
  
    X& operator=(X&& obj);    // move assign operator  
  
    ~X();                // destructor  
};
```

## Move constructor semantic

```
X(X&& obj);
```

- (1) *Shallow copy* of `obj` data members (in contrast to deep copy)
- (2) *Release* any `obj` resources and reset all data members (pointer to `nullptr`, size to 0, etc.)

## Move assignment semantic

```
X& operator=(X&& obj);
```

- (1) *Release* any resources of `this`
- (2) *Shallow copy* of `obj` data members (in contrast to deep copy)
- (3) *Release* any `obj` resources and reset all data members (pointer to `nullptr`, size to 0, etc.)
- (4) Return `*this`

## Move constructor

```
Array(Array&& obj) {  
    _size      = obj._size; // (1) shallow copy  
    _array     = obj._array; // (1) shallow copy  
    obj._size  = 0;         // (2) release obj (no more valid)  
    obj._array = nullptr;  // (2) release obj  
}
```

## Move assignment

```
Array& operator=(Array&& obj) {  
    delete[] _array; // (1) release this  
    _size      = obj._size; // (2) shallow copy  
    _array     = obj._array; // (2) shallow copy  
    obj._array = nullptr;  // (3) release obj  
    obj._size  = 0;         // (3) release obj  
    return *this;         // (4) return *this  
}
```

**C++11** provides the method `std::move` ( `<utility>` ) to indicate that an object may be “moved from”

It allows to efficient transfer resources from an object to another one

```
#include <vector>

int main() {
    std::vector<Array> vector;
    vector.push_back( Array{1000} );    // call "push_back(Array&&)"

    Array arr{1000};
    vector.push_back( arr );           // call "push_back(const Array&)"

    vector.push_back( std::move(arr) ); // call "push_back(Array&&)"
                                        // efficient!!

    // "arr" is not more valid here
}
```

## Move Semantic Notes

If an object requires the *copy constructor/assignment*, then it should also define the *move constructor/assignment*. The opposite could not be true

The *defaulted move constructor/assignment* `=default` recursively applies the move semantic to its *base class* and *data members*.

Important: *it does not release the resources*. It is very dangerous for classes with manual resource management

```
// Suppose: Array(Array&&) = default;
Array x{10};
Array y = std::move(x); // call the move constructor
// "x" calls ~Array() when it is out of scope, but now the internal pointer
// "_array" is NOT nullptr -> double free or corruption!!
```



# Move Semantic and Code Reuse

Some operations can be expressed as a function of the move semantic

```
A& operator=(const A& other) {  
    *this = std::move(A{other}); // copy constructor + move assignment  
    return *this;  
}
```

```
void init(... /* any paramters */) {  
    *this = std::move(A{...}); // user-declared constructor + move assignment  
}
```

## Special Members

compiler implicitly declares

|                     | default constructor | destructor    | copy constructor | copy assignment | move constructor | move assignment |
|---------------------|---------------------|---------------|------------------|-----------------|------------------|-----------------|
| Nothing             | defaulted           | defaulted     | defaulted        | defaulted       | defaulted        | defaulted       |
| Any constructor     | not declared        | defaulted     | defaulted        | defaulted       | defaulted        | defaulted       |
| default constructor | user declared       | defaulted     | defaulted        | defaulted       | defaulted        | defaulted       |
| destructor          | defaulted           | user declared | defaulted        | defaulted       | not declared     | not declared    |
| copy constructor    | not declared        | defaulted     | user declared    | defaulted       | not declared     | not declared    |
| copy assignment     | defaulted           | defaulted     | defaulted        | user declared   | not declared     | not declared    |
| move constructor    | not declared        | defaulted     | deleted          | deleted         | user declared    | not declared    |
| move assignment     | defaulted           | defaulted     | deleted          | deleted         | not declared     | user declared   |

user declares

# Class Declaration Semantic

---

| User-declared Entity                     | Meaning / Implications  |
|--|---|
| <code>non- static const members</code>   | <i>Copy/Move constructors</i> are not trivial (not provided by the compiler). <i>Copy/move assignment</i> is not supported  |
| <code>reference members</code>           | <i>Copy/Move constructors/assignment</i> are not trivial (not provided by the compiler)   |
| <code>destructor</code>                  | The resource management is not trivial. <i>Copy constructor/assignment</i> is very likely to be implemented   |
| <code>copy constructor/assignment</code> | Resource management is not trivial. <i>Move constructors/assignment</i> need to be implemented by the user  |
| <code>move constructor/assignment</code> | There is an efficient way to move the object. <i>Copy constructor/assignment</i> cannot fall back safely to <i>copy constructors/assignment</i> , so they are deleted |

---

# Universal Reference and Perfect Forwarding

---

The `&&` syntax has two different meanings depending on the context it is used

- **rvalue reference**
- **Universal reference**: Either **rvalue reference** or **lvalue reference**

**Universal references** (also called *forwarding references*) are **rvalues** that appear in a type-deducing context. `T&&`, `auto&&` accept any expression regardless it is an **lvalue** or **rvalue** and preserve the `const` property

```
void f1(int&& t) {} // rvalue reference

template<typename T>
void f2(T&& t) {} // universal reference

int&& v1 = ...; // rvalue reference
auto&& v2 = ...; // universal reference
```

```
int          f_copy()          { return x; }
int&        f_ref(int& x)      { return x; }
const int&  f_const_ref(const int& x) { return x; }

auto        v1 = ...; // f_copy(), f_const_ref(), only lvalues
auto&      v2 = ...; // f_ref(), only lvalue ref
const auto& v3 = ...; // f_copy(), f_ref(), f_const_ref()
                // only const lvalue ref (decay), cannot be modified
const auto&& v4 = ...; // f_copy(), only rvalues, cannot be modified

auto&&     v5 = ...; // everything
```

```
struct A {};  
void f1(A&& a) {} // rvalue only  
  
template<typename T>  
void f2(T&& t) {} // universal reference  
  
A a;  
f1(A{}); // ok  
// f1(a); // compile error (only rvalue)  
f2(A{}); // universal reference  
f2(a); // universal reference  
  
A&& a2 = A{}; // ok  
// A&& a3 = a; // compile error (only rvalue)  
auto&& a4 = A{}; // universal reference  
auto&& a5 = a; // universal reference
```

## Universal Reference - Misleading Cases

```
template<typename T>
void f(std::vector<T>&&) {} // rvalue reference

template<typename T>
void f(const T&&) {}      // rvalue reference (const)

const auto&& v = ...;    // const rvalue reference
```



# Reference Collapsing Rules

Before C++11 (C++98, C++03), it was not allowed to take a reference to a reference ( `A& &` causes a compile error)

C++11, by contrast, introduces the following **reference collapsing rules**:

```
template<typename T>
void f(T&) {} // compile error in C++98/03 (with gcc),
              // no errors in C++11 (and clang with C++98/03)

int a = 3;    //
f<int&>(a);   //
```

| Type | Reference |   | Result |
|------|-----------|---|--------|
| A&   | &         | → | A&     |
| A&   | &&        | → | A&     |
| A&&  | &         | → | A&     |
| A&&  | &&        | → | A&&    |

# Perfect Forwarding

*Perfect forwarding* allows preserving argument *value category* and *const/volatile* modifiers

`std::forward` ( `<utility>` ) forwards the argument to another function with the *value category* it had when passed to the calling function (*perfect forwarding*)

```
#include <utility> // std::forward
template<typename T> void f(T& t) { cout << "lvalue"; }
template<typename T> void f(T&& t) { cout << "rvalue"; } // overloading

template<typename T> void g1(T&& obj) { f(obj); } // call only f(T&)
template<typename T> void g2(T&& obj) { f(std::forward<T>(obj)); }

struct A{};
f ( A{10} ); // print "rvalue"
g1( A{10} ); // print "lvalue"!!
g2( A{10} ); // print "rvalue"
```

# Value Categories

---

## Taxonomy (simplified)

Every expression is either an **rvalue** or an **lvalue**

- An **lvalue** (*left* value of an assignment for historical reason or *locator* value) represents an expression that occupies an *identity*, namely a memory location (it has an address)
- An **rvalue** is movable; an **lvalue** is not

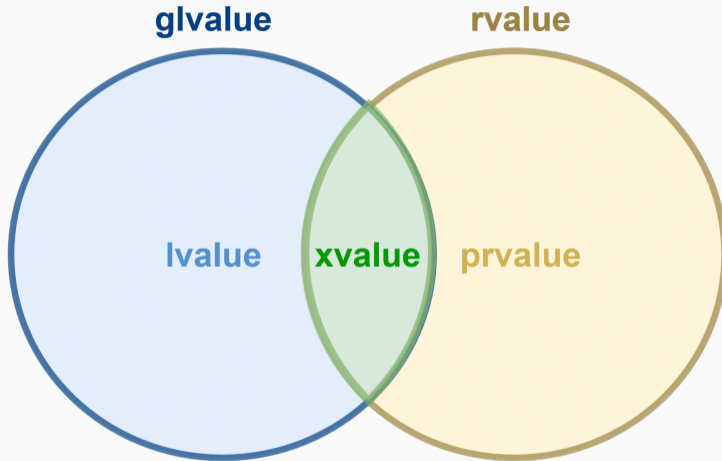
**glvalue** (*generalized lvalue*) is an expression that has an identity

**lvalue** is a **glvalue** but it is not movable (it is not an **xvalue**). An *named rvalue reference* is a **lvalue**

**xvalue** (*eXpiring*) has an identity and it is movable. It is a **glvalue** that denotes an object whose resources can be reused. An *unnamed rvalue reference* is a **xvalue**

**prvalue** (*pure rvalue*) doesn't have identity, but is movable. It is an expression whose evaluation initializes an object or computes the value of an operand of an operator

**rvalue** is movable. It is a **prvalue** or an **xvalue**



# Examples

```
struct A {
    int x;
};

void f(A&&) {}
A&& g();
//-----
int a = 4;      // "a" is an lvalue, "4" is a prvalue
f(A{4});       // "A{4}" is a prvalue

A&& b = A{3};   // "A&& b" is a named rvalue reference → lvalue

A c{4};
f(std::move(c)); // "std::move(c)" is a xvalue
f(A{}.x);        // "A{}.x" is a xvalue
g();             // "A&&" is a xvalue
```

# **&, && Ref-qualifiers and volatile Overloading**

---



C++11 allows overloading member functions depending on the **lvalue/rvalue** property of their object. This is also known as **ref-qualifiers overloading** and can be useful for optimization purposes, namely, moving a variable instead of copying it

```
struct A {  
    // void f()    {} // already covered by "f() &"  
    void f() & {}  
    void f() && {}  
};  
  
A a1;  
a1.f();           // call "f() &"  
  
A{}.f();         // call "f() &&"  
std::move(a1).f(); // call "f() &&"
```

*Ref-qualifiers overloading* can be also combined with `const` methods

```
struct A {  
    // void f() const    {} // already covered by "f() const &"  
    void f() const & {}  
    void f() const && {}  
};  
  
const A a1;  
a1.f();           // call "f() const &"  
  
std::move(a1).f(); // call "f() const &&"
```

A simple example where *ref-qualifiers overloading* is useful

```
struct ArrayWrapper {  
    ArrayWrapper(/*params*/) { /* something expensive */ }  
  
    ArrayWrapper copy() const & { /* expensive copy with std::copy() */ }  
    ArrayWrapper copy() const && { /* just move the pointer as the original  
                                     object is no more used */ }  
};
```

## volatile Overloading

```
struct A {  
    void f()          {}  
    void f() volatile {} // e.g. propagate volatile to data members  
    void f() const volatile {}  
// void f() volatile &      {} // combining ref-qualifier and volatile  
// void f() const volatile & {} // overloading is also fine  
// void f() volatile &&     {}  
// void f() const volatile && {}  
};  
  
volatile A a1;  
a1.f(); // call "f() volatile"  
  
const volatile A a2;  
a2.f(); // call "f() const volatile"
```

# Copy Elision and RVO

---

## Copy Elision and RVO

**Copy elision** is a compiler optimization technique that eliminates unnecessary copying/moving of objects (it is defined in the C++ standard)

A compiler avoids omitting copy/move operations with the following optimizations:

- **RVO (Return Value Optimization)** means the compiler is allowed to avoid creating *temporary* objects for return values
- **NRVO (Named Return Value Optimization)** means the compiler is allowed to return an object (with automatic storage duration) without invokes copy/move constructors

## RVO Example

Returning an object from a function is *very expensive* without RVO/NVRO:

```
struct Obj {
    Obj() = default;

    Obj(const Obj&) { // non-trivial
        cout << "copy constructor\n";
    }
};

Obj f() { return Obj{}; } // first copy

auto x1 = f();           // second copy (create "x")
```

If provided, the compiler uses the *move constructor* instead of *copy constructor*

## RVO - Where it works

*RVO Copy elision* is always guaranteed if the operand is a `prvalue` of the same class type and the *copy constructor* is trivial and non-deleted

```
struct Trivial {
    Trivial()           = default;
    Trivial(const Trivial&) = default;
};

// single instance
Trivial f1() {
    return Trivial{}; // Guarantee RVO
}

// distinct instances and run-time selection
Trivial f2(bool b) {
    return b ? Trivial{} : Trivial{}; // Guarantee RVO
}
```



## Guaranteed Copy Elision (C++17)

In C++17, *RVO Copy elision* is always guaranteed if the operand is a `prvalue` of the same class type, even if the *copy constructor* is not trivial or deleted

```
struct S1 {
    S1()          = default;
    S1(const S1&) = delete; // deleted
};

struct S2 {
    S2()          = default;
    S2(const S2&) {}      // non-trivial
};

S1 f() { return S1{}; }
S2 g() { return S2{}; }

auto x1 = f(); // compile error in C++14
auto x2 = g(); // RVO only in C++17
```

NRVO is not always guarantee even in C++17

```
Obj f1() {  
    Obj a;  
    return a; // most compilers apply NRVO  
}  
  
Obj f2(bool v) {  
    Obj a;  
    if (v)  
        return a; // copy/move constructor  
    return Obj{}; // RVO  
}
```

```
Obj f3(bool v) {  
    Obj a, b;  
    return v ? a : b;    // copy/move constructor  
}  
  
Obj f4() {  
    Obj a;  
    return std::move(a); // force move constructor  
}  
  
Obj f5() {  
    static Obj a;  
    return a;           // only copy constructor is possible  
}
```

```
Obj f6(Obj& a) {  
    return a; // copy constructor (a reference cannot be elided)  
}  
  
Obj f7(const Obj& a) {  
    return a; // copy constructor (a reference cannot be elided)  
}  
  
Obj f8(const Obj a) {  
    return a; // copy constructor (a const object cannot be elided)  
}  
  
Obj f9(Obj&& a) {  
    return a; // copy constructor (the object is instantiated in the function)  
}
```

# Type Deduction

---

**When you call a template function, you may omit any template argument that the compiler can determine or deduce (inferred) by the usage and context of that template function call [IBM]**

- The compiler tries to deduce a template argument by comparing the type of the corresponding template parameter with the type of the argument used in the function call
- Similar to function default parameters, (any) template parameters can be deduced only if they are at end of the parameter list

Full Story: [IBM Knowledge Center](#)

## Example

```
template<typename T>
int add1(T a, T b) { return a + b; }

template<typename T, typename R>
int add2(T a, R b) { return a + b; }

template<typename T, int B>
int add3(T a) { return a + B; }

template<int B, typename T>
int add4(T a) { return a + B; }

add1(1, 2);           // ok
// add1(1, 2u);      // the compiler expects the same type
add2(1, 2u);         // ok (add2 is more generic)
add3<int, 2>(1);     // "int" cannot be deduced
add4<2>(1);          // ok
```

# Type Deduction - Pass by-Reference

## Type deduction with references

```
template<typename T>
void f(T& a) {}

template<typename T>
void g(const T& a) {}

int      x = 3;
int&     y = x;
const int& z = x;
f(x);    // T: int
f(y);    // T: int
f(z);    // T: const int // <-- !! it works...but it does not
g(x);    // T: int      //      for "f(int& a)"!!
g(y);    // T: int      //      (only non-const references)
g(z);    // T: int      // <-- note the difference
```



## Type deduction with pointers

```
template<typename T>
void f(T* a) {}

template<typename T>
void g(const T* a) {}

int*      x = nullptr;
const int* y = nullptr;
auto      z = nullptr;
f(x);     // T: int
f(y);     // T: const int
// f(z);  // compile error!! z: "nullptr_t != T*"
g(x);     // T: int
g(y);     // T: int  <-- note the difference
```

```
template<typename T>
void f(const T* a) {} // pointer to const-values

template<typename T>
void g(T* const a) {} // const pointer

int*          x = nullptr;
const int*    y = nullptr;
int* const    z = nullptr;
const int* const w = nullptr;
f(x);        // T: int
f(y);        // T: int
f(z);        // T: int
// g(x); // compile error!! objects pointed are not constant
// g(y); // the same (the pointer itself is constant)
g(z);        // T: int
g(w);        // T: const int
```

## Type deduction with values

```
template<typename T>
void f(T a) {}

template<typename T>
void g(const T a) {}

int      x = 2;
const int y = 3;
const int& z = y;
f(x);    // T: int
f(y);    // T: int!! (drop const)
f(z);    // T: int!! (drop const&)
g(x);    // T: int
g(y);    // T: int
g(z);    // T: int!! (drop reference)
```

```
template<typename T>
void f(T a) {}

int*      x = nullptr;
const int* y = nullptr;
int* const z = x;
f(x);    // T = int*
f(y);    // T = const int*
f(z);    // T = int* !! (const drop)
```

# Type Deduction - Array

## Type deduction with arrays

```
template<typename T, int N>
void f(T (&array)[N]) {}    // type and size deduced

template<typename T>
void g(T array) {}

int      x[3] = {};
const int y[3] = {};
f(x);    // T: int, N: 3
f(y);    // T: const int, N: 3
g(x);    // T: int*
g(y);    // T: const int*
```

```
template<typename T>
void add(T a, T b) {}
```

```
template<typename T, typename R>
void add(T a, R b) {}
```

```
template<typename T>
void add(T a, char b) {}
```

```
add(2, 3.0f);           // call add(T, R)
// add(2, 3);          // error!! ambiguous match
add<int>(2, 3);         // call add(T, T)
add<int, int>(2, 3);    // call add(T, R)
add(2, 'b');           // call add(T, char) -> nearest match
```

```
template<typename T, int N>
void f(T (&array)[N]) {}

template<typename T>
void f(T* array) {}

// template<typename T>
// void f(T array) {} // ambiguous

int x[3];
f(x); // call f(T*) not f(T(&)[3]) !!
```

# auto Deduction

- `auto x =` copy by-value/by-const value
- `auto& x =` copy by-reference/by-const-reference
- `auto* x =` copy by-pointer/by-const-pointer
- `auto&& x =` copy by-universal reference
- `decltype(auto) x =` automatic type deduction

```
int          f1(int& x) { return x; }
int&        f2(int& x) { return x; }
auto        f3(int& x) { return x; }
decltype(auto) f4(int& x) { return x; }
```

```
int v = 3;
int x1 = f1(v);
int& x2 = f2(v);
// int& x3 = f3(v); // compile error 'x' is copied by-value
int& x4 = f4(v);
```



# const **C**orrectness

---

**const correctness** refers to guarantee object/variable const consistency throughout its lifetime and ensuring safety from unintentional modifications

References:

- Isocpp: `const-correctness`
- GotW: `Const-Correctness`
- Abseil: `Meaningful 'const' in Function Declarations`
- `const` is a contract
- `Why const Doesn't Make C Code Faster`
- `Constant Optimization?`

- `const` entities do not change their values at run-time. This does not imply that they are evaluated at compile-time
- `const T*` is different from `T* const`. The first case means “*the content does not change*”, while the later “*the value of the pointer does not change*”
- Pass *by-const-value* and *by-value* parameters imply the *same* function signature
- Return *by-const-value* and *by-value* have different meaning
- `const_cast` can *break* const-correctness

**const** and member functions:

- `const` member functions do not change the internal status of an object
- `mutable` fields can be modified by a `const` member function (they should not change the external view)

**const** and code optimization:

- `const` keyword purpose is for correctness (*type safety*), not for performance
- `const` may provide performance advantages in a few cases, e.g. non-trivial copy semantic

## Function Declarations Example

```
void f(int);  
void f(const int); // the declaration is exactly the same of  
                  // "void f(int)"!!  
  
void f(int*);  
void f(const int*); // different declaration  
  
void f(int&);  
void f(const int&); // different declaration
```

```
int      f();  
// const int f(); // compile error conflicting declaration
```

## const Return Example

```
const int const_value = 3;

const int& f2() { return const_value; }
// int&      f1() { return const_value; } // WRONG
int       f3() { return const_value; }   // ok
```

```
struct A {
    void f()      { cout << "non-const"; }
    void f() const { cout << "const";    }
};
```

```
const A getA() { return A{}; }
```

```
auto a = getA(); // "a" is a copy
a.f();          // print "non-const"
```

```
getA().f();     // print "const"
```

## struct Example

```
struct A {           // struct A_const { // equal to "const A"
    int* ptr;        //     int* const ptr;
    int  value;     //     const int  value;
};                  // };

void f(A a) {
    a.value = 3;
    a.ptr[0] = 3;
}

void g(const A a) { // the same with g(const A&)
// a.value = 3;    // compile error
    a.ptr[0] = 3;  // "const" does not apply to "ptr" content!!
}

A a{new int[10]};
f(a);
g(a);
```

## Member Functions Example

```
struct A {  
    int value = 0;  
  
    int&      f1() { return value; }  
    const int& f2() { return value; }  
  
    // int&      f3() const { return value; } // WRONG  
    const int& f4() const { return value; }  
  
    int      f5() const { return value; } // ok  
    const int f6() const { return value; }  
};
```