New C++ -- New Traps

Zoltán Porkoláb

gsd@inf.elte.hu
http://aszt.inf.elte.hu

WARMING UP

```
#include <iostream>

int main()
{
    int i = 1;
    std::cout << i << ++i << std::endl;
}
```

WARMING UP

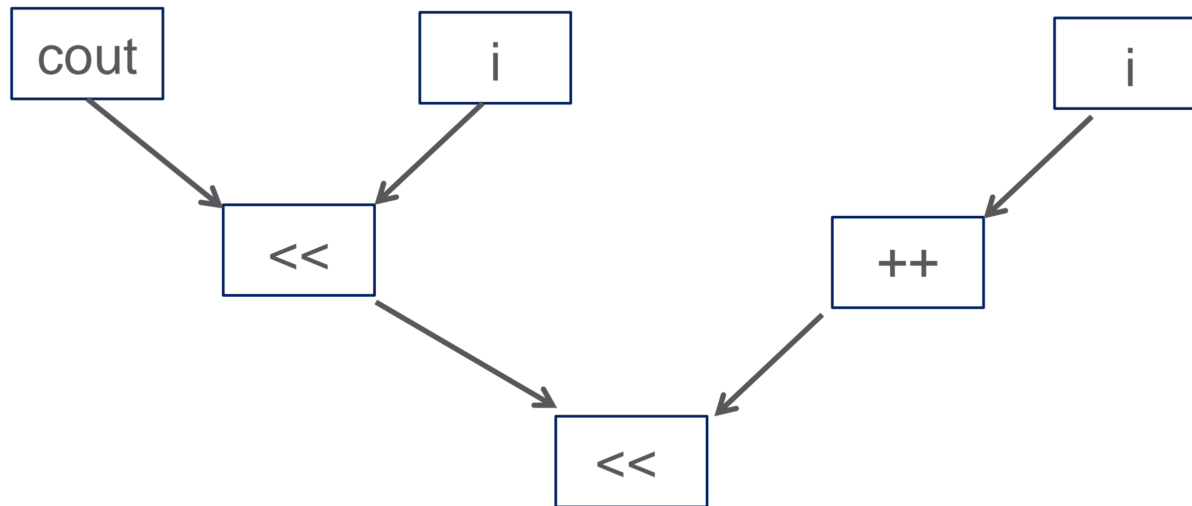
```
#include <iostream>

int main()
{
    int i = 1;
    std::cout << i << ++i << std::endl;
}

$ g++ plusplus.cpp  &&  ./a.out
12
```

WARMING UP

```
#include <iostream>

int main()
{
    int i = 1;
    std::cout << i << ++i << std::endl;
}

$ g++ plusplus.cpp  &&  ./a.out
22
```

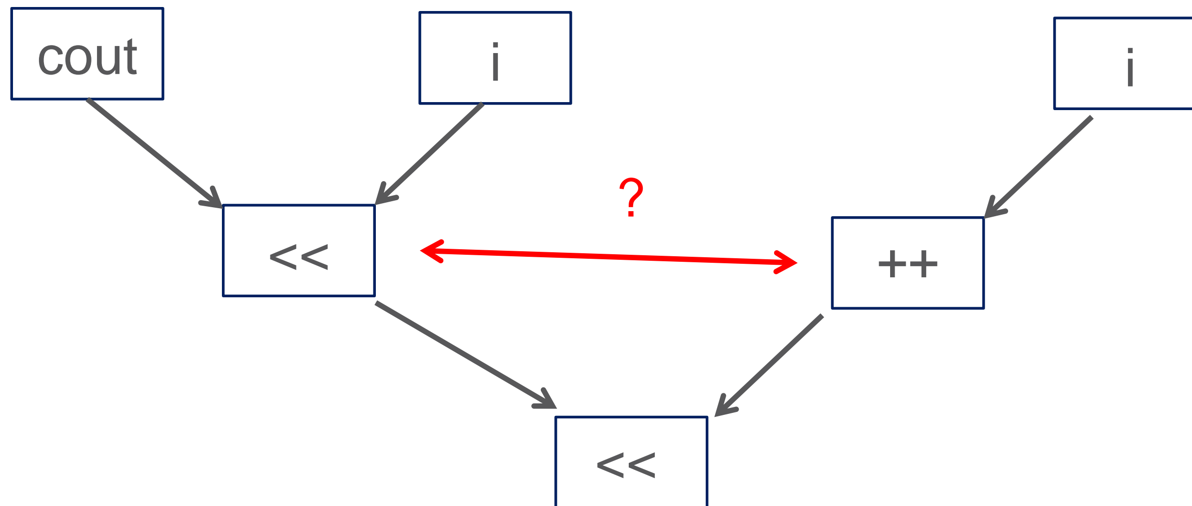
WARMING UP

```
#include <iostream>

int main()
{
    int i = 1;
    std::cout << i << ++i << std::endl;
}

$ g++ plusplus.cpp  &&  ./a.out
?2
```



WARMING UP

```
#include <iostream>

int main()
{
    int i = 1;
    std::cout << i << ++i << std::endl;
}

$ g++ plusplus.cpp  &&  ./a.out
?2
```



WARMING UP

```
#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;
}
```

WARMING UP

```
#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;
}

$ g++ v.cpp  &&  ./a.out
```

WARMING UP

```
#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;
}

$ g++ v.cpp  &&  ./a.out
1 1 1 1 1 2
```

WARMING UP

```
#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;
}
```

```
$ g++ v.cpp  &&  ./a.out
1 1 1 1 1 2
```

```
#include <vector>

explicit vector (size_type n,
                 const value_type& val = value_type(),
                 const allocator_type& alloc = allocator_type());
```

WARMING UP

```
#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;
}

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

WARMING UP

```
#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;
}

$ g++ -std=c++11 v.cpp  &&  ./a.out
1 2 3 4 5 6
```

WARMING UP

```
#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;
}
```

```
#include <vector>     // C++11

explicit vector (size_type n);

vector (size_type n,
        const value_type& val,
        const allocator_type& alloc = allocator_type());
```

```
$ g++ -std=c++11 v.cpp  &&  ./a.out
1 2 3 4 5 6
```

WARMING UP

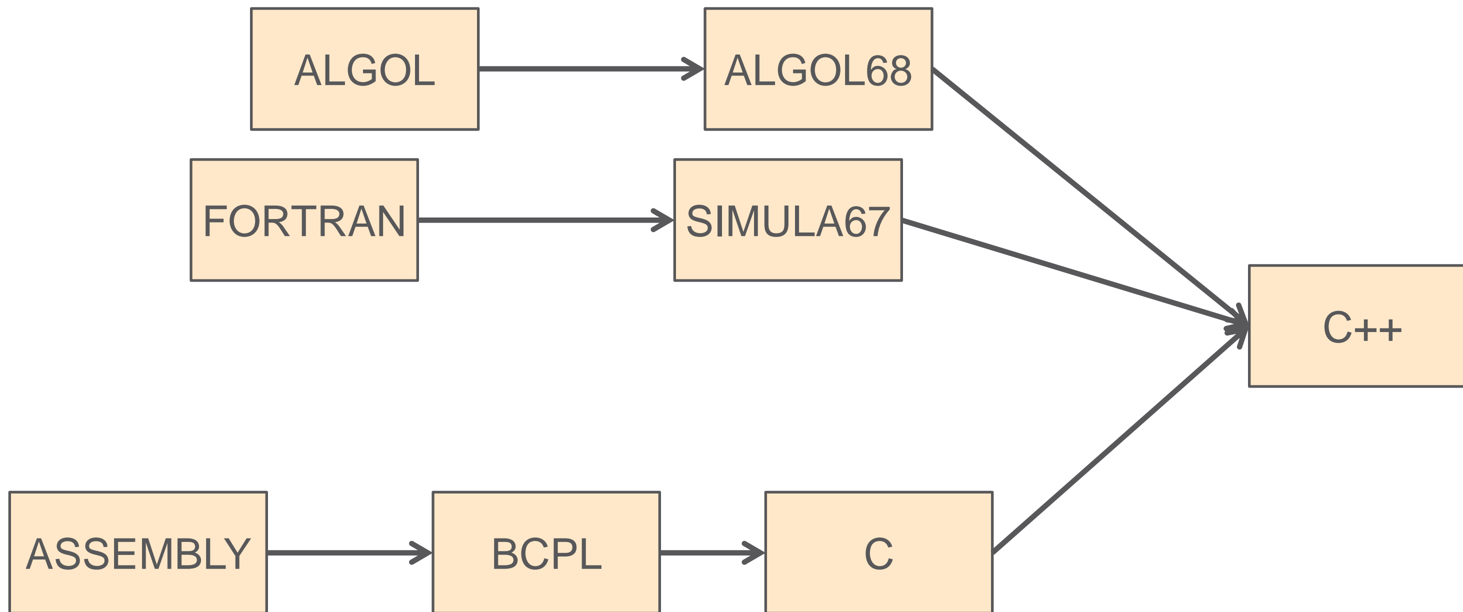
```
#include <iostream>
#include <vector>
#include <memory>

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

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

    for (std::size_t i = 0; i < sv.size(); ++i)
        if ( nullptr == v[i] )    std::cout << “nullptr ”;
        else                      std::cout << v[i]->a << “”;
     std::cout << std::endl;
}
$ g++ -std=c++11 v.cpp  &&  ./a.out
nullptr nullptr nullptr nullptr nullptr 1
```

```
#include <vector>     // C++11

explicit vector (size_type n);

          vector (size_type n,
                   const value_type& val,
                   const allocator_type& alloc = allocator_type());
```

AGENDA

- The past of C++
  - Origins and history
  - What are the C++ “invariants”
- C++ now
  - New features in C++11
  - Move semantics
  - C++11 issues
- Near future (C++14, C++17)
- Behind C++17

PAST OF C++



C++ DESIGN GOALS

- Type safety
- Resource safety
  - Not just memory, but all resources
- Performance
  - High performance/Real time applications
  - Low energy consumption
- Predictability
  - Large systems
  - Orthogonal features should work well together (mostly)
- Learnability
  - From expert friendly to novice friendly
- Readability
  - Special goal for C++11/14

C++11 (was C++0x)

- Runtime improvements
  - Rvalue references and Move semantics
  - Constexpr
  - New definition of POD (trivial class and standard layout)
- Usability
  - New memory model, thread locals
  - Better type inference (auto and decltype)
  - Lambda functions, range based for
  - Initializer lists, uniform initialization, delegated constructors
  - Template aliases, variadic templates, user defined literals
  - Overrides, final, strongly typed enums, static assert, attributes
  - Right angle bracket!!! vector<list<int>>

C++11 STANDARD LIBRARY

- Libraries from Boost
  - Hash tables (unordered_...)
  - Smart pointers (but unique_ptr instead of scoped_ptr)
  - Function objects and wrappers
  - Tuple, Array
  - Type traits
  - Regular expressions
- Other libraries
  - Threading facilities (thread, future, promise, mutex, guard …)
  - Math

LVALUE & RVALUE

- 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

LVALUE & RVALUE

› Lvalues

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

int *p = &i;
i = 99;
j = 88;
*p = 77;

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

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

LVALUE & RVALUE

› Lvalues

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

int *p = &i;
i = 99;
j = 88;
*p = 77;

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

*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
```

LVALUE & RVALUE

› Lvalues

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

int *p = &i;
i = 99;
j = 88;
*p = 77;

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

*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
```

› Rvalue reference

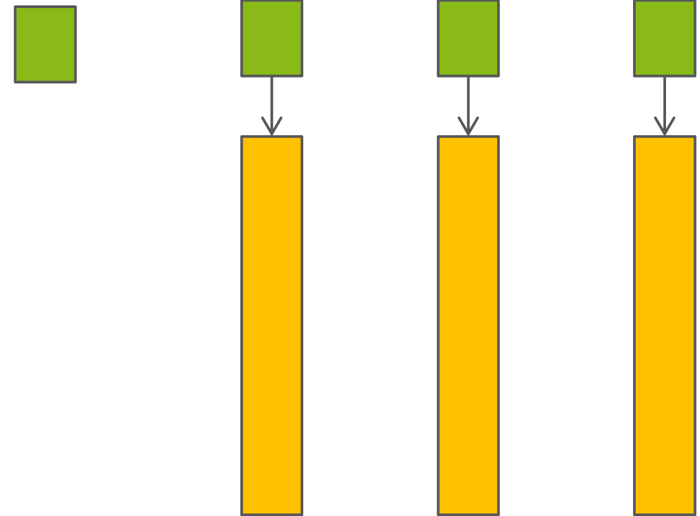
```
void f(X& arg_) // lvalue reference parameter
void f(X&& arg_) // rvalue reference parameter

X x;
X g();

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

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

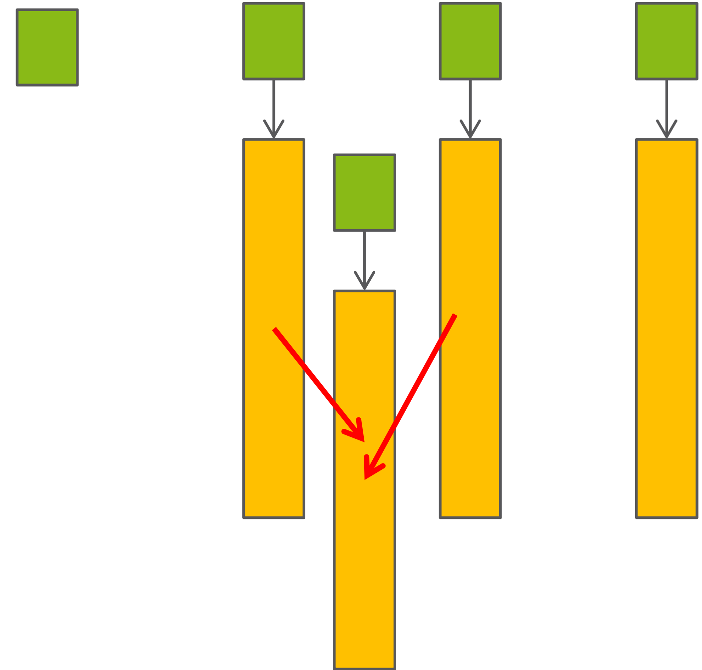
VALUE SEMANTICs (1)

```
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;
}
```



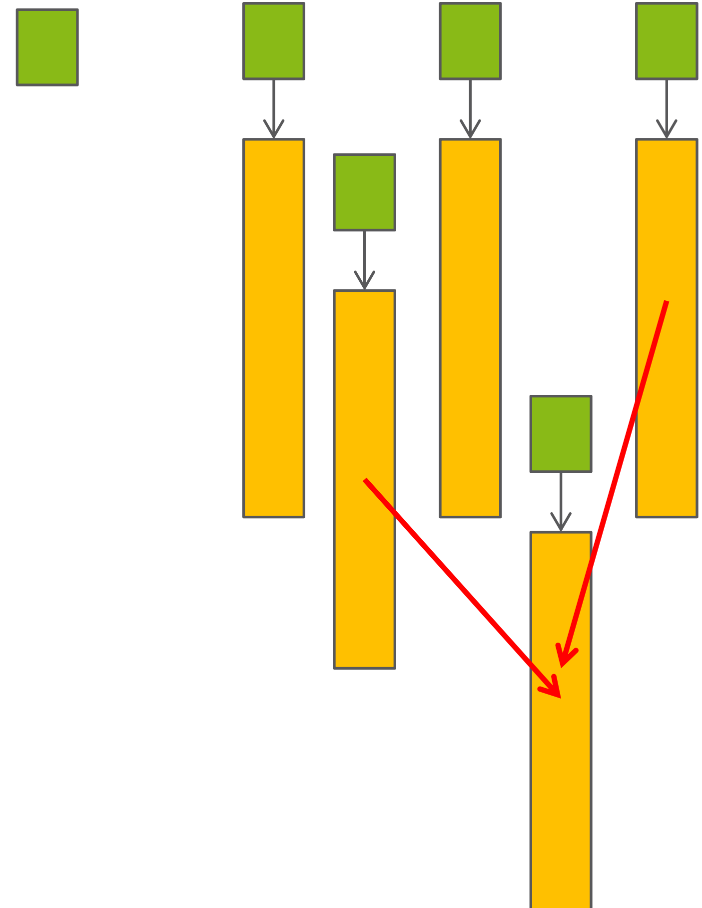
VALUE SEMANTICS (2)

```
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;
}
```



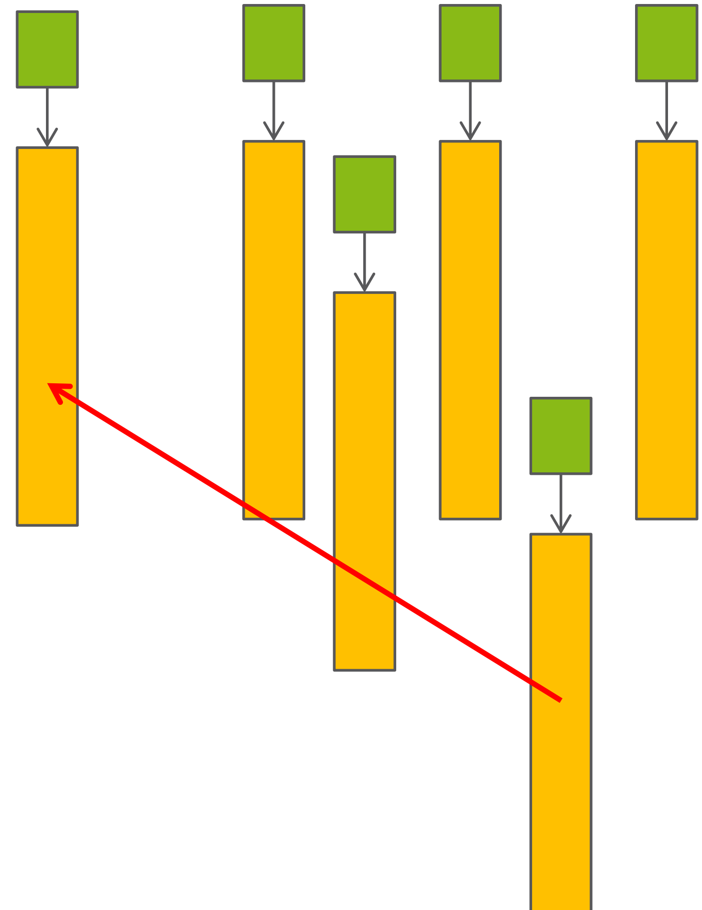
VALUE SEMANTICS (3)

```
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;
}
```



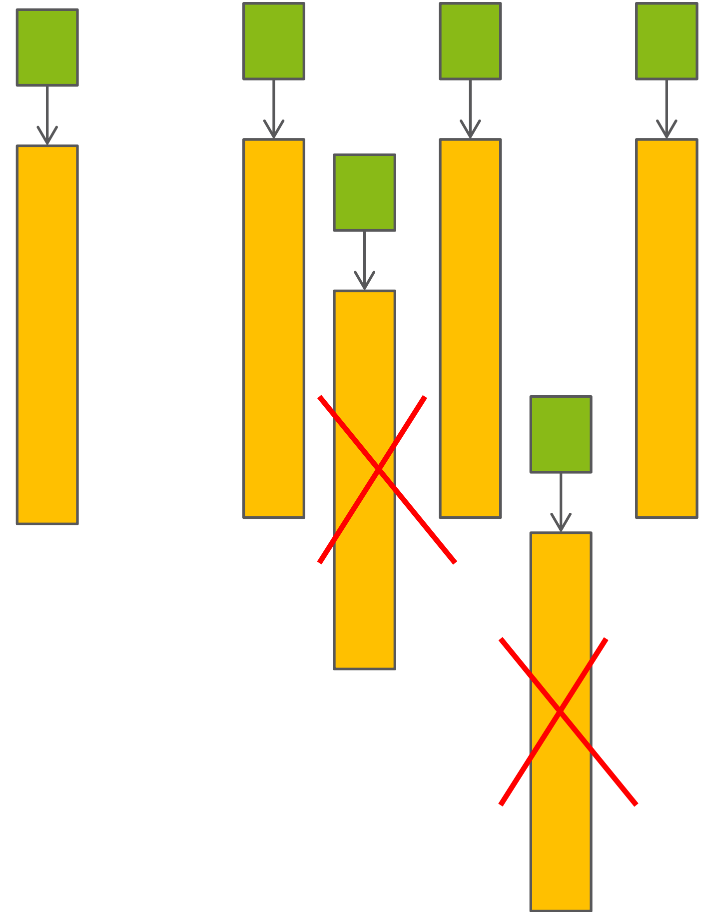
VALUE SEMANTICS (4)

```
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;
}
```



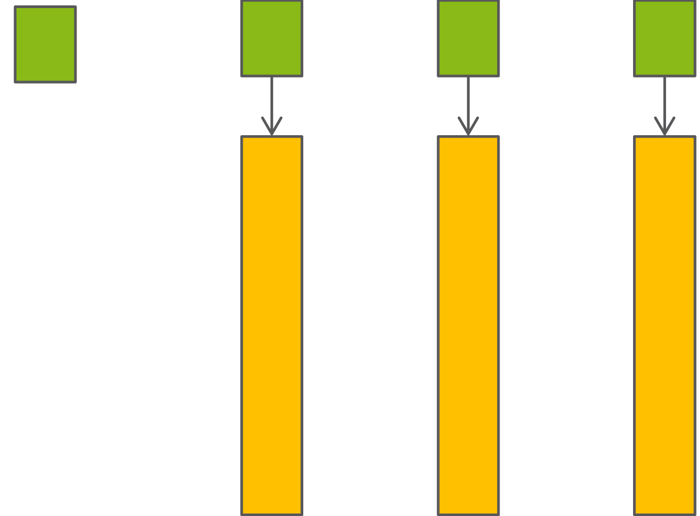
VALUE SEMANTICS (5)

```
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;
}
```



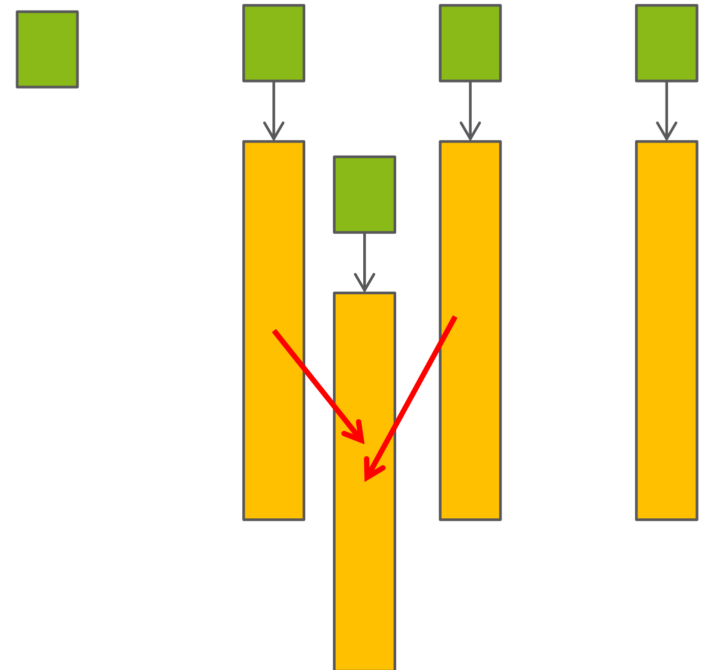
MOVE SEMANTICS (1)

```
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;
}
```



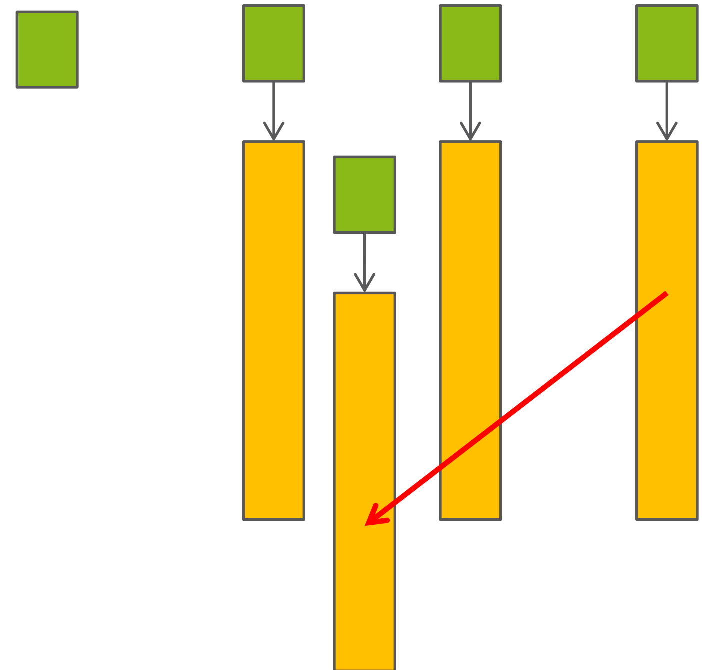
MOVE SEMANTICS (2)

```
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;
}
```



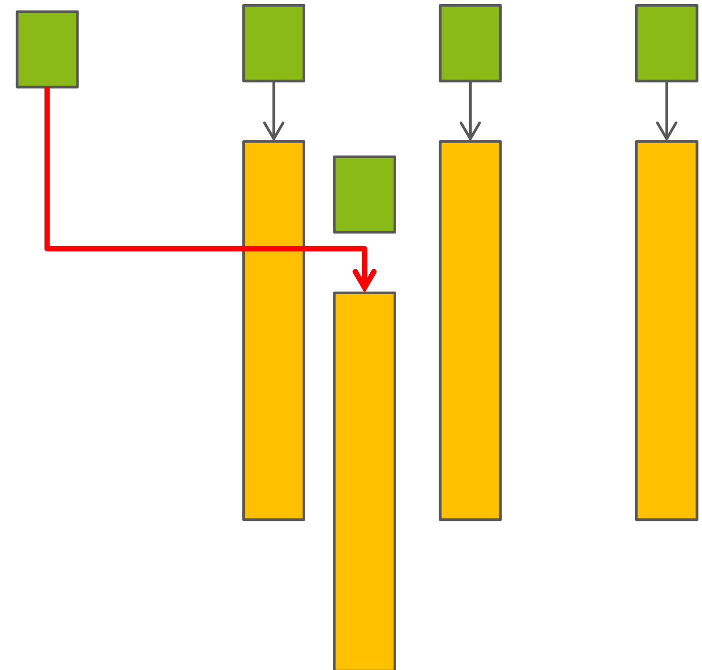
MOVE SEMANTICS (3)

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



MOVE SEMANTICS (4)

```
class Array
{
public:
   Array (const Array&);
   Array (Array&&);
   Array& operator=(const Array&);
   Array& operator=(const 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;
}
```



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

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

MOVE SEMANTICS ISSUE

```
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= ”; }
   S& operator=(S&& rhs)        { a = rhs.a; std::cout << “move= ”;}
    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);
}
```

MOVE SEMANTICS ISSUE

```
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= ”; }
   S& operator=(S&& rhs)       { a = rhs.a; std::cout << “move= ”;}
    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
A() A() copyCtr copy= copy=
```

MOVE SEMANTICS ISSUE

```
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= "; }
   S& operator=(S&& rhs)       { a = rhs.a; std::cout << "move= ";}
    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
A() A() copyCtr copy= copy=
```

If it has a name: LVALUE

MOVE SEMANTICS ISSUE

```
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= ”; }
   S& operator=(S&& rhs)        { a = rhs.a; std::cout << “move= ”;}
    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);
}
```

MOVE SEMANTICS ISSUE

```
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= ”; }
   S& operator=(S&& rhs)       { a = rhs.a; std::cout << “move= ”;}
    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
A() A() moveCtr move= move=
```

MOVE SEMANTICS ISSUE

```
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…
}
```

MOVE SEMANTICS ISSUE

```
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…
}
```

MOVE SEMANTICS ISSUE

```
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…
}
```

REVISIT STD::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= ”; }
   S& operator=(S&& rhs)        { a = rhs.a; std::cout << “move= ”;}
    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;
}
```

REVISIT STD::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= ”; }
   S& operator=(S&& rhs)        { a = rhs.a; std::cout << “move= ”;}
    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  &&  ./a.out
A() A() A() A() A() A() moveCrt
copyCrt copyCrt copyCrt copyCrt copyCrt
1 2 3 4 5 6
```

REVISIT STD::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= "; }
   S& operator=(S&& rhs)        { a = rhs.a; std::cout << "move= ";}
    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  &&  ./a.out
A() A() A() A() A() A() moveCrt
copyCrt copyCrt copyCrt copyCrt copyCrt
1 2 3 4 5 6
```

REVISIT STD::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= "; }
   S& operator=(S&& rhs)        { a = rhs.a; std::cout << "move= ";}
    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;
}
```

REVISIT STD::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= ”; }
    S& operator=(S&& rhs)        { a = rhs.a; std::cout << “move= ”;}
     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  &&  ./a.out
A() A() A() A() A() A()
moveCrt moveCrt moveCrt moveCrt moveCrt moveCrt
1 2 3 4 5 6
```

STD::MOVE(b,e,b2)

```
#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= ”; }
   S& operator=(S&& rhs)         { a = rhs.a; std::cout << “move= ”;}
    int a ;
    static int cnt;
};
int S::cnt = 0;

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

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

STD::MOVE(b,e,b2)

```
#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= "; }
    S& operator=(S&& rhs)         { a = rhs.a; std::cout << "move= ";}
     int a ;
     static int cnt;
};
int S::cnt = 0;

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

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

```
$ g++ -std=c++11  &&  ./a.out
A() A() A() A() A()
copyCrt copyCrt copyCrt copyCrt copyCrt
A() A() A() A() A() A()
move= move= move= move= move=
1 2 3 4 5 6
```

STD::MOVE(b,e,b2)

```
#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= ”; }
   S& operator=(S&& rhs)        { a = rhs.a; std::cout << “move= ”;}
    int a ;
    static int cnt;
};
int S::cnt = 0;
bool operator< (const A& x, const A& y) { return x.a < y.a; }
int main()
{
   std::set<S>        sl = { A(), A(), A(), A(), A() };
   std::vector<S>  sv(5);
   std::move( l.begin(), l.end(), v.begin());

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

STD::MOVE(b,e,b2)

```
#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= "; }
    S& operator=(S&& rhs)         { a = rhs.a; std::cout << "move= ";}
     int a ;
     static int cnt;
};
int S::cnt = 0;
bool operator< (const A& x, const A& y) { return x.a < y.a; }
int main()
{
    std::set<S>        sl = { A(), A(), A(), A(), A() };
    std::vector<S>  sv(5);
    std::move( l.begin(), l.end(), v.begin());

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

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

STD::SET

- Pre C++11
  - iterator — Bidirectional iterator
  - const_iterator — Constant bidirectional iterator
- Post C++11
  - iterator — Constant bidirectional iterator
  - const_iterator — Constant bidirectional iterator
- Const reference does not convertible to rvalue reference

- The same problem exists with std::priority_queue

C++14 (C++1y)

- Minor release
  - The committee draft was finalized 15 May 2013
  - The working draft was published 2 March 2014
  - C++14 has been accepted 18 August 2014
- Features
  - Return type deduction
  - [[deprecated]] attribute
  - Digit separators 1'000'000'000
  - Initialized lambda capture
  - Generic/Polymorphic lambda `auto add = [](auto a,auto b){return a + b;}`
  - Variable templates `template<typename T> constexpr T pi = T(3.1415);`
  - Runtime-sized automatic arrays (in C since C99)
  - Constexpr extensions if/else/switch, loops, etc…
  - Sized delete operator void `operator delete(void*, std::size_t) noexcept;`

C++17 STUDY GROUPS

- Core language
  - SG1 Concurrency
  - SG2 Modules
  - SG5 Transactional memory
  - SG7 Reflection
  - SG8 Concepts
  - SG10 Feature test
  - SG12 Undefined behavior
- Library
  - SG3 Filesystem
  - SG4 Networking
  - SG6 Numerics
  - SG9 Ranges
  - SG11 Databases

Evolution WG

Core WG

Library evolution WG

Library WG

REFERENCES

- The C++ standard committee site: http://isocpp.org/
- Andrew Sutton on Constraints: http://isocpp.org/blog/2013/02/concepts-lite-constraining-templates-with-predicates-andrew-sutton-bjarne-s
- Thomas Becker on Move semantics: http://thbecker.net/articles/rvalue_references/section_01.html
- David Abrahams on RVO and Move semantics http://cpp-next.com/archive/2009/08/want-speed-pass-by-value
- LLVM/Clang fully supporting C++14: http://clang.llvm.org/
- LLVM/Clang static analyzer: http://clang-analyzer.llvm.org/