Selected Topics in Computer Programming #1

last update: 2008-03-28

Selected Topics in Computer Programming #1

To Copy or Not To Copy: A Deeper Look at Values in C++

Walter E. Brown, Ph.D.
Computing Division
Fermi National Accelerator Laboratory

A little about me

- B.A. (math's); M.S., Ph.D. (computer science).
- Professional programmer for nearly 40 years.
- Experienced in both academia and industry:
  - Founded Comp.Sci. Dept.; served as Professor and Dept. Head; taught/mentored at all levels.
  - Managed/mentored programming staff for a computer reseller; self-employed as a software consultant and commercial trainer.
- At Fermilab since 1996; now in Computing Division/LSC Dept., specializing in C++ consulting and programming.
- Participant in the international C++ standardization process.
- Be forewarned: Based on the above training and experience, I hold some rather strong opinions about computer software and programming methodology — these opinions are not shared by all programmers, but they should be! ☺

Today's topics

- Values and their role in modern C++ ("C++03")
  - Behind the scenes: the two kinds of values
  - The impact of context in values' use
  - Value copying: prevalence, cost, and mitigation
- New uses of values in the next C++ ("C++0X")
  - A new kind of reference
  - Application for significantly improved performance
  - Solution to a previously unsolved programming problem

Our most important resource: memory

- A digital computer's memory is modelled as a sequence of cells (bytes, words, registers, units, …):
  - The size (capacity, width, …) of a cell is measured in bits.
  - All cells within a memory have identical capacity, *w* bits.
- Associated with each cell are:
  - Its memory address, a unique unsigned integer denoting that cell's permanent position in the sequence, and …
  - Its contents, a specific pattern of *w* bits.

Memory address characteristics

- A given cell's address is determined by the circuitry:
  - Two cells are neighbors iff their addresses differ by 1.
  - Neighboring cells are described as contiguous or adjacent.
  - Cells are remote from each other if they are not adjacent.
- From a programming perspective, memory addresses are considered lvalues:
  - High-level programming languages let programmers use symbols (names, identifiers) in lieu of addresses.
  - Programmers then rely on a compiler to select addresses and then map their program's names to those addresses.

Memory contents characteristics

- As conceived by J. von Neumann (following A. Turing), any cell's contents can represent:
  - ① An encoded instruction, or …
  - ② An encoded data value.

- From a programming perspective, memory contents are considered rvalues:
  - Since all bits look alike, we can't tell by inspecting an rvalue what kind of information it encodes.
  - We therefore don't know how to decode the rvalue unless we have some external knowledge about it.
  - Programmers rely on a compiler to track each rvalue's type, so that any rvalue's bits can be properly interpreted (encoded and decoded).

Cooperating cells form important abstractions

- A function is an organized collection of instructions that cooperatively denote a logical task:
  - Lets us think and reason about the task as if it were a single (composite) instruction.
- A function is conventionally identified via the address of the cell holding its leading (initial/first) instruction.
- A data structure is an organized collection of data values that cooperatively denote a logical object:
  - Lets us think and reason about the object as if it were a single (composite) data value.
- An object is conventionally identified via the address of the cell holding its leading (initial/first) datum.
  - Use the leading address of the object's principal part if the object is linked to remote parts.

A historical perspective

"About 1,000 instructions is a reasonable upper limit for the complexity of problems now envisioned."

— *Herman H. Goldstine & John von Neuman*, 1946


Instruction architecture

- Each instruction:
  - Has one operator, …
  - Has zero or (usually) more associated operands, and …
  - (Usually) produces a result.
- Each operand and each result represents a point of interaction with the computer's memory:
  - Each instruction documents the nature of each such interaction either as a data value or as an address.
  - At a low level, the distinction affects the circuitry that is activated to deal with the operand/result.
  - At a high level, the distinction affects the code that is generated.

In the context of a high-level expression

- Sometimes an lvalue (address) operand is needed:
  - *E.g.*, the left operand of a traditional assignment (operators =, +=, −=, *etc.*).
- Other times an rvalue (data value) operand is needed:
  - *E.g.*, the right operand of a traditional assignment.
- Some operators give an lvalue result, others give an rvalue.
- In addition to a result, other side effects may also ensue:
  - *E.g.*, I/O, further memory updates, thrown exceptions, *etc.*

Recognizing lvalues and rvalues in C++

- An operand is an lvalue:
  - If it is named (*i.e.*, an alias for a cell's address), or ...
  - If it has reference type (also an alias for an address), or …
  - If it has array type (more about arrays shortly), ...
  - Otherwise it is an rvalue.
- A few interesting cases:
  - Literals are rvalues, usually corresponding to some encoded bit pattern that need not occupy program storage; ...
  - But a string literal is an lvalue, since it corresponds to an in-memory array of characters.
  - The result of a function call is unnamed, hence is an rvalue unless it's of reference type.

Examples of lvalues and rvalues

- Literals:
  - 3 *// an rvalue (of type* int*)*
  - "abc" *// an lvalue (of type* char const [4] *)*
- After declaring int i; :
  - i *// an lvalue*
  - (i) *// an lvalue (unaffected by parentheses)*
  - i + 3 *//* int *addition yields an rvalue*
  - i = 3 *//* int *assignment yields an lvalue*
- After declaring int f(int); :
  - f(3) *// call to* f *yields an rvalue*
  - f(i) *// call to* f *yields an rvalue*
- After declaring int & g(int); :
  - g(3) *// call to* g *yields an (anonymous) lvalue*
  - g(i) *// call to* g *yields an (anonymous) lvalue*

The lvalue-to-rvalue conversion

- Any address can be used to obtain its cell's value:
  - *E.g.*, via a microcoded memory fetch or read.
  - Analogously, any C++ lvalue is convertible to its cell's rvalue.
- Such conversions are very, very common:
  - They will happen implicitly whenever an lvalue is supplied in a right-hand context (one which demands an rvalue).
  - *I.e.*, a conversion happens each time a programmer supplies an lvalue operand to an operator needing an rvalue there.
  - Example: a = b; // rhs lvalue converted to rvalue before assigning
  - The cost of the conversion depends on the rvalue's type.

Mutable vs. immutable values

- Each value (whether an lvalue or an rvalue) is further classified as modifiable or nonmodifiable:
  - The sole criterion is the value's mutability/constness.
  - Thus each named variable is an lvalue, whether const or not.
  - The result of a function call can be a modifiable rvalue.

```
class C {
private:
  int i;
public:
  C ( ) : i ( 0 ) { }
  void incr ( ) { ++ i; }
};
```

```
C( ).incr( )
```

updates the newly-created mod. rvalue

c'tor produces a modifiable rvalue

Analyzing some traditional operators

- Arithmetic, relational, and shift operators:
  - Take two rvalues as operands.
  - Yield an rvalue result.
- Assignment operators:
  - Take one modifiable lvalue and one rvalue as operands.
  - Yield an lvalue result.
- Increment and decrement operators:
  - Take one modifiable lvalue operand.
  - Prefix forms ++ i and -- i yield an lvalue result, but ...
  - Postfix forms i ++ and i -- yield an rvalue result.

Using pointer types

- Use of a pointer value is an rvalue:
  - Same as using a value of any other type.
- Use of a pointer variable's name is an lvalue:
  - Same as using a named variable of any other type.
- An lvalue of pointer type can implicitly decay (be converted) into an rvalue:
  - Happens via an ordinary lvalue-to-rvalue conversion, …
  - Same as using an lvalue of any other type.
  - The result of such a decay is a pointer value.

Analyzing some pointer-related operators

- Address-of operator (unary &):
  - Takes one lvalue operand.
  - Yields a pointer value (*i.e.*, an rvalue) as its result.
- Instantiation operator (new):
  - Allocates (obtains) memory for an unnamed variable via an allocation function (operator new), then …
  - Initializes that memory via the appropriate c'tor.
  - Yields a pointer value (*i.e.*, an rvalue) as its result.
- Dereferencing operator (unary ∗):
  - Takes one rvalue operand of pointer type.
  - Yields an lvalue as its result.

Arrays' relationship to pointers

- Use of an array's name is a nonmodifiable lvalue:
  - When needed, decays into a pointer value (*i.e.*, an rvalue).
    - An ordinary lvalue-to-rvalue conversion for an array type.
  - The pointer value denotes the array's leading item.
- Array instantiation operator (new [ ]):
  - Yields a pointer value (*i.e.*, an rvalue) …
  - That denotes the new array's leading item.
- Indexing/subscripting operator ([ ]):
  - Recall that a [b] is defined as * (a + b) .
  - Hence its operands are rvalues (same as binary +) …
  - And it yields an lvalue (same as unary *).

19

Functions' relationship to pointers

- Use of a function's name is a nonmodifiable lvalue:
  - Decays implicitly into a pointer value (*i.e.*, an rvalue).
    - An ordinary lvalue-to-rvalue conversion for a function type.
    - Use of a function template-id f<…> is likewise a nonmodifiable lvalue that decays implicitly.
  - The pointer value typically denotes the function's leading instruction.
- Call operator ( ) takes two operands:
  - Left: an rvalue designating the callee (function to be called).
  - Right: an argument list (a sequence of lvalues and rvalues consistent with the callee's type).
  - Yields an rvalue result unless the callee's return type specifies an lvalue result.

20

Simple decay can be just what's needed

- Function example:
  - ```
    typedef void F( int );
    void f( int );
    F * fp = & f;
    (*fp)( 3 );
    ```
- Array example:
  - ```
    typedef int A[10];
    A a;
    A * ap = & a;
    (*ap)[ 3 ];
    ```

```
        // pointer lvalue ① decays to an rvalue,
        // that then ② is dereferenced to obtain an lvalue,
        // that then ③ decays to yield the rvalue left operand
fp( 3 );                    ap[ 3 ];
        // equivalent semantics via a single decay
```

21

Using traditional reference types

- A value of traditional reference type is an lvalue, no matter how it was produced:
  - ```
    float const & pi( ) { // yields a nonmodifiable lvalue when called
        static float const pi = 3.1415926F;
        return pi;
    }
    ```
- Use of a named variable of reference type is an lvalue:
  - Same as using any other named variable.
- Given an lvalue reference (lvalue of reference type), what can be bound to (initialize) it?
  - If modifiable, only a modifiable lvalue.
  - If nonmodifiable, any lvalue or any rvalue.

22

Examples of lvalue reference bindings

- Can bind only a modifiable lvalue to a modifiable lvalue reference (maintains const-correctness):
  - ```
    int m;
    int & r = m;
    ```
- Can bind any lvalue/any rvalue to a nonmodifiable lvalue reference:
  - ```
    int m;
    int const & r = m;      // binding a modifiable lvalue
    ```
  - ```
    int const n = 0;
    int const & r = n;      // binding a nonmodifiable lvalue
    ```
  - `int const & r = int( );  // binding a modifiable rvalue`
  - `int const & r = 0;       // binding a nonmodifiable rvalue`

23

Binding during call/return

- Initialization semantics also apply to parameter passage:
  - Before the caller gives over control to the callee, …
  - Each argument from the argument list is bound …
  - To its corresponding parameter (function-local variable).
- Initialization semantics also apply to result return:
  - Before the caller regains control from the callee, the return statement's value is bound to some ephemeral (temporary) object owned by the caller.
  - RVO (return value optimization): a compiler <u>may</u> elide this binding <u>if</u> the caller immediately binds his ephemeral to another target; *i.e.*, 1 binding may replace a sequence of 2.

24

Copying in today's C++

- Always involves an lvalue target (destination).
- Non-native types carry out copying via member functions:
  - Namely, copy c'tors and copy assignment operators, …
  - With each taking a source (original) as its parameter.
- Such copy functions have several possible signatures, *e.g.*:
  - *// copy without modifying the source:*
    T (T const & src);
    T & operator = (T const & src);
  - *// copy from only a modifiable lvalue source*
    // (not generally recommended, but used, e.g., by std::auto_ptr<>*):*
    T (T & src);
    T & operator = (T & src);

But copying isn't always what it seems to be

- Neither arrays nor functions have copy operators, so:
  - An apparent array copy first produces a decay, then copies only that decayed rvalue.
  - An apparent function copy first produces a decay, then copies only that decayed rvalue.
- Array example:
  - void f(int [10]);
    int a[10];
    f(a); *// a pointer rvalue is bound, not the array* a
- Function example:
  - void g(int (int));
    int h(int);
    g(h); *// a pointer rvalue is bound, not the function* h

Why should we programmers care about copying?

- Per the C++ Standard, copying occurs frequently "in various contexts" [12.2] during execution, *e.g.*:
  - When "binding an rvalue to a reference,
  - returning an rvalue,
  - a conversion that creates an rvalue,
  - throwing an exception,
  - entering a handler,
  - and in some initializations."
- Copying can be (very) expensive.
  - The costs depend heavily on the source's data type.
  - *E.g.*, copying a std::vector<T> is O(n·t) in time and space.

Example: unnecessary temporaries [Sutter, 2000]

- "A programmer has written the following function, which uses unnecessary temporary objects […]."

```
typedef  std::list<Employee>  e_list;
std::string  FindAddr( e_list        e
                     , std::string  name  )  {
  for ( e_list::const_iterator  it  =  e.begin( )
      ;  it  !=  e.end( ) ;  it ++ )  {
    if ( * it  ==  name )
      return  it –> addr;
  }
  return  " ";
}
```


Strategies to mitigate costly or repeated copying

- Avoid implicit copy operations when not needed:
  - *E.g.*, prefer ++ i to i ++.
  - *E.g.*, prefer a += 10 to a = a + 10.
- Precompute and cache const values to avoid recalculation.
- Choose parameter passage with size in mind:
  - Pass small, cheap-to-copy objects (e.g., ints) by-value.
  - Pass larger, costly-to-copy objects by-const-reference; references are always cheap to initialize.
- Prefer smart pointers to native pointers:
  - Copying a native pointer is cheap, but introduces ownership (lifetime management) issues.
  - Details presented in course dedicated to pointers.

Introducing a new technique

- In general, reduce costs via strength reduction:
  - Use a semantically equivalent but cheaper operation …
  - In place of a more expensive operation.
- In C++0X, prefer move semantics to copy semantics:
  - By making selected types movable, and …
  - By making selected client code move-aware.

Selected Topics in Computer Programming #1

To Copy or Not To Copy: A Deeper Look at Values in C++

End of Part 1

Part 2 will discuss lvalues and rvalues
in the context of the next C++ standard,
emphasizing new coding opportunities
that lead to improved runtime performance.

Today's topics

- ✓ Values and their role in modern C++
  - ✓ Behind the scenes: the two kinds of values
  - ✓ The impact of context in values' use
  - ✓ Value copying: prevalence/cost/mitigation
- New uses of values in the next C++
  - A new kind of reference
  - Application for significantly improved performance
  - Solution to a previously unsolved problem

Overview of move semantics

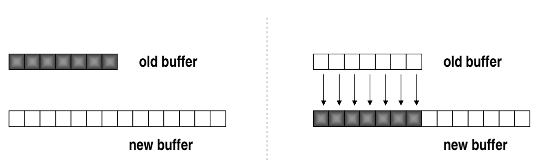
- Moving can safely replace copying whenever:
  - The source's value is about to be replaced, or …
  - The source is about to go out of existence *in toto*.
  - *I.e.*, we won't again use that value from that source.
- We can then safely move (pilfer ☺) from that source:
  - Provided that the source is left with some value …
  - That is consistent with at least basic exception safety (invariants hold and no resources are leaked).
  - Beyond that, a client doesn't/can't care what that value is:
    - Since that value is about to be replaced or to go away …
    - The larceny that is move semantics is an acceptable strength reduction for much traditional copying.

Copy semantics *vis-à-vis* move semantics

- Copy assignment (and copy c'tor) semantics:
  - assert(b == orig); *// precondition*
    a = b; *// side effect on* a *only;* b *is unaffected*
    assert(a == b && b == orig); *// postcondition*
- Move assignment/move c'tor has weaker semantics:
  - assert(b == orig && &b != &orig);
    a = std::move(b); *// side effect on* a *and maybe also on* b
    assert(a == orig);
- For some types, a move is <u>much</u> cheaper than a copy:
  - The postcondition is weaker, so there's often less work.
  - A compiler will automatically optimize move-aware code whenever it's applied to a movable type.

Advantages of a move-aware std::vector, part 1

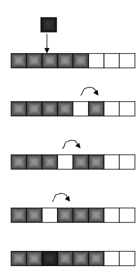
- std::vector<T> can make good use of T's movability when creating a new internal buffer:



  - Elements are now moved (not copied) to the new buffer.
  - Why? Since the entire old buffer is about to be destroyed, we care little about its elements' post-move values.

Advantages of a move-aware std::vector, part 2

- std::vector<T> can make good use of T's movability when inserting (or erasing) within a single buffer:



  - Elements can be moved (not copied) within the buffer to create a "hole" for the new element.
  - Why? Since each "hole" quickly gets a new value, we care little about its post-move value.

Advantage of a moveable std::vector

- Example:
  - std::vector< T > f(···) {
    std::vector< T > result;
    // calculate result*, then …*
    return result; *// will exploit* vector<>*'s movability*
    }
  - Why? Move semantics are applicable in the above return because of the imminent end of result's lifetime.
- Moving a std::vector is <u>far</u> less expensive than copying it:
  - Copying entails allocating a new buffer, then copying each element of the old buffer into the new one, but …
  - Moving entails only two cheap pointer assignments:
    ① Take possession of the old buffer, and
    ② Leave a vacuous buffer behind!

Applicability of move semantics

- Move semantics can be exploited explicitly:
  - *E.g.*, a = std::move(b);
  - Typically to take advantage of an algorithm's pattern of memory access, as determined by a programmer.
- Move semantics can also be exploited implicitly:
  - By a compiler …
  - Whenever the source of a copy is a modifiable rvalue …
  - As is true of most ephemerals!
- A type must be movable before move-aware code (such as the above) can exploit it:
  - Movable types and move-aware code are made possible via a new C++ language feature, the rvalue reference.

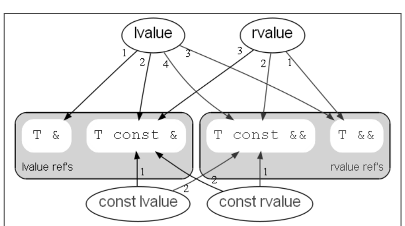
What is an rvalue reference? [H. Hinnant, 2002-2006]

- A compound type, much like a traditional reference:
  - Formed by placing && after a type name: T && .
- Examples:
  - T && r = T( ); *// rvalue bound to modifiable rvalue ref* r
  - T t;
    T && r = t; *// lvalue* t *bound to modifiable rvalue ref* r
- Today's rules remain unchanged:
  - T & r = T( ); *// no: still can't bind rvalue to modifiable <u>l</u>value ref*
  - T const & r = T( ); *// still okay when* r *is nonmodifiable*

New function overloading options

- Can now overload a function on lvalue/rvalue parameters:
  - void f(int const &) { ··· } *// #1*
    void f(int const &&) { ··· } *// #2*
- Overload resolution will pick the correct version to call:
  - f(i); *// lvalue argument; calls #1 as a better match*
    f(i + 1); *// rvalue argument; calls #2 as a better match*
- A function taking rvalue references is often most valuable when it overloads functions that take lvalues.

Bindings to references



Permitted bindings to C++ reference types;
overload resolution preferences are numbered.

Type deduction from an rvalue reference

- Today's deduction rules don't cover this new scenario:
  - template < class T >
    void f (T &&) *// how to deduce* T *when* f *is called?*
    { ··· }
- Additional deduction rules (current rules unchanged):
  - When the above function template is invoked via f(3)
    (*i.e.*, with an rvalue argument of type int),
    T will be deduced as int, calling f<int>(3).
  - When the template is invoked via f(i)
    (*i.e.*, with an lvalue argument of type int),
    T will be deduced as int &, calling f<int &>(i).

Reference-to-reference types

- C++ has long prohibited the formation of any reference-to-reference type:
  - *E.g.*, if T is deduced as int &, then returning T & is equivalent to returning int &, a simple reference.
- We introduce analogous rules for rvalue references:

Deduced T	Returning	Produces
int &	T &	int &
int &	T &&	int &
int &&	T &	int &
int &&	T &&	int &&

The new library component std::move( )

- Designed:
  - To accept a modifiable (lvalue/rvalue) argument, and …
  - To return that argument as an rvalue, but …
  - To use only references (to avoid copying any object).


```
template < class T >
std::remove_reference<T>::type &&
  move ( T && t )
{ return t; }
```

- Combined with type deduction rules (shown earlier), std::move( ) lets us write move-aware code that can take advantage of any movable type T .

Evolution of a move-aware standard algorithm

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

- Each line in the body copies a source to a target:
  - If T is a class, uses potentially expensive copy-function calls.
  - We'd strongly prefer <u>no</u> copies; we just want to swap!
- Let's recode the body to be move-aware such that:
  - If T is movable, swap( ) can avoid the copying and so provide improved performance.
  - If T is not movable, swap( )'s behavior is unchanged (preserving backwards compatibility).

How to make a class movable

- Augment the class by adding ① a move c'tor, and ② a move assignment operator:
- Skeletal example:

```
class C {
private:
  some_type * p;
public:
  C ( ) : p( 0 ) { }
  C ( C && src ) : p( src.p ) { src.p = 0; }
  C & operator = ( C && src ) {
    std::swap( p, src.p );
    return * this;
  }
  ~C ( ) { delete p; }
};
```

Transfers the resource from the source to the target

Then leaves behind a resource-free source

Exchanges the resources of the source and the target

Movability is orthogonal to copyability

- A class can be copyable, movable, both, or neither:
  - It's up to the class designer/implementor.
  - No class is movable by default; a programmer must explicitly provide the pair of move functions.
- A class can usefully be movable even if not copyable!
  - *E.g.*, today's standard streams are noncopyable by design, but will become movable in the next C++, allowing …
  - std::vector<std::ofstream> v;
v.push_back(std::ofstream("myfile"));
  - This will work because the move-aware std::vector<> will require only movability, not copyability, of its elements.

Phasing in move semantics

① Existing code retains existing behavior:
  - Except that any use of standard components may show improved performance …
  - Just by recompiling/relinking with a move-aware library.
  - *E.g.*, several std::vector<> operations immediately become <u>much</u> faster, on average!

② Algorithms can gradually be made move-aware:
  - To take greater advantage of movable components …
  - Often just by inserting judicious calls to std::move( ).

③ Classes can gradually be made movable:
  - So that move-aware client and library code can take advantage of performance improvement opportunities.
  - Not every class needs to be made movable.

Move-aware std utility components

- std::move( )
- std::move_iterator< >
- std::make_move_iterator( )
- Example (std:: omitted for clarity):
  - list<string> s;
  - *// C++03: copy sequence of strings into* v
    ```
    vector<string>  v ( s.begin( )
                      , s.end( )   );
    ```
  - *// C++0X: move sequence of strings into* v
    ```
    vector<string>  v ( make_move_iterator( s.begin( ) )
                      , make_move_iterator( s.end( ) )   );
    ```


When to make a class movable?

- Let M denote a type such that:
  - M has direct resource-ownership semantics, or …
  - M is already movable (read M's documentation!).
- A class C will likely benefit from move semantics if:
  - C has an M-like direct base class, or …
  - C has an M-like nonstatic data member.
- What about a class template with a base/member of a generic type T?
  - Advice: make the template movable, because ...
  - There is potential for a substantial performance gain when T is M-like, and …
  - There's no performance loss when T is not M-like (the attempted move in that case just copies, as before).

The forwarding problem

- We want a call f (a_1, a_2, $\cdots$, a_n) that will:
  - Internally forward to (call) g (a_1, a_2, $\cdots$, a_n) such that …
  - f takes an argument list of n arbitrary types and …
  - Passes that list to g, lvalues as lvalues, rvalues as rvalues.
- Additional constraints:
  - Valid uses (calls) of g must also be valid uses of f.
  - Invalid uses of g must also be invalid uses of f.
  - f must be implementable in at worst O(n).
- No solution is possible in today's C++:
  - Several come close, but none is perfect.
  - C++ with rvalue references does allow a perfect solution.

Consider just the generic two-argument case

- template< class T, class A1, class A2 >
  std::shared_ptr<T>
  factory(A1 & a1, A2 & a2); *// forward* a1, a2 *to* T*'s c'tor*
- template< class T, class A1, class A2 >
  std::shared_ptr<T>
  factory(A1 const & a1, A2 const & a2);
- template< class T, class A1, class A2 >
  std::shared_ptr<T>
  factory(A1 const & a1, A2 & a2);
- template< class T, class A1, class A2 >
  std::shared_ptr<T>
  factory(A1 & a1, A2 const & a2);
- And even all these don't cover:
  - volatile and const volatile (admittedly rare, but possible).
  - Call-by-rvalue-reference, plus all its cv variants.

Solving the forwarding problem in the next C++

- ```
  template< class T, class A1, class A2 >
  std::shared_ptr<T>
    factory ( A1 &&  a1, A2 &&  a2 )
  {
    return std::shared_ptr<T> ( new T( std::forward<A1>( a1 )
                                     , std::forward<A2>( a2 )
                                     )        );
  }
  ```
- This example twice uses the new library component std::forward<>( ) …
  - To forward two arguments, each of arbitrary type, to a two-parameter c'tor of type T…
  - Preserving each argument's lvalue/rvalue-ness.

In sum

- Programmer attention to lvalues and rvalues is very important to achieve effective and efficient code in today's C++ programs.
- Future C++ programs will be able to take increasing advantage of lvalue/rvalue distinctions:
  - To improve performance, sometimes dramatically, under common circumstances, and …
  - To apply coding techniques not previously possible.
  ```

FIN

Selected Topics in Computer Programming #1

To Copy or Not To Copy:
A Deeper Look at Values in C++

Walter E. Brown, Ph.D.
Computing Division
Fermi National Accelerator Laboratory