References, Pointers, and other links

Links to objects in C++

- References

```
T &, const T &, T &&
```

  - Built in C++
  - Must be initialized to point to an object, **cannot be redirected** later
  - Syntactically identical to values when used (r.a)
- Raw pointers

```
T *, const T *
```

  - Built in C/C++
  - Requires special operators to access the referenced value (*p, p->a)
  - **Pointer arithmetics** allows to access adjacent values residing in arrays
  - Ownership semantics requires manual deallocation – **not recommended after C++11**
- Smart pointers

```
std::shared_ptr< T>, std::unique_ptr< T>
```

  - Class templates in standard C++ library
  - Operators to access the referenced value same as with raw pointers (*p, p->a)
  - **Represents ownership** - automatic deallocation on destruction of the last reference
- Iterators

```
K::iterator, K::const_iterator
```

  - Classes associated to every kind of container (K) in standard C++ library
  - Returned by container functions as **pointers to container elements**
  - Operators to access the referenced value same as with raw pointers (*p, p->a)
  - **Pointer arithmetics** allows to access adjacent values in the container

References in C++

- References may be used only in some contexts
  - Formal arguments of functions (almost always safe and useful)
    - Like passing by reference in other languages (but more complex)
  - Return values of functions (dangerous but sometimes necessary)
  - Local variables (sometimes useful, particularly with **auto &&**)
  - Static variables, data members of classes (limited usability, use a pointer or std::ref instead)
- References must be initialized and cannot be redirected later
  - All uses of references work as if they were the referenced object
- References have three flavors
  - Modifiable L-value reference

```
T &
```

    - The actual argument (init value) must be an L-value, i.e. a repeatedly accessible object
  - Const (L-value) reference

```
const T &
```

    - The actual argument may be anything of type T
  - R-value reference

```
T &&
```

    - The actual argument must be an R-value, i.e. a temporary object or marked with std::move()

References in C++

- (Modifiable) L-value reference

`T &`

  - The initializing expression must be a modifiable **L-value**, i.e. a repeatedly accessible object without the **const** flag:
    - **x**, **y.x** - A variable or a data member of type **T**, **T&** or **T&&**
    - **f()**, **cout<<x** - A call of a function (or an user-defined operator) returning **T&**
    - **(T&)e** - A cast to **T&**
    - ***p**, **p[i]** - Dereference of a pointer of type **T***, element of an array of type **T[]** (for smart pointers, these are user-defined operators)
    - Other rarely occurring cases

- Const (L-value) reference

`const T &`

  - The initializing expression may be any expression of type T, i.e. an L-value or an R-value, in addition, it may have modification prohibited (using the const flag)
    - The term "const L-value reference" corresponds to the syntax, not the semantics
    - Formally, an expression is never of reference type because any reference always behaves as the object being referenced. The L-value/R-value and modifiability is used instead

- R-value reference

`T &&`

  - The initializing expression must be an **R-value**:
    - **42**, **'c', true**, **nullptr** - a literal (except for string literals)
    - **f(), a+b** - the result of a function or operator returning **T** or **T&&** (e.g. **std::move**)
    - **(T)e**, **(T&&)e** - A cast to **T** or **T&&**
    - Other rarely occurring cases

References in C++

- Why **T &** requires an L-value
  - T & often used as a "result parameter"
  - Prohibiting R-value protects again programming mistakes

```
void setup(int & p) { p = 42; }

int x;
setup(x);           // OK
setup(x + 1);       // Error
setup(1);           // Error
```

- Why **const T &** accepts both L-values and R-values
  - const T & means "input parameter"
  - the only reason to use a reference is the speed of passing

```
std::string operator+(const std::string & a, const std::string & b);

std::string x, y, z, u;
u = x + y + z;      // u = ((x + y) + z)
```

  - (x + y) is an R-value but we still want to pass it by reference

References in C++

- The purpose of R-value references **T &&**
    - The initializing expression must be an **R-value**:
        - **42**, **'c'**, **true**, **nullptr** - a literal (except for string literals)
        - **f(), a+b** - the result of a function or operator returning **T** or **T&&** (e.g. **std::move**)
        - **(T)e**, **(T&&)e** - A cast to **T** or **T&&**
        - Other rarely occurring cases
    - There are two sub-cases of R-values
        - *prvalue* - the expression is an anonymous temporary which will not be accessible again
        - *xvalue* **-** a call to a function or a cast returning **T&&**
- The object in an R-value expression may be arbitrarily modified because
    - for a prvalue, the modified object will not be seen
        - but beware, the destructor will sooner or later run on the object
    - for an xvalue, the programmer explicitly agreed to that
        - the semantics of T&& is "a reference to an object for which the owner no longer cares"
        - if the object is used again, the owner must start with assigning a new value
- The most important uses of R-value references
    - Stealing and reusing of the dynamically allocated blocks owned by the object
        - Pointers inside the objects must be set to null to avoid deallocation by destructor
        - Often a hidden optimization (but documented as mandatory for containers)
    - Passing ownership of a block held by a smart pointer
        - Technically equal to stealing but explicitly documented as part of smart pointer semantics
- A function with a parameter of type T&& will probably steal from it

References in C++

- **std::move(x)** returns T&&, therefore, this expression is an R-value
  - inside, there is a cast to T&&
- A function with a parameter of type T&& will probably steal from it
  - Most often a move-constructor or a move-assignment
    - both functions move the contents to the target object
    - in addition, move-assignment cleans the previous content of the target object

```
class T {
public:
  T(T && b) noexcept : data_(b.data_) { b.data_ = nullptr; }
  T& operator=(T && b) noexcept {
    T tmp(std::move(b));         // calls move-ctor, i.e. empties b
    std::swap(data_, tmp.data_);// swap the pointers
    return * this;               // destructor on tmp will be called here
  }
  ~T() { if (data_) delete data_; }    // the destructor does the clean-up
private:
  X * data_;    // with unique_ptr<X>, the three functions would not be needed
};
```

-
  -
    - Implementing the move-functions is considered advanced programming
    - It usually can be avoided by the use of smart pointers etc.
  - Other than in the two move-functions, T&& is used for non-copyable types or as an advanced optimization

- **std::move(x)** returns T&&, therefore, this expression is an R-value
  - inside, there is a cast to T&&
  - example usage:

```
void swap(T & a, T & b)
{
  T tmp(std::move(a));    // call move-constructor, a is probably emptied
  a = std::move(b);       // call move-assignment, b is probably emptied
  b = std::move(tmp);     // call move-assignment, tmp is no longer needed
}                         // call destructor on tmp
```

    - The real std::swap is a template - the correct implementation is more complex
  - Three move operations are significantly faster than three copy operations
    - If T holds some dynamically allocated data that can be stolen
    - std::string, std::vector<U> and many others, including classes containing them
  - Řešení s přesuny funguje i pro nekopírovatelné typy
    - std::unique_ptr<X>, std::fstream, std::mutex, ...

- The ability to move things quickly has an impact on program architecture
  - It is possible to avoid dynamic allocation by moving contents of variables
  - E.g., an fstream may be opened inside a function and returned by value
    - A return statement with a local variable is automatically implemented by moving

References with function templates and auto

- Forwarding (a.k.a. universal) reference
  - As template function argument

```
template< typename T>
void f( T && p)
```

  - As auto variable

```
auto && x = /*...*/;
```

  - May be bound to both R-values and L-values
    - Beware, there are *reference combining rules*

```
U a;
auto && x = a;  // decltype(x) == U &
f( a);    // T == U &
```

Function arguments

Passing arguments in C++

- Guidelines for formal **argument** types
  - If the function needs to modify the object
    - use modifiable reference: **T &**
  - otherwise, if copying of T is really cheap (numbers, complex, observer pointers)
    - pass by value: **T**
  - otherwise, if the type does not support copying
    - pass by R-value reference: **T &&**
  - [advanced] otherwise, if the function stores a copy of the object somewhere
    - if you want really fast implementation
      - implement two versions of the function, with **const T &** and **T &&**
    - simplified approach
      - pass by value: **T**
      - use std::move() when storing the object
  - otherwise
    - use const reference: **const T &**

  - These guidelines do **not** apply to return values and other contexts

Read-only arguments

- For read-only arguments passed by reference, **const is necessary**
  - Global function:

```
std::string concat( const std::string & a, const std::string & b)
{ auto tmp = a; tmp.append( b); return tmp; }
```

    - A concatenation operator from the standard library:

```
std::string operator+( const std::string & a, const std::string & b);
```

    - Read-only operators shall be defined as global functions – this allows implicit conversions on the left operand, like in "abc"+s
  - Member functions:

```
class my_string {
public:
  my_string concat(const my_string & b) const;
  my_string operator+(const my_string & b) const;
};
```

    - the ending const changes the implicit declaration of this: const my_string* this

  - Otherwise, the argument could not be bound to an R-value

```
std::string concat(std::string & a, std::string & b);   // WRONG
std::string operator+(std::string & a, std::string & b);   // WRONG

u = concat( concat( x, y), z);      // ERROR
u = x + y + z;      // ERROR
```

Return types of functions

Returning values from functions

- Function **return type** guidelines
  - If the function provides access to an element of a data structure (e.g. operator[])
    - and if you want to allow modification to the element
      - use modifiable L-value reference: **T &**
    - otherwise
      - use constant reference: **const T &**
    - The returned object **must survive** at least a moment **after exiting the function**
  - In all other cases
    - pass by value: **T**
    - Do not use **std::move()** in the return statement if the returned object is a local variable
      - Except when the type T does not support copying
      - The compiler will optimize the return by *copy-elision:* The local variable will be placed in the space reserved for the return value by the calling function
  - **If a function computes** or somehow constructs the returned value, **it cannot return by reference**
    - the computed value is stored only in local objects which are destroyed before the function is exited
      - the only accessible object which survives is the temporary created by the calling function to hold the returned value (the return statement initializes this temporary)

Returning by reference

Returning by reference

- Returning by reference is possible only if the function provides access to data which will exist after returning from the function
  - All these attempts are seriously defective:

    - anonymous variable

```
const Complex & add( const Complex & a, const Complex & b) // WRONG
{ return Complex(a.re+b.re, a.im+b.im);
  // ERROR: the temporary object will disappear before exiting the function
}
```

    - local variable

```
const Complex & add( const Complex & a, const Complex & b) // WRONG
{ Complex tmp(a.re+b.re, a.im+b.im);
  return tmp;  // ERROR: tmp will disappear before exiting the function
}
```

    - global (static) variable

```
Complex tmp;
const Complex & add( const Complex & a, const Complex & b) // WRONG
{ tmp.re = a.re+b.re; tmp.im = a.im+b.im;
  return tmp;  // correct, but unusable for the users
}
Complex v = (x + y) - (z + u);   // which values are passed to the '-'?
```

    - dynamic allocation

```
const Complex & add( const Complex & a, const Complex & b) // WRONG
{ Complex * tmpp = new Complex(a.re+b.re, a.im+b.im);
  return * tmpp;   // MEMORY LEAK: nobody will deallocate the block
}
```

    - smart pointer

```
const Complex & add( const Complex & a, const Complex & b) // WRONG
{ std::shared_ptr<Complex> tmpp = std::make_shared<Complex>(a.re+b.re, a.im+b.im);
  return * tmpp;
  // ERROR: the block will be deallocated before exiting the function
}
```

  - Beware: All these implementations may appear like functioning

Functions returning by reference

- Returning by reference is possible only if the function provides access to data which will exist after returning from the function
  - To provide usable access, **two** functions are usually required
    - Different headers, usually (syntactically) the same body
    - Global functions:

```
T & get_element( std::vector< T> & v, std::size_t i)
{ return v[ i]; }
const T & get_element( const std::vector< T> & v, std::size_t i)
{ return v[ i]; }
```

    - Member functions:

```
class my_hidden_vector {
public:
  T & get_element( std::size_t i)
  { return v_[ i]; }
  const T & get_element(std::size_t i) const
  { return v_[ i]; }
private:
  std::vector< T> & v_;
};
```

Functions returning by reference

- Non-symmetric versions are usually wrong
  - Voluntarily dropping the right to modify:

```
const T & get_element(std::vector< T> & v, std::size_t i) { return v[i]; }
```

    - This method does not need modifiable v
    - This situation shall be solved by a symmetric double-const version (and without the no-const version)

  - Attempting to break the protection (compilation error):

```
T & get_element(const std::vector< T> & v, std::size_t i) { return v[i]; }
```

    - v is const, therefore v[i] invokes this function:

```
const T & std::vector<T>::operator[](std::size_t) const;
```

    - Therefore, v[i] is of type const T and cannot be converted to T& by the return statement

- Logical const-ness
  - The data returned by [] are physically outside the vector
  - Logically, they are considered a part of the vector, therefore, operator[] propagates const-ness
    - The language rules do not force vector<T>::operator[] to return const T&

```
template< typename T> class vector { public:
  /*const*/ T & operator[](std::size_t i) const { return *(begin_ + i); }
private:
  T * begin_, * end_, * block_end_;
};
```

    - **const** flag of a member-function affects **this** and is propagated to member data
      - except members declared as **mutable**
      - in a const member-function, the member data appear as if declared const:

```
  T * const begin_, * const end_, * const block_end_;
```

    - the type of the data pointed to by the vector is not affected
      - returning non-const T& from operator[] is technically possible

Returning by reference

- Overload resolution in C++
  - C++ resolves overloaded function calls using only the types of arguments
    - The use of the result is not considered

```
T & get_element( std::vector< T> & v, std::size_t i);
const T & get_element( const std::vector< T> & v, std::size_t i);
void example1(std::vector<T> & w)
{
  get_element(w,1) = get_element(w,2);    // both calls invoke the first version
}
T example2(const std::vector<T> & w)
{
  return get_element(w,2);  // invokes the second version
}
```

  - With member functions:

```
class my_hidden_vector { public:
  T & get_element( std::size_t i);
  const T & get_element(std::size_t i) const;
};
```

    - The const flag on the object on the left of '.' participates in overload resolution

```
void example1(my_hidden_vector & w)
{
  w.get_element(1) = w.get_element(2);    // both calls invoke the first version
}
T example2(const my_hidden_vector<T> & w)
{
  return w.get_element(2);  // invokes the second version
}
```

Returning by reference

- Interface design in C++
  - Influenced by the life-time considerations for returned values
    - The object must survive the return from the function if returned by reference
  - Returning by value may be slow and exception-unsafe
    - Remedied by C++11/14/17 but still of some concern

```
template< typename T> class vector {
public:
```

- back() returns the last element which will remain on the stack
  - it may allow modification of the element

```
  T & back();
  const T & back() const;
```

- pop_back() removes the last element from the stack
  - if it had to return the removed value, it would have to return by value!
  - before C++11: returning by value was slow and exception-unsafe

~~T pop_back(); // NO SUCH FUNCTION IN std::vector~~

- therefore, in standard library, the pop_back() function returns nothing

```
  void pop_back();
  // ...
};
```

- For math-inspired interfaces like operator+, returning by value is necessary
  - Such functions return newly calculated values – no chance to return by reference

Returning by value

Typical function implementations returning by value

- Returning a newly constructed value
  - With a local variable

```
std::string concat( const std::string & a, const std::string & b)
{ auto tmp = a; tmp.append( b); return tmp; }
```

  - With an anonymous temporary

```
Complex add( const Complex & a, const Complex & b)
{ return Complex( a.re + b.re, a.im + b.im); }
```

  - Sometimes, a simplified syntax may be used

```
Complex add( const Complex & a, const Complex & b)
{ return { a.re + b.re, a.im + b.im}; }
```

How compilers translate returning by value

- Returning structures/classes by value - example

```
std::string concat( const std::string & a, const std::string & b)
{ auto tmp = a; tmp.append( b); return tmp; }
```

  - Called from an initialization:

```
void f() { std::string x = concat( y, z); use(x); }
```

  - Called from an assignment statement:

```
void g() { std::string x; x = concat( y, z); use(x); }
```

- Translation before C++11 and without *copy-elision*
  - Explained using a hypothetical "C language with member functions":

```
void concat( std::string * r, const std::string * a, const std::string * b)
{ std::string tmp; tmp.copy_ctor(a); tmp.append(b);
  r->copy_ctor(&tmp); tmp.dtor(); }
```

  - The compiler assumes that the function called invokes the construction of the object *r:

```
void f() { std::string x; concat( &x,&y,&z); use(&x); x.dtor(); }
void g() { std::string x,t; x.ctor();
  concat(&t,&y,&z); x.copy_asgn(&t); t.dtor();
  use(&x); x.dtor();}
```

  - In assignment, the temporary t is required, because x was already initialized
  - ctor, copy_ctor, copy_asgn a dtor are translated versions of these C++ member functions:

```
string();                                       // default-constructor
string(const string &);                         // copy-constructor
string & operator=(const string &);             // copy-assignment
~string();                                          // destructor
```

  - In many low-level classes like string, these member functions must be explicitly implemented
  - In well-designed higher-level classes, the compiler can supply an applicable implementation

How compilers translate returning by value

- Returning structures/classes by value - example

```
std::string concat( const std::string & a, const std::string & b)
{ auto tmp = a; tmp.append( b); return tmp; }
```

  - Called from an initialization:

```
void f() { std::string x = concat( y, z); use(x); }
```

  - Called from an assignment statement:

```
void g() { std::string x; x = concat( y, z); use(x); }
```

- Translation in C++11 without *copy-elision*
  - Explained using a hypothetical "C language with member functions":

```
void concat( std::string * r, const std::string * a, const std::string * b)
{ std::string tmp; tmp.copy_ctor(a); tmp.append(b);
  r->move_ctor(&tmp); tmp.dtor(); }
```

  - The compiler assumes that the function called invokes the construction of the object *r:

```
void f() { std::string x; concat( &x,&y,&z); use(&x); x.dtor(); }
void g() { std::string x,t; x.ctor();
  concat(&t,&y,&z); x.move_asgn(&t); t.dtor();
  use(&x); x.dtor();}
```

  - move_ctor and move_asgn correspond to newly defined member-functions in C++11:

```
string(string &&);                // move-constructor
string & operator=(string &&);   // move-assignment
```

  - for classes carrying data in attached dynamically allocated blocks, move functions may be implemented faster than copy functions, by stealing the data blocks

How compilers translate returning by value

- Returning structures/classes by value - example

```
std::string concat( const std::string & a, const std::string & b)
{ auto tmp = a; tmp.append( b); return tmp; }
```

  - Called from an initialization:

```
void f() { std::string x = concat( y, z); use(x); }
```

  - Called from an assignment statement:

```
void g() { std::string x; x = concat( y, z); use(x); }
```

- Translation using *copy-elision*
  - The tmp variable is removed, *r is used instead
  - The return statement does nothing
  - Explained using a hypothetical "C language with member functions":

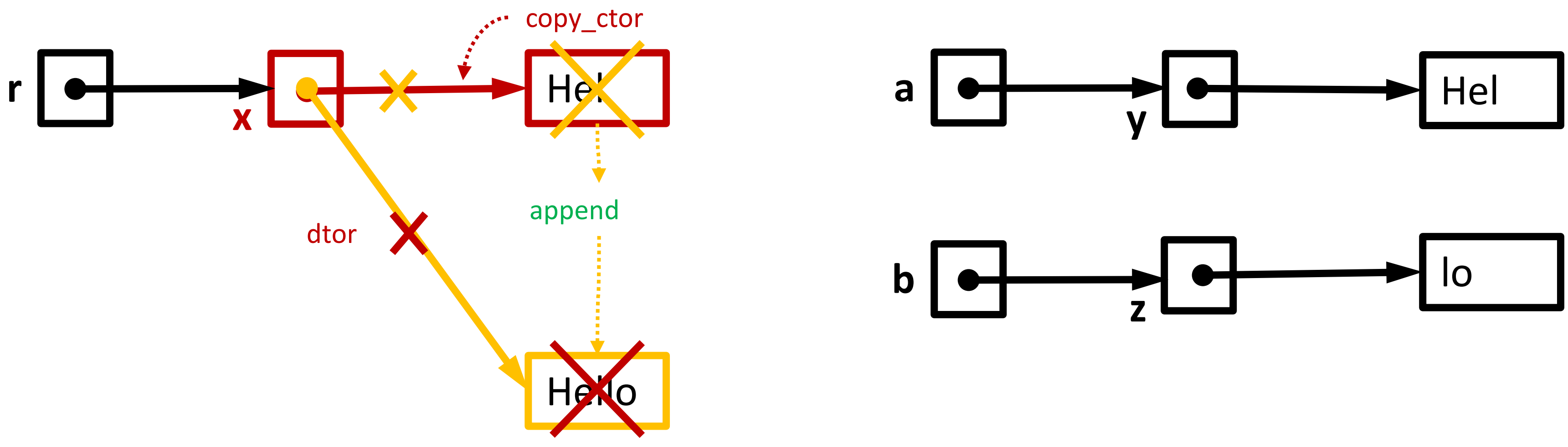
```
void concat( std::string * r, const std::string * a, const std::string * b)
{ r->copy_ctor(a); r->append(b); }
```

  - At the calling side, the compiler does not know whether copy-elision is applied inside
  - The calling code is the same

```
void f() { std::string x; concat( &x,&y,&z); use(&x); x.dtor(); }
void g() { std::string x,t; x.ctor();
  concat(&t,&y,&z); x.move_asgn(&t); t.dtor();
  use(&x); x.dtor(); }
```

  - The move-assignment may be used because t will be killed anyway

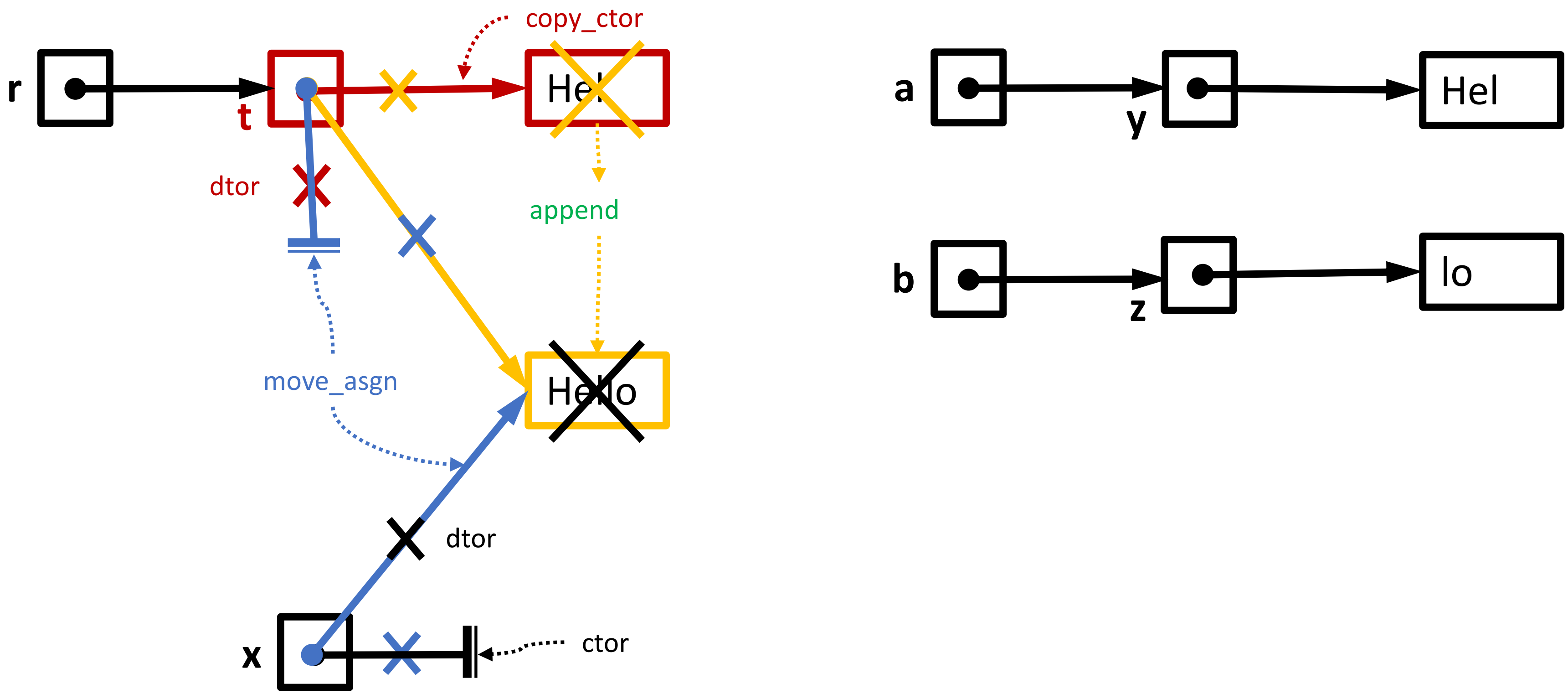
How compilers translate returning by value (in initialization)



- C-like equivalent

```
void concat( std::string * r, const std::string * a, const std::string * b)
{ r->copy_ctor(a); r->append(b); }
void f() {
  std::string x; concat(&x,&y,&z);  // std::string x = concat(y,z);
  use(&x); x.dtor(); }
```

How compilers translate returning by value (in assignment)



- C-like equivalent

```
void concat( std::string * r, const std::string * a, const std::string * b)
{ r->copy_ctor(a); r->append(b); }
void g()
{ std::string x,t; x.ctor();                         // std::string x;
  concat(&t,&y,&z); x.move_asgn(&t); t.dtor();   // x = concat(y,z);
  use(&x); x.dtor();}
```

Returning by value

- With move semantics and copy-elision, T pop_back() might be implemented
  - it still must return by value, but we can move the value quickly

```
template< typename T> class vector_ng {
public:
  T pop_back()
  {
    T tmp = std::move(arr_[size_-1]);  // move-constructor
    arr_[size_-1].T::~T();             // destruct the object in the array
    --size_;                           // mark the object as unused
    return tmp;                        // copy-elision or move
  }
private:
  T * arr_;
  std::size_t size_;
};
```

- The speed relies on move-constructor and move-assignment of the type T
- Programmers shall equip their classes with fast move operations
  - If all the data members have fast move operations, the class will acquire it automatically
  - Otherwise, programmers need to implement the move (and copy) operations explicitly