# Contemporary C++: Learning Modern C++ in a Modern Way

الماس فناوری ابری پاسارگاد- آلفا

مدرس: سعید امراللهی بیوکی

# Agenda 11/24

## Session 11. Classes invariants, Classes essential operations, RAII, move semantics, and more

- Constructors and class invariants
- C++98 Classes essential operations: Construction, Copy operations, and Destruction
- Law of big three
- The C++11 R-value references and move semantics
- Move operations: Move constructor and Move assignment operators
- Extending Classes essential operations: Construction, Copy and Move operations, and Destruction
- Law of big five
- Construction, Destruction, and Resource Acquisition Is Initialization
- Q&A

150 min (incl. Q & A)

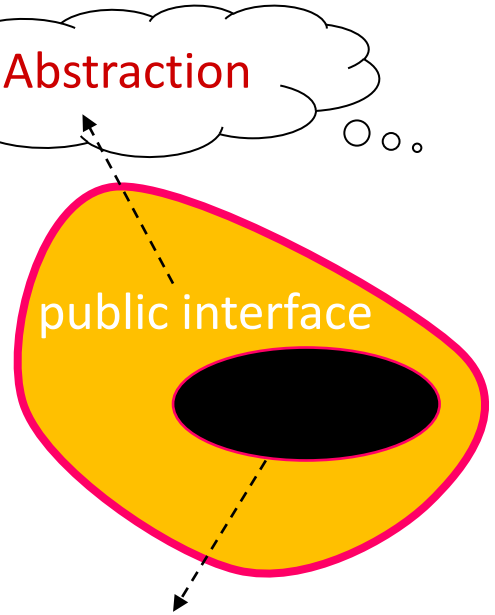
# Class Invariants

- Object-Oriented: Avoid *public* data members.

```
class Address { // very common
public:
     string get_country() const;
     void set_country(const string&);
     string get_city() const;
     void set_city(const string&);
     void set_street(const string&);
     string get_street() const;
private:
   string country;
   string city;
   string street;
   // …
};
```

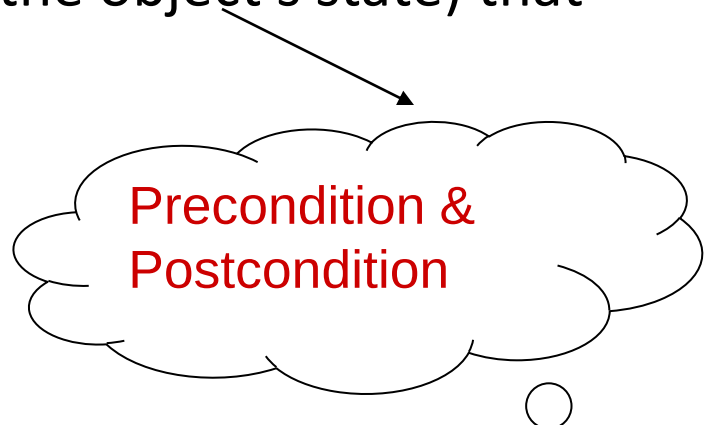
*vs.*

```
struct Address { // POD
   string country;
   string city;
   string street;
   // …
};
```





- Constructor: establishment an *invariant* for the class.
- Invariant: a condition of the representation of an object (the object's state) that should hold each time an *interface function* is called.




Rob Murray:

Representation Invariant: A *Representation invariant* is a predicate that is guaranteed to be true for every fully constructed object of the class.

# Class Invariant: an example

```
struct Dollar {
  Dollar(int d, int c) : dollars(d), cents(c) {}
  int dollars;
  int cents;
};
void f()
{
  Dollar d(10, 30);
  d.dollars = 12; // fine
  d.cents = 90; // fine
  d.dollars -= 12; // d.Dollars = 0: fine
  d.cents += 15; // d.Cents = 105; the invariant is broken!
}
```

- Invariant for Dollar class: cents must not be more than 99.

```
class Dollar { // invariant: <= 0 cents < 100
public:
  Dollar(int d, int c) : dollars(d), cents(c) {}
  std::pair<int, int> value() const { return std::make_pair<int, int>(d, c) {}
  double get_as_dollar() const { return dollars + (cents + 0.0) / 100; }
  void add(const Dollar& d) { dollars += d.dollars; cents += d.cents; }
  void add(int d, int c) { dollars += d; cents += c; }
private:
  int dollars, cents;
};
```

invariant may be invalidated

# Invariant cont.

1

```
// check invariant
inline bool Dollar::is_valid() const
{
  if(::abs(cents) > 99) return false;
  if((cents < 0) != (dollars < 0)) return false;

  return true;
}
```

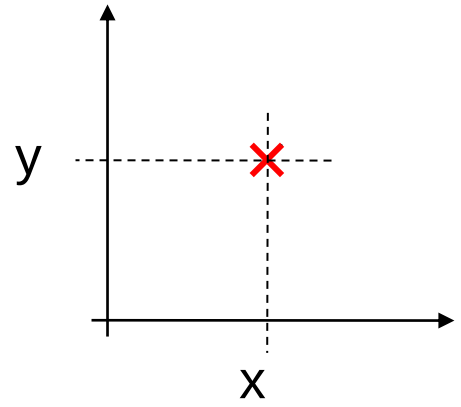
2

```
class RangeError {
  // …
};

class Dollar { // invariant: <= 0 cents < 100
public:
  Dollar(int d, int c) : dollars(d), cents(c) {
    if (!is_valid()) throw RangeError(); // check invariant
  }
  void add(const Dollar& d) { dollars += d.dollars; cents += d.cents;
    if (!is_valid()) throw RangeError(); // check invariant
  }
  void add(int d, int c) { dollars += d; cents += c;
    if (!is_valid()) throw RangeError(); // check invariant
  }
private:
  bool is_valid() const;
  int dollars, cents;
};
```

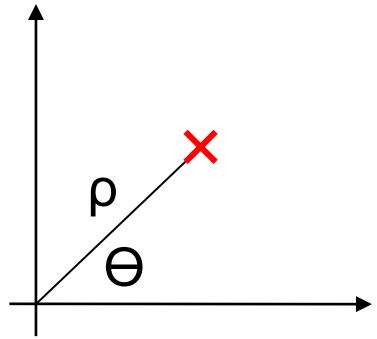
# Invariant and encapsulation cont.

```
class Complex { // Cartesian representation
  double real, image;
public:
  Complex(double r = 0.0, double i = 0.0) :
        real_(r), image_(i) {}
  double real() const { return real_; }
  double image() const { return image_; }
  void real(double r) { real_ = r; }
  void image(double i) { image_ = i; }
};
```




```
#include <cmath>
class Complex { // polar representation
  double r_, theta;
public:
  Complex(double r = 0.0, double i = 0.0) :
        r_(sqrt(r * r + i * i)), theta(atan(i/r)) {}
  double real() const { return r_ * cos(theta); }
  double image() const { return r_ * sin(theta); }

  void real(double r) { double y = r_ * sin(theta);
        r_= sqrt(r * r + y * y);  theta = acos(r / r_); }
  void image(double i) { double x = r_ * cos(theta);
        r_= sqrt( x * x + i * i); theta = asin(i / r_); }
};
```

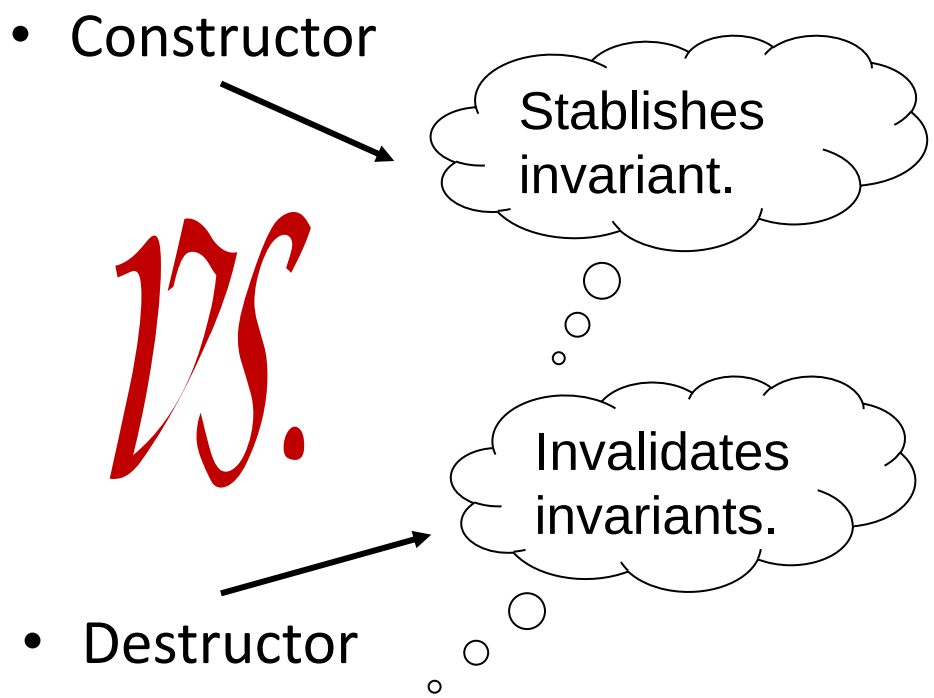


# Class invariants cont.

• Date, Dollar, Rational number, Stack, String, Vector, … have invariant. They should be represented as a class.

• Address, Pair, Personal Record, … don't have invariant. They are POD. It is better to represent them with struct.

• Stroustrup: Every class should have some invariant.




Classes should Enforce Invariants.

# Get/Set Interface

- Get/set member functions related to poorly designed interfaces.
- Get/set member functions are often used as Band-Aids to patch broken interfaces.
- Drawbacks:
  - expose implementation technique/object secrets
  - The ripple effect: changing the implementation technique breaks user code
  - Cluttered interface!

```
class BadAccount {
public:
    void set_balance(double);
    double get_balance();
};
```

- Use the vocabulary of problem domain (real-world)
- Class design: export data vs. provide services.

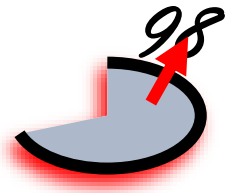
- Is this really the way bank account works?
- Do your set a new balance at the ATM?
- Does this class have purpose at all (invariant)

Change the balance by adding and subtracting

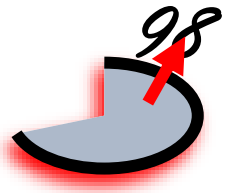
```
class BadAccount {
public:
    void deposit(double);
    void withdraw(double);
    double balance() const;
};
```

# Essential operations

# Essential operations



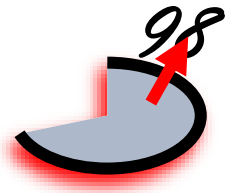
# Essential operations



```
class X {
public:
    X(Sometype); // "Ordinary constructor": create an object

    X(); // default constructor
    X(const X&); // copy constructor
    X& operator=(const X&); // copy assignment operator
    ~X(); // destructor
};
```

# Essential operations

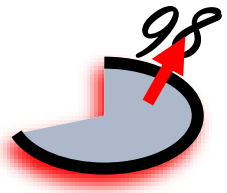



Special member functions

```
class X {
public:
    X(Sometype); // "Ordinary constructor": create an object

    X(); // default constructor
    X(const X&); // copy constructor
    X& operator=(const X&); // copy assignment operator
    ~X(); // destructor
};
```

# Essential operations



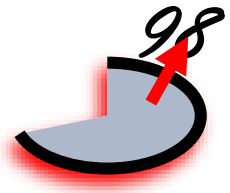

Special member functions

```
class X {
public:
    X(Sometype); // "Ordinary constructor": create an object

    X(); // default constructor
    X(const X&); // copy constructor
    X& operator=(const X&); // copy assignment operator
    ~X(); // destructor
};
```

- Ordinary constructor: They are as varied as the classes they serve

# Essential operations



Special member functions

```
class X {
public:
    X(Sometype); // "Ordinary constructor": create an object

    X(); // default constructor
    X(const X&); // copy constructor
    X& operator=(const X&); // copy assignment operator
    ~X(); // destructor
};
```

- Ordinary constructor: They are as varied as the classes they serve

```
class SocialSecurityNumber { // A typical national number wrapper
public:
    SocialSecurityNumber(const std::string& nn) : national_number{nn} {}
    // …
private:
    std::string national_number;
};
```

# Default constructor

C++ standard working draft

- A *default constructor* for a class X is a constructor of class X for which each parameter … has a default argument including the case of a constructor with no parameters.

Bjarne Stroustrup

- A constructor that can be invoked without an argument is called a *default constructor*.
- When does it make sense to have a default constructor?

- When we can establish the invariant for the class with a meaningful and obvious default value.

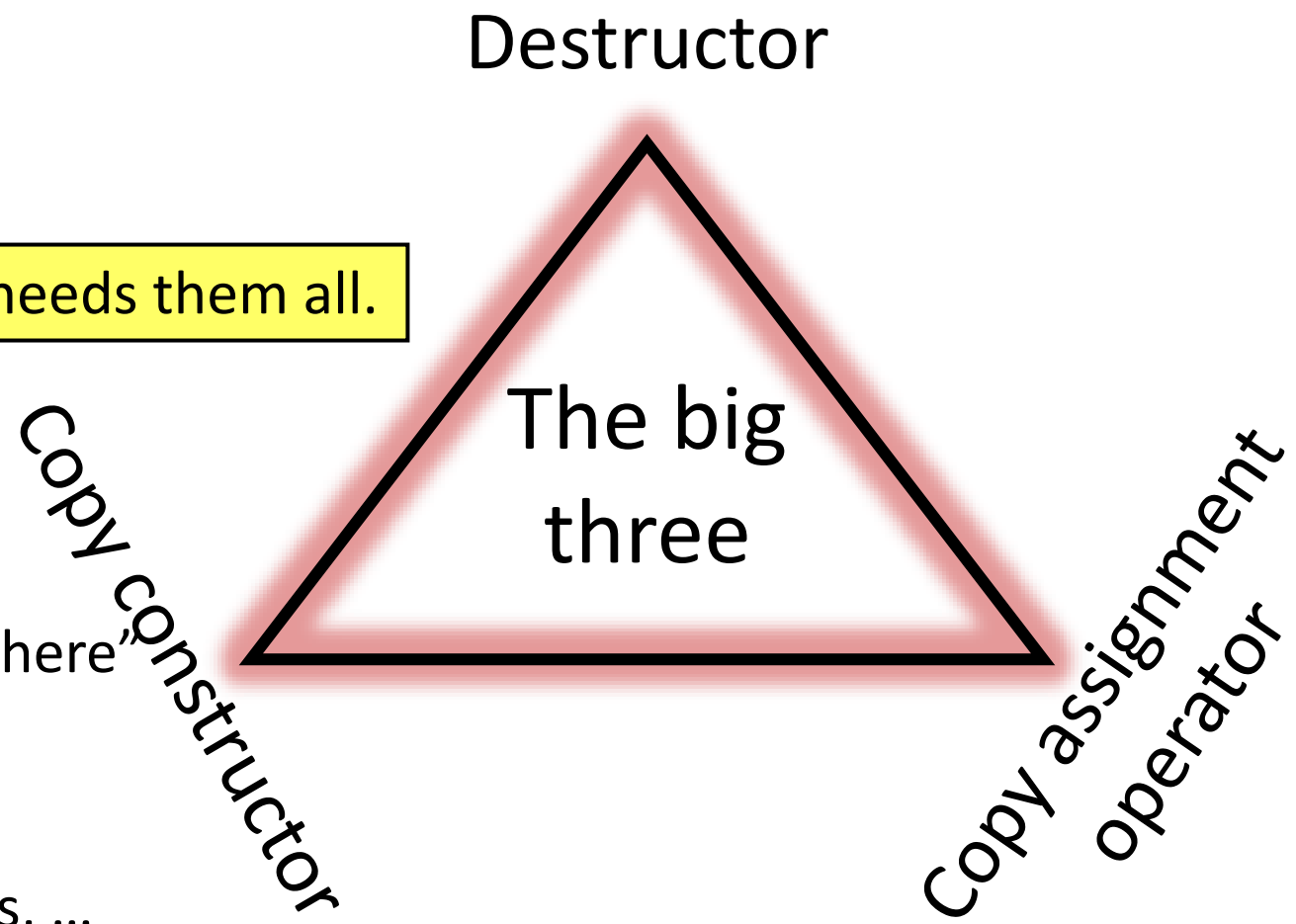
- For every type T, T() or T{} is default value.

```
class Point; // forward declaration
class Date;
class Rational;
class Time;
class X {
public:
    X() = delete;
};
void f()
{
    std::vector<Point> vp; // OK
    std::array<Date, 10> da; // OK
    Rational r[10]; // OK
    std::vector<Time> vt; // OK
    std::vector<X> vx; // error
    std::vector<X*> vxp; // OK
}
```

# law of Big three

- 1991, comp.lang.c++ discussion group

If a class needs any of the Big Three, it needs them all.




- Examples: the classes Vector or String
- Destructor: resource release
- Resource: Something you "get from somewhere" and that you must give back once you have finished using it.
- Memory, files, locks, thread handles, sockets, …
- A class that needs a destructor almost always also needs copy constructor and a copy assignment.
- If a class has a pointer or reference member, if often needs the big three.

# Copy terminology

# Copy terminology

*vs.*

# Copy terminology

Copy a pointer or reference

*vs.*

# Copy terminology

Copy a pointer or reference

*vs.*

Copy the information pointed to (referred to)

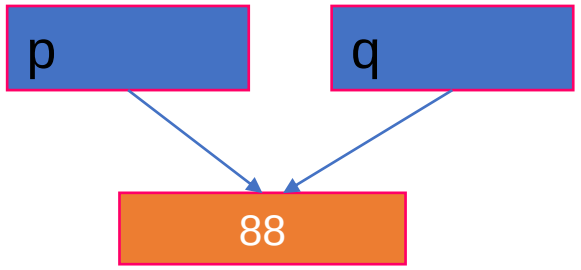
# Copy terminology

Copy a pointer or reference

*vs.*

Copy the information pointed to (referred to)

```
int* p = new int{77};
int* q = p;
*p = 88
```



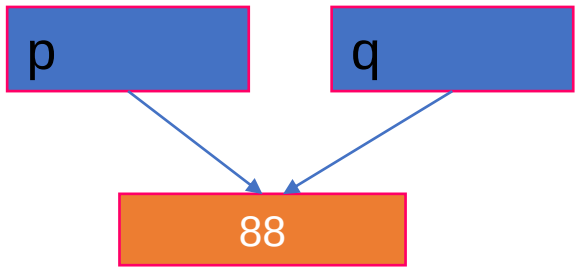
# Copy terminology

Copy a pointer or reference

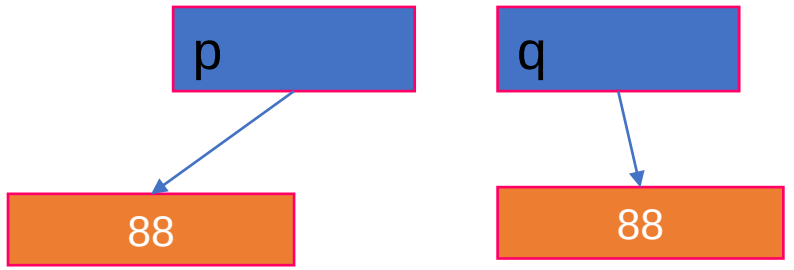
*vs.*

Copy the information pointed to (referred to)

```
int* p = new int{77};
int* q = p;
*p = 88
```




```
int* p = new int{77};
int* q = new int{*p};
*p = 88
```



# Copy terminology

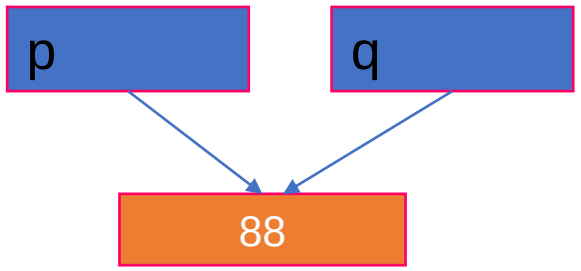
Copy a pointer or reference

*vs.*

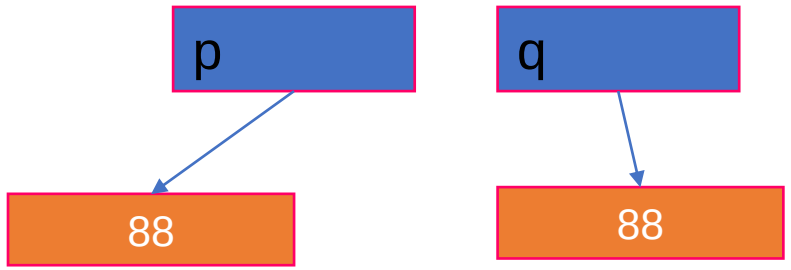
Copy the information pointed to (referred to)

Shallow copy

```
int* p = new int{77};
int* q = p;
*p = 88
```




```
int* p = new int{77};
int* q = new int{*p};
*p = 88
```



# Copy terminology

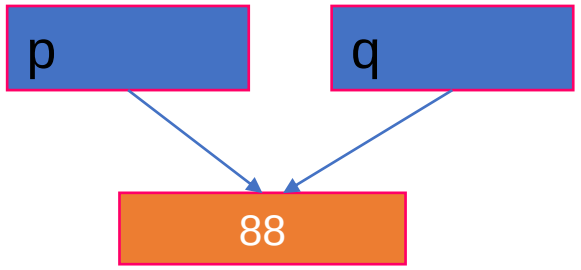
Copy a pointer or reference

*vs.*

Copy the information pointed to (referred to)

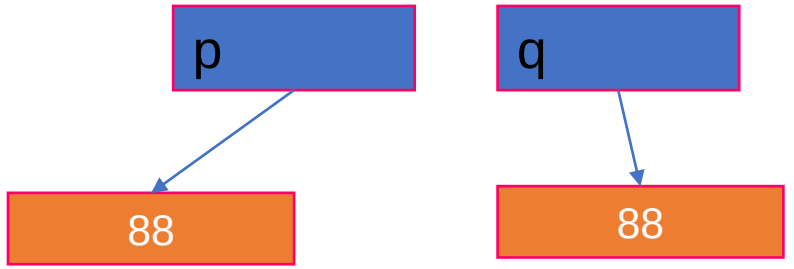
Shallow copy

```
int* p = new int{77};
int* q = p;
*p = 88
```




Deep copy

```
int* p = new int{77};
int* q = new int{*p};
*p = 88
```



# Copy terminology

Copy a pointer or reference

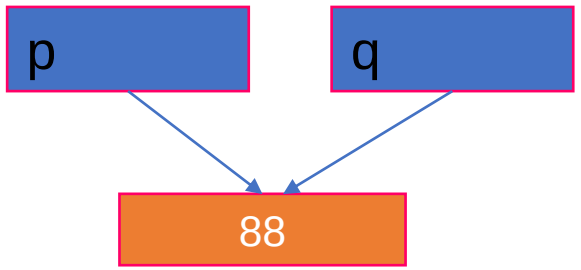
*vs.*

Copy the information pointed to (referred to)

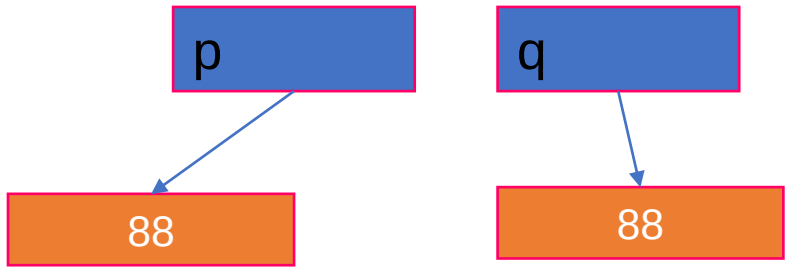
Shallow copy

Deep copy

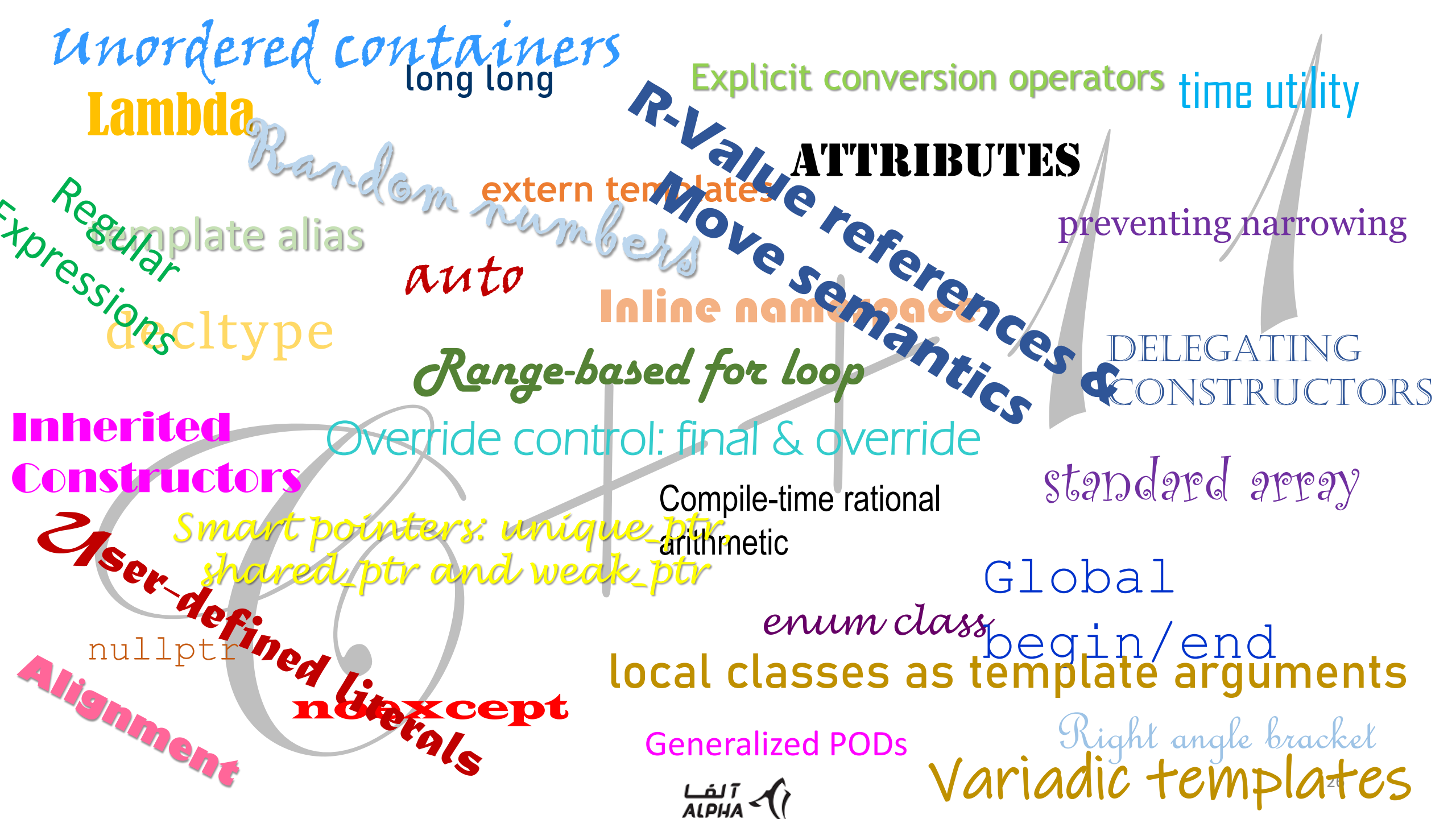
```
int* p = new int{77};
int* q = p;
*p = 88
```

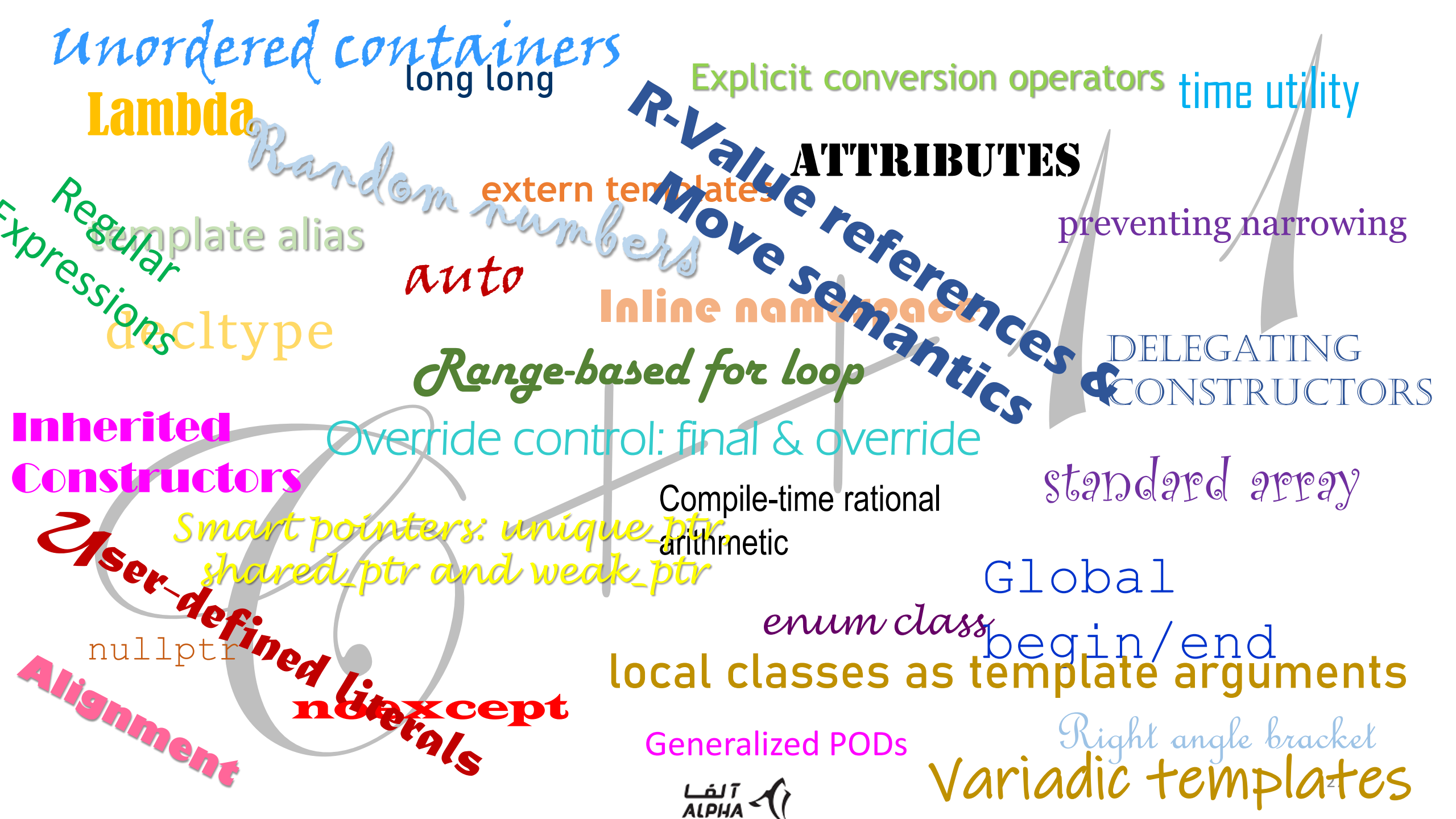



```
int* p = new int{77};
int* q = new int{*p};
*p = 88
```




- Pointer semantics or Reference semantics
- Value semantics → work just like integers


Unordered containers
long long
Explicit conversion operators
time utility
Lambda
Random numbers
R-Value references &
Move semantics
ATTRIBUTES
extern templates
Regular Expressions
template alias
preventing narrowing
auto
Inline namespace
decltype
Range-based for loop
DELEGATING CONSTRUCTORS
Override control: final & override
Inherited Constructors
standard array
Compile-time rational arithmetic
Smart pointers: unique_ptr
shared_ptr and weak_ptr
User-defined literals
Global
enum class
begin/end
nullptr
local classes as template arguments
noexcept
Alignment
Generalized PODs
Right angle bracket
Variadic templates
ALPHA

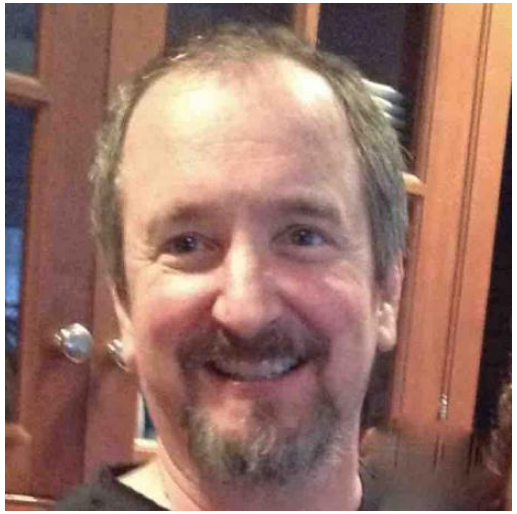

Unordered containers
long long
Explicit conversion operators
time utility
Lambda
Random numbers
R-Value references &
Move semantics
ATTRIBUTES
extern templates
Regular Expressions
template alias
preventing narrowing
auto
Inline namespace
decltype
Range-based for loop
DELEGATING CONSTRUCTORS
Override control: final & override
Inherited Constructors
standard array
Compile-time rational arithmetic
Smart pointers: unique_ptr
shared_ptr and weak_ptr
User-defined literals
Global
enum class
begin/end
nullptr
local classes as template arguments
noexcept
Alignment
Generalized PODs
Right angle bracket
Variadic templates
ALPHA

# R-value references & Move semantics

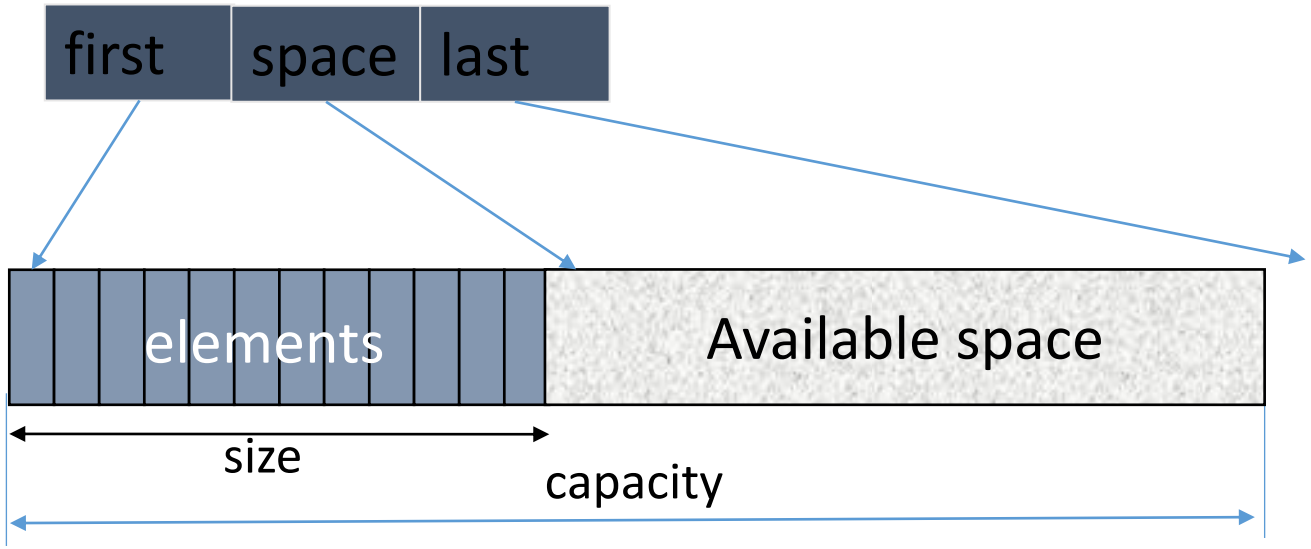
- The problem with copy semantic
- Move semantics
- Copy operations vs. Move operations
- Under the hood: R-value reference

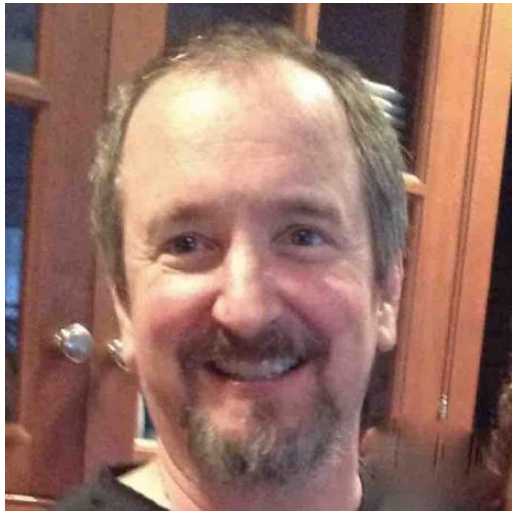
# A motivating example: standard vector implementation

# A motivating example: standard vector implementation

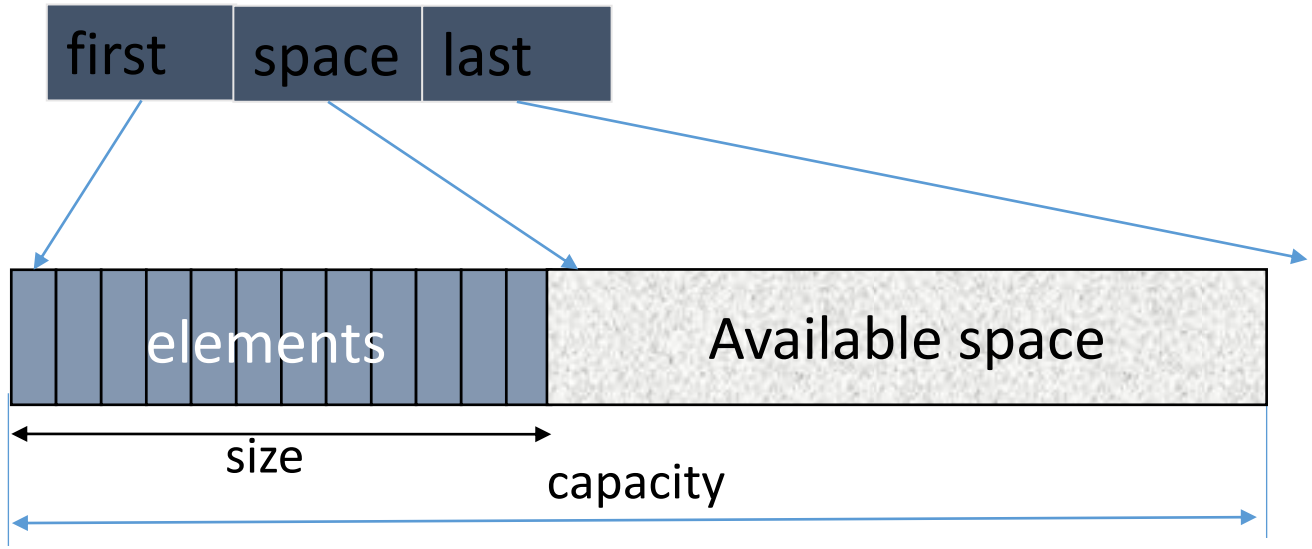


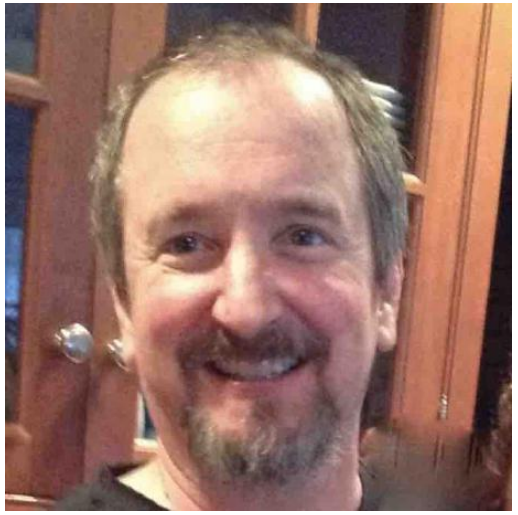
# A motivating example: standard vector implementation



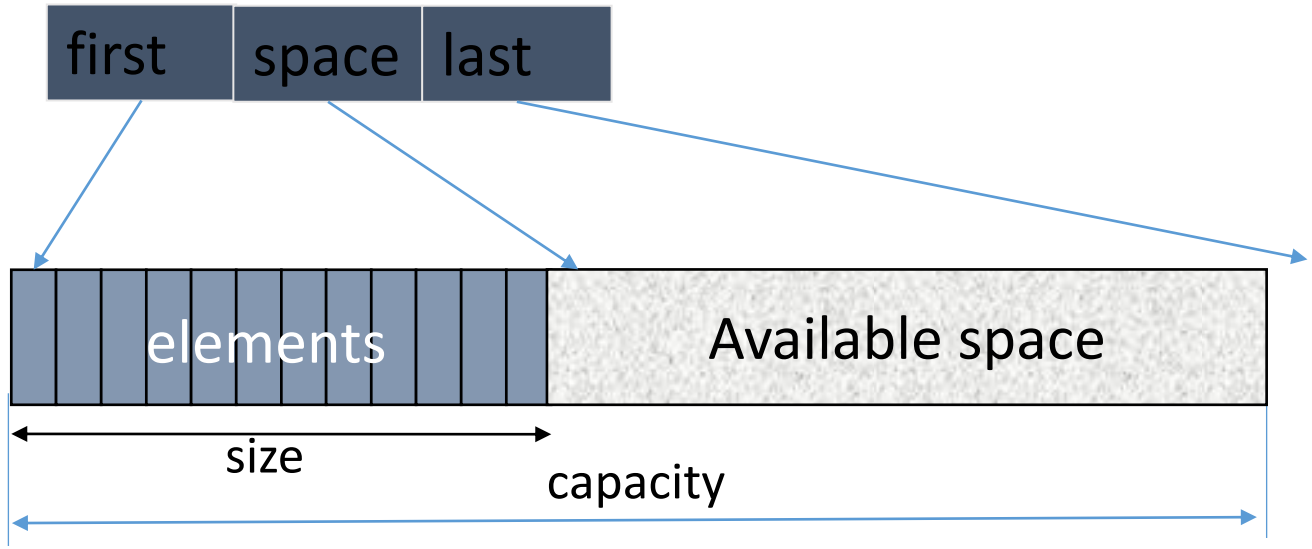
# A motivating example: standard vector implementation

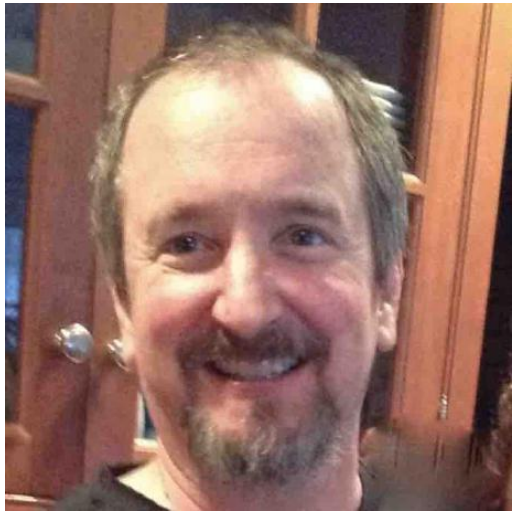


```
template <class T /*, allocator */ >
class vector {
  T* first;
  T* space;
  T* last;
  // …
}; // 24 bytes (on 64 bits systems)
```

# A motivating example: standard vector implementation



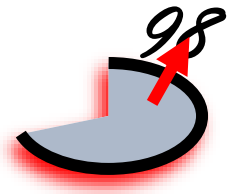



```
template <class T /*, allocator */ >
class vector {
  T* first;
  T* space;
  T* last;
  // …
}; // 24 bytes (on 64 bits systems)
```

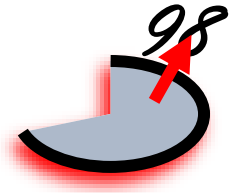
first → begin()
space → end()
last → capacity()
size() → end() – begin()

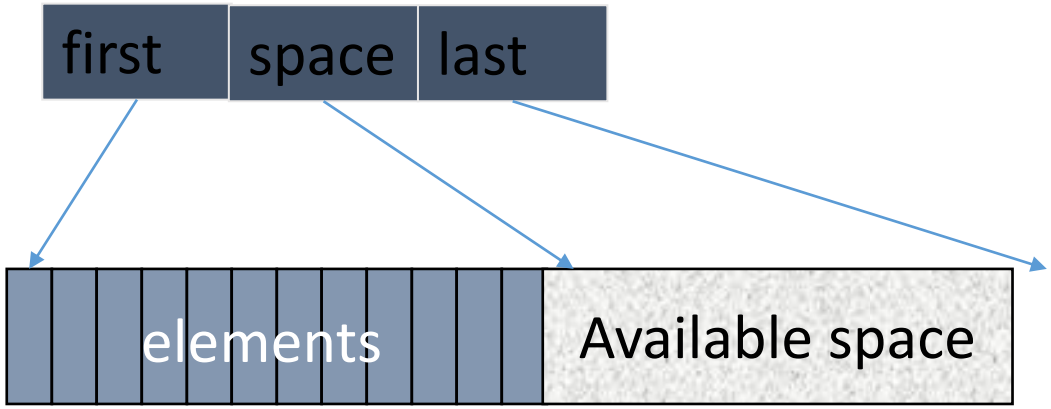
# A motivating example: standard vector implementation

# A motivating example: standard vector implementation

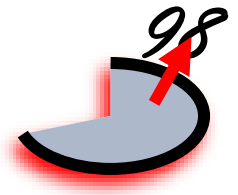


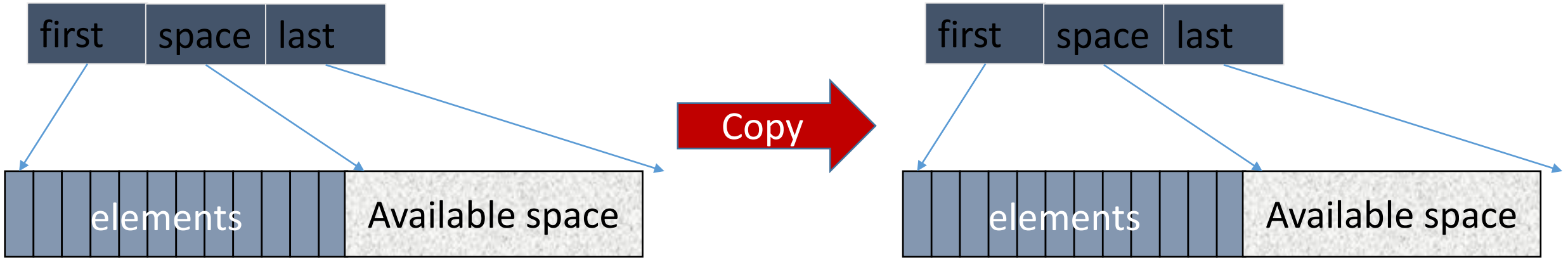
# A motivating example: standard vector implementation



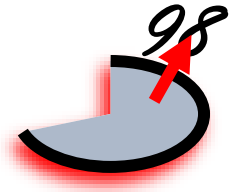


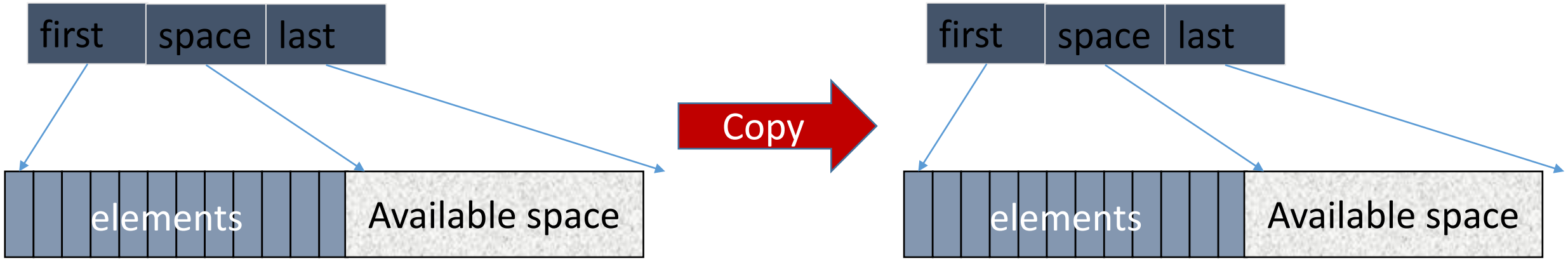
# A motivating example: standard vector implementation





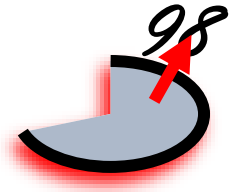
# A motivating example: standard vector implementation

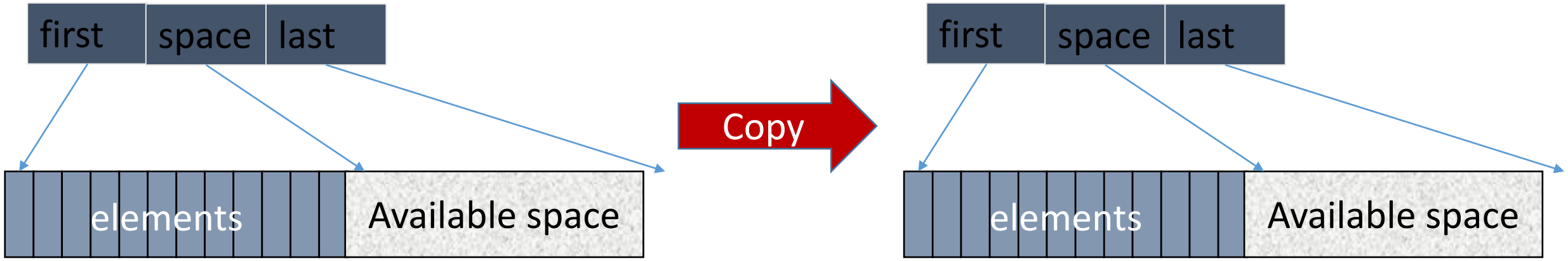





- Copying can be costly specially for large objects.
    1. allocate enough memory to hold the requested number of elements
    2. copy the elements into the newly allocated memory
    3. set up the target vector's data structure to use the new storage

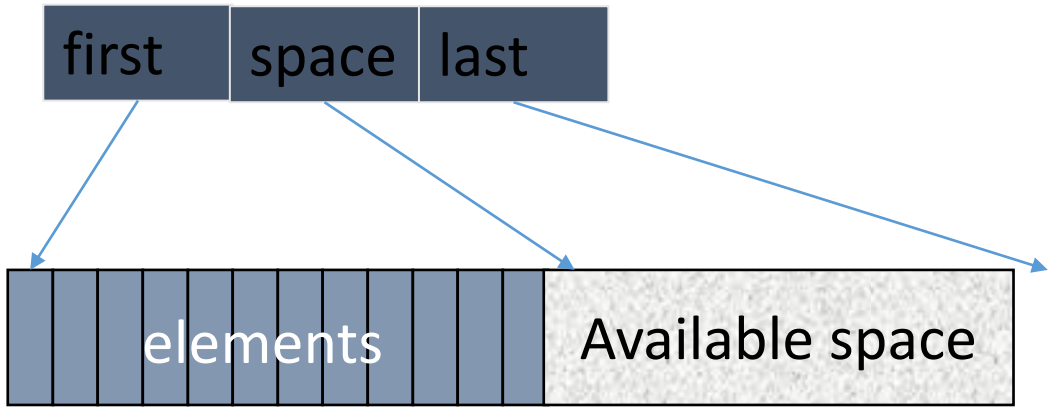
# A motivating example: standard vector implementation



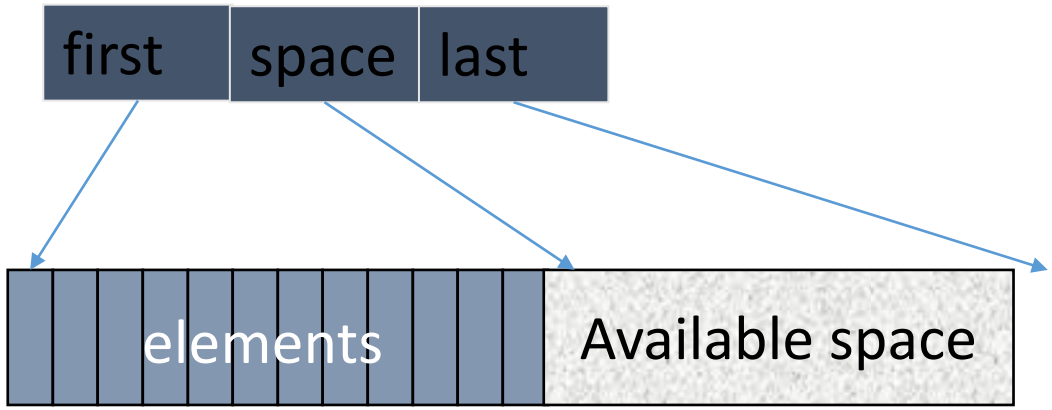



- Copying can be costly specially for large objects.
    1. allocate enough memory to hold the requested number of elements
    2. copy the elements into the newly allocated memory
    3. set up the target vector's data structure to use the new storage
- Most of the time we don't need to original
    4. destroy the original sequence of elements
    5. deallocate the original memory.

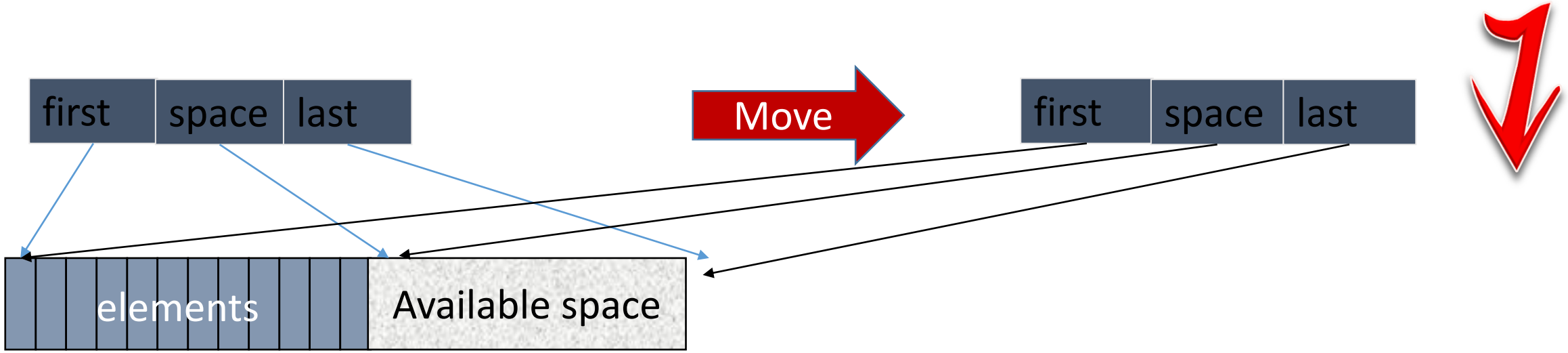
# A motivating example: standard vector implementation



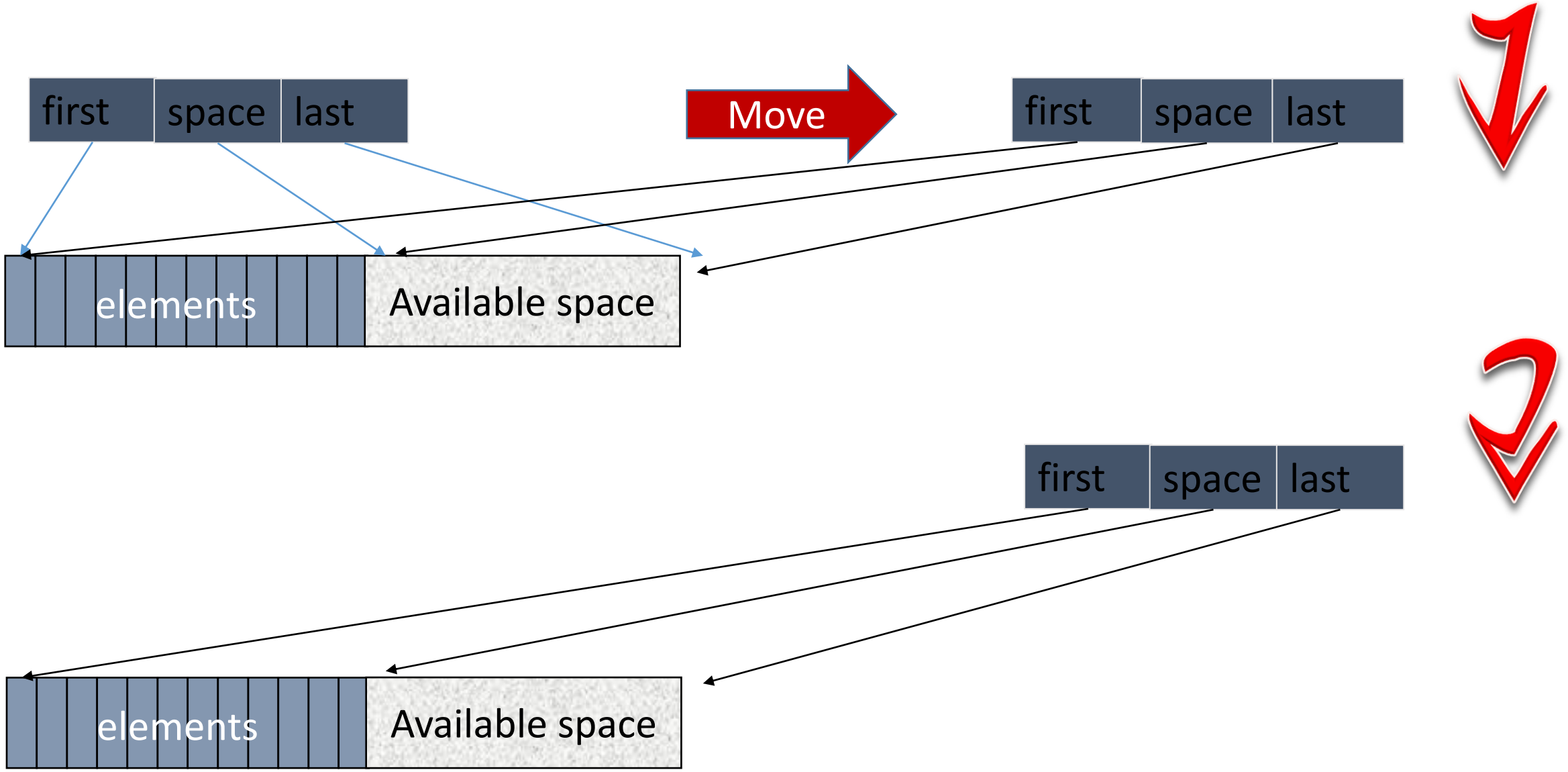
# A motivating example: standard vector implementation



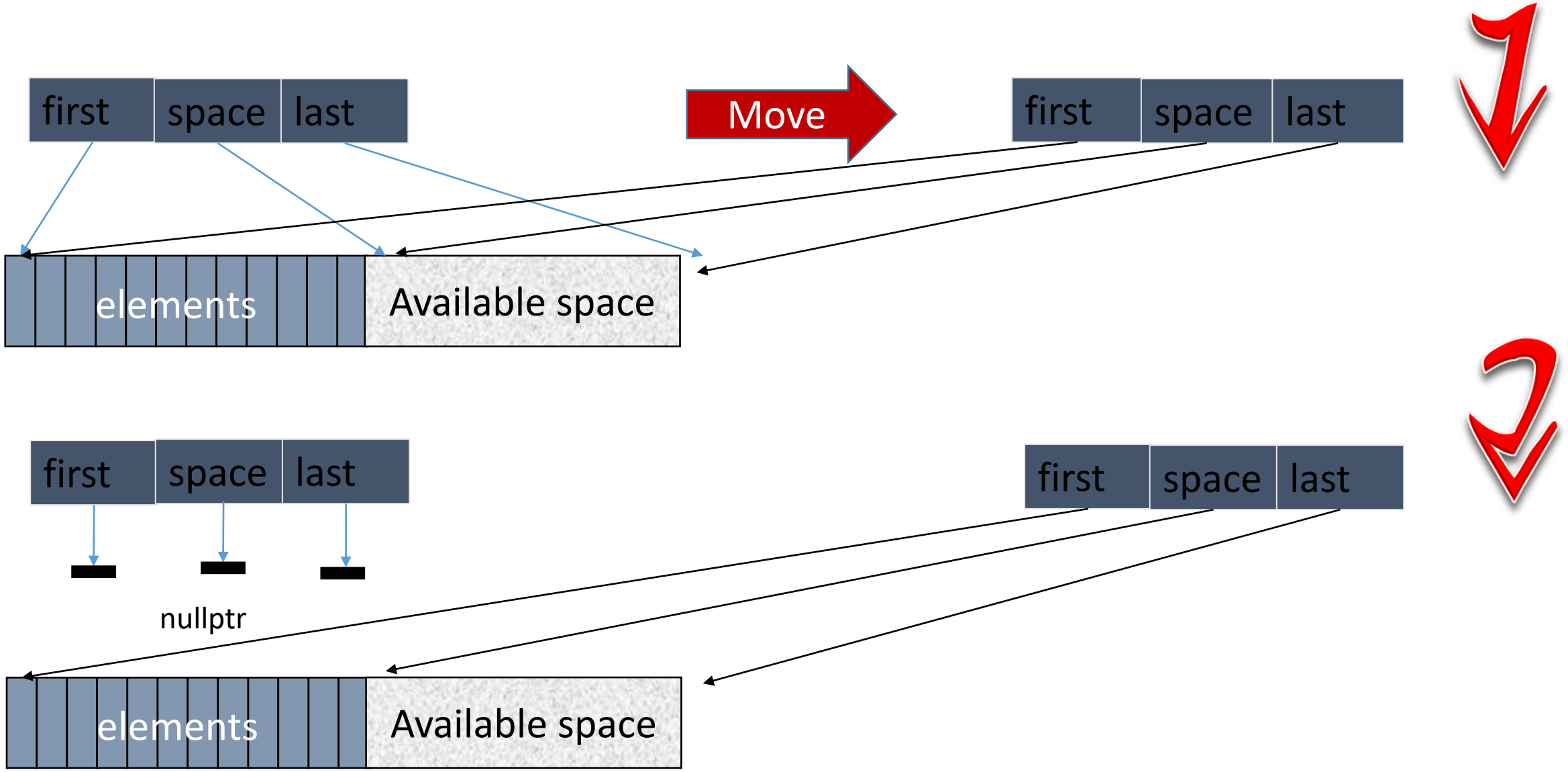
# A motivating example: standard vector implementation



# A motivating example: standard vector implementation



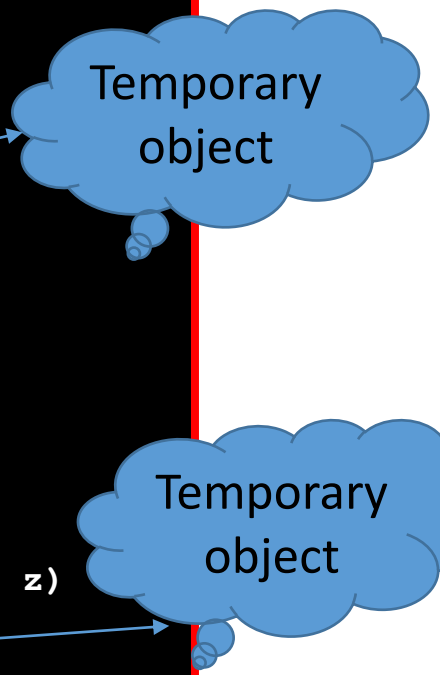
# A motivating example: standard vector implementation



# Temporary objects

```
using std::vector;
using std::lenth_error;
vector<double> operator+(const vector<double>& a, const vector<double>& b)
{
    if (a.size() != b.size())
        throw lenth_error("different length error");
    vector<double> res(a.size()); // temporary object
    for (int i = 0; i != a.size(); ++i) {
        res[i] = a[i] + b[i];
    }
    return res;
} // res is destroyed

void f(const vector<double>& x, const vector<double>& y, const vector<double>& z)
{
    vector<int> r;
    // …
    r = x + y + z; // 2 copy operations: one for each +: a lot of temporary objects
    // …
}
```

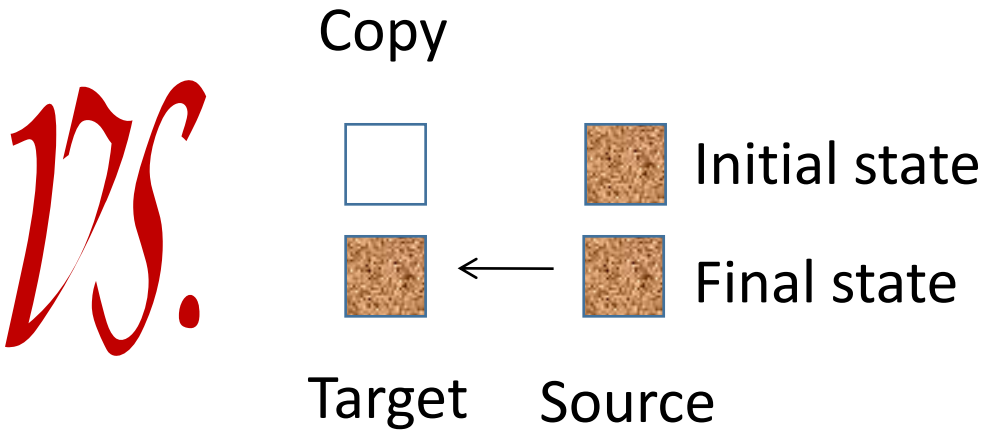



- We didn't really want a copy, we just wanted to get the result out of a function: we wanted to move a vector rather than to copy it.
- If you want to borrow my phone, I pass my phone to you rather than making you your own copy.

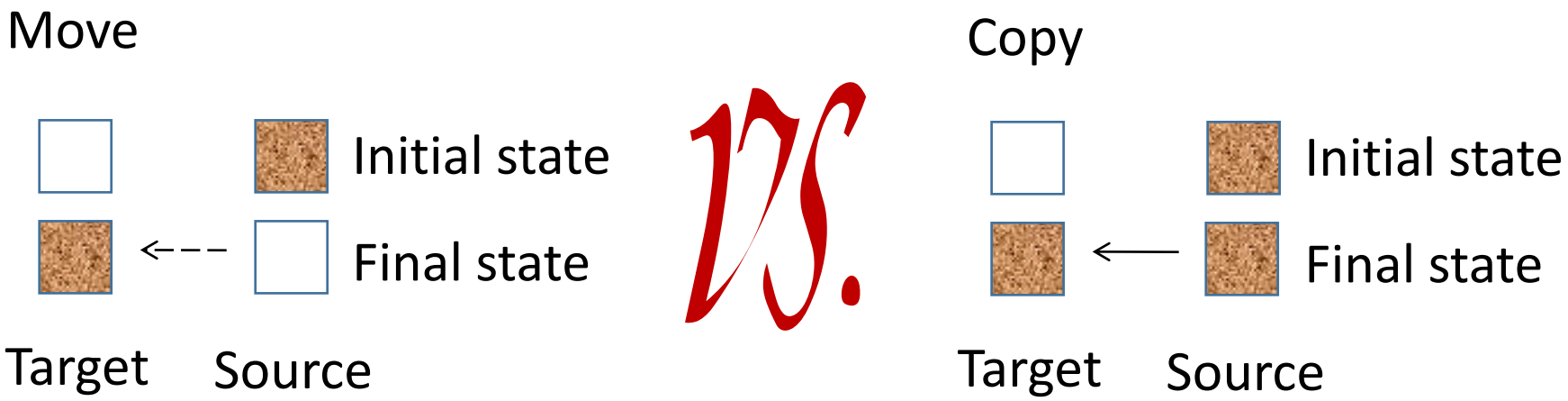
# Move vs. copy

# Move vs. copy

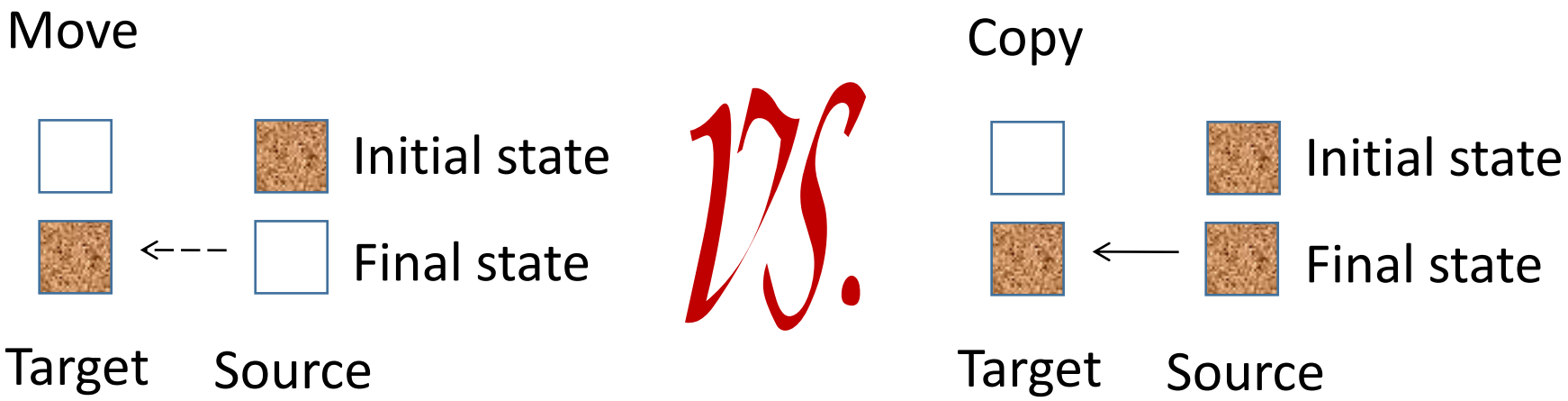
# Move vs. copy



# Move vs. copy

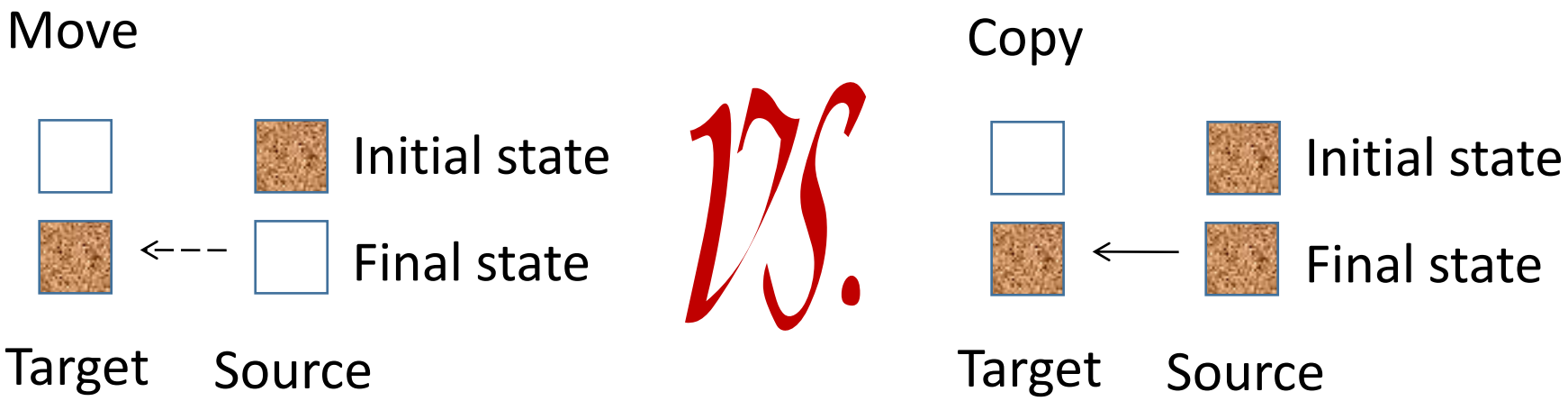

Move
Initial state
Final state
Target
Source
vs.
Copy
Initial state
Final state
Target
Source

# Move vs. copy



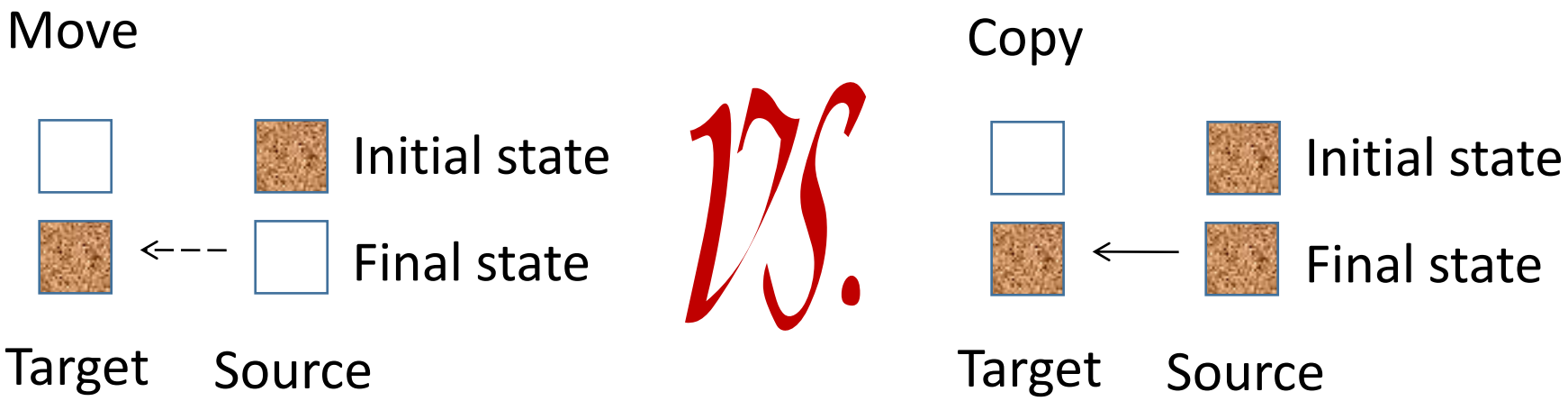

- The difference between a copy and a move is that a copy leaves the source unchanged.

# Move vs. copy



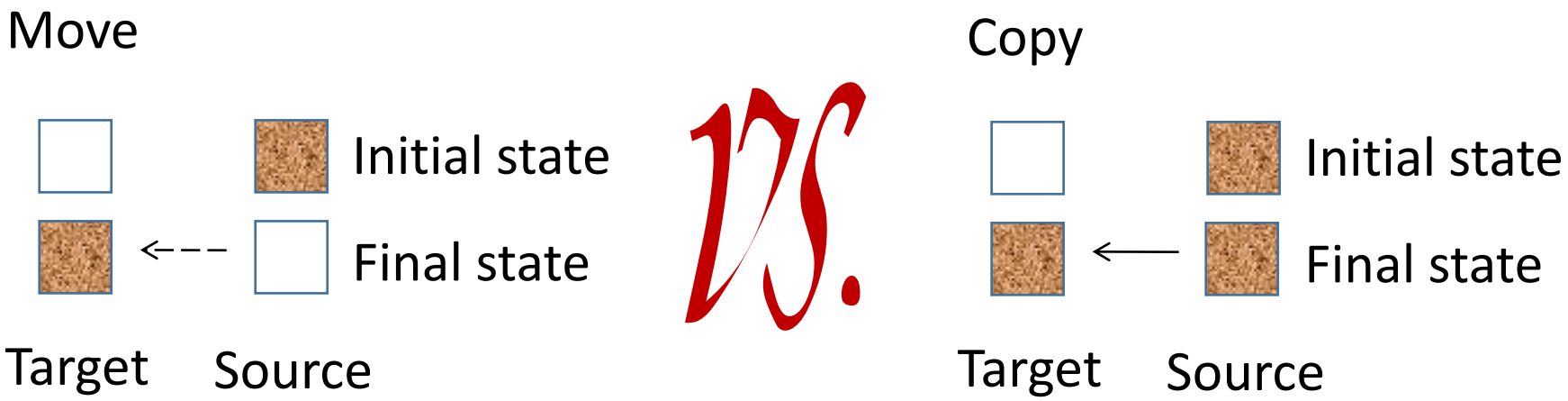

- The difference between a copy and a move is that a copy leaves the source unchanged.
- Move operation: Avoid temporary objects

# Move vs. copy



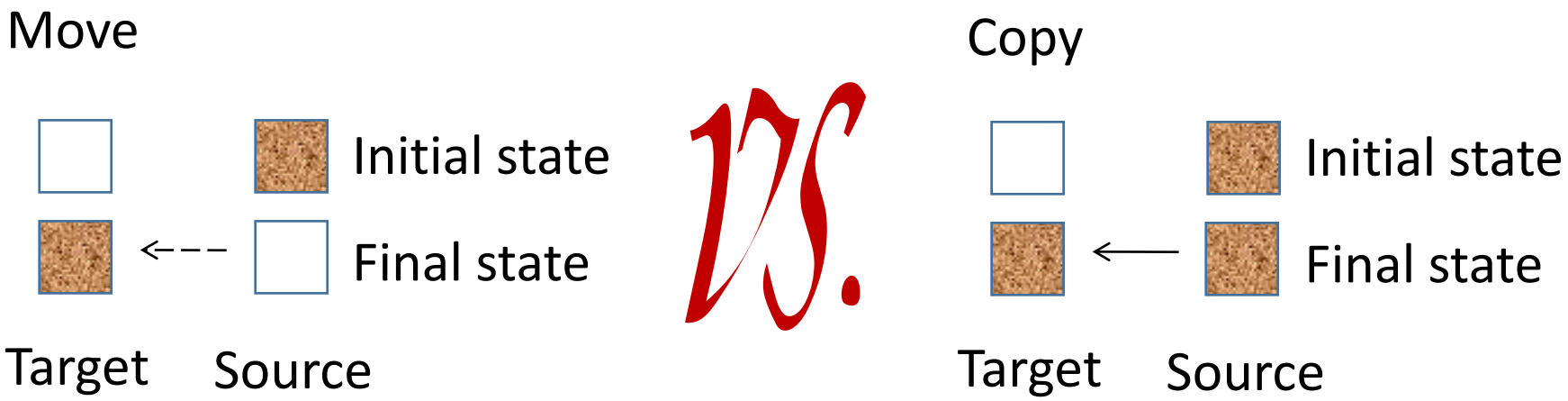

- The difference between a copy and a move is that a copy leaves the source unchanged.
- Move operation: Avoid temporary objects
- Copying can be costly for large objects.

# Move vs. copy




- The difference between a copy and a move is that a copy leaves the source unchanged.
- Move operation: Avoid temporary objects
- Copying can be costly for large objects.
- Copying can be costly for large number of objects.

# Move vs. copy




- The difference between a copy and a move is that a copy leaves the source unchanged.
- Move operation: Avoid temporary objects
- Copying can be costly for large objects.
- Copying can be costly for large number of objects.
- By default, objects are copied.

# Extending special member functions

- The default constructor, copy constructor, copy assignment operator and destructor are *special member functions*.

  - International Standard ISO/IEC 14882. Programming Languages- C++, 2nd edition, 2003.

Default constructor + Copy operations + Destructor

- The *default constructor*, *copy constructor* and *copy assignment operator*, *move constructor* and *move assignment operator*, and *destructor* are *special member functions.*

  - International Standard ISO/IEC JTC1 SC22 WG21 N3290. Programming Languages- C++, 2011.

Default constructor + Copy operations + Move operations + Destructor

# Classes Special member functions

# Classes Special member functions

- C++ Language-technical rule:

> Provide as good support for user-defined types as for built-in types.

# Classes Special member functions

- C++ Language-technical rule:

> Provide as good support for user-defined types as for built-in types.

- The *default constructor*, *copy constructor* and *copy assignment operator*, *move constructor* and *move assignment operator*, and *destructor* are *special member functions.*

# Classes Special member functions

- C++ Language-technical rule:

> Provide as good support for user-defined types as for built-in types.

- The *default constructor*, *copy constructor* and *copy assignment operator*, *move constructor* and *move assignment operator*, and *destructor* are *special member functions.*
- The implementation will implicitly define them if they

are used.

# Classes Special member functions

- C++ Language-technical rule:

> Provide as good support for user-defined types as for built-in types.

- The *default constructor*, *copy constructor* and *copy assignment operator*, *move constructor* and *move assignment operator*, and *destructor* are *special member functions.*
- The implementation will implicitly define them if they are used.

```
class X { // the empty class
};
```

# Classes Special member functions

- C++ Language-technical rule:

> Provide as good support for user-defined types as for built-in types.

- The *default constructor*, *copy constructor* and *copy assignment operator*, *move constructor* and *move assignment operator*, and *destructor* are *special member functions.*
- The implementation will implicitly define them if they are used.

```
class X { // the empty class
};
```

Compiler generates

```
class X {
public:
    X(Sometype); // "Ordinary constructor": create an object

    X() =default; // default constructor
    X(const X&) =default; // copy constructor
    X(X&&) =default; // move constructor
    X& operator=(const X&) =default; // copy assignment operator
    X& operator=(X&&) =default; // move assignment operator
    ~X() = default; // destructor
};
```

# Classes Special member functions

- C++ Language-technical rule:

> Provide as good support for user-defined types as for built-in types.

- The *default constructor*, *copy constructor* and *copy assignment operator*, *move constructor* and *move assignment operator*, and *destructor* are *special member functions.*
- The implementation will implicitly define them if they are used.
- Constructors, destructors, and copy and move operations for a type are not logically separate. We must define them as a matched set or suffer logical or performance problems.

```
class X { // the empty class
};
```

Compiler generates

```
class X {
public:
    X(Sometype); // "Ordinary constructor": create an object

    X() =default; // default constructor
    X(const X&) =default; // copy constructor
    X(X&&) =default; // move constructor
    X& operator=(const X&) =default; // copy assignment operator
    X& operator=(X&&) =default; // move assignment operator
    ~X() = default; // destructor
};
```

# Copy and move applications

- There are five situations in which an object is copied or moved:
  - As the source of an assignment
  - As an object initializer
  - As a function argument
  - As a function return value
  - As an exception

NoisySpecialMemberFunctions

Prog.

# Special member functions: Implementation

```
class Vector { // dynamic array
    int* elem_;
    int size_;
public:
    explicit Vector(int s) : elem_{ new int[ size_ = s ] } { /* initialize v */ } // memory allocated
    Vector(const Vector&); // copy ctor: exactly as before
    Vector& operator=(const Vector&); // copy assignment op.: exactly as before
    Vector(Vector&&); // move ctor
    Vector& operator=(Vector&&); // move assignment op.
    ~Vector() { delete [] elem_; } // memory recycled (released)
};
```

- Copy constructor

```
Vector::Vector(const Vector& v) :
    elem_{new int[size_ = v.size_]}
    // "copy the elements" from v
{
    for (int i = 0; i != size; ++i)
        elem_[i] = v.elem_[i]
}
```

- Move constructor

```
Vector::Vector(Vector&& v) :
    elem_{v.elem_}, size_{v.size_}
    // "grab the elements" from v
{
    v.elem_ = nullptr; // now v has no elements
    v.size_ = 0;
}
```

- A move constructor does not take a const argument.
- A move assignment operator does not take a const argument.

# Special member functions: Implementation cont.

- Copy assignment operator

```
Vector& Vector::operator=(const Vector& v)
{
    if (&v != this) { // handle self-assignment
        delete[] elem_;
        elem_ = new int[size_ = v.size_];
        for (int i = 0; i < v.size_; ++i)
                elem_[i] = v.elem_[i];
    }
    return *this;
}
```

- Move assignment operator

```
Vector& Vector::operator=(Vector&& a)
{
    // grap the elements from v
    elem_ = v.elem_;
    size_ = v.size_;

    v.elem_ = nullptr; // now v has no elements
    v.size_ = 0;

    return *this;
}
```

- *Grab* the elements = *Steal* the states of original objects = Pilfering resources

- Move semantics is mostly about performance optimization: the ability to move an expensive object from one address in memory to another, while pilfering resources of the source in order to construct the target with minimum expense.

*from the original proposal Howard Hinnant, Peter Dimov, Dave Abrahams. N1377=02-0035-A Proposal to Add Move Semantics Support to the C++ Language.*


Vector4
Prog.

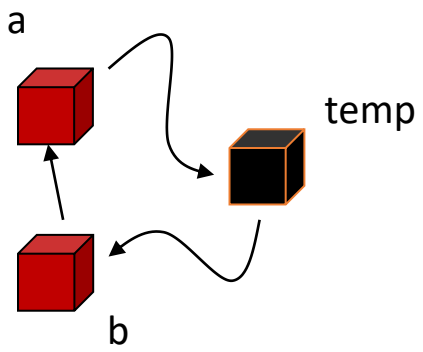
# another example: Swap

- Copy-based implementation

# another example: Swap

- Copy-based implementation

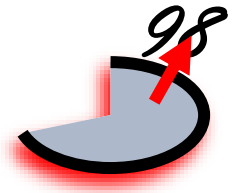
```
template<class T>
void swap(T& a, T& b)
{
  const T temp = a;
  a = b;
  b = temp;
}
```

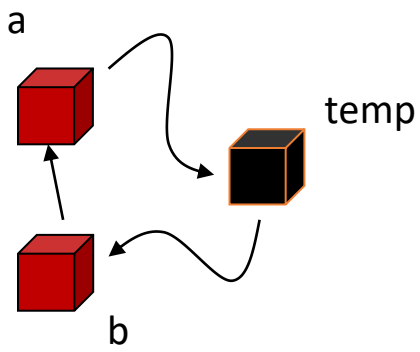


# another example: Swap

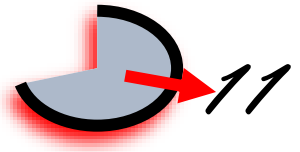
- Copy-based implementation

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



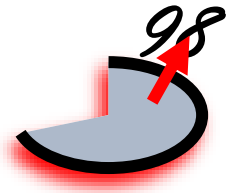


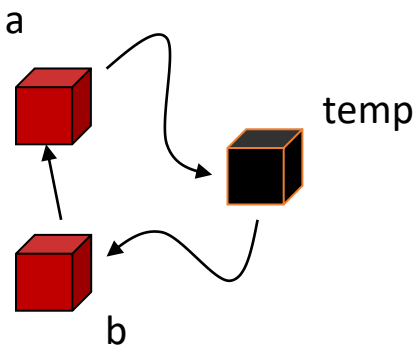
# another example: Swap



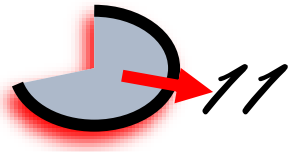

- Copy-based implementation

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



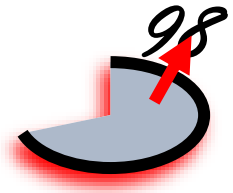



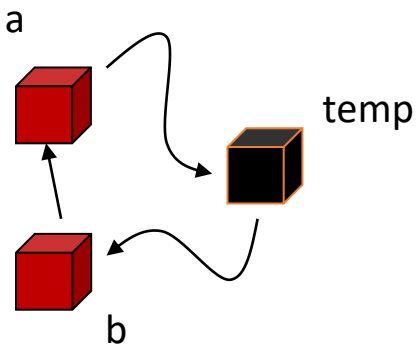
# another example: Swap




- Copy-based implementation

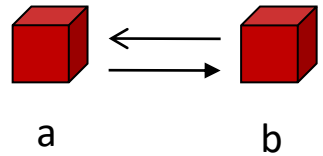
```
template<class T>
void swap(T& a, T& b)
{
  const T temp = a;
  a = b;
  b = temp;
}
```



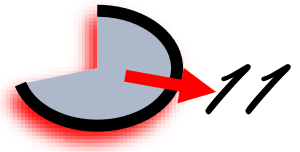




- Move-based implementation

```
template<class T>
void swap(T& a, T& b) // "perfect swap"
{
     T temp = std::move(a);
     a = std::move(b);
     b = std::move(temp);
}
```

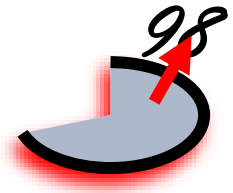


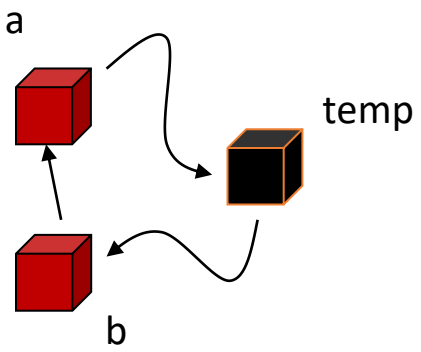
# another example: Swap




- Copy-based implementation

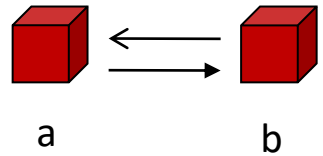
```
template<class T>
void swap(T& a, T& b)
{
  const T temp = a;
  a = b;
  b = temp;
}
```



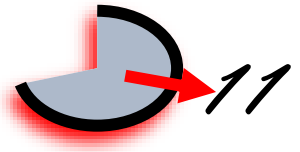

- Move-based implementation

```
template<class T>
void swap(T& a, T& b) // "perfect swap"
{
    T temp = std::move(a);
    a = std::move(b);
    b = std::move(temp);
}
```



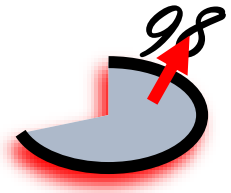

```
void f(string& s1, string& s2,
       Vector<string>& v1, Vector<string>& v2,
       Matrix& m1, Matrix& m2)
{
    swap(s1, s2);
    swap(v1, v2);
    swap(m1, m2);
}
```

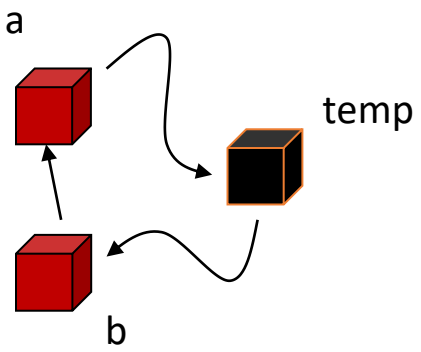
# another example: Swap




- Copy-based implementation

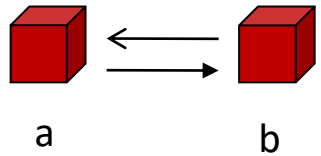
```
template<class T>
void swap(T& a, T& b)
{
  const T temp = a;
  a = b;
  b = temp;
}
```




- Move-based implementation

```
template<class T>
void swap(T& a, T& b) // "perfect swap"
{
     T temp = std::move(a);
     a = std::move(b);
     b = std::move(temp);
}
```



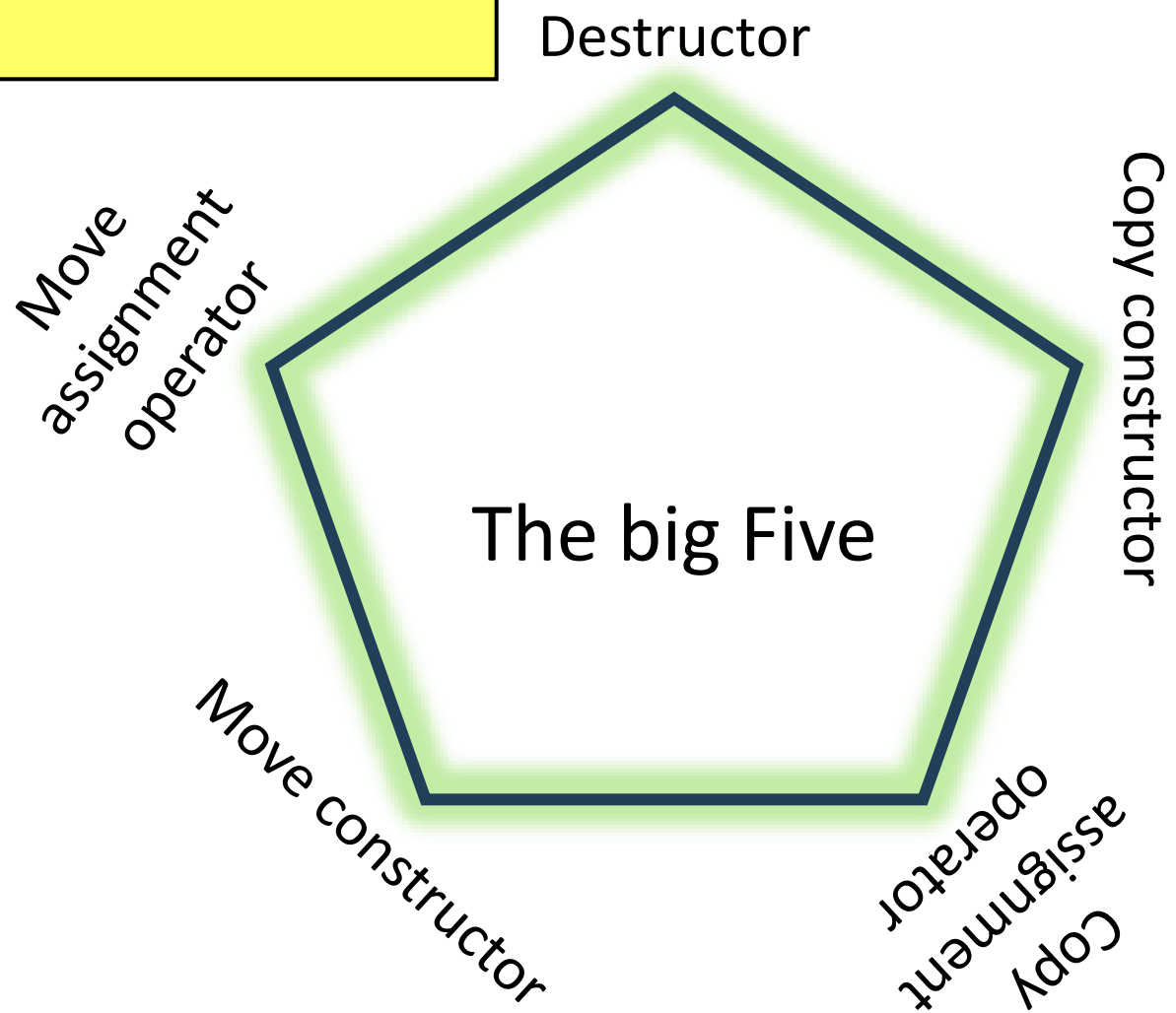

```
void f(string& s1, string& s2,
       Vector<string>& v1, Vector<string>& v2,
       Matrix& m1, Matrix& m2)
{
     swap(s1, s2);
     swap(v1, v2);
     swap(m1, m2);
}
```

- The move() is a standard-library function returning an r-value reference to its argument. move(x) means "give me an r-value reference to x". That is, move(x) does not move anything, instead it allows a user to move a.

# law of Big five

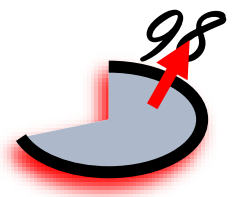
If a class needs move operations, has to declare all five special member functions.

- Examples: the classes Vector or String




- Unlike law of big three, failing to provide move constructor and move assignment is usually not an error, but a missed optimization opportunity.

# R-value references



- L-value Reference
  - An (L-value) reference is an *alternative* name for an object.
  - The notation T& means *reference to T.*

Type Specifier& Reference = Referent;

```
int i = 0;
int& ir = i; // ir is another name for i
ir++; // i = 1
long long& ll_ref; // error: reference should be initialized
long long& ll_ref = 1LL; // error: reference should be initialized with l-value
long long& ll_ref = long long(100); // error: bind to temporary object
long long& = i; // error: different types
long long LL = 1000;
long long& ll_ref = LL; // done
```

- L-values vs. R-values

Left Value
or
Location Value

a = a + 1

Assignment operator

Right Value
or
Read Value

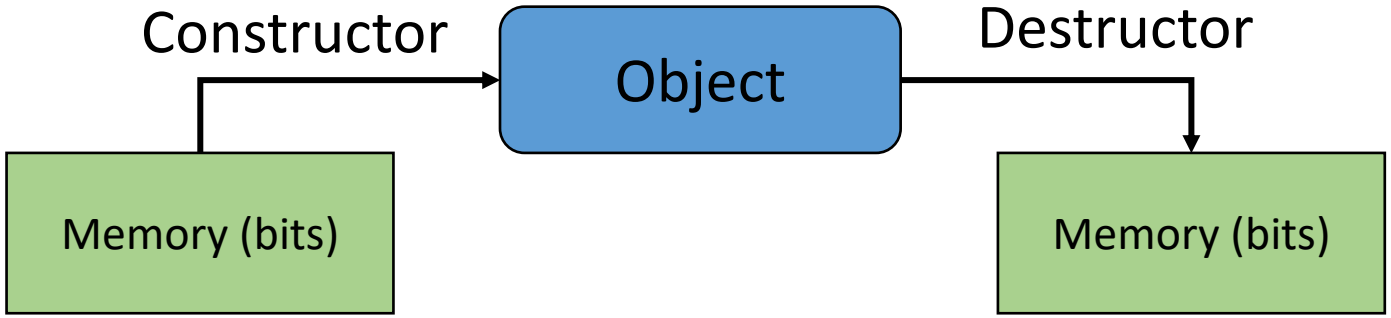
# R-value references cont.

- R-value reference 

```
class X {};
long l = 2L;
long&& lr = long(); // OK
int f() { return 1; }
int&& irr = f(); // OK
X&& xrr = X() // OK
```

- The && means r-value reference. An r-value reference can bind to an r-value.
- An r-value reference is a reference to something that nobody else can assign to. The && means r-value reference.
- Copy constructor and assignment take l-value references, whereas move constructor and move assignment take r-value references. For a return value, the move constructor is chosen.

# Constructors & destructors




Marshal P. Cline:

- Constructors build objects from dust.
- Constructors turn a pile of arbitrary bits into a living object.
- A destructor gives an object its last rites. Destructors are a "prepare to die" member function.

# RAII- Resource Acquisition is Initialization

# RAII- Resource Acquisition is Initialization

• One of the key tasks of any non-trivial program is to manage resources. For a long-running program, failing to release a resource in a timely manner can cause serious performance degradation and possibly crash.

• Resource: any entity that a program acquires and releases.

Examples: memory, file handles, thread handles, locks, sockets, timer, transactions, network connection, data base connection, …

# RAII- Resource Acquisition is Initialization

• One of the key tasks of any non-trivial program is to manage resources. For a long-running program, failing to release a resource in a timely manner can cause serious performance degradation and possibly crash.

• Resource: any entity that a program acquires and releases.

Examples: memory, file handles, thread handles, locks, sockets, timer, transactions, network connection, data base connection, …

• RAII is about Resource Management. RAII is about *Resource-safety*.

• A basic technique for *resource management* based on scopes.

- Bjarne Stroustrup *programming Principles and Practice Using C++, page 1175.*

# RAII- Resource Acquisition is Initialization

• One of the key tasks of any non-trivial program is to manage resources. For a long-running program, failing to release a resource in a timely manner can cause serious performance degradation and possibly crash.

• Resource: any entity that a program acquires and releases.

Examples: memory, file handles, thread handles, locks, sockets, timer, transactions, network connection, data base connection, …

• RAII is about Resource Management. RAII is about *Resource-safety*.

• A basic technique for *resource management* based on scopes.

- Bjarne Stroustrup *programming Principles and Practice Using C++, page 1175.*

• A simple technique for *handling resources* in programs using exceptions.

One of the keys to *exception safety*.

- Bjarne Stroustrup Glossary page http://www.stroustrup.com/glossary.html

# RAII- Resource Acquisition is Initialization

• One of the key tasks of any non-trivial program is to manage resources. For a long-running program, failing to release a resource in a timely manner can cause serious performance degradation and possibly crash.

• Resource: any entity that a program acquires and releases.

Examples: memory, file handles, thread handles, locks, sockets, timer, transactions, network connection, data base connection, …

• RAII is about Resource Management. RAII is about *Resource-safety*.

• A basic technique for *resource management* based on scopes.

- Bjarne Stroustrup *programming Principles and Practice Using C++, page 1175.*

• A simple technique for *handling resources* in programs using exceptions.

One of the keys to *exception safety*.

- Bjarne Stroustrup Glossary page http://www.stroustrup.com/glossary.html

• RAII is a programming *idiom* or *technique* used for exception-safe resource management.

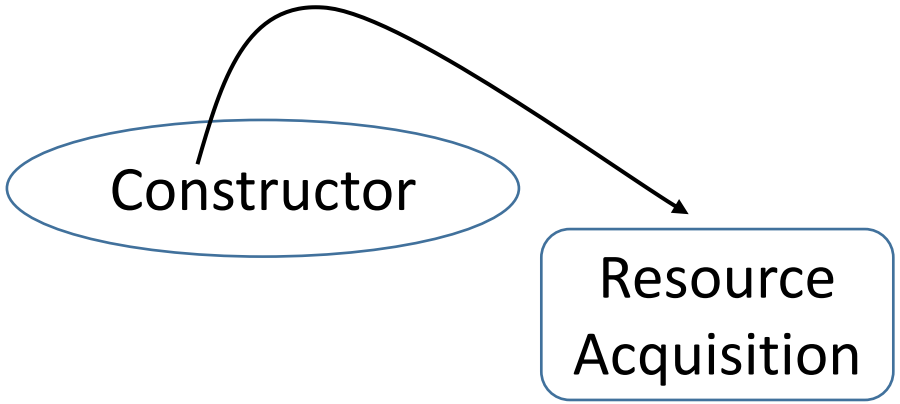
- Wikipedia *http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization*

# Constructors, destructors and resources

- How to implement RAII?

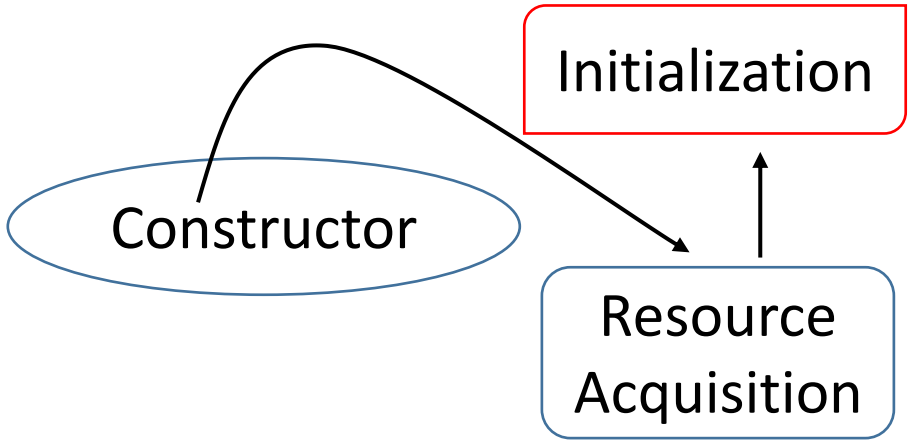
# Constructors, destructors and resources

- How to implement RAII?



# Constructors, destructors and resources

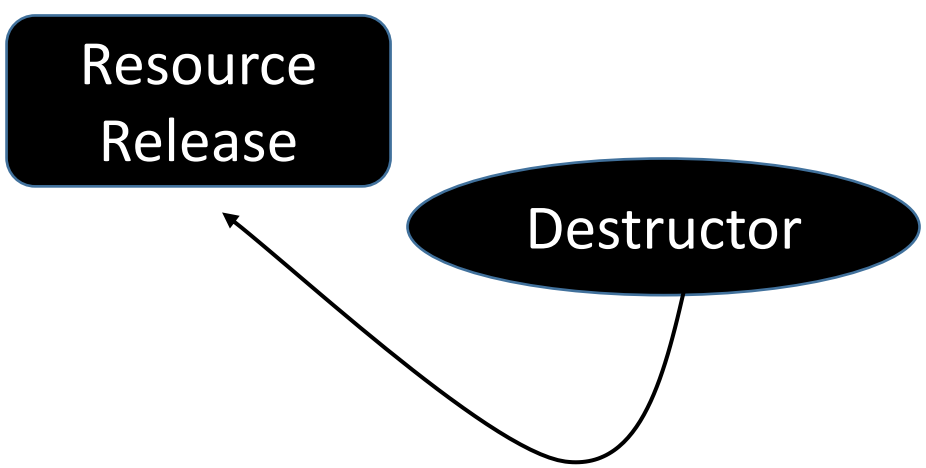
- How to implement RAII?

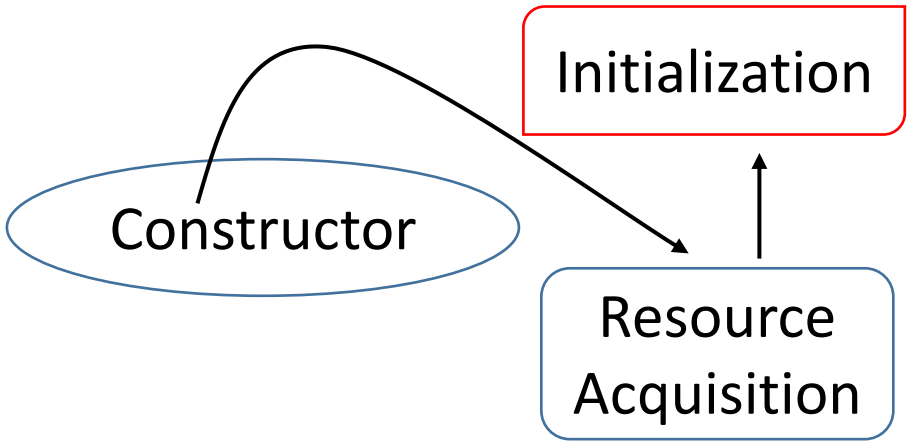



• A constructor is used to initialize objects of its class type.

# Constructors, destructors and resources

- How to implement RAII?

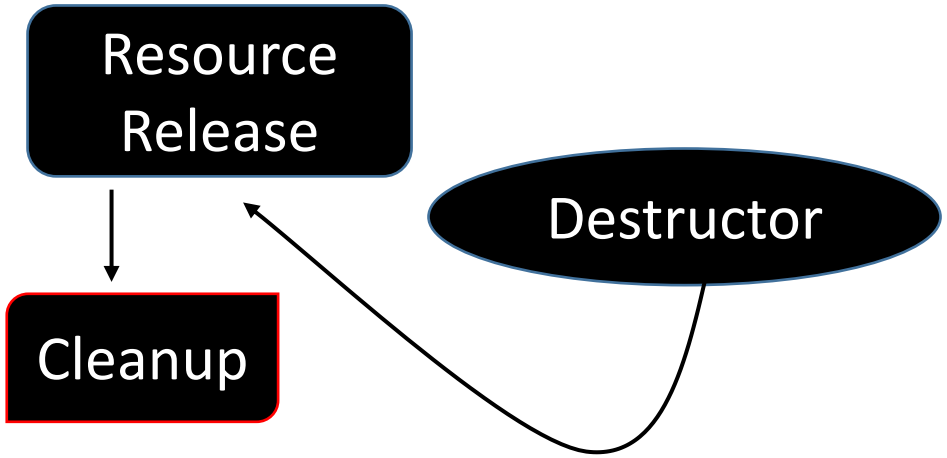


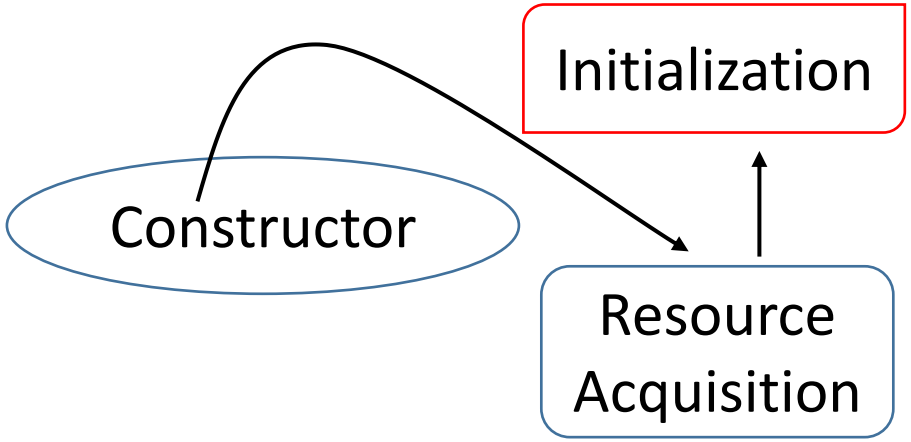




• A constructor is used to initialize objects of its class type.

# Constructors, destructors and resources

- How to implement RAII?



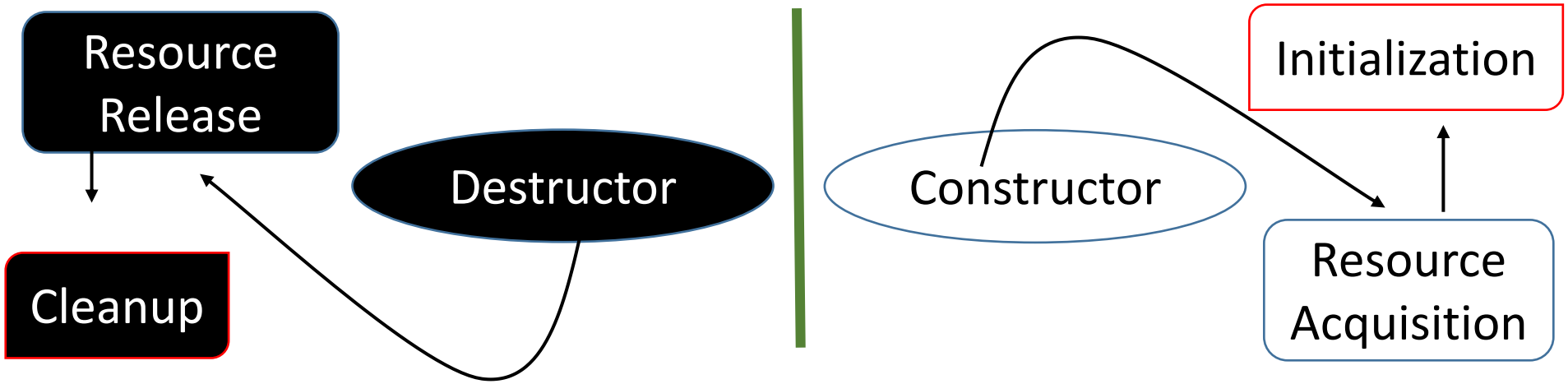

• A destructor is used to destroy objects of its class type.

• Destroy = cleanup

• A constructor is used to initialize objects of its class type.

# Constructors, destructors and resources

- How to implement RAII?

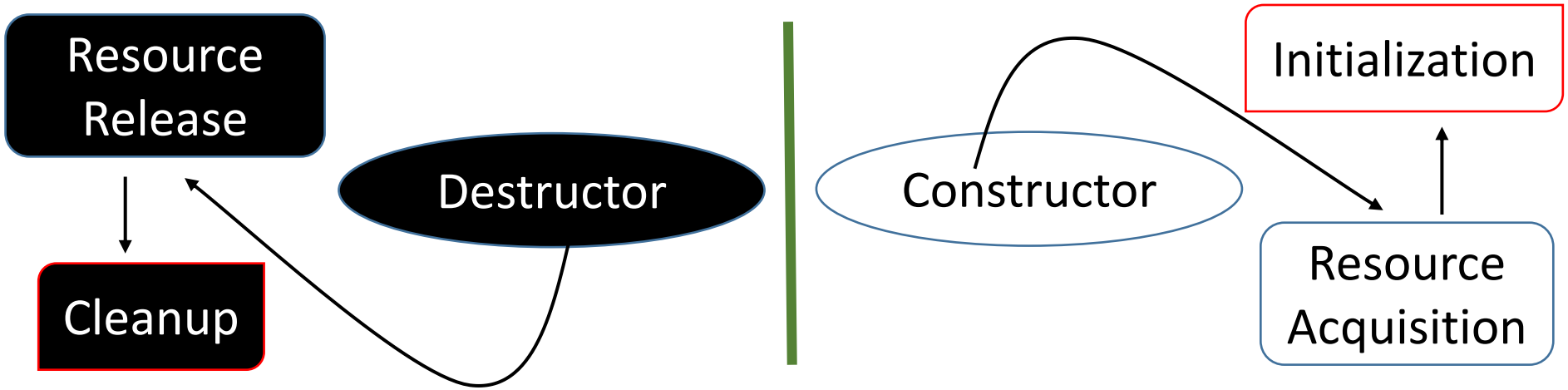



- A destructor is used to destroy objects of its class type.
- Destroy = cleanup

- A constructor is used to initialize objects of its class type.

# Constructors, destructors and resources

- How to implement RAII?




• A destructor is used to destroy objects of its class type.

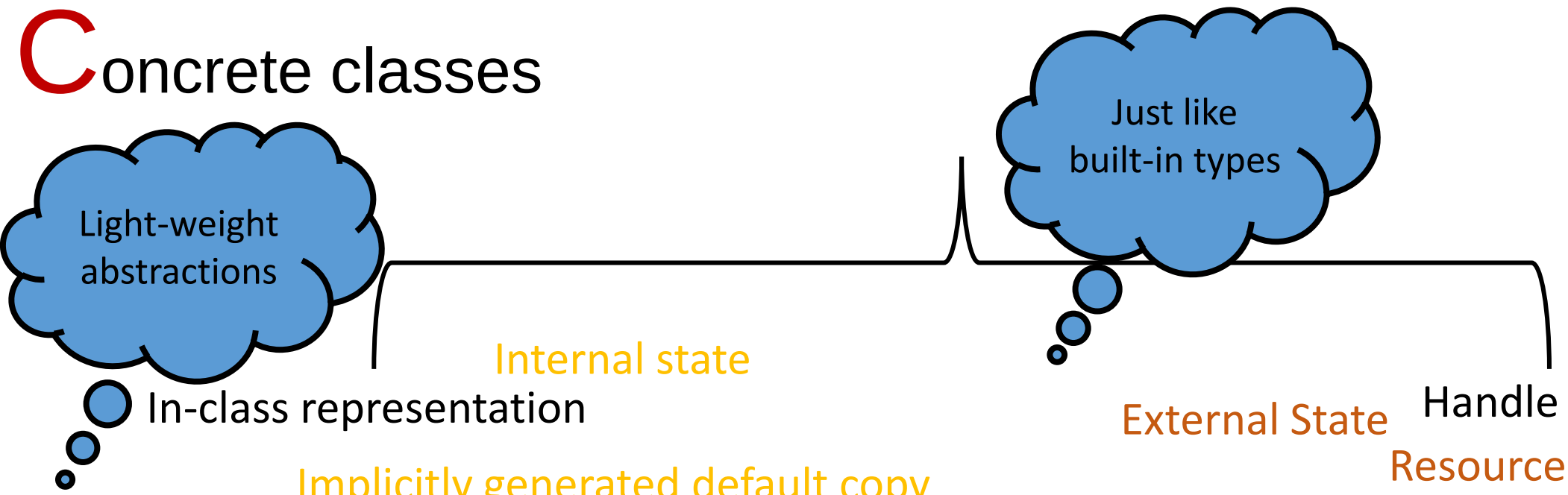
• A constructor is used to initialize objects of its class type.

• Destroy = cleanup

• An object is not considered constructed until its constructor completes.

• Variables with automatic storage duration declared in the block are destroyed on exit from the block. from *Committee Draft*

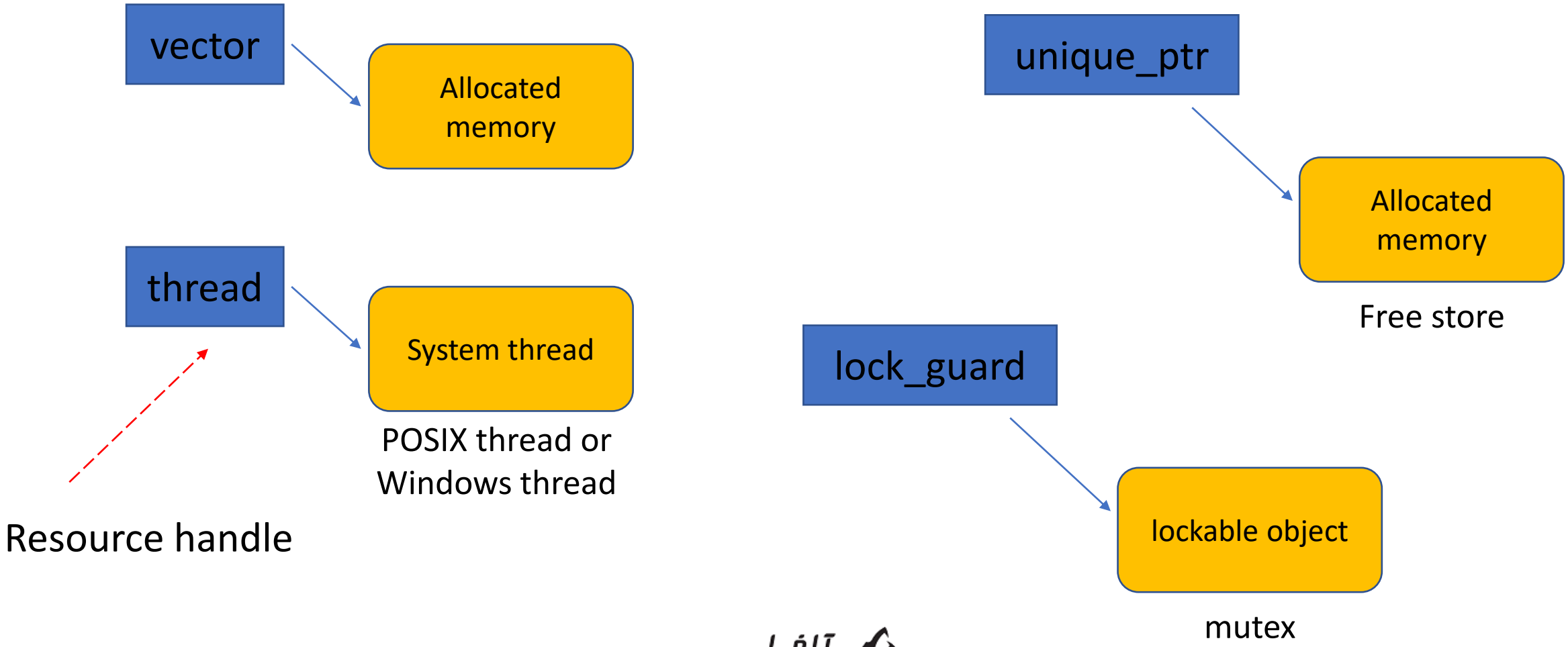
# Concrete classes




- Concrete classes = Value-type classes
- Value-oriented programming

# Handle

- Resource handle
- Handle-to-data model
- Resource: memory, file, thread, …



# *Thanks for your patience …*

A man who asks a question is a fool for minute,
The man who does not ask, is a fool for a life.
- Confucius

Learning to ask the right (often hard) questions is an essential part of learning to think as a programmer.
- Bjarne Stroustrup *programming Principles and Practice Using C++, page 4.*

There is no stupid question, but there is stupid answer.
- Howard Hinnant