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 with 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;
};
```

23.3.2020

3/23/2020

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->*p3)();
}

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

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

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

"ABC" + "CDE"

s = s1 + s2 + s3 + s4 + s5

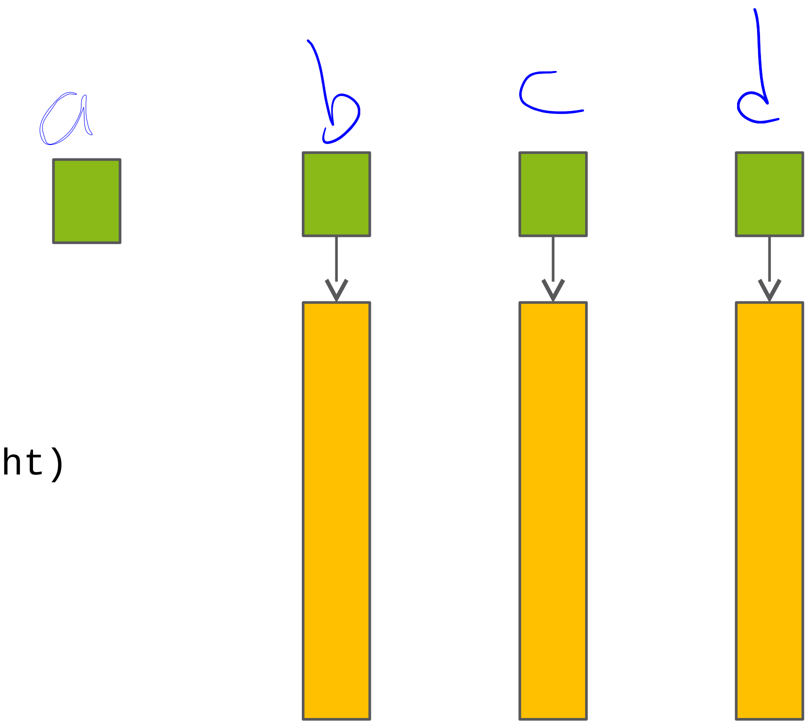
a = 7;

Mong m1, m2;

m1 = m2;

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;
}
```



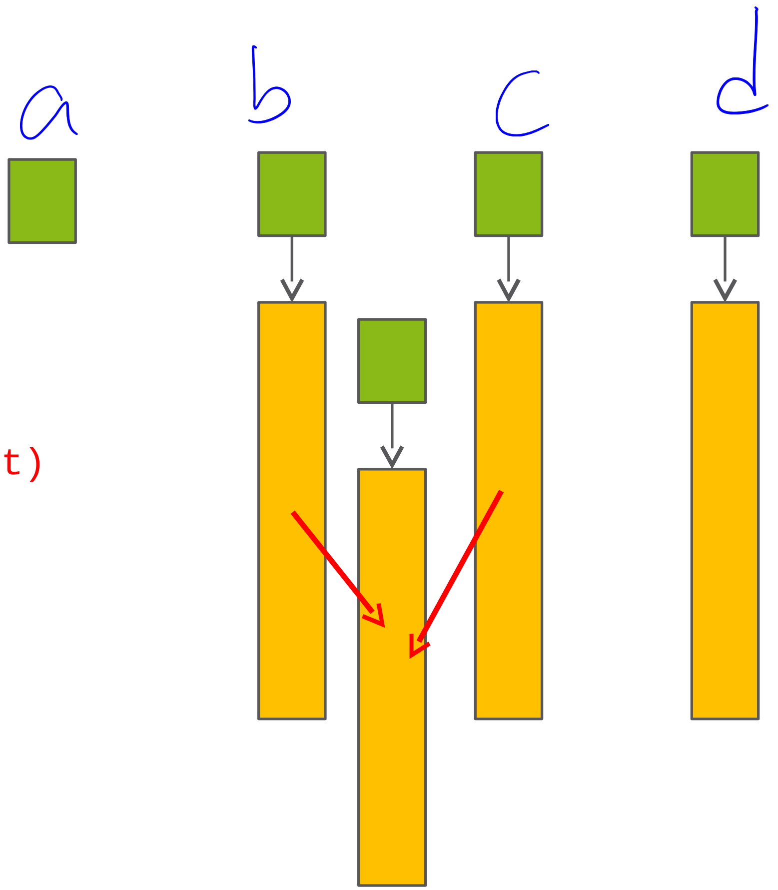
(1.0 7.2 9.1)

+ (1.5 0.2 3.1)

(2.5 7.4 12.2)

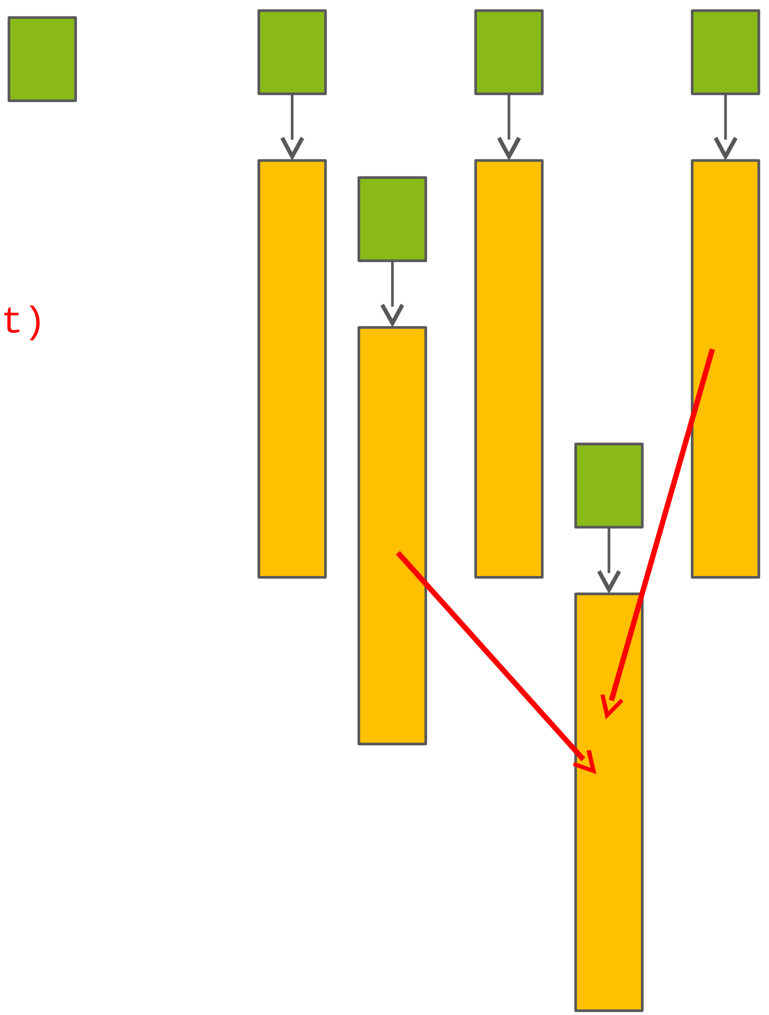
Value semantics

```
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;
}
```



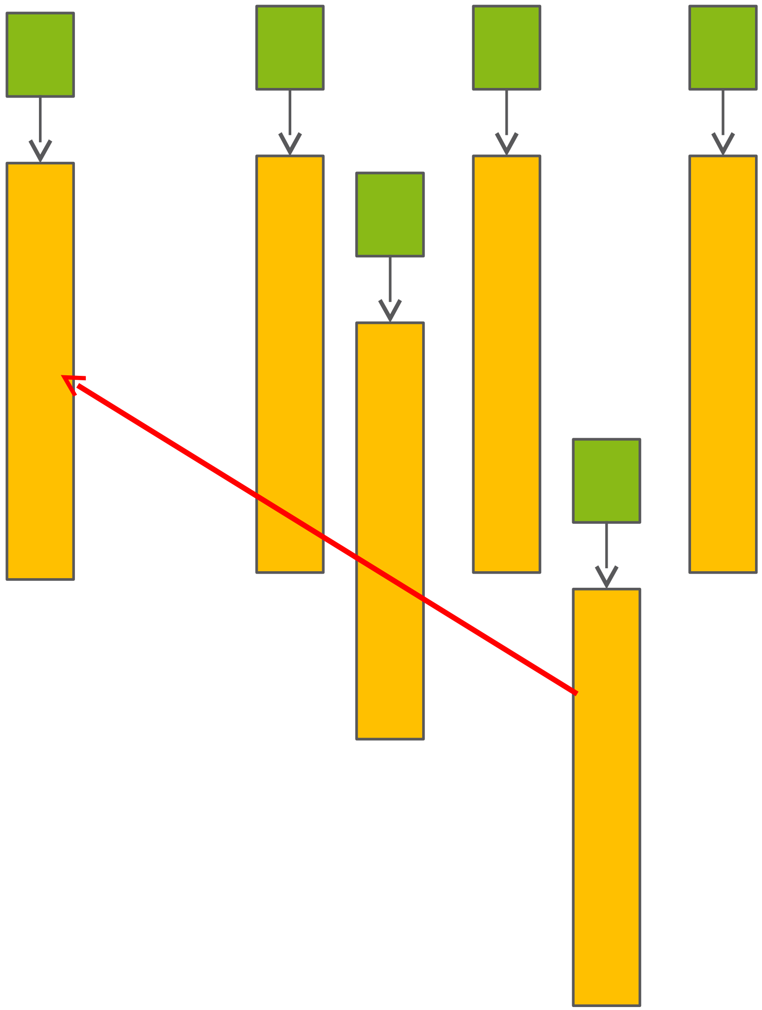
Value semantics

```
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;
}
```



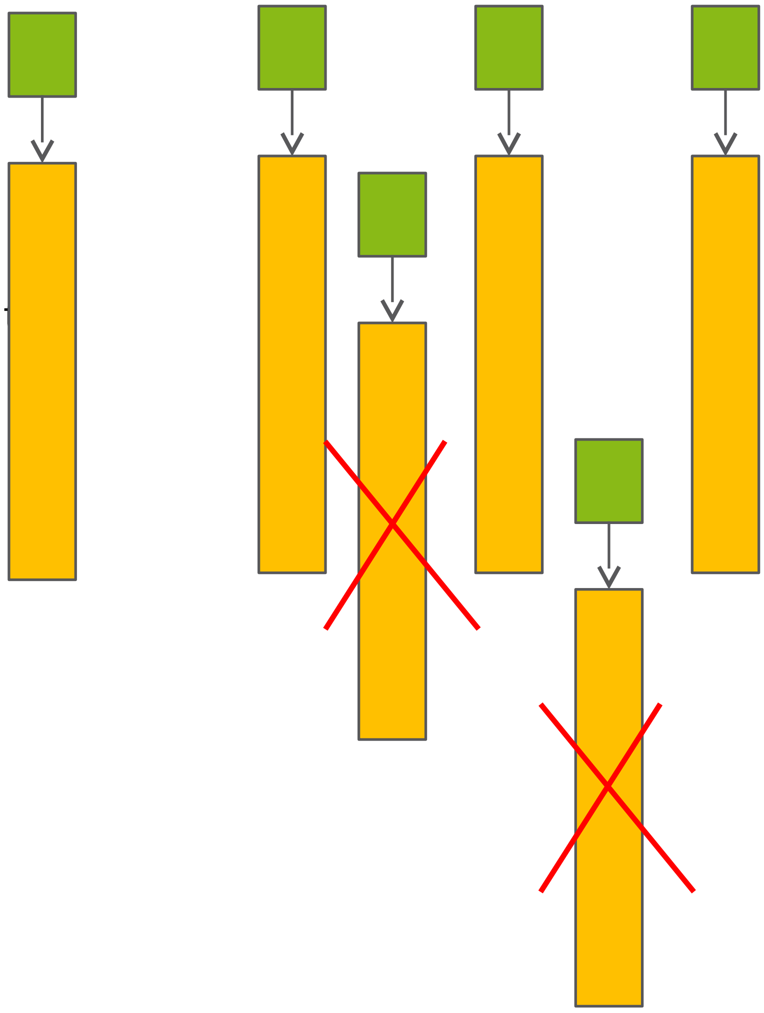
Value semantics

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

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



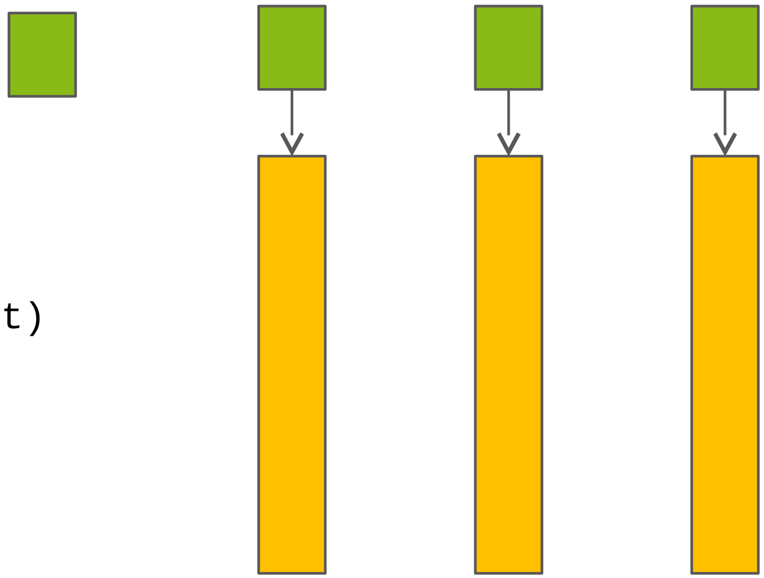
Value semantics

```
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;
}
```



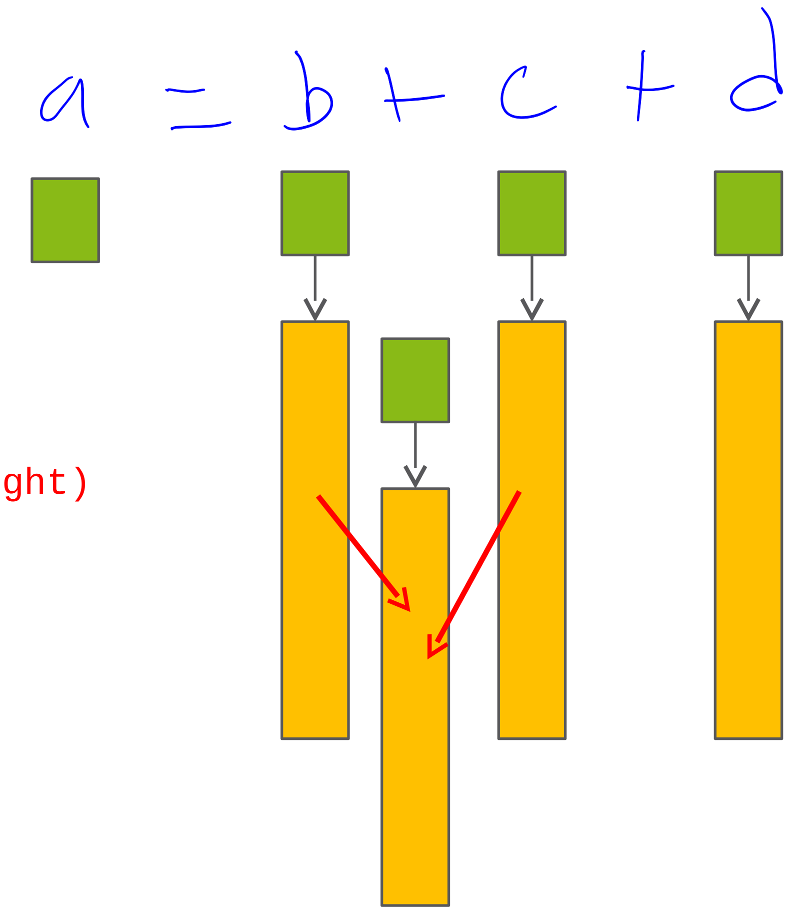
Move semantics

```
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;
}
```



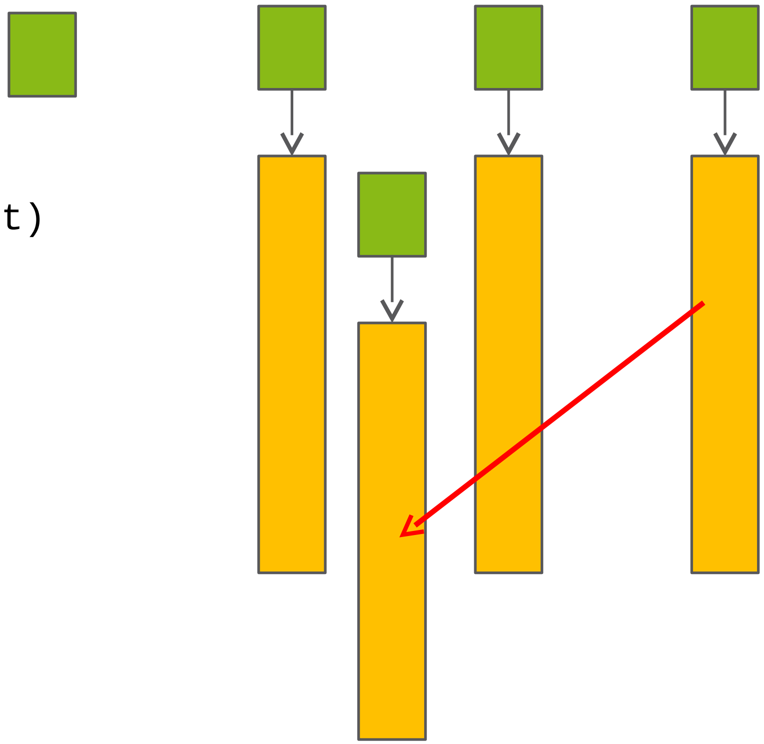
Move semantics

```
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

Big three

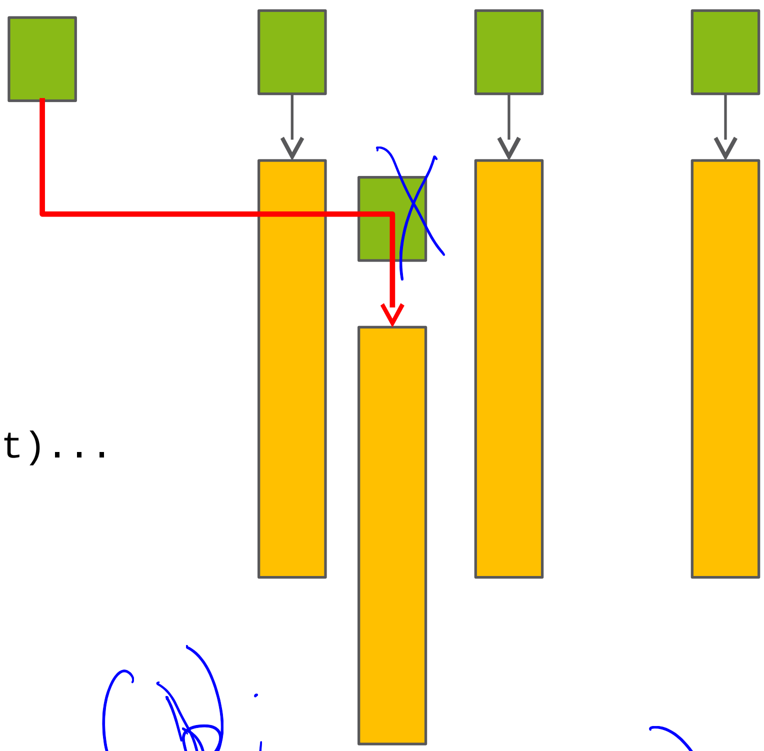
"Big" Five

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

move constr.

move assignment

a



```
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;
}
```

Array a(b);

Array a(f(x))

Right value reference

- For overloading, 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
void f(const X& arg_) // const lvalue 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

$a = f(x)$

- 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**, than **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

std::move

class

~~struct~~

```
struct S
{    public:
    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);
}
```

cnt $\boxed{2}$

$\boxed{S() \quad S()}$ $\boxed{\text{copyCtr}}$ $\boxed{\text{copy=}}$ $\boxed{\text{copy=}}$

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=
```

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);
}
```

void * p = ...

(int *) p

S() S() moveCtr move= move=

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

Vector

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

struct S      public:
{
    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;
}
```

class

S() S() S() S() S()

S() moveCtr

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() 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() 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() 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> 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 <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> 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()
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/

Forwarding/Universal reference

Forwarding/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 ReturnType = remove_reference_t<T>&&;
  return static_cast<ReturnType>(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&;
};
```