# Improving Array Implementation

**Giovanni Dicanio**
AUTHOR, SOFTWARE ENGINEER

https://blogs.msmvps.com/gdicanio

# Overview

**cout << myArray**

**Copying arrays**

- Shallow vs. deep copy
- Copy-and-swap idiom

**Move semantics**

**Generic Array<T> using templates**

```
int n{64};
cout << n;
```

```
cout << myArray;
```

# Printing to cout

```
int n{64};
cout << n;
```

*Insertion* operator


# Printing to cout

```
int n{64};
cout << n;
```

*Output stream*

std::ostream

# Printing to cout

Function with a «special» name

RETURN

PARAMETERS

```
ostream& operator<<(ostream& os, const IntArray& a) {
    …
}
```

# Printing to cout

**Overloading the insertion operator (<<)**

Output stream
(target)

```
ostream& operator<<(ostream& os, const IntArray& a) {
    …
}
```

# Printing to cout

**Overloading the insertion operator (<<)**

```
ostream& operator<<(ostream& os, const IntArray& a) {
    …
}
```

Object to print (source)

const X&

# Printing to cout

## Overloading the insertion operator (<<)

*Observing Parameters with const References*

«C++11 from Scratch»

Output stream

```
ostream& operator<<(ostream& os, const IntArray& a) {
    …
}
```

# Printing to cout

**Overloading the insertion operator (<<)**

```
ostream& operator<<(ostream& os, const IntArray& a) {
    …
}
```

Implementation

# Printing to cout

**Overloading the insertion operator (<<)**

```
ostream& operator<<(ostream& os, const IntArray& a) {
  // Code to print IntArray objects to 'os'
  // …
  return os;
}
```

# Printing to cout

**Overloading the insertion operator (<<)**

```
// e.g. «C64 is 36 years old»
cout << name << " is " << age << " years old \n";
```

string

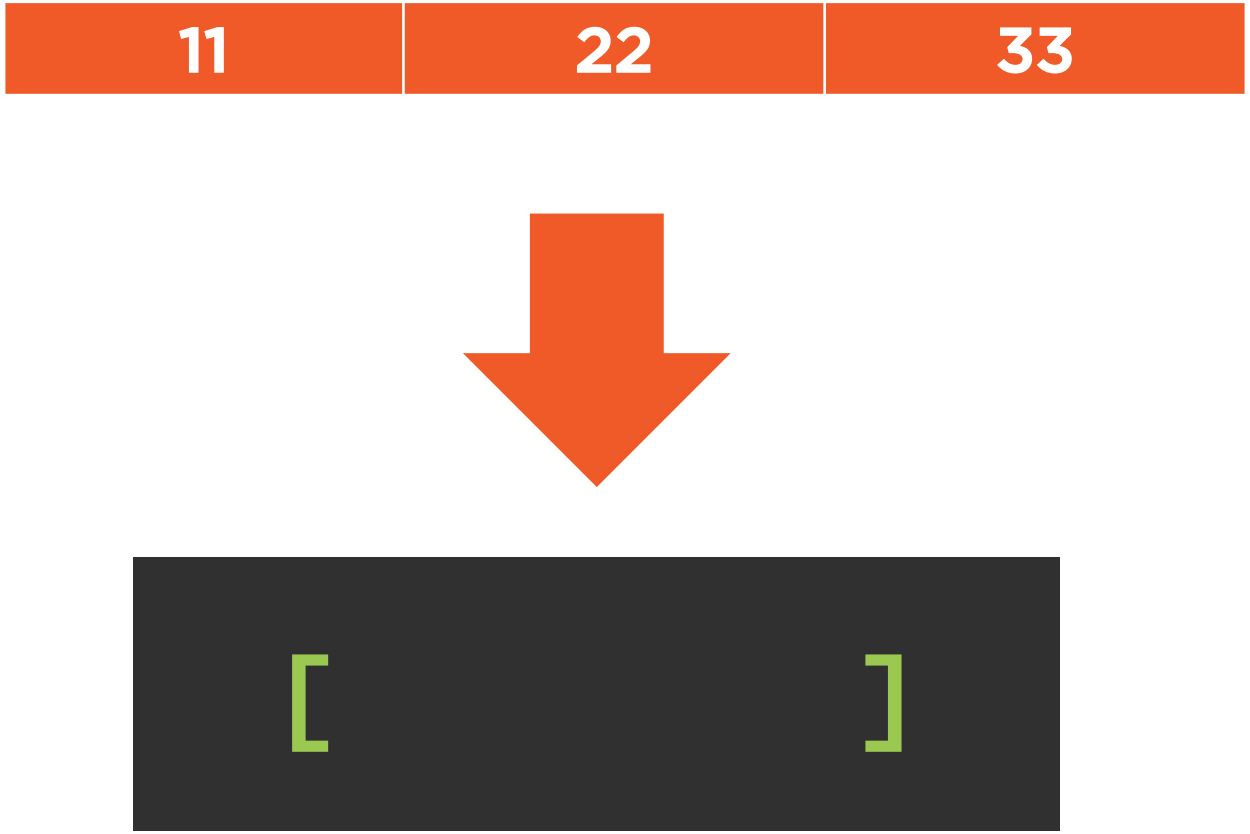
int

Extensible to your own classes

# Chained Calls to Insertion Operator Overloads

```
ostream& operator<<(ostream& os  const X& a) {
  // Code to print X object to 'os'
  // ...
  return os;
}
```

# How to Format Arrays?



# How to Format Arrays?


[ 11 22 33 ]

Print code in
operator<< overload

# Overloading operator<< for Arrays

```
ostream& operator<<(ostream& os, const IntArray& a) {


  //
  // Print array elements using cout
  //




}
```

# Overloading operator<< for Arrays

```
ostream& operator<<(ostream& os, const IntArray& a) {

  //
  // Print array elements using
  //

}
```

Generic output stream

# Overloading operator<< for Arrays

```
ostream& operator<<(ostream& os, const IntArray& a) {
  os << "[ ";

  os << "]";

}
```

Brackets around elements

# Overloading operator<< for Arrays

```
ostream& operator<<(ostream& os, const IntArray& a) {
  os << "[ ";

    os << a[i]

  os << "]";



}
```

Print *i-th* element

# Overloading operator<< for Arrays

```
ostream& operator<<(ostream& os, const IntArray& a) {
  os << "[ ";

    os << a[i] << ' ';

  os << "]";



}
```

Separate elements

# Overloading operator<< for Arrays

```
ostream& operator<<(ostream& os, const IntArray& a) {
  os << "[ ";
  for (int i = 0; i < a.Size(); i++) {
    os << a[i] << ' ';
  }
  os << "]";



}
```

Repeat *for each* element

# Overloading operator<< for Arrays

```
ostream& operator<<(ostream& os, const IntArray& a) {
  os << "[ ";
  for (int i = 0; i < a.Size(); i++) {
    os << a[i] << ' ';
  }
  os << "]";

  return os;
}
```

Allow *chained* calls

cout << x << y << ...

# Overloading operator<< for Arrays

```
ostream& operator<<(ostream& os, const IntArray& a) {
  os << "[ ";
  for (int i = 0; i < a.Size(); i++) {
    os << a[i] << ' ';
  }
  os << "]";

  return os;
}
```

```
cout << myArray;
```

DESTINATION SOURCE

```
IntArray b = a;
```

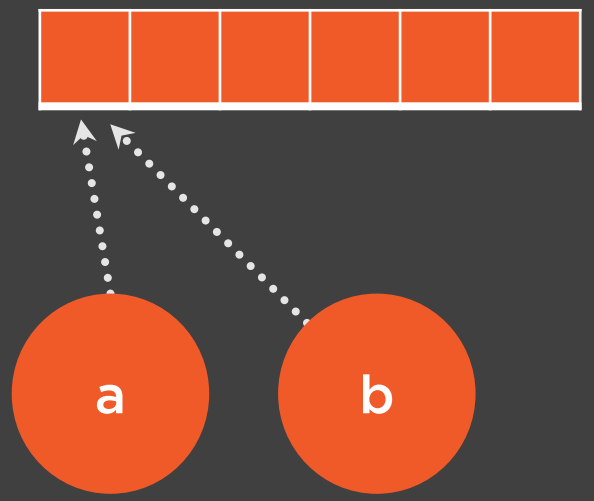
Copy Initialization

COPY CONSTRUCTOR

**Default:**
***member-wise* copy**

```
class IntArray {
 private:
  int* m_ptr;
  int m_size;
  …
};
```

```
IntArray b = a;

b.m_ptr  = a.m_ptr;
b.m_size = a.m_size;
```




*Shallow* copy

# Copy Initialization

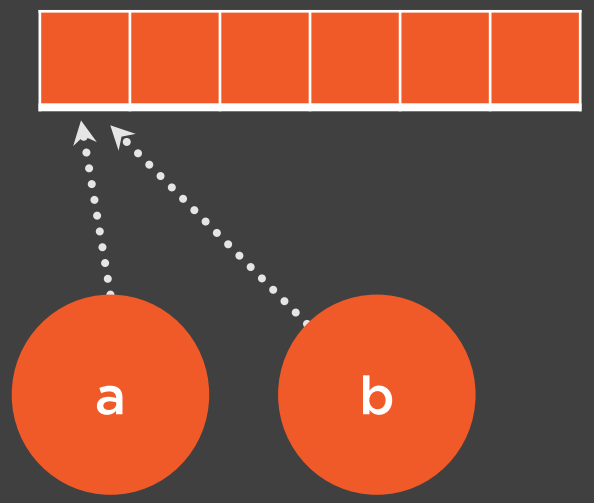
**Default member-wise copy**

b[1] = 100;

Modifications to b are *reflected* to a

```
{
  IntArray a{…};
  …
  IntArray b = a;
  …
}
```

IntArray's destructor invoked on *b*



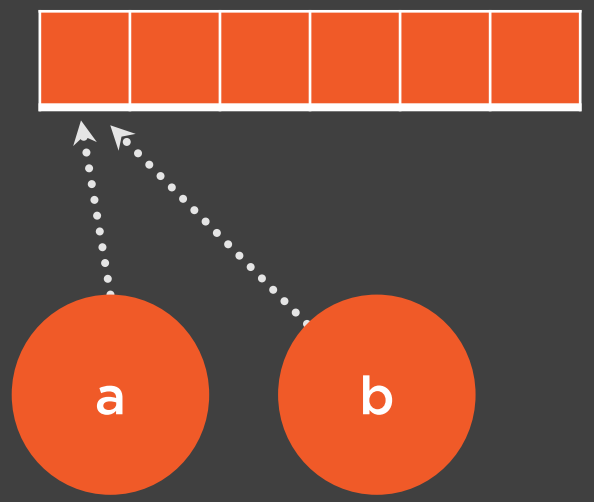

# Copy Initialization

**Subtle memory *bug* with *shallow* copies**

```
{
  IntArray a{…};
  …
  IntArray b = a;
  …
}
```

IntArray's destructor invoked on *b*

*delete*[] b.m_ptr;




# Copy Initialization

**Subtle memory *bug* with *shallow* copies**

```
{
  IntArray a{…};
  …
  IntArray b = a;
  …
}
```

IntArray's destructor invoked on *b*

*delete*[] b.m_ptr;

a

# Copy Initialization

**Subtle memory *bug* with *shallow* copies**

```
{
  IntArray a{…};
  …
  IntArray b = a;
  …
}
```

IntArray's destructor invoked on *a*

*delete*[] a.m_ptr;

Memory was *already freed*

a

# Copy Initialization

**Subtle memory *bug* with *shallow* copies**

**Double-delete Bug**

```
class IntArray {
  // Copy constructor
  IntArray(const IntArray&)
  …
};
```

# Disable Compiler-generated Copy Constructor

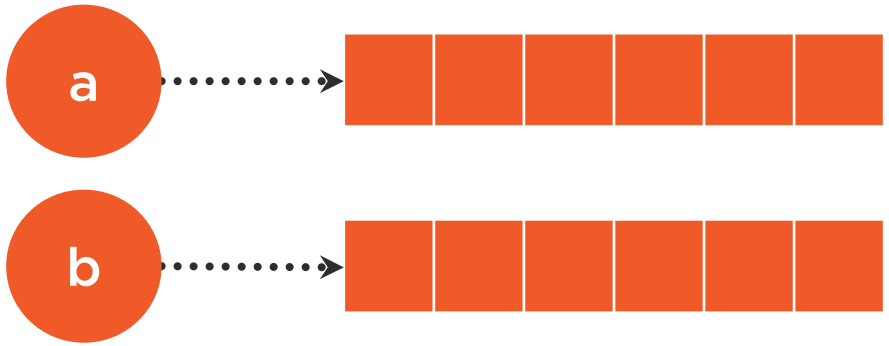
```
class IntArray {
  // Copy constructor
  IntArray(const IntArray&) = delete;
  …
};
```

Disable Compiler-generated Copy Constructor

```
IntArray(const IntArray& source) {
  // Allocate memory to store array elements
  // Copy elements from source to this
}
```

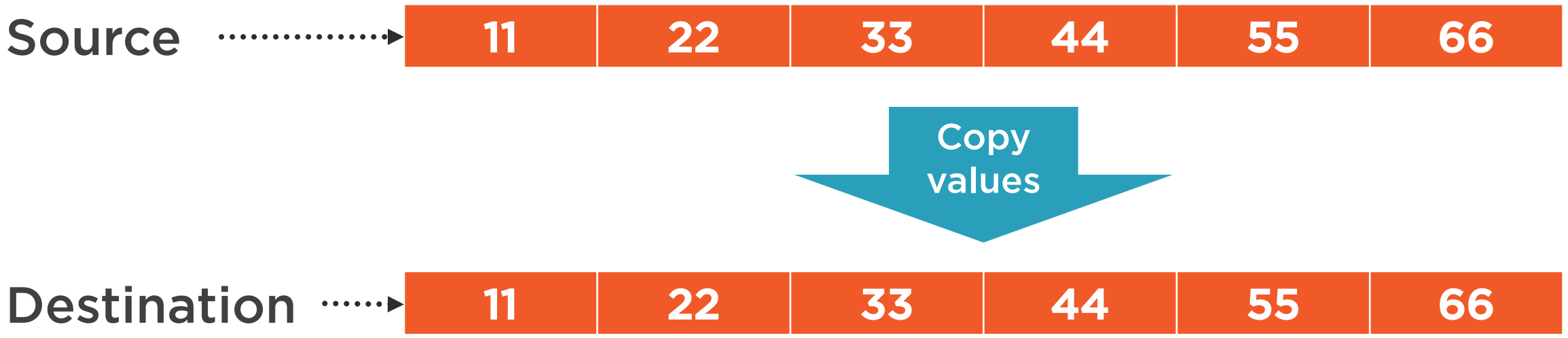
# Implement a Custom Copy Constructor

**Make *deep* copies**

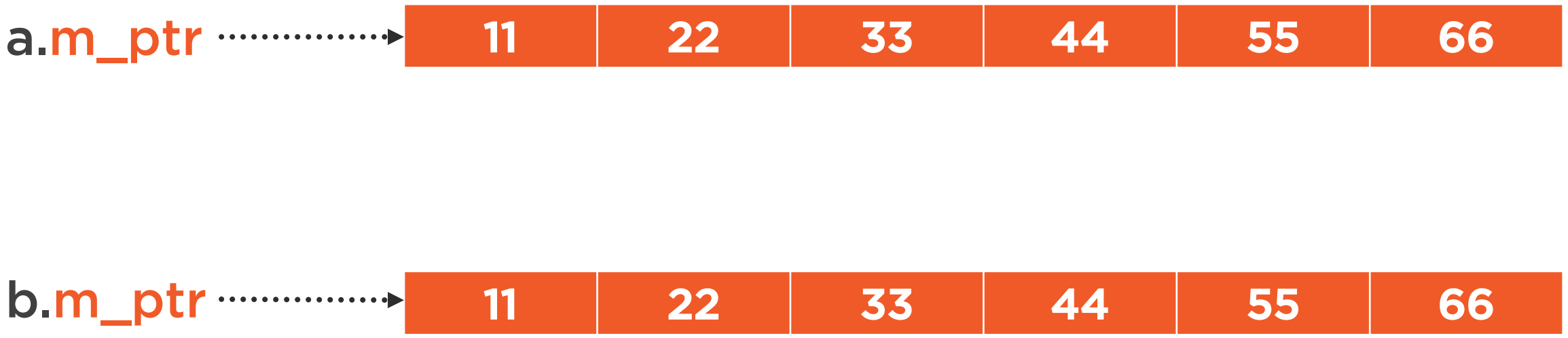


# Deep Copy → Distinct Arrays

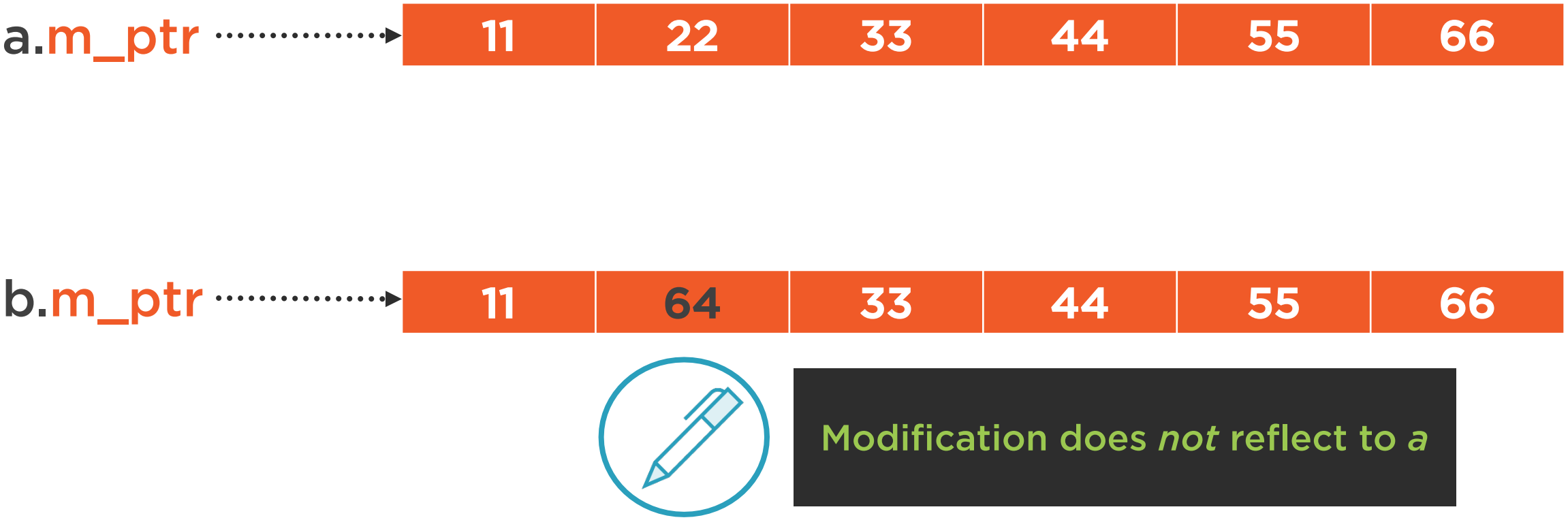
# Deep Copy → Distinct Arrays



# Deep Copy → Distinct Arrays



# Deep Copy → Distinct Arrays




Modification does *not* reflect to *a*

# Deep Copy → Distinct Arrays

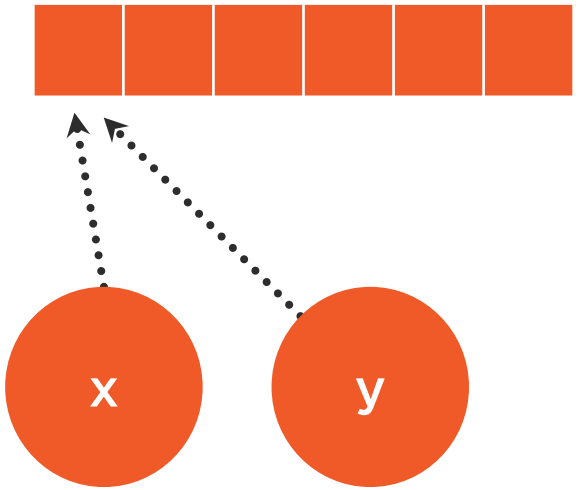

No more
*double-delete* bugs

# Copying with the Assignment Operator

y = x;

**Default:**
**Member-wise copy**

*Shallow* copy



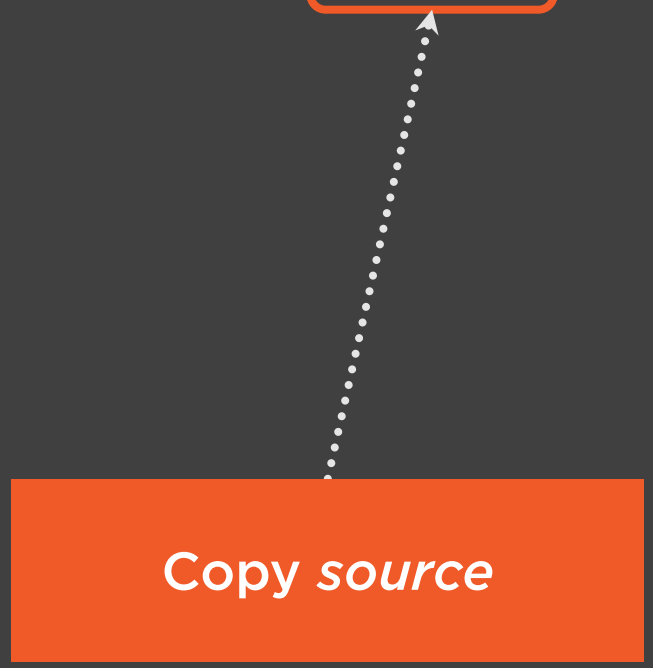

*Deep* copy
requires
custom code

# Implementing Copy Assignment

```
IntArray& operator=(const IntArray& source)
```

# Implementing Copy Assignment

```
IntArray& operator=(const IntArray& source)
```



# Implementing Copy Assignment

```
IntArray& operator=(const IntArray& source)
```

Reference to destination (*this*)

*this* : predefined keyword

# Implementing Copy Assignment

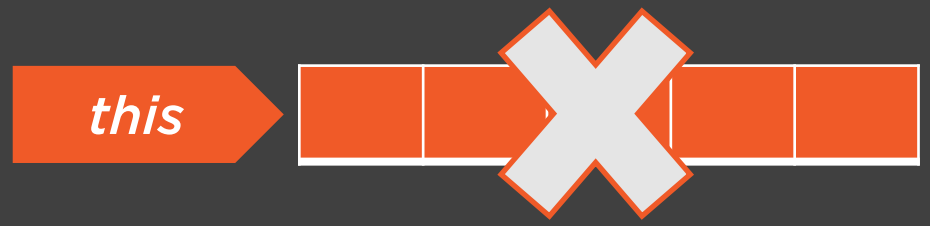
```
IntArray& operator=(const IntArray& source) {

  return *this;
}
```

# Implementing Copy Assignment

```
IntArray& operator=(const IntArray& source) {
  // Prevent self-assignment (x = x)
  if (&source != this) {
    // Do the copy …




  }
  return *this;
}
```

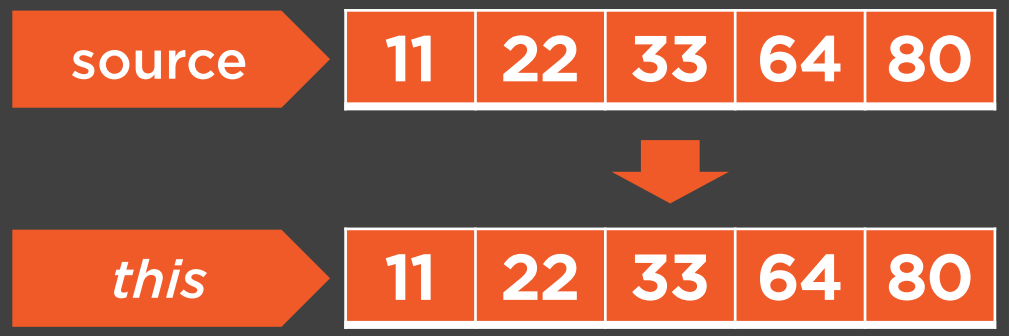
# Implementing Copy Assignment

```
IntArray& operator=(const IntArray& source) {
  // Prevent self-assignment (x = x)
  if (&source != this) {
    // Do the copy …
    // Don't forget to release the previous array block

  }
  return *this;
}
```



# Implementing Copy Assignment

```
IntArray& operator=(const IntArray& source) {
  // Prevent self-assignment (x = x)
  if (&source != this) {
    // Do the copy …
    // Don't forget to release the previous array block
    // Implement proper deep-copy from 'source' to this
  }
  return *this;
}
```

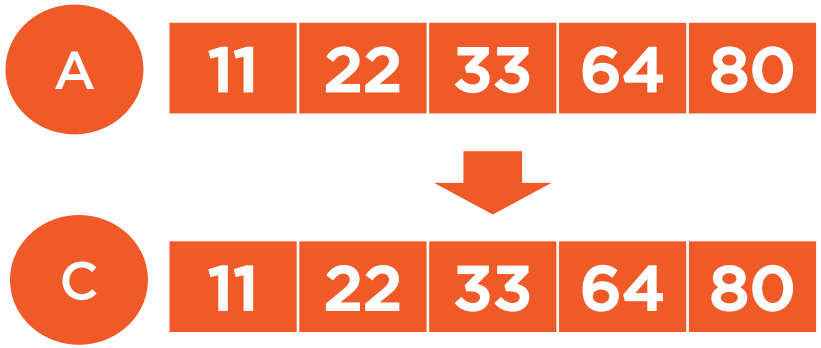


# Copy-and-swap Idiom

**Deep Copy**

Reuse *copy constructor*

# Copy-and-swap Idiom

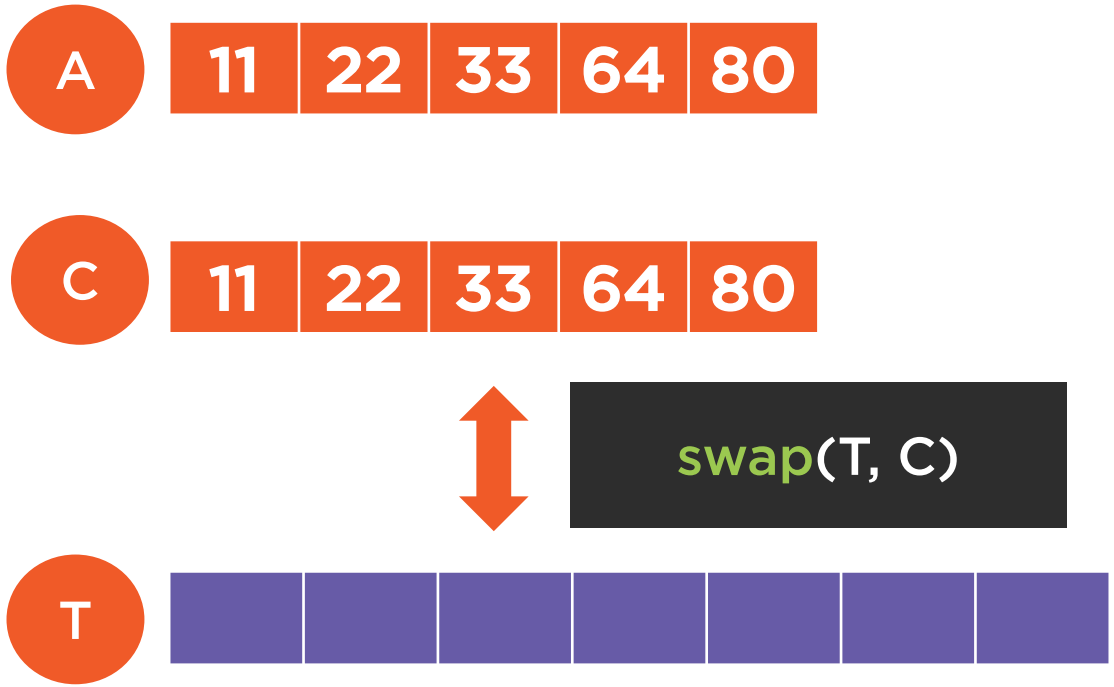



Reuse *copy constructor*

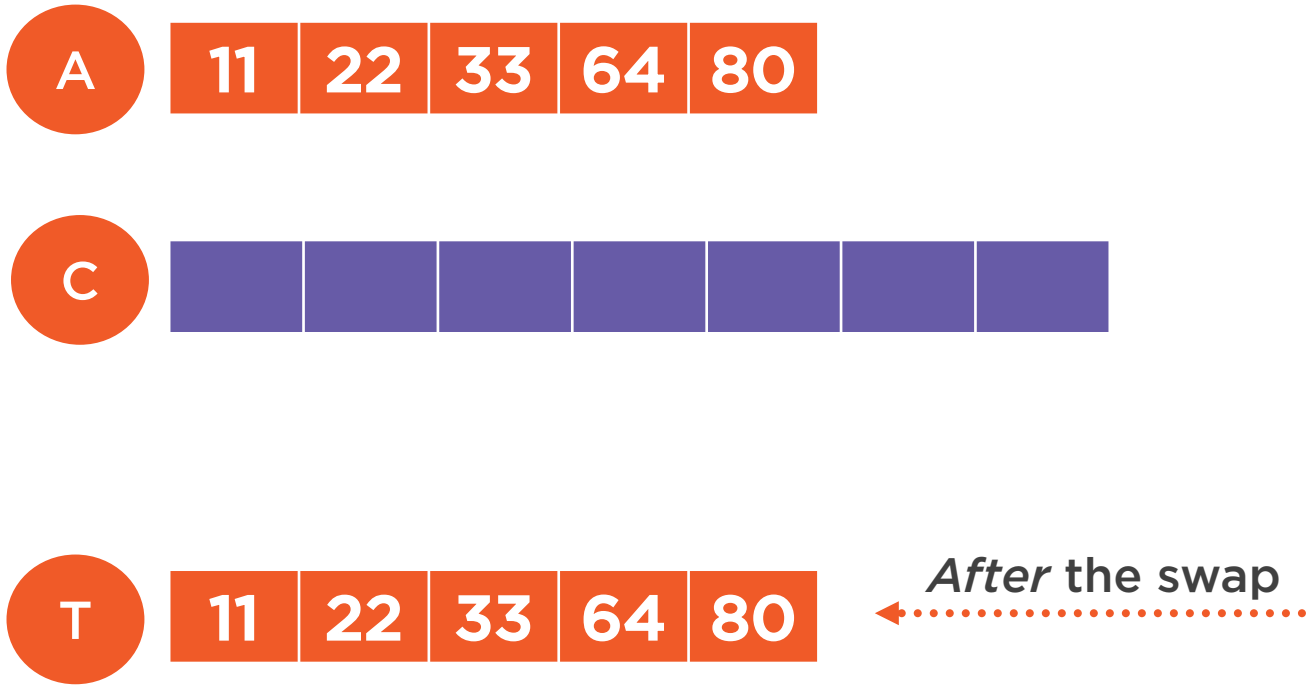
# Copy-and-swap Idiom

"C++11 from Scratch"

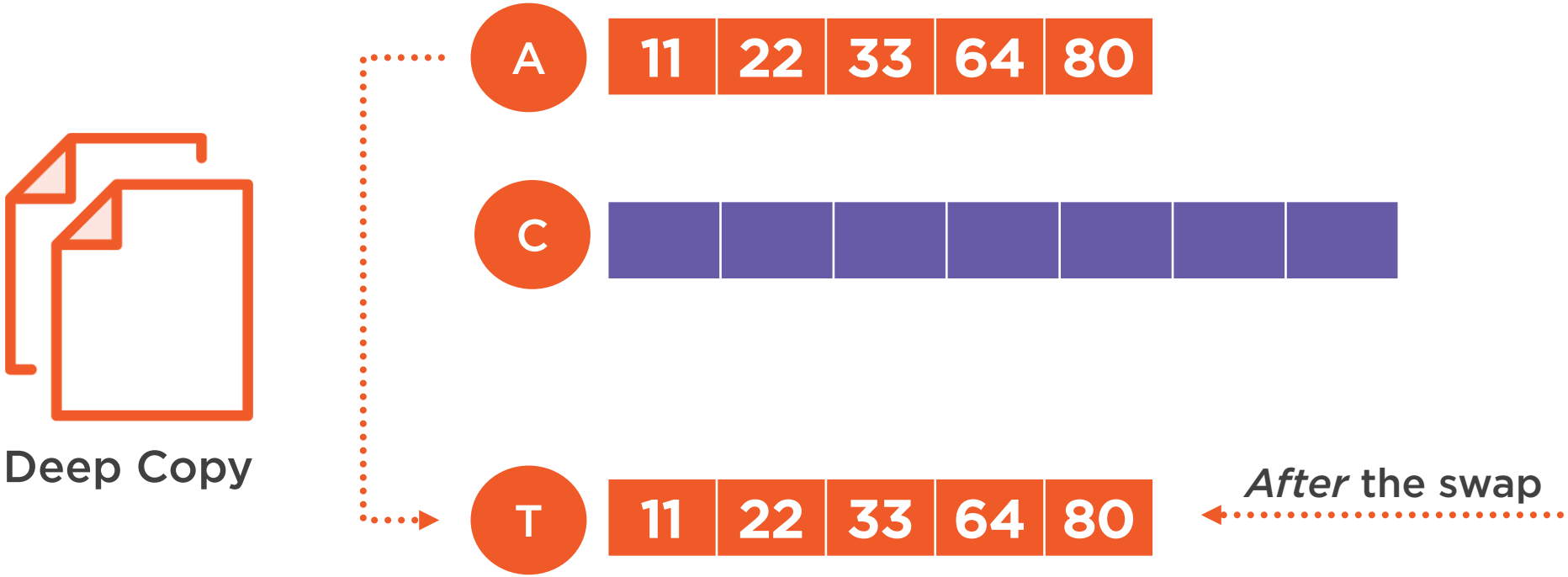
**The Swap Algorithm**



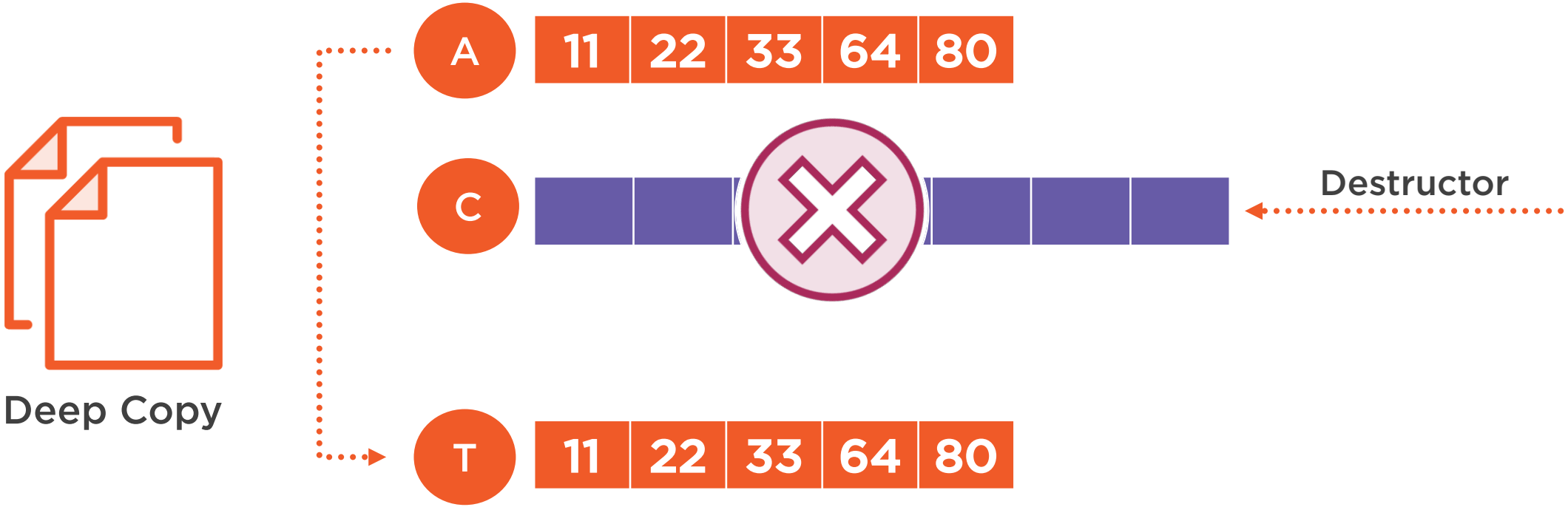
# Copy-and-swap Idiom



# Copy-and-swap Idiom

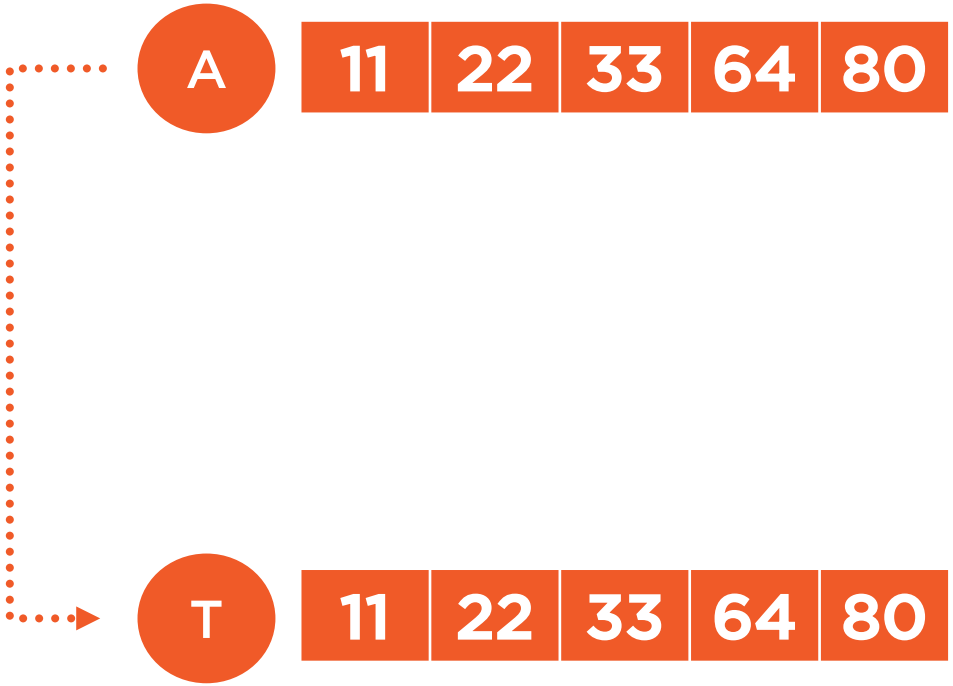


# Copy-and-swap Idiom



# Copy-and-swap Idiom

Deep Copy



# Copy-and-swap Idiom

Copy constructor

Swap function

Destructor

*Memberwise*
swap

```
IntArray& operator=(IntArray source) {
  swap(*this, source);
  return *this;
}
```

# Assignment Operator Using Copy-and-swap

COPY
Pass by value

```
IntArray& operator=(IntArray source) {
  swap(*this, source);
  return *this;
}
```

# Assignment Operator Using Copy-and-swap

```
IntArray& operator=(IntArray source) {
  swap(*this, source);
  return *this;
}
```

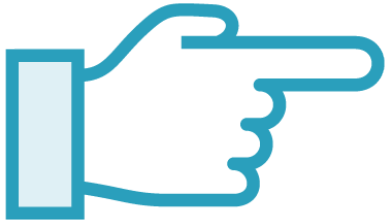
SWAP

# Assignment Operator Using Copy-and-swap

```
IntArray& operator=(IntArray source) {
  swap(*this, source);
  return *this;
} ←..... DESTRUCTOR
```

# Assignment Operator Using Copy-and-swap

# Refresher on Basic C++ Concepts



"C++11 from Scratch"

Local Variables and Scope

Passing Parameters by Value vs. by Reference

Basic Rules for Parameter Passing in C++

```
friend void swap(IntArray& a, IntArray& b) {
  using std::swap;
  // Memberwise swap
  swap(a.m_ptr,  b.m_ptr);
  swap(a.m_size, b.m_size);
}
```

swap(a, b)

# Implementing Swap for the Array Class

**Swap array objects by swapping their data members**

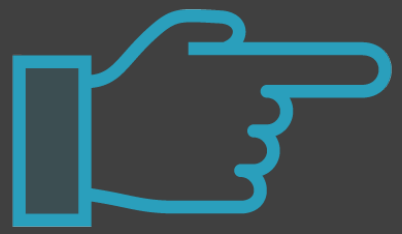
```
friend void swap(IntArray& a, IntArray& b) {
  using std::swap;
  // Memberwise swap
  swap(a.m_ptr,  b.m_ptr);
  swap(a.m_size, b.m_size);
}
```

ACCESS PRIVATE MEMBERS
IntArray's *m_ptr* and *m_size*

# Implementing Swap for the Array Class

**Swap array objects by swapping their data members**

```
friend void swap(IntArray& a, IntArray& b) {
  using std::swap;
  // Memberwise swap
  swap(a.m_ptr,  b.m_ptr);
  swap(a.m_size, b.m_size);
}
```



"C++11 from Scratch"

**The Swap Algorithm**

# Implementing Swap for the Array Class

**Swap array objects by swapping their data members**

```
friend void swap(IntArray& a, IntArray& b) {
  using std::swap;
  // Memberwise swap
  swap(a.m_ptr,  b.m_ptr);
  swap(a.m_size, b.m_size);
}
```

# Implementing Swap for the Array Class

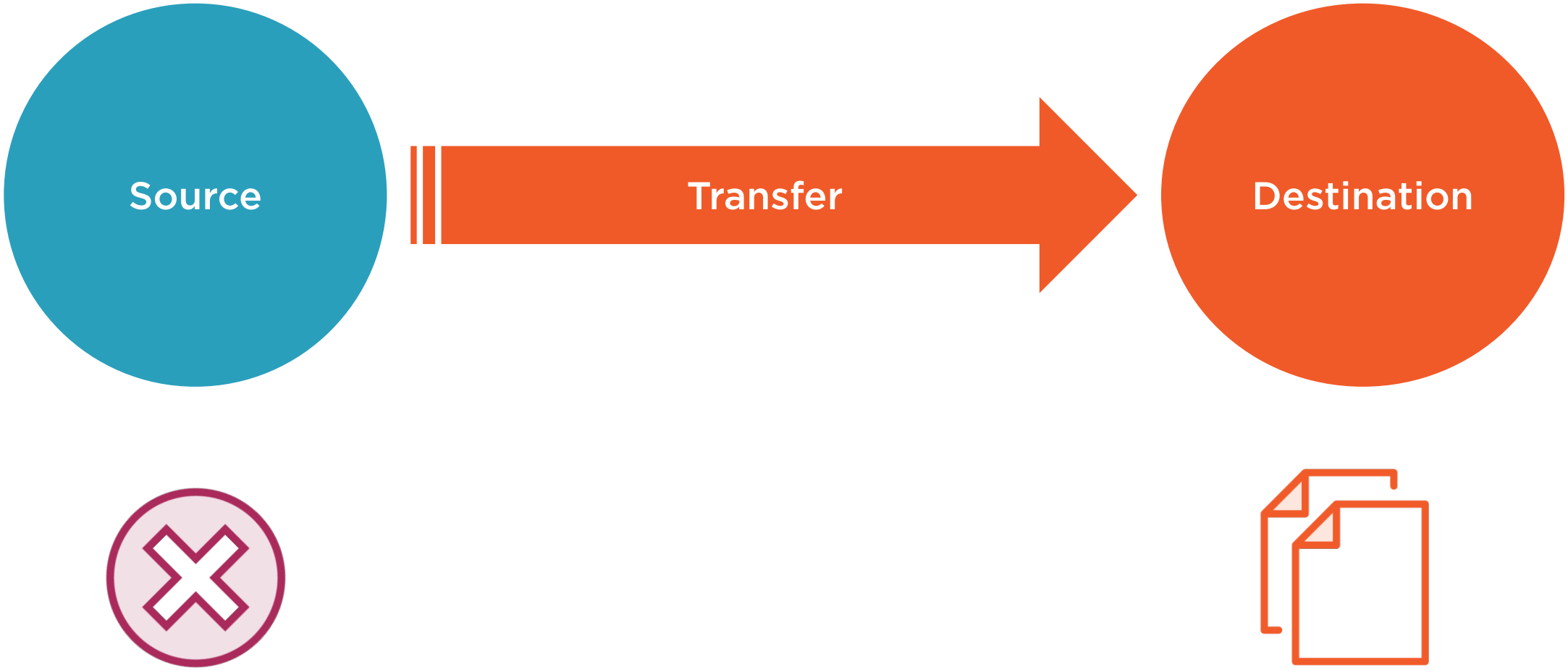
**Swap array objects by swapping their data members**

```
friend void swap(IntArray& a, IntArray& b) noexcept {
  using std::swap;
  // Memberwise swap
  swap(a.m_ptr,  b.m_ptr);
  swap(a.m_size, b.m_size);
}
```

# Implementing Swap for the Array Class

**Use *noexcept* for *non-throwing* swap**

# Transfer Objects



# Transfer Objects

# Returning Data

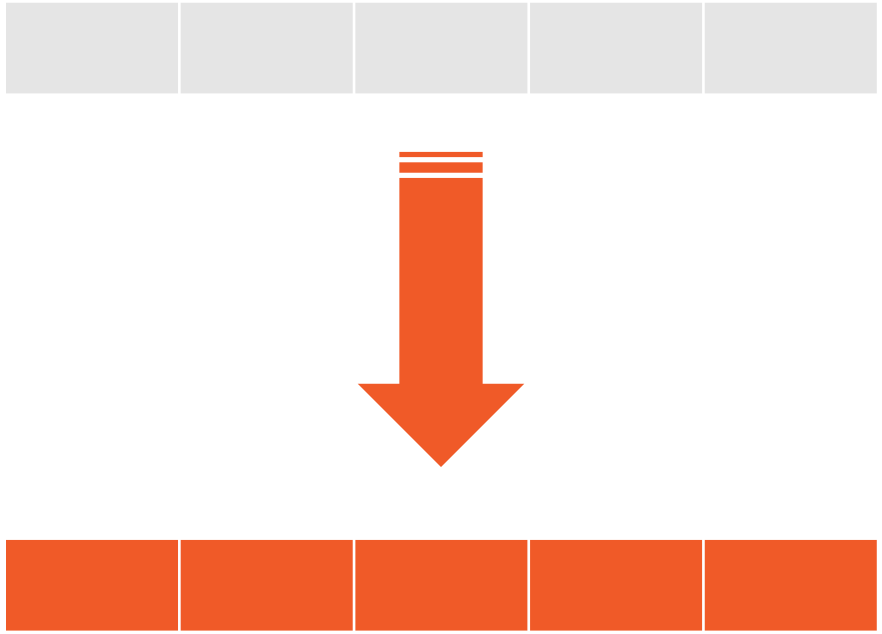

Cheap to copy

```
ReturnType ProcessSomething() {

   …

   return something;

}
```

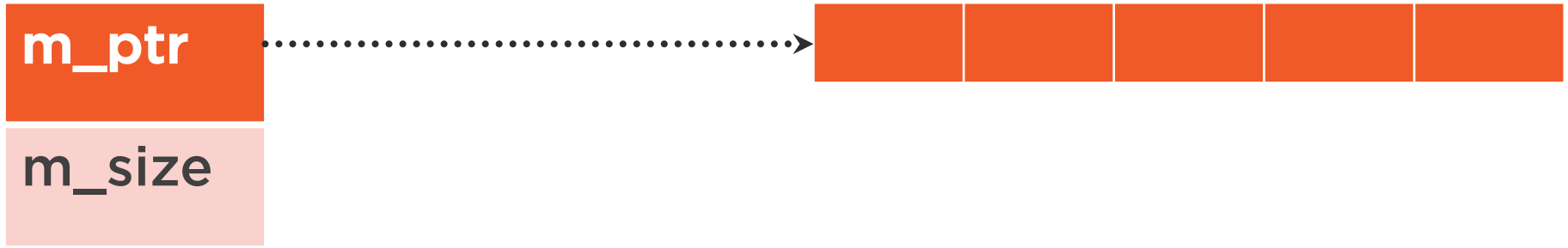

Expensive to copy

# Move Semantics

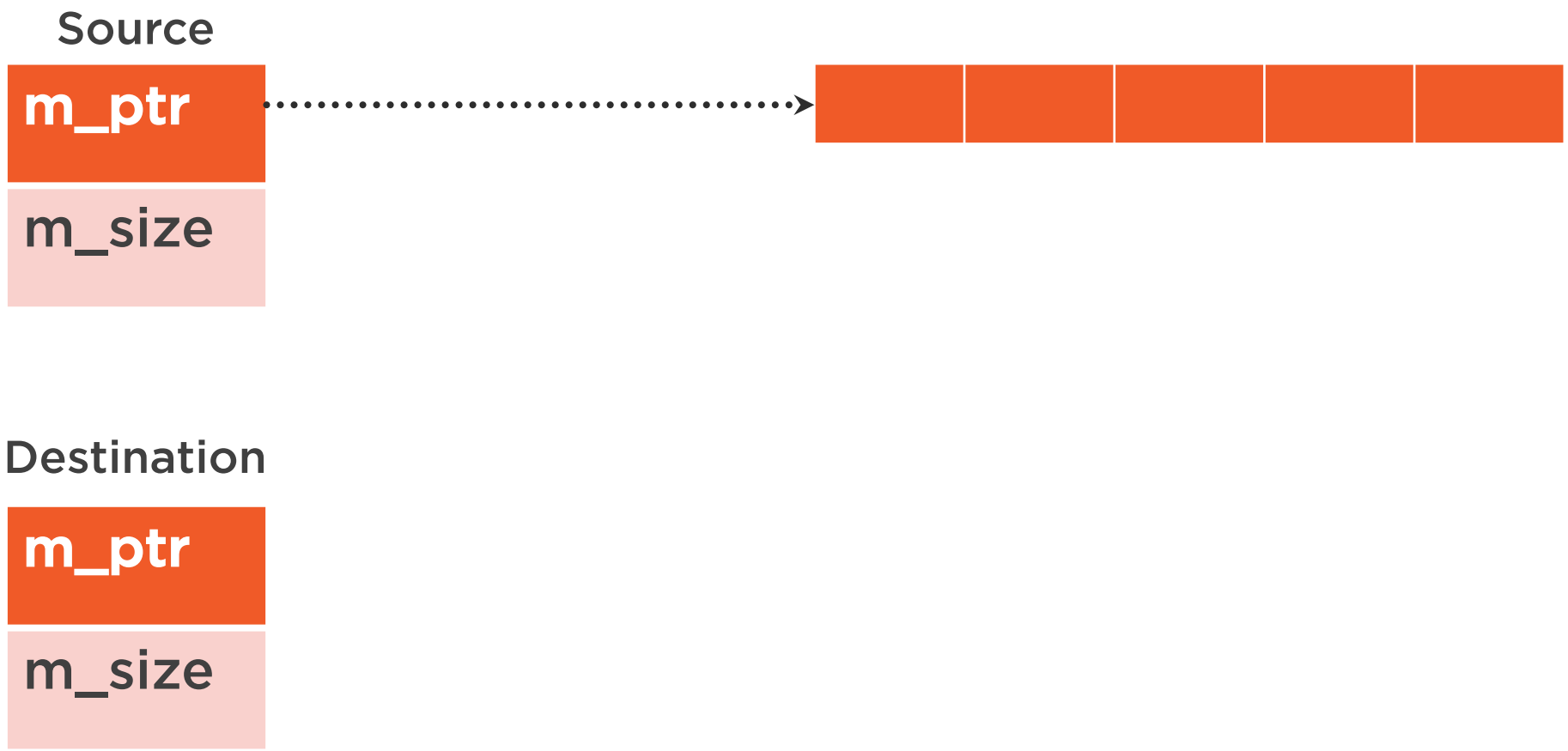
# Move Constructor



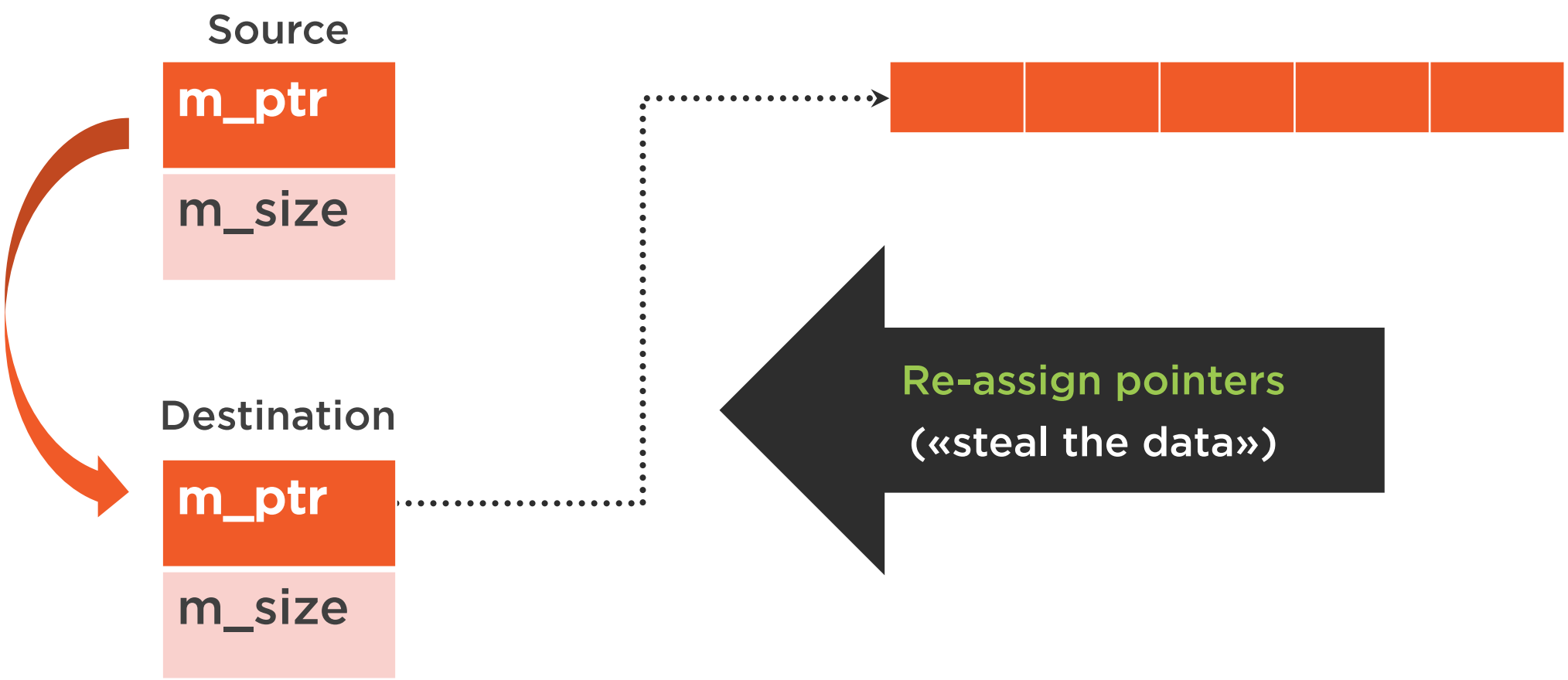
# Moving Arrays



# Moving Arrays



# Moving Arrays



# Move Constructor for the Array Class

```
IntArray(IntArray&& source) {

}
```

**R-VALUE REFERENCE**

**Temporary object:**
**Can safely «steal» the data from it**

# Move Constructor for the Array Class

```
IntArray(IntArray&& source) {
  // Transfer ownership (steal data) from source
  m_ptr  = source.m_ptr;
  m_size = source.m_size;




}
```

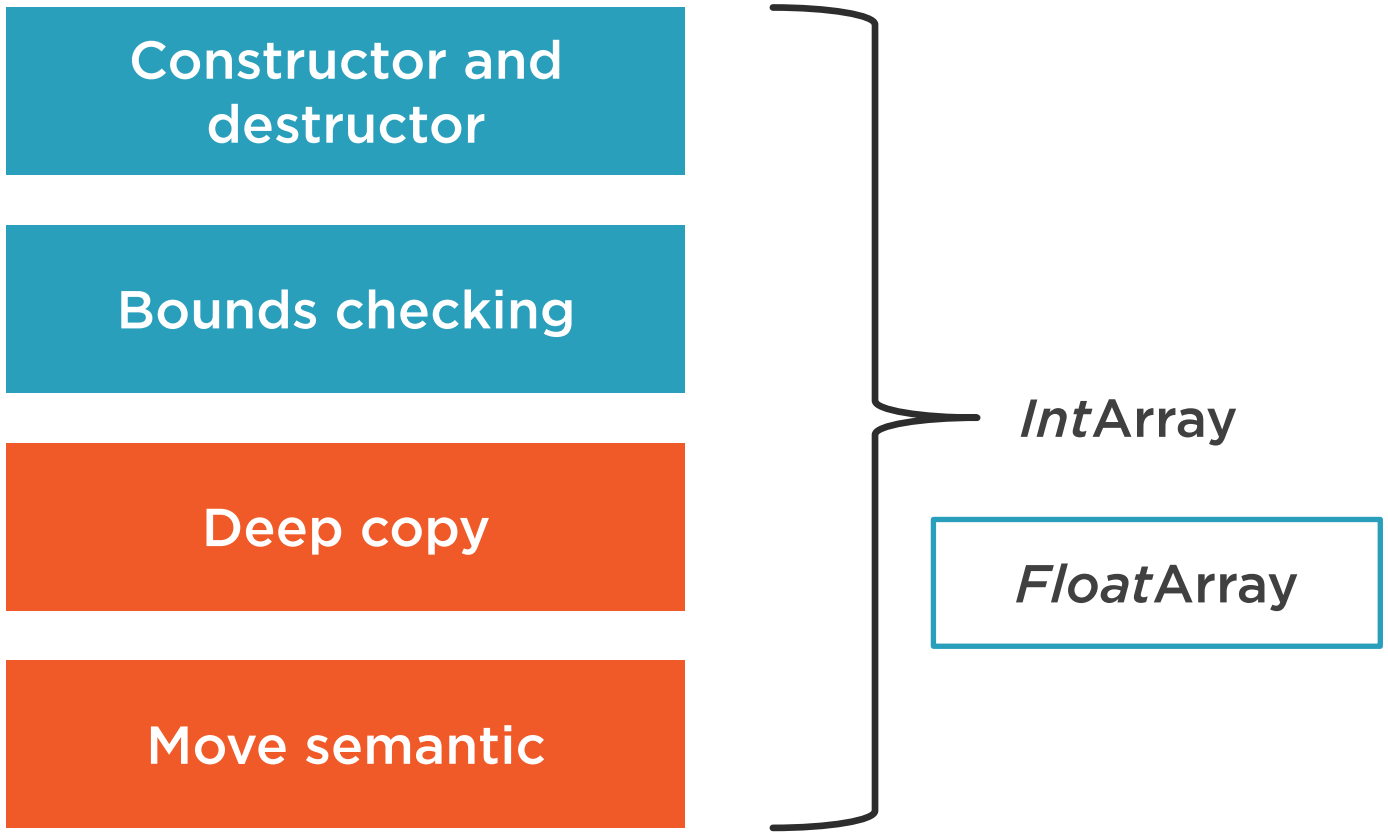
# Move Constructor for the Array Class

```
IntArray(IntArray&& source) {
  // Transfer ownership (steal data) from source
  m_ptr  = source.m_ptr;
  m_size = source.m_size;

  // Clear source
  source.m_ptr  = nullptr;
  source.m_size = 0;
}
```

# Move Constructor for the Array Class

```
IntArray(IntArray&& source) {
  // Transfer ownership (steal data) from source
  m_ptr  = source.m_ptr;
  m_size = source.m_size;

  // Clear source
  source.m_ptr  = nullptr;
  source.m_size = 0;
}
```

# The Array Class Journey



```
class IntArray {
 private:
  int* m_ptr;
   …

};
```

class *Float*Array

*float** m_ptr;

# Generalizing the Array Class

```
class IntArray {
 private:
  int* m_ptr;
   …

};
```

class *Float*Array

*float** m_ptr;

Use *templates* for writing *generic* code

# Generalizing the Array Class

```
template <typename T>
class Array {
 private:
  T* m_ptr;
   …
};
```

Use *generic type* T

C++ compiler will *synthesize* array *classes*

Use *templates* for writing *generic* code

# Generalizing the Array Class

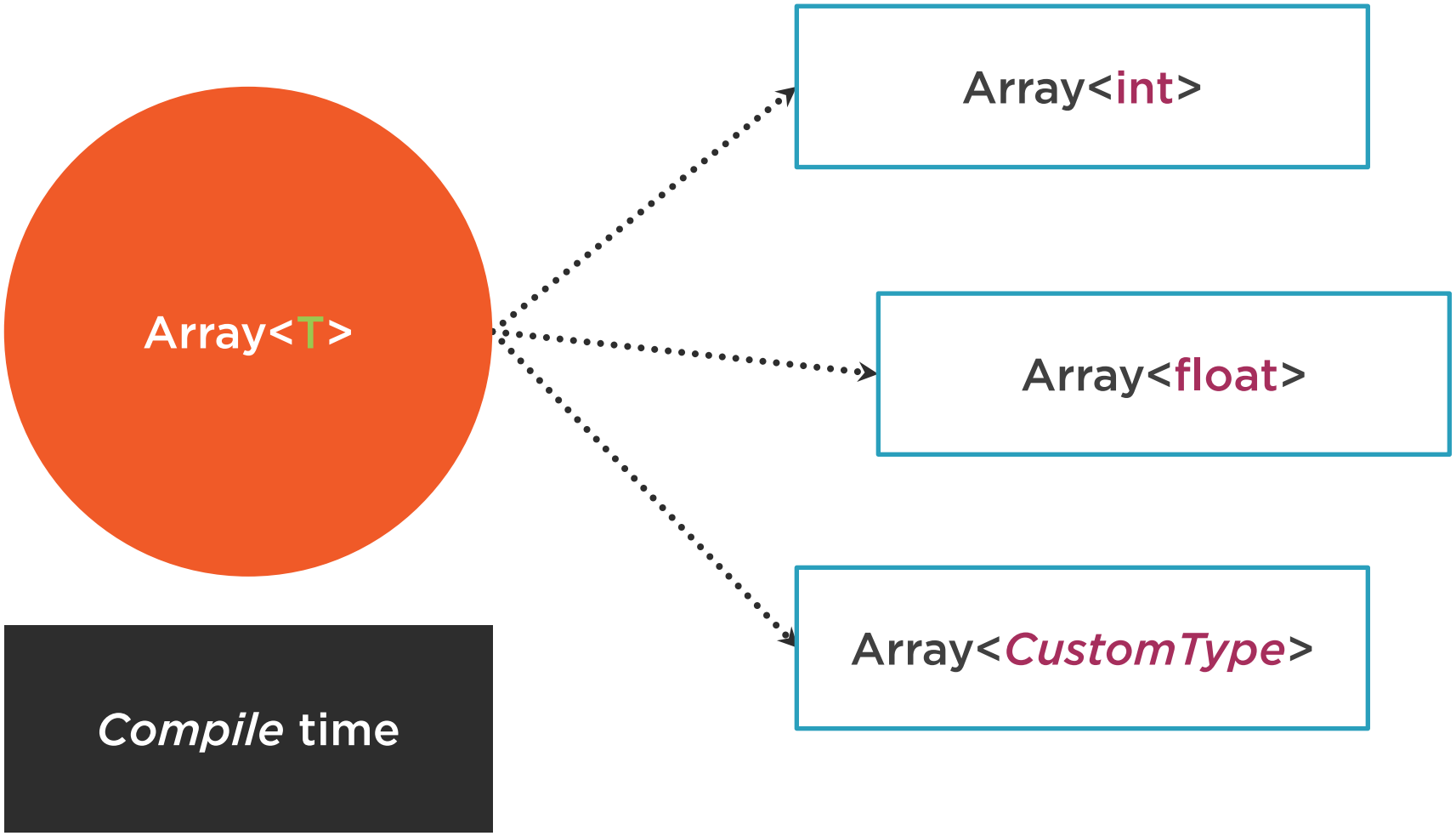
```
template <typename T>
class Array {
 private:
  T* m_ptr;
  …
};
```

Introduce class *template*

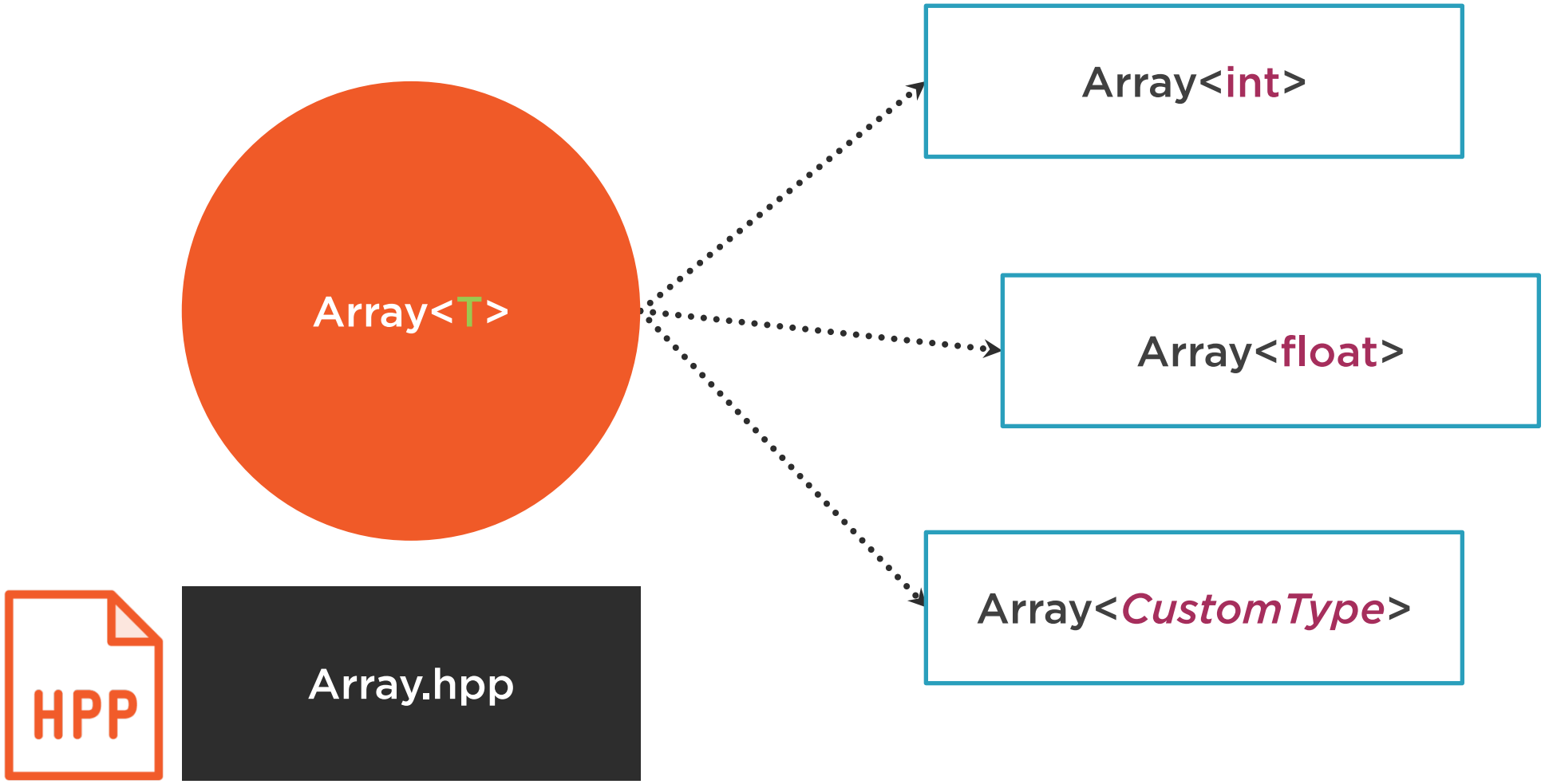
Use *templates* for writing *generic* code

# Generalizing the Array Class

# From Class Template to Concrete Classes



# From Class Template to Concrete Classes



# Summary

**Overload <<**

**Copy semantics**

- Shallow vs. deep copy
- Copy-and-swap idiom

**Move semantics**

**Generic Array<T> class template**