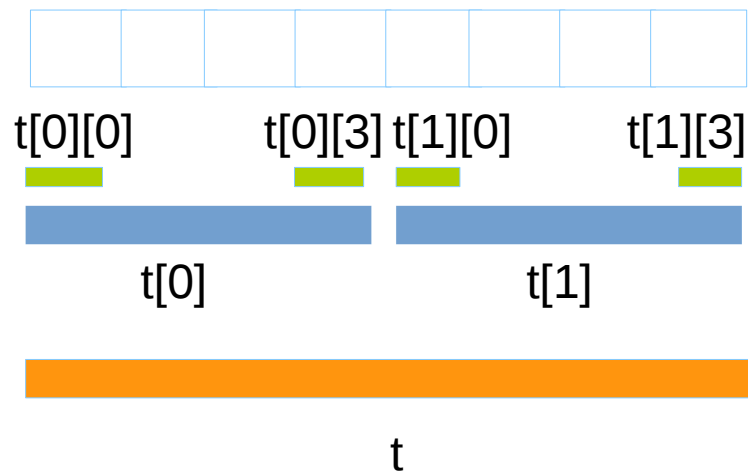


# Move semantics

- Pointers and arrays
- References
- Value vs move semantics
- Right value references
- Move semantics in C++11
- Perfect forwarding
- Traps and pitfalls

# Arrays

- An array is a strictly continuous memory area
- Arrays do not know their size, but: `sizeof(t) / sizeof(t[0])`
- Array names are converted to pointer value to first element
- No multidimensional arrays. But are array of arrays.
- No operations on arrays, only on array elements



```
int t[2][4];
```

```
assert(sizeof(t) == 8*sizeof(int));  
assert(sizeof(t[0]) == 4*sizeof(int));  
assert(sizeof(t[0][0]) == sizeof(int));
```

```
t[0][1] = t[1][1];  
// t[0] = t[1]; syntax error
```

# Pointers

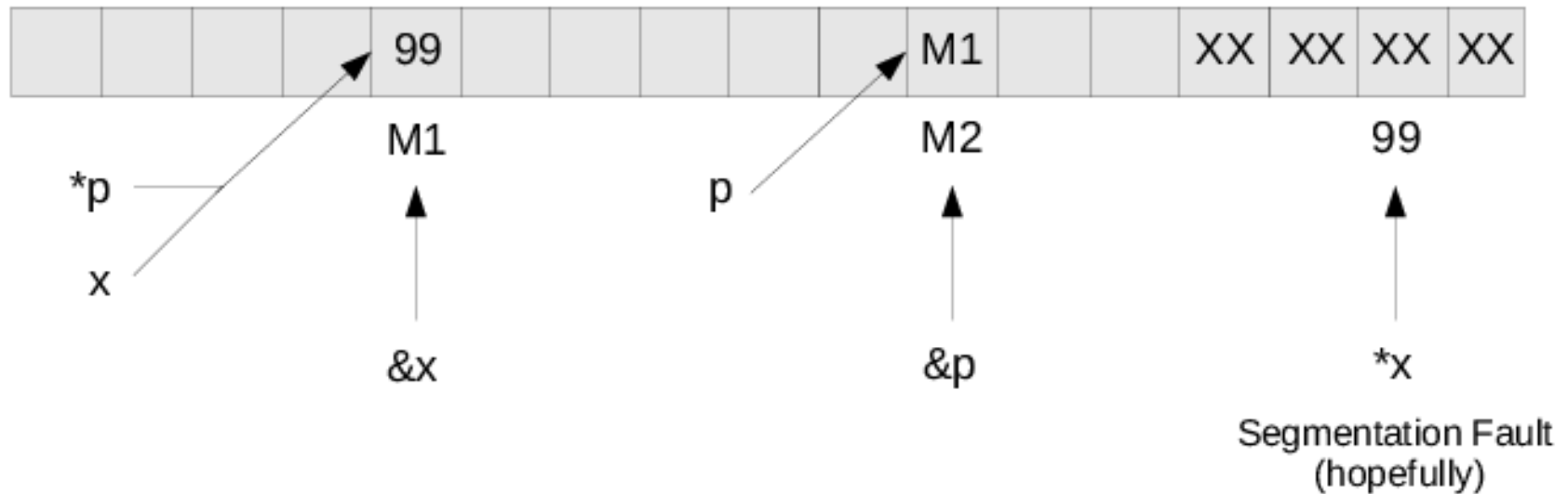
## C Code

```
int x;  
int *p;
```

```
x = 99;    //holds a value  
p = &x;   //holds an address of a value
```

## Pointers in C

## Memory



# Pointers

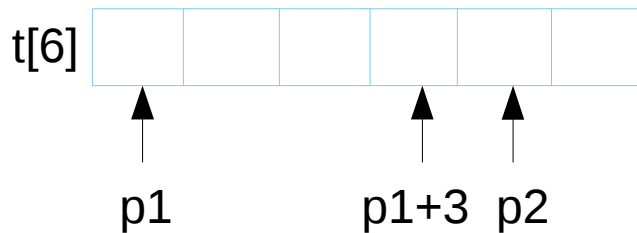
- Pointer is a value that refers to a(n abstract) memory location
- Pointers can refer to ANY valid memory locations (unlike e.g. PASCAL)
- **nullptr** (NULL) is a universal null-pointer value
- Non-null pointers are considered as TRUE value

```
char *ptr = (char *) malloc(1024);
```

```
if ( ptr )  
{  
    // ptr != NULL here  
}
```

# Pointer arithmetics

- Integers can be added and subtracted from pointers
- Pointers pointing to the same array can be subtracted
- Pointer arithmetics **depends on the pointed type!**
- No pointer arithmetics on **void \***



```
p = &t[0];  
p = t;  
p + k == &t[k]  
*(p + k) == t[k]
```

```
int t[6];  
int *p1 = &t[0];  
int *p2 = &t[4];
```

```
assert( &t[3] == p1+3 );  
assert( p2 - p1 == 4 );  
assert( p1 + 4 == p2 );
```

# Pointers and arrays

- Pointers and array names can be used similarly in expressions

```
int t[10];  
int i = 3;
```

```
int *p = t;      // t is used as pointer to the first element
```

```
p[i] = *(t + i); // p used as array, t is used as pointer
```

**But pointers ARE NOT EQUIVALENT TO arrays !!!**

# Pointers and arrays

```
// s1.c
#include <stdio.h>

int t[] = {1,2,3};

void f( int *par)
{
    printf("%d", par[1]);
    printf("%d", t[1]);
}
int main()
{
    int *p = t;
    printf("%d", t[1]);
    printf("%d", p[1]);
    f(t);
    g(t);
}
```

```
// s2.c
#include <stdio.h>

extern int *t;

void g( int *par)
{
    printf("%d", par[1]);

    // the program crashes here
    printf("%d", t[1]);
}
```

# Pointers and arrays

```
// s1.c
#include <stdio.h>

int t[] = {1,2,3};

void f( int *par)
{
    printf("%d", par[1]);
    printf("%d", t[1]);
}

int main()
{
    int *p = t;
    printf("%d", t[1]);
    printf("%d", p[1]);
    f(t);
    g(t);
}
```

```
// s2.c
#include <stdio.h>

extern int *t;

void g( int *par)
{
    printf("%d", par[1]);

    // the program crashes here
    printf("%d", t[1]);
}
```



# Member pointers

- Data Member pointer: Referencing to an offset inside a class
- Member function pointer: Referencing to a (possible virtual) member function of a class
- Works via 2 components: **this + mptr**

```
Type Class::*dmptr;  
Type (Class::*fmptr)(P1 par1, P2 par2, ...);
```

```
Class obj;  
Class *ptr = &obj;
```

```
obj.*dmptr = ...;  
ptr->*dmptr = ...;
```

```
obj.*fmptr(par1, par2);  
ptr->*fmptr(par1, par2);
```

# Member pointers

```
#include <iostream>

class Date
{
public:
    void set (int y, int m, int d);
    int  getYear() const { return _year; }
    int  getMonth() const { return _month; }
    int  getDay() const  { return _day; }

    void print(std::ostream& os) const;
    void hu();
    void us();
private:
    int _year;
    int _month;
    int _day;

    int Date::*p1;
    int Date::*p2;
    int Date::*p3;
    char sep;
};
```

# Member pointers

```
void Date::hu()
{
    sep = '.';
    p1 = &Date::_year;
    p2 = &Date::_month;
    p3 = &Date::_day;
}
void Date::us()
{
    sep = '/';
    p1 = &Date::_month;
    p2 = &Date::_day;
    p3 = &Date::_year;
}
int main()
{
    Date d;
    d.set(2017,4,20);
    d.hu();
    std::cout << d << std::endl;
    d.us();
    std::cout << d << std::endl;
}
```

```
void Date::set(int y, int m, int d)
{
    _year = y;
    _month = m;
    _day = d;
}
void Date::print(std::ostream& os) const
{
    os << this->*p1 << sep << this->*p2
      << sep << this->*p3;
}
std::ostream& operator<<(
    std::ostream& os, const Date& d)
{
    d.print(os);
    return os;
}
2017.4.20
4/20/2017
```

# Member pointers

```
int (Date::*g1)() const;
int (Date::*g2)() const;
int (Date::*g3)() const;
};

void Date::hu()
{
    sep = '.';
    g1 = &Date::getYear;
    g2 = &Date::getMonth;
    g3 = &Date::getDay;
}
void Date::us()
{
    sep = '/';
    g1 = &Date::getYear;
    g2 = &Date::getMonth;
    g3 = &Date::getDay;
}
int main()
{
    Date d;
    d.set(2017,4,20);
    d.hu();
    std::cout << d << std::endl;
    d.us();
    std::cout << d << std::endl;
}
```

```
void Date::set(int y, int m, int d)
{
    _year = y;
    _month = m;
    _day = d;
}

void Date::print(std::ostream& os) const
{
    os << (this->*g1)() << sep
        << (this->*g2)() << sep
        << (this->*g3)();
}

std::ostream& operator<<(
    std::ostream& os, const Date& d)
{
    d.print(os);
    return os;
}
```

```
2017.4.20
4/20/2017
```

# Nullptr

- nullptr is a new literal since C++11 of type std::nullptr\_t
- Helps to overload between pointers and integer
- Automatic conversion from null pointer of any type and from NULL

```
void f(int*); // 1
void f(int);  // 2

f(0);        // calls 2
f(nullptr);  // calls 1
```

# Reference

- In modern languages definitions hide two important but orthogonal concepts:
  - Allocate memory for a variable
  - Bind a name with special scope to it
- In most languages this is inseparable
- In C++ we can separate the two steps

```
void f()
{
    int i;           // allocate memory, bind i as name
    int &i_ref = i;  // binds a new name
    int *iptr = new int; // allocate memory, no binded name
    int &k = *iptr;  // binds a new name to unnamed int area
    delete iptr;    // memory invalidated name k still lives
    k = 5;          // compiles, later run-time error
}                  // k goes out of scope
```

# Pointer vs reference

- Pointers have extremal element: nullptr
  - nullptr means: pointing to no valid memory
- References always should refer to valid memory
  - Use exception, if something fails

```
if ( Derived *dp = dynamic_cast<Derived*>(bp) )
{
    // use dp as Derived*
}

try
{
    Derived &dr = dynamic_cast<Derived&>>(*bp); // may throw
    // use dr as Derived&
}
catch(bad_cast &e) { . . . }
```

# Parameter passing

- Parameter passing in C++ follows initialization semantics
  - Value initialization copies the object
  - Reference initialization just set up an alias

```
void f1( int x, int y) { ... }  
void f2( int &xr, int &yr) { ... }
```

```
int i = 5, j = 6;
```

```
f1( i, j);    int x = i;    // creates local x and copies i to x  
              int y = j;    // creates local y and copies j to y
```

```
f2( i, j);    int &xr = i; // binds xr name to i outside  
              int &yr = j; // binds yr name to j outside
```



# Swap before move semantics

```
void swap( int &x, int &y)
{
    int tmp = x;
    x = y;
    y = tmp;
}

int main()
{
    int i = 5;
    int j = 6;

    swap( i, j);
    assert(i==6 && j==5);
}
```

# Reference binding

- Non-const (left) reference binds only to left value
- Const reference binds to right values too

```
int i = 5, j = 6; double d = 7.0;
```

```
swap(i, j); // ok
```

```
swap(i, 7); // error: could not bind reference to 7
```

```
int &ir1 = 7; // error: could not bind reference to 7
```

```
swap(i, d); // error: int(d) creates non-left value
```

```
int &ir2 = int(3.14); // error: int(3.14) creates non-left value
```

```
const int &ir3 = 7; // ok, lifetime extension
```

```
const int &ir4 = int(3.14); // ok, lifetime extension
```

# Returning with reference

- By default C++ functions return with copy
- Returning reference just binds the function result to an object

```
int f1()
{
    int i = 5;
    return i;    // ok, copies i before evaporating
}
```

```
int &f2()
{
    int i = 5;
    return i;    // oops, binds to evaporating i
}
```

# Usage example

```
class date
{
public:
    date& setYear(int y) { _year = y; return *this; }
    date& setMonth(int m) { _month = m; return *this; }
    date& setDay(int d) { _day = d; return *this; }

    date& operator++() { ++_day; return *this; }
    date operator++(int) // should return temporary
        { date curr(*this); ++_day; return curr; }

private:
    int _year;
    int _month;
    int _day;
};

void f()
{
    date d;
    ++d.setYear(2011).setMonth(11).setDay(11); // still left value
}
```

# Usage example

```
template <typename T>
class matrix
{
public:
    T& operator()(int i, int j)          { return v[i*cols+j]; }
    const T& operator()(int i, int j) const { return v[i*cols+j]; }
    matrix& operator+=( const matrix& other)
    {
        for (int i = 0; i < cols*rows; ++i)
            v[i] += other.v[i];
        return *this;
    }
private:
    // ...
    T* v;
};

template <typename T> matrix<T> // returns value
operator+(const matrix<T>& left, const matrix<T>& right)
{
    matrix<T> result(left); // copy constructor
    result += right;
    return result;
}
```

# Left vs right value

- Assignment in earlier languages work the following way:  
<variable> = <expression>, like `x = a+5;`
- In C/C++ however it can be:  
<expression> = <expression>, like `*++ptr = *++qtr;`
- But not all expressions are valid, like `a+5 = x;`
  
- An **lvalue** is an expression that refers to a memory location and allows us to take the address of that memory location via the `&` operator. An **rvalue** is an expression that is not an lvalue
  
- A rigorous definition of lvalue and rvalue: <http://accu.org/index.php/journals/227>

# Left value vs. right value

```
int i = 42;  
int &j = i;  
int *p = &i;
```

```
  i = 99;  
  j = 88;  
 *p = 77;
```

```
int *fp() { return &i; } // returns pointer to i: lvalue  
int &fr() { return i; }  // returns reference to i: lvalue
```

```
*fp() = 66; // i = 66  
fr() = 55;  // i = 55
```

```
// rvalues:
```

```
int f() { int k = i; return k; } // returns rvalue
```

```
  i = f(); // ok  
  p = &f(); // bad: can't take address of rvalue  
f() = i;    // bad: can't use rvalue on left-hand-side
```

# Value semantics

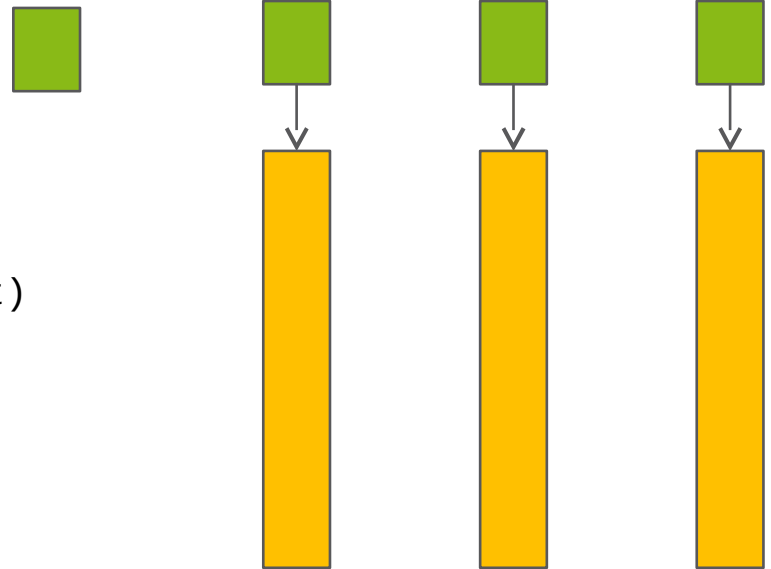
- C++ has value semantics
  - Clear separation of memory areas
  - Significant performance loss when copying large objects
  - This can lead to improper use of (smart) pointers



# Value semantics

```
class Array
{
public:
    Array (const Array&);
    Array& operator=(const Array&);
    ~ Array ();
private:
    double *val;
};
Array operator+(const Array& left,const Array& right)
{
    Array res = left;
    res += right;
    return res;
}

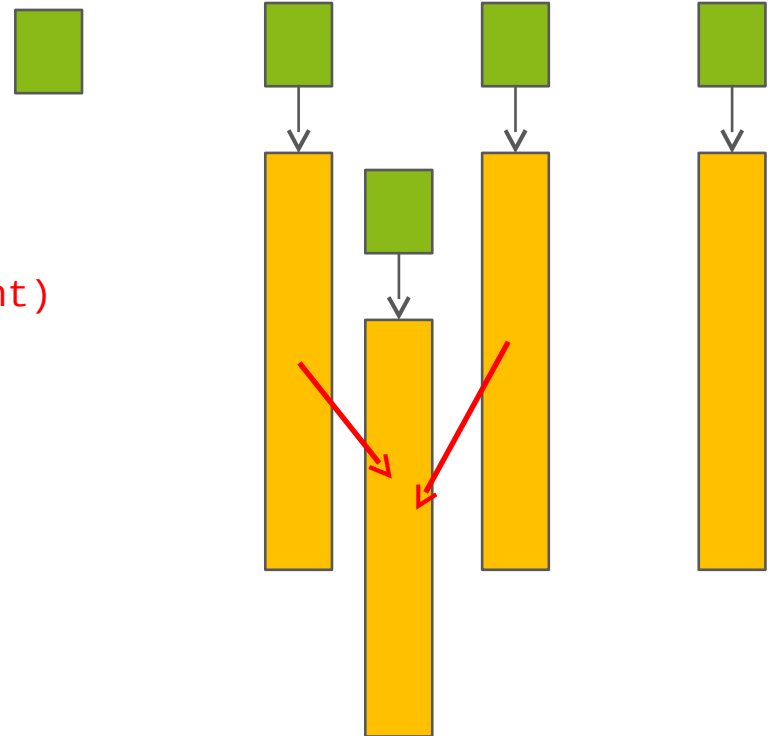
void f()
{
    Array b, c, d;
    ...
    Array a = b + c + d;
}
```



# Value semantics

```
class Array
{
public:
    Array (const Array&);
    Array& operator=(const Array&);
    ~ Array ();
private:
    double *val;
};
Array operator+(const Array& left, const Array& right)
{
    Array res = left;
    res += right;
    return res;
}

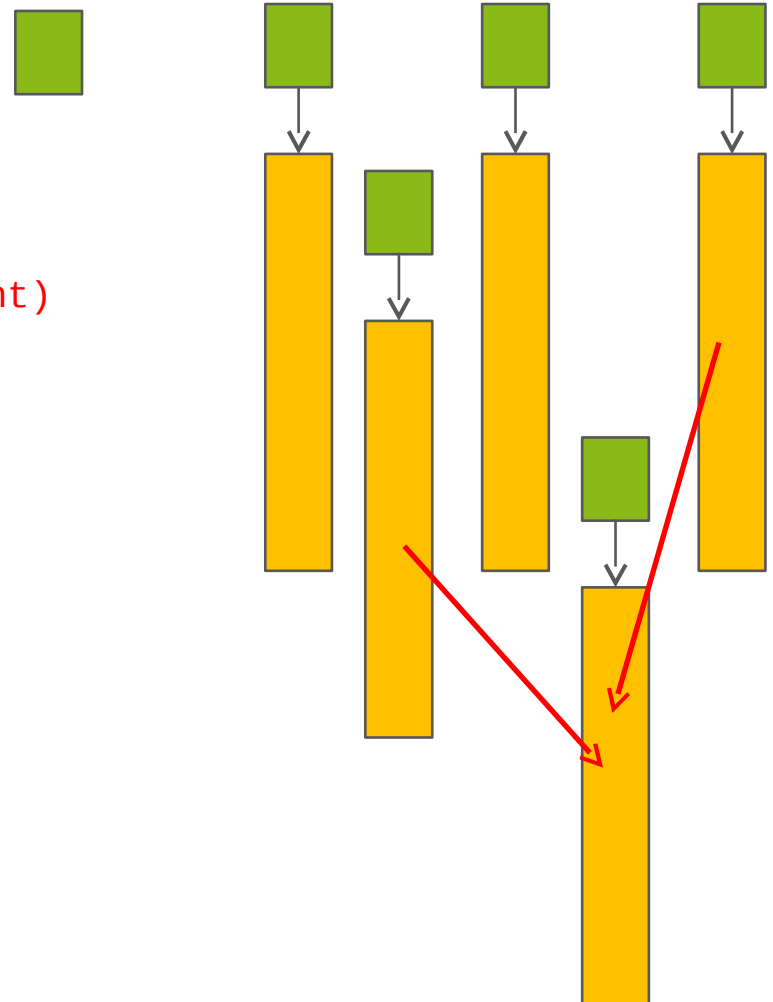
void f()
{
    Array b, c, d;
    ...
    Array a = b + c + d;
}
```



# Value semantics

```
class Array
{
public:
    Array (const Array&);
    Array& operator=(const Array&);
    ~ Array ();
private:
    double *val;
};
Array operator+(const Array& left, const Array& right)
{
    Array res = left;
    res += right;
    return res;
}

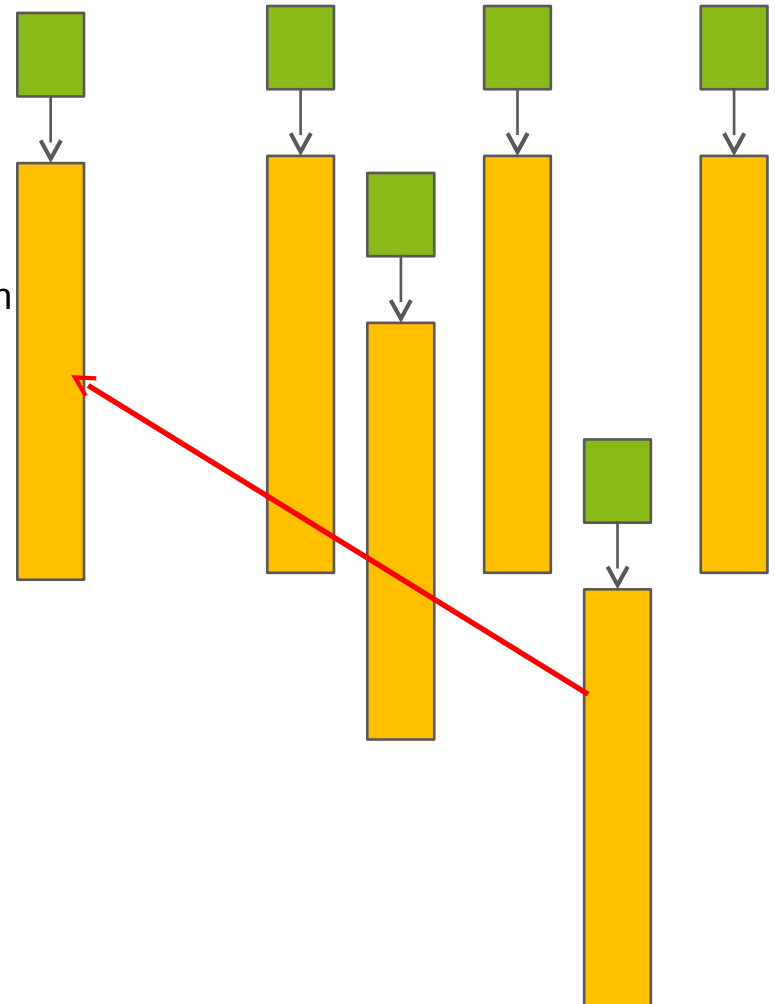
void f()
{
    Array b, c, d;
    ...
    Array a = b + c + d;
}
```



# Value semantics

```
class Array
{
public:
    Array (const Array&);
    Array& operator=(const Array&);
    ~ Array ();
private:
    double *val;
};
Array operator+(const Array& left, const Array& right)
{
    Array res = left;
    res += right;
    return res;
}

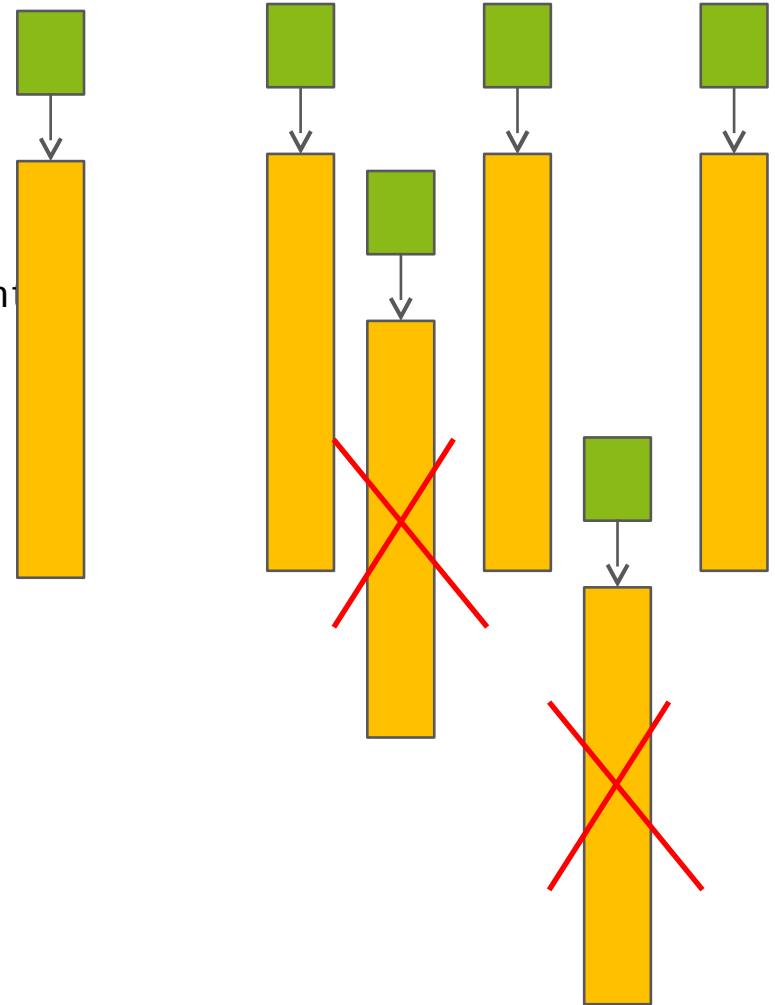
void f()
{
    Array b, c, d;
    ...
    Array a = b + c + d;
}
```



# Value semantics

```
class Array
{
public:
    Array (const Array&);
    Array& operator=(const Array&);
    ~ Array ();
private:
    double *val;
};
Array operator+(const Array& left, const Array& right)
{
    Array res = left;
    res += right;
    return res;
}

void f()
{
    Array b, c, d;
    ...
    Array a = b + c + d;
}
```



# Move semantics

```
class Array
{
public:
    Array (const Array&);
    Array& operator=(const Array&);
    ~ Array ();
private:
    double *val;
};
Array operator+(const Array& left, const Array& right)
{
    Array res = left;
    res += right;
    return res;
}

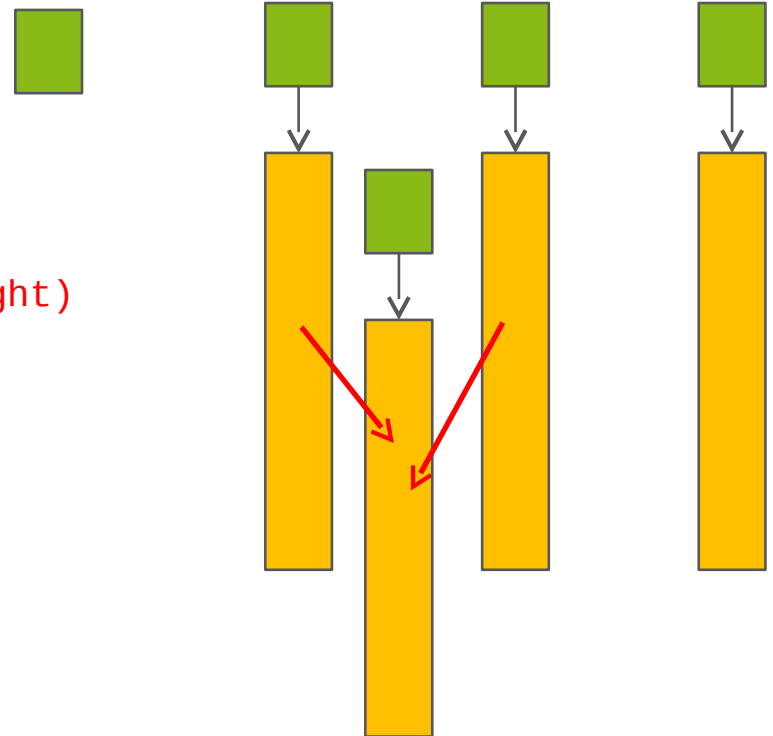
void f()
{
    Array b, c, d;
    ...
    Array a = b + c + d;
}
```



# Move semantics

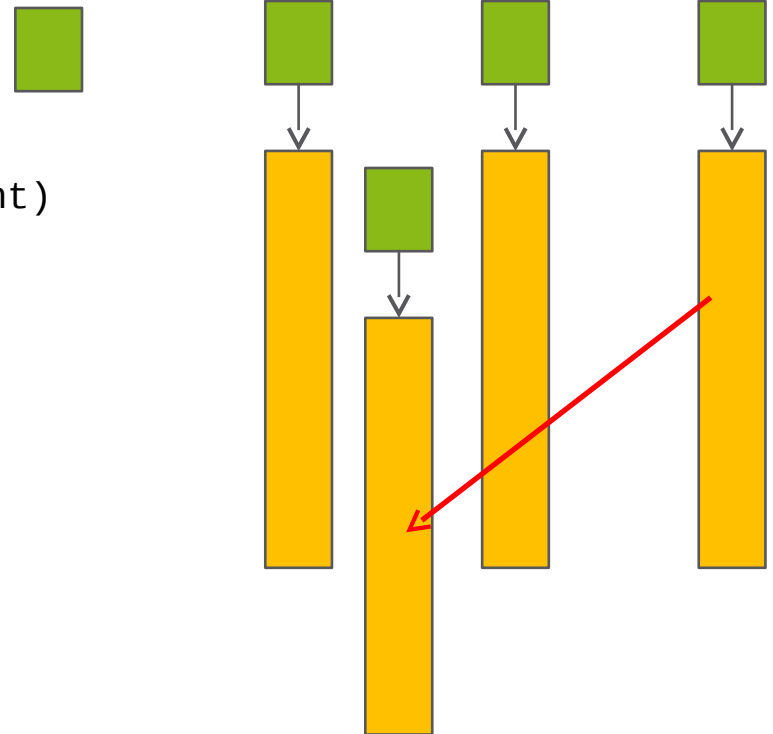
```
class Array
{
public:
    Array (const Array&);
    Array& operator=(const Array&);
    ~ Array ();
private:
    double *val;
};
Array operator+(const Array& left, const Array& right)
{
    Array res = left;
    res += right;
    return res;
}

void f()
{
    Array b, c, d;
    ...
    Array a = b + c + d;
}
```



# Move semantics

```
class Array
{
public:
    Array (const Array&);
    Array& operator=(const Array&);
    ...
};
Array operator+(const Array& left, const Array& right)
{
    Array res = left;
    res += right;
    return res;
}
Array operator+(Array&& left, const Array& right)
{
    left += right;
    return left;
}
void f()
{
    Array b, c, d;
    ...
    Array a = b + c + d;
}
```



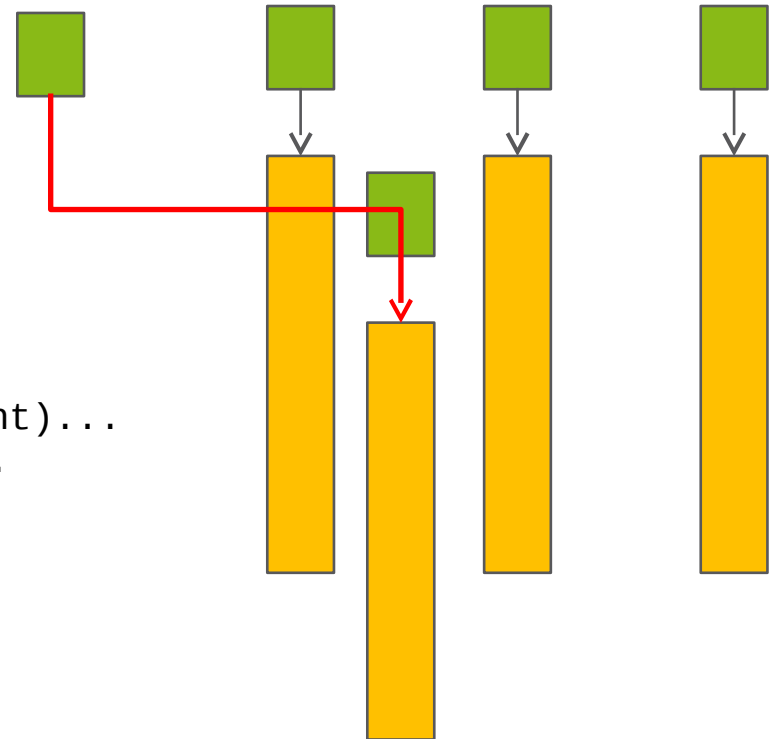


# Move semantics

```
class Array
{
public:
    Array (const Array&);
    Array (Array&&);
    Array& operator=(const Array&);
    Array& operator=(Array&&);
    ~ Array ();
private:
    double *val;
};
```

```
Array operator+(const Array& left, const Array& right)...
Array operator+(Array&& left, const Array& right)...
```

```
void f()
{
    Array b, c, d;
    ...
    Array a = b + c + d;
}
```



# Right value reference

- For overloadind, we need a new type
  - Reference type for performance reasons
  - Overload resolution should prefer this new type on rvalue objects

```
void f(X& arg_) // lvalue reference parameter  
void f(X&& arg_) // rvalue reference parameter
```

```
X x;  
X g();
```

```
f(x); // lvalue argument --> f(X&)  
f(g()); // rvalue argument --> f(X&&)
```

# Move semantics

- Move semantics
  - Instead of copying **steal** the resources
  - Leave the other object in a **destructible state**
  - Rule of three becomes rule of five
  - All standard library components were extended
- Reverse compatibility
  - If we implement the old-style member functions with lvalue reference but do not implement the rvalue reference overloading versions  
we keep the old behaviour -> gradually move to move semantics.
  - If we implement only rvalue operations we cannot call these on lvalues -> no default copy ctor or **operator=** will be generated.
- Serious performance gain
  - Except some rare RVO situations

# Special memberfunctions

```
class X
{
public:
    X(const X& rhs);
    X(X&& rhs);

    X& operator=(const X& rhs);    // = default or = delete
    X& operator=(X&& rhs);
private:
    // ...
};
X& X::operator=(const X& rhs)
{
    // free old resources than allocate and copy resource from rhs
    return *this;
}
X& X::operator=(X&& rhs) // draft version, will be revised
{
    // free old resources than move resource from rhs
    // leave rhs in a valid, destructable state
    return *this;
}
```

# Generation of special memberfunc.

1. The two **copy operations** (copy constructor and copy assignment) **are independent**. Declaring copy constructor does not prevent compiler to generate copy assignment (and vice versa). (same as in C++98)
2. **Move operations are not independent**. Declare either prevents the compiler to generate the other.
3. If any of the **copy operation is declared**, then **none of the move** operation will be generated.
4. If any of the **move operation is declared**, then **none of the copy** operation will be generated. This is the opposite rule of (3).
5. If a **destructor is declared**, then **none of the move** operation will be generated. Copy operations are still generated for reverse compatibility with C++98.
6. **Default constructor** generated only no constructor is declared. (same as in C++98)

# Move operations

- For reverse compatibility, move operations are generated only when
  - No copy operations are declared
  - No move operations are declared
  - No destructor is declared
- Function templates do not considered here
  - Templated copy constructor, assignment does not prevent move operation generations
  - Same rule since C++98 with copy operations
- Play safe!
  - ... it is easy...
- Really?

# A simple program

```
#include <iostream>
#include <vector>

struct S
{
    S() { a = ++cnt; }
    int a;
    static int cnt;
};
int S::cnt = 0;

int main()
{
    std::vector<S> sv(5);
    sv.push_back(S());

    for (std::size_t i = 0; i < sv.size(); ++i)
        std::cout << sv[i].a << " ";
    std::cout << std::endl;
}
```

# First amendment to the C++ standard

"The committee shall make no rule that prevents C++ programmers from shooting themselves in the foot."

quoted by Thomas Becker

[http://thbecker.net/articles/rvalue\\_references/section\\_04.html](http://thbecker.net/articles/rvalue_references/section_04.html)



# std::move

```
struct S
{
    S() { a = ++cnt; std::cout << "S() "; }
    S(const S& rhs) { a = rhs.a; std::cout << "copyCtr "; }
    S(S&& rhs) { a = rhs.a; std::cout << "moveCtr "; }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    S& operator=(S&& rhs) { a = rhs.a; std::cout << "move= "; return *this; }
    int a;
    static int cnt;
};
int S::cnt = 0;

template<class T>
void swap(T& a, T& b)
{
    T tmp(a);
    a = b;
    b = tmp;
}
int main()
{
    S a, b;
    swap( a, b);
}
```

# std::move

```
struct S
{
    S() { a = ++cnt; std::cout << "S() "; }
    S(const S& rhs) { a = rhs.a; std::cout << "copyCtr "; }
    S(S&& rhs) { a = rhs.a; std::cout << "moveCtr "; }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    S& operator=(S&& rhs) { a = rhs.a; std::cout << "move= "; return *this; }
    int a ;
    static int cnt;
};
int S::cnt = 0;

template<class T>
void swap(T& a, T& b)
{
    T tmp(a);
    a = b;
    b = tmp;
}

int main()
{
    S a, b;
    swap( a, b);
}
```

\$ ./a.out

S() S() copyCtr copy= copy=

# std::move

```
struct S
{
    S() { a = ++cnt; std::cout << "S() "; }
    S(const S& rhs) { a = rhs.a; std::cout << "copyCtr "; }
    S(S&& rhs) { a = rhs.a; std::cout << "moveCtr "; }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    S& operator=(S&& rhs) { a = rhs.a; std::cout << "move= "; return *this; }
    int a ;
    static int cnt;
};
int S::cnt = 0;

template<class T>
void swap(T& a, T& b)
{
    T tmp(a);
    a = b;
    b = tmp;
}

int main()
{
    S a, b;
    swap( a, b);
}
```

\$ ./a.out

S() S() copyCtr copy= copy=

If it has a name: LVALUE

# std::move

```
struct S
{
    S() { a = ++cnt; std::cout << "S() "; }
    S(const S& rhs) { a = rhs.a; std::cout << "copyCtr "; }
    S(S&& rhs) { a = rhs.a; std::cout << "moveCtr "; }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    S& operator=(S&& rhs) { a = rhs.a; std::cout << "move= "; return *this; }
    int a ;
    static int cnt;
};
int S::cnt = 0;

template<class T>
void swap(T& a, T& b)
{
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}

int main()
{
    S a, b;
    swap( a, b);
}
```

# std::move

```
struct S
{
    S() { a = ++cnt; std::cout << "S() "; }
    S(const S& rhs) { a = rhs.a; std::cout << "copyCtr "; }
    S(S&& rhs) { a = rhs.a; std::cout << "moveCtr "; }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    S& operator=(S&& rhs) { a = rhs.a; std::cout << "move= "; return *this; }
    int a ;
    static int cnt;
};
int S::cnt = 0;

template<class T>
void swap(T& a, T& b)
{
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}

int main()
{
    S a, b;
    swap( a, b);
}
```

```
$. /a.out
S() S() moveCtr move= move=
```

# std::move(x)

- Right value reference cast
- Usually has positive effect of performance
  - Many standard lib function utilize right-value references
- Sometimes we have to use it
  - Movable non-copyable classes
  - std::unique\_ptr, std::fstream, std::thread
- Might be dangerous
  - A variable with name left with unspecified value

# Rvalue and constness

```
#include <iostream>

struct MyString {
    MyString() { std::cerr << "defCtor" << std::endl; }
    MyString(const MyString& rhs) { std::cerr << "copyCtor" << std::endl; }
    MyString(MyString&& rhs) { std::cerr << "moveCtor" << std::endl; }
};

class MoveConstr {
public:
    MoveConstr( MyString s) : value(std::move(s)) { }
private:
    MyString value;
};

int main() {
    MoveConstr mc(MyString());
    return 0;
}
```

# Most vexing parse

MoveConstr mc(MyString())

- Function declaration:
  - mc returning MoveConstr type
  - One parameter: a pointer to function returns MyString and having no parameter

```
$ clang++ -std=c++11 -Wvexing-parse m1.cpp
m1.cpp:21:16: warning: parentheses were disambiguated as a function declaration
      [-Wvexing-parse]
  MoveConstr mc(MyString());
                ^~~~~~
m1.cpp:21:17: note: add a pair of parentheses to declare a variable
  MoveConstr mc(MyString());
                ^
                (      )
1 warning generated.
```



# Rvalue and constness

```
#include <iostream>

struct MyString {
    MyString() { std::cerr << "defCtor" << std::endl; }
    MyString(const MyString& rhs) { std::cerr << "copyCtor" << std::endl; }
    MyString(MyString&& rhs) { std::cerr << "moveCtor" << std::endl; }
};

class MoveConstr {
public:
    MoveConstr( MyString s) : value(std::move(s)) { }
private:
    MyString value;
};

int main() {
    MoveConstr mc((MyString()));
    return 0;
}

$ ./a.out
defCtor
moveCtor
```

# Rvalue and constness

```
#include <iostream>

struct MyString {
    MyString() { std::cerr << "defCtor" << std::endl; }
    MyString(const MyString& rhs) { std::cerr << "copyCtor" << std::endl; }
    MyString(MyString&& rhs) { std::cerr << "moveCtor" << std::endl; }
};

class MoveConstr {
public:
    MoveConstr( const MyString s) : value(std::move(s)) {}
private:
    MyString value;
};

int main() {
    MoveConstr mc((MyString()));
    return 0;
}

$ ./a.out
defCtor
copyCtor
```

# Inheritance

```
class Base
{
public:
    Base(const Base& rhs); // non-move semantics
    Base(Base&& rhs);      // move semantics
};
class Derived : public Base
{
    Derived(const Derived& rhs); // non-move semantics
    Derived(Derived&& rhs);      // move semantics
};
Derived(Derived const & rhs) : Base(rhs) // non-move semantics
{
    // copy derived specific...
}
Derived(Derived&& rhs) : Base(rhs)      // move semantics
{
    // move derived specific...
}
```

# Inheritance

```
class Base
{
public:
    Base(const Base& rhs); // non-move semantics
    Base(Base&& rhs);      // move semantics
};
class Derived : public Base
{
    Derived(const Derived& rhs); // non-move semantics
    Derived(Derived&& rhs);      // move semantics
};
Derived(Derived const & rhs) : Base(rhs) // non-move semantics
{
    // copy derived specific...
}
Derived(Derived&& rhs) : Base(rhs)      // wrong!
{
    // move derived specific...
}
```

# Inheritance

```
class Base
{
public:
    Base(const Base& rhs); // non-move semantics
    Base(Base&& rhs);      // move semantics
};
class Derived : public Base
{
    Derived(const Derived& rhs); // non-move semantics
    Derived(Derived&& rhs);      // move semantics
};
Derived(Derived const & rhs) : Base(rhs) // non-move semantics
{
    // copy derived specific...
}
Derived(Derived&& rhs) : Base(std::move(rhs)) // good, calls Base(Base&& rhs)
{
    // move derived specific...
}
```

# Vector

```
#include <iostream>
#include <vector>

struct S
{
    S() { a = ++cnt; std::cout << "S() "; }
    S(const S& rhs) { a = rhs.a; std::cout << "copyCtr "; }
    S(S&& rhs) { a = rhs.a; std::cout << "moveCtr "; }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy="; return *this; }
    S& operator=(S&& rhs) { a = rhs.a; std::cout << "move="; return *this; }
    int a;
    static int cnt;
};
int S::cnt = 0;

int main()
{
    std::vector<S> sv(5);
    sv.push_back(S());

    for (std::size_t i = 0; i < sv.size(); ++i)
        std::cout << sv[i].a << " ";
    std::cout << std::endl;
}
```

# Vector

```
#include <iostream>
#include <vector>

struct S
{
    S() { a = ++cnt; std::cout << "S() "; }
    S(const S& rhs) { a = rhs.a; std::cout << "copyCtr "; }
    S(S&& rhs) { a = rhs.a; std::cout << "moveCtr "; }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    S& operator=(S&& rhs) { a = rhs.a; std::cout << "move= "; return *this; }
    int a;
    static int cnt;
};
int S::cnt = 0;

int main()
{
    std::vector<S> sv(5);
    sv.push_back(S());

    for (std::size_t i = 0; i < sv.size(); ++i)
        std::cout << sv[i].a << " ";
    std::cout << std::endl;
}
```

```
$ g++ -std=c++11 vec.cpp && ./a.out
S() S() S() S() S() moveCtr
copyCtr copyCtr copyCtr copyCtr copyCtr
1 2 3 4 5 6
```

# Vector

```
#include <iostream>
#include <vector>

struct S
{
    S() { a = ++cnt; std::cout << "S() "; }
    S(const S& rhs) { a = rhs.a; std::cout << "copyCtr "; }
    S(S&& rhs) { a = rhs.a; std::cout << "moveCtr "; }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy="; return *this; }
    S& operator=(S&& rhs) { a = rhs.a; std::cout << "move="; return *this; }
    int a;
    static int cnt;
};
int S::cnt = 0;

int main()
{
    std::vector<S> sv(5);
    sv.push_back(S());

    for (std::size_t i = 0; i < sv.size(); ++i)
        std::cout << sv[i].a << " ";
    std::cout << std::endl;
}
```

```
$ g++ -std=c++11 vec.cpp && ./a.out
S() S() S() S() S() moveCtr
copyCtr copyCtr copyCtr copyCtr copyCtr
1 2 3 4 5 6
```



# Vector

```
#include <iostream>
#include <vector>

struct S
{
    S() { a = ++cnt; std::cout << "S() "; }
    S(const S& rhs) { a = rhs.a; std::cout << "copyCtr "; }
    S(S&& rhs) noexcept { a = rhs.a; std::cout << "moveCtr "; }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy="; return *this; }
    S& operator=(S&& rhs) { a = rhs.a; std::cout << "move="; return *this; }
    int a;
    static int cnt;
};
int S::cnt = 0;

int main()
{
    std::vector<S> sv(5);
    sv.push_back(S());

    for (std::size_t i = 0; i < sv.size(); ++i)
        std::cout << sv[i].a << " ";
    std::cout << std::endl;
}
```

# Vector

```
#include <iostream>
#include <vector>

struct S
{
    S() { a = ++cnt; std::cout << "S() "; }
    S(const S& rhs) { a = rhs.a; std::cout << "copyCtr "; }
    S(S&& rhs) noexcept { a = rhs.a; std::cout << "moveCtr "; }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy="; return *this; }
    S& operator=(S&& rhs) { a = rhs.a; std::cout << "move="; return *this; }
    int a;
    static int cnt;
};
int S::cnt = 0;

int main()
{
    std::vector<S> sv(5);
    sv.push_back(S());

    for (std::size_t i = 0; i < sv.size(); ++i)
        std::cout << sv[i].a << " ";
    std::cout << std::endl;
}
```

```
$ g++ -std=c++11 vec.cpp && ./a.out
```

```
S() S() S() S() S()
```

```
moveCtr moveCtr moveCtr moveCtr moveCtr moveCtr
```

```
1 2 3 4 5 6
```

# std::move(b,e,b2)

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>

struct S
{
    S() { a = ++cnt; std::cout << "S() "; }
    S(const S& rhs) { a = rhs.a; std::cout << "copyCtr "; }
    S(S&& rhs) noexcept { a = rhs.a; std::cout << "moveCtr "; }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    S& operator=(S&& rhs) { a = rhs.a; std::cout << "move= "; return *this; }
    int a ;
    static int cnt;
};
int S::cnt = 0;

int main()
{
    std::list<S> sl = { S(), S(), S(), S(), S() };
    std::vector<S> sv(5);
    std::move( sl.begin(), sl.end(), sv.begin());

    for (const S& s : sv)
        std::cout << s.a << " ";
    std::cout << std::endl;
}
```

# std::move(b,e,b2)

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>

struct S
{
    S() { a = ++cnt; std::cout << "S() "; }
    S(const S& rhs) { a = rhs.a; std::cout << "copyCtr "; }
    S(S&& rhs) noexcept { a = rhs.a; std::cout << "moveCtr "; }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    S& operator=(S&& rhs) { a = rhs.a; std::cout << "move= "; return *this; }
    int a ;
    static int cnt;
};
int S::cnt = 0;

int main()
{
    std::list<S> sl = { S(), S(), S(), S(), S() };
    std::vector<S> sv(5);
    std::move( sl.begin(), sl.end(), sv.begin());

    for (const S& s : sv)
        std::cout << s.a << " ";
    std::cout << std::endl;
}
```

```
$ g++ -std=c++11 && ./a.out
S() S() S() S() S() copyCtr
copyCtr copyCtr copyCtr copyCtr
S() S() S() S() S()
move= move= move= move= move=
1 2 3 4 5
```

# std::move(b,e,b2)

```
#include <iostream>
#include <vector>
#include <set>
#include <algorithm>

struct S
{
    S() { a = ++cnt; std::cout << "S() "; }
    S(const S& rhs) { a = rhs.a; std::cout << "copyCtr "; }
    S(S&& rhs) noexcept { a = rhs.a; std::cout << "moveCtr "; }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    S& operator=(S&& rhs) { a = rhs.a; std::cout << "move= "; return *this; }
    int a ;
    static int cnt;
};
int S::cnt = 0;
bool operator<(const S& x, const S& y) { return x.a < y.a; }
int main()
{
    std::set<S> s1 = { S(), S(), S(), S(), S() };
    std::vector<S> sv(5);
    std::move( s1.begin(), s1.end(), sv.begin());

    for (const S& s : sv)
        std::cout << s.a << " ";
    std::cout << std::endl;
}
```

# std::move(b,e,b2)

```
#include <iostream>
#include <vector>
#include <set>
#include <algorithm>

struct S
{
    S() { a = ++cnt; std::cout << "S() "; }
    S(const S& rhs) { a = rhs.a; std::cout << "copyCtr "; }
    S(S&& rhs) noexcept { a = rhs.a; std::cout << "moveCtr "; }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    S& operator=(S&& rhs) { a = rhs.a; std::cout << "move= "; return *this; }
    int a ;
    static int cnt;
};
int S::cnt = 0;
bool operator<(const S& x, const S& y) { return x.a < y.a; }
int main()
{
    std::set<S> s1 = { S(), S(), S(), S(), S() };
    std::vector<S> sv(5);
    std::move( s1.begin(), s1.end(), sv.begin());

    for (const S& s : sv)
        std::cout << s.a << " ";
    std::cout << std::endl;
}
```

```
$ g++ -std=c++11 && ./a.out
S() S() S() S() S() copyCtr
copyCtr copyCtr copyCtr copyCtr
S() S() S() S() S()
copy= copy= copy= copy= copy=
1 2 3 4 5
```

# RVO

```
std::vector<double> fill();

int main()
{
    std::vector<double> vd = fill();
    // ...
}

std::vector<double> fill()
{
    std::vector<double> local;
    // fill the elements
    return local;    // return or move?
}
```

David Abrahams has an article on this:

<http://cpp-next.com/archive/2009/08/want-speed-pass-by-value/>

# Universal reference

Universal reference is used in case of type deduction

```
class X;
Void f(X&& param)           // rvalue reference
X&& var1 = X();             // rvalue reference
auto&& var2 = var1;        // NOT rvalue reference: universal reference
template <typename T>
void f(std::vector<T>&& param); // rvalue reference (1)
template <typename T>
void f(T&& param);          // NOT rvalue reference: universal reference (2)
template <typename T>
void f(const T&& param);    // rvalue reference

X var;
f(var);                     // lvalue passed: param type is: X& (2)
f(std::move(var));         // rvalue passed: param type is: X&& (2)
std::vector<int> v;
f(v);                      // syntax error: can't bind lvalue to rvalue (1)
```



# Universal reference

Universal reference is used in case of type deduction

```
template <class T, class Allocator = allocator<T>>
class vector
{
public:
    void push_back(T&& x); // rvalue reference, no type deduction here
    // ...
};
```

```
template <class T, class Allocator = allocator<T>>
class vector
{
public:
    template <class... Args>
    void emplace_back(Args&&... args); // universal reference, type deduction
    // ...
};
```

# Perfect forwarding

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg arg)
{
    return shared_ptr<T>(new T(arg));    // call T(arg) by value. Bad!
}
```

// A half-good solution is passing arg by reference:

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg& arg)
{
    return shared_ptr<T>(new T(arg));
}
```

But this does not work for rvalue parameters:

```
factory<X>(f());           // error if f() returns by value
factory<X>(42);           // error
```

# Perfect forwarding

```
// If f() called on "lvalue of A"  T --> A&      argument type --> A&  
// If f() called on "rvalue of A"  T --> A       argument type --> A
```

```
template<typename T, typename Arg>  
shared_ptr<T> factory(Arg&& arg)  
{  
    return shared_ptr<T>(new T(std::forward<Arg>(arg)));  
}  
template<class S>  
S&& forward(typename remove_reference<S>::type& a) noexcept  
{  
    return static_cast<S&&>(a);  
}
```

```
// Reference collapsing:
```

```
A& &    --> A&  
A& &&   --> A&  
A&& &   --> A&  
A&& &&  --> A&&
```

# Perfect forwarding

```
shared_ptr<A> factory(X&& arg)
{
    return shared_ptr<A>(new A(std::forward<X>(arg)));
}
X&& forward(X& a) noexcept // std::forward keeps move semantic
{
    return static_cast<X&&>(a);
}
```

```
template <typename T> // C++11
typename remove_reference<T>::type&&
std::move(T&& a) noexcept
{
    typedef typename remove_reference<T>::type&& RvalRef;
    return static_cast<RvalRef>(a);
}
template <typename T> // C++14
decltype(auto) std::move(T&& a)
{
    using ReturnT = remove_reference_t<T>&&;
    return static_cast<ReturnT>(a);
}
```

# Overloading on right value

```
T &      value(optional<T> &      par_);  
T &&    value(optional<T> &&    par_);  
T const& value(optional<T> const& par_);
```

But in object-oriented programming, sometimes we want to overload on the `this` parameter too.

```
template <typename T>  
class optional  
{  
    // ...  
    T&      value() &;  
    T&&    value() &&;  
    T const& value() const&;  
};
```