Advanced Topics

CSE 333 Winter 2021

Instructor: John Zahorjan

Teaching Assistants:

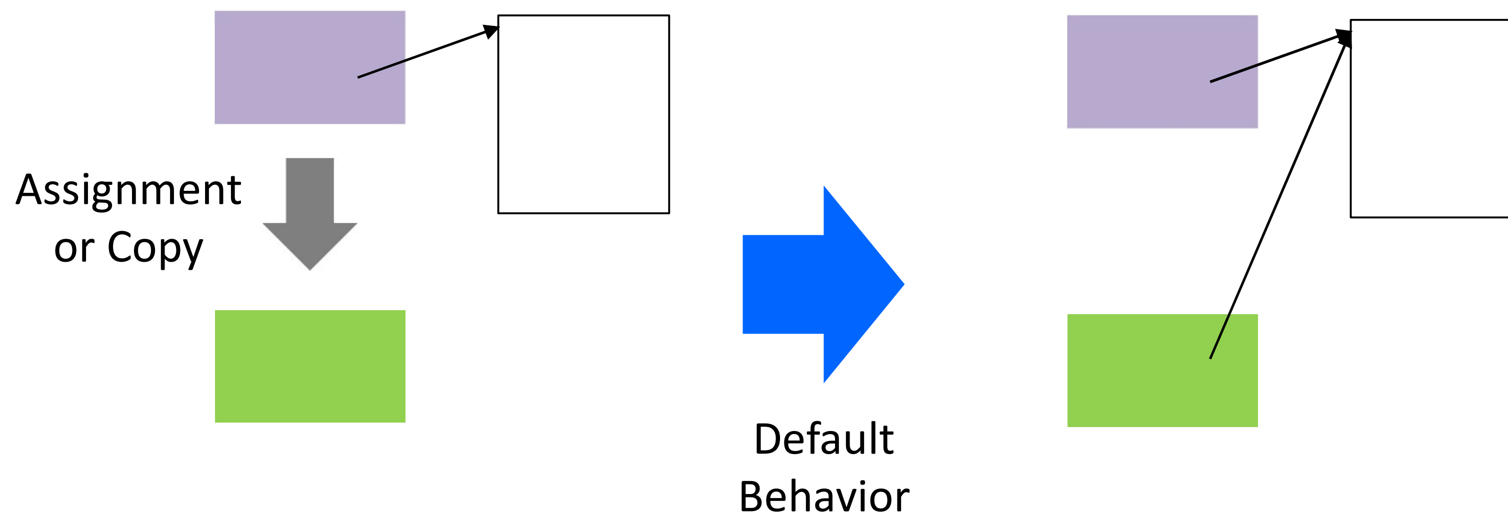
Matthew Arnold	Nonthakit Chaiwong	Jacob Cohen
Elizabeth Haker	Henry Hung	Chase Lee
Leo Liao	Tim Mandzyuk	Benjamin Shmidt
Guramrit Singh		

Advanced Topics

- Move semantics
  - What is it?
  - Why would you want it?
    - Functionality
    - Performance
  - How do you get it?
- Conversion Operators
- Functors
  - Function objects

Default Assignment and Copy

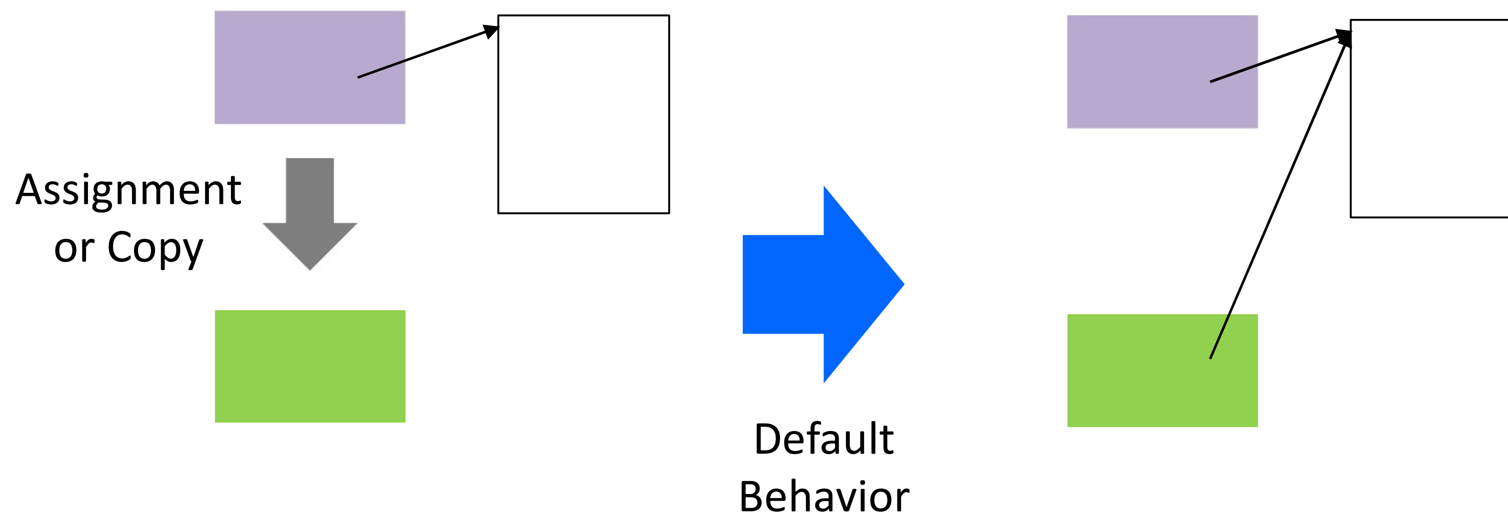
- The version of assignment we inherited from C is byte copy
  - This leads to *shallow copy* when objects contain pointers



Now may have double free, dangling pointer, and/or memory leak issues

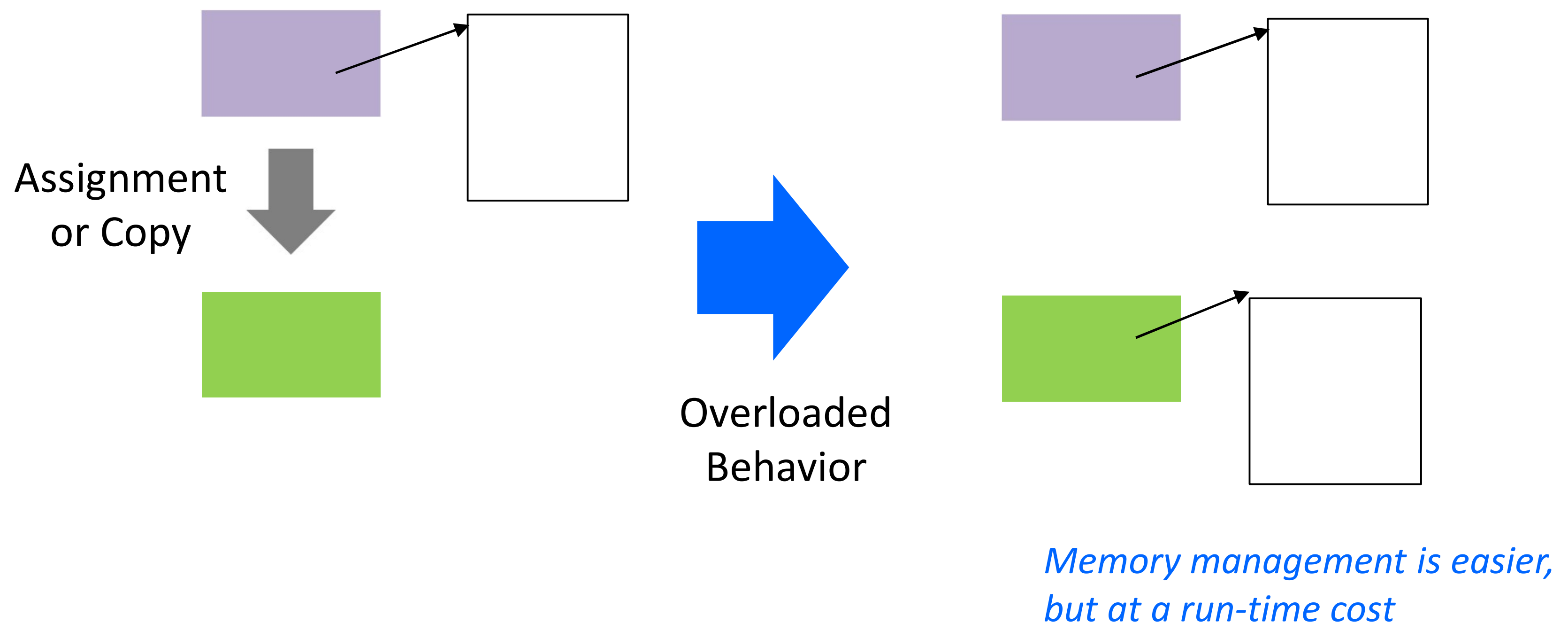
Default Assignment and Smart Pointers

- std::shared_ptr<T> makes memory management problems less likely, when they can be used
- Cannot have this semantics with std::unique_ptr<T>

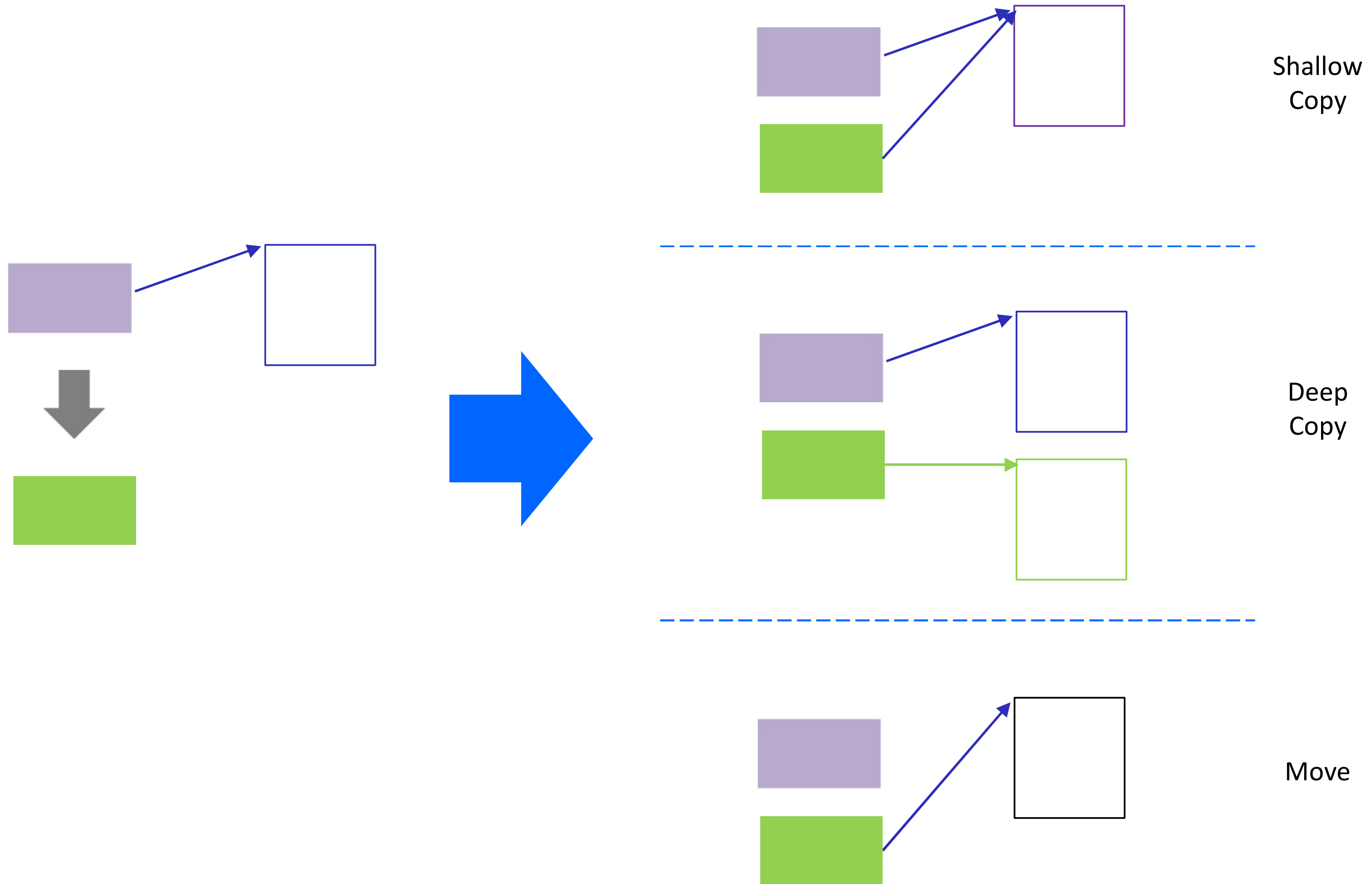


Deep Copy Semantics

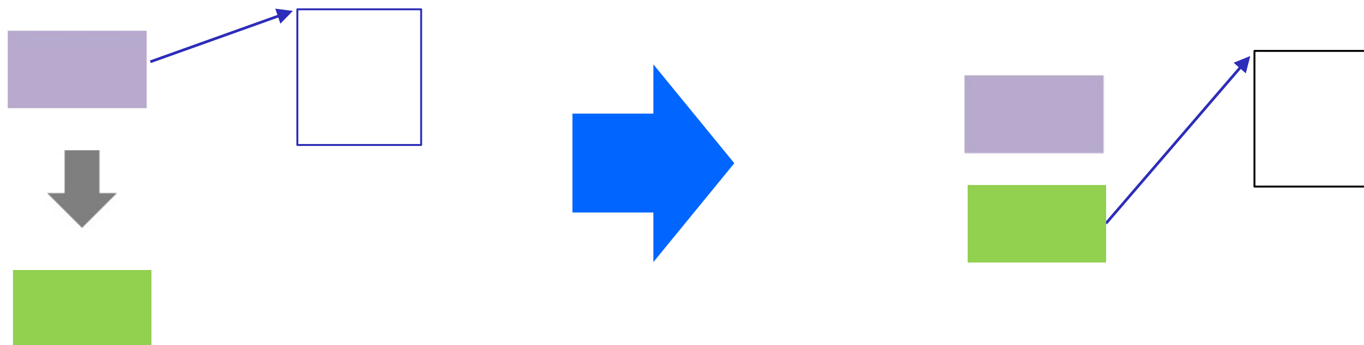
- By overloading the copy constructor, assignment, and the destructor (the Rule of Three) we can implement deep copy



Move Semantics



Implementing Move Semantics



- Implementation is by overloading of the assignment operator and copy construction
  - Just like for deep copy

- What if sometimes I want deep copy and sometimes I want move?
  a) Why would I?
  b) How can I? There can't be more than one copy constructor or one assignment operator.
  c) Or can there…?

Why Would I Want Move Semantics

- I'd like to transfer ownership (move) the dynamically allocated space from the old object to the new object when:
  - The old object will never touch the dynamically allocated space again
- Often/usually because the old object is a compiler managed temporary
- There are no names for the temporaries
  - So, the code can't manipulate them after the statement in which they are produced and consumed

High Level Example

```
class SparseMatrix {
   SparseMatrix() { …}  // news some data structure
    ~Matrix() {…}
   SparseMatrix operator*(const SparseMatrix & other) { …; return newMatrix;}
   …
};
int main(int argc, char *argv[]) {
   SparseMatrix A();
   SparseMatrix B();
   …
   A = A * B;
    …
}
```

- Why is operator*() returning a new object by value?
  - What if it builds the result on the stack?
  - What if it builds the result in the heap?
  - What if it put the result in the lhs (left-hand-side) object?

High Level Example

```
class SparseMatrix {
   SparseMatrix() { …}  // news some data structure
    ~Matrix() {…}
   SparseMatrix operator*(const SparseMatrix & other) { …; return newMatrix;}
   …
};
int main(int argc, char *argv[]) {
   SparseMatrix A();
   SparseMatrix B();
   …
   A = A * B;
    …
}
```

- A*B is a new object that doesn't have a name
- After the = is performed, the program can't possibly access the memory pointed at by the A*B object
  - It has no way to refer to that object
  - In any case, the compiler will have destroyed it because it's a temporary

rvalue and lvalue reference

- Informally, an lvalue reference is a reference to something that can appear to the left of an assignment operator
  - lvalue reference is indicated by ‘&’
- (Informally) An rvalue reference is to something that can’t
  - rvalue reference is indicated by ‘&&’

```
int i = 42;
int &r = i;       // okay, refers to i
int &&rr = i;  // error, cannot bind an rvalue ref to an lvalue

int &r2 = i * 42;            // error, i * 42 is an rvalue
const int &r3 = i * 42; // ok, can bind a ref to a const to an rvalue
int &&rr2 = i * 42;       // ok, i * 42 is an rvalue
```

Why Distinguish rvalues and lvalues?

- The two kinds of references are distinct <u>types</u>
  - So, can create distinct copy constructors and assignment operators for them

- ```
  MyObj::MyObj(MyObj & other) {
     // normal copy constructor
     // perform deep copy
  }

  MyObj::MyObj(MyObj && other ) {
    // move copy constructor
    this->my_ptr_ = other.my_ptr_;   // take ownership of dynamic memory
    other.my_ptr_ = nullptr;         // prepare other obj for destruction
  }
  ```
  ```

An Example

```
class MyObject {
public:
  MyObject(int n) { pN_ = new int(n); }
  MyObject( const MyObject & other ) {
    pN_ = new int(other.pN_ ? *other.pN_ : 0);
  }
  ~MyObject() { if (pN_) delete pN_; }
  MyObject operator*(const MyObject & other) {
    MyObject result(*this->pN_ * *other.pN_);
    return result;
  }
  MyObject & operator=(MyObject && other ) {
    std::cout << "Move assignment: " << *other.pN_ << std::endl;
    if ( this-> pN_ ) delete pN_;   // free my memory
    this->pN_ = other.pN_;          // take other's memory
    other.pN_ = nullptr;            // prepare other for destruction
    return *this;
  }
  explicit operator int() const {
    if ( pN_ ) return *pN_;
    else return 0;
  }
private:
  int * pN_;
};
```

Should check if this == &other...

Move assignment operator

Bonus: conversion operator

An Example

```
class MyObject {
public:
  MyObject(int n) { pN_ = new int(n); }
  MyObject( const MyObject & other ) {
    pN_ = new int(other.pN_ ? *other.pN_ : 0);
  }
  ~MyObject() { if (pN_) delete pN_; }
  MyObject operator*(const MyObject & other) {
    MyObject result(*this->pN_ * *other.pN_);
    return result;
  }
  MyObject & operator=(MyObject && other ) {
    std::cout << "Move assignment: " << *other.pN_ << std::endl;
    if ( this-> pN_ ) delete pN_;   // free my memory
    this->pN_ = other.pN_;          // take other's memory
    other.pN_ = nullptr;            // prepare other for destruction
    return *this;
  }
  explicit operator int() const {
    if ( pN_ ) return *pN_;
    else return 0;
  }
private:
  int * pN_;
};
```

```
int main(int argc, char *argv[])
{
  MyObject A(2);
  MyObject B(3);
  A = A * B;
  std::cout << "A = " << (int)A << std::endl;
  return 0;
}
```

```
attu> ./a.out
Move assignment: 6
A = 6
```

Explicit Conversion Operator

```
class MyObject {
public:
 ...
 explicit operator int() const {
  if ( pN_ ) return *pN_;
  else return 0;
 }
private:
 int * pN_;
};
```

```
int main(int argc, char *argv[])
{
 MyObject A(2);
 MyObject B(3);
 A = A * B;
 std::cout << "A = " << A << std::endl;
 return 0;
}
```

```
test.cc: In function ‘int main(int, char**)’:
test.cc:36:23: error: no match for ‘operator<<’ (operand types are ‘std::basic_ostream<char>’ and ‘MyObject’)
  36 |   std::cout << "A = " << A << std::endl;
     |   ~~~~~~~~~~~~~~~~~~~ ^~ ~
```

Non-Explicit Conversion Operator

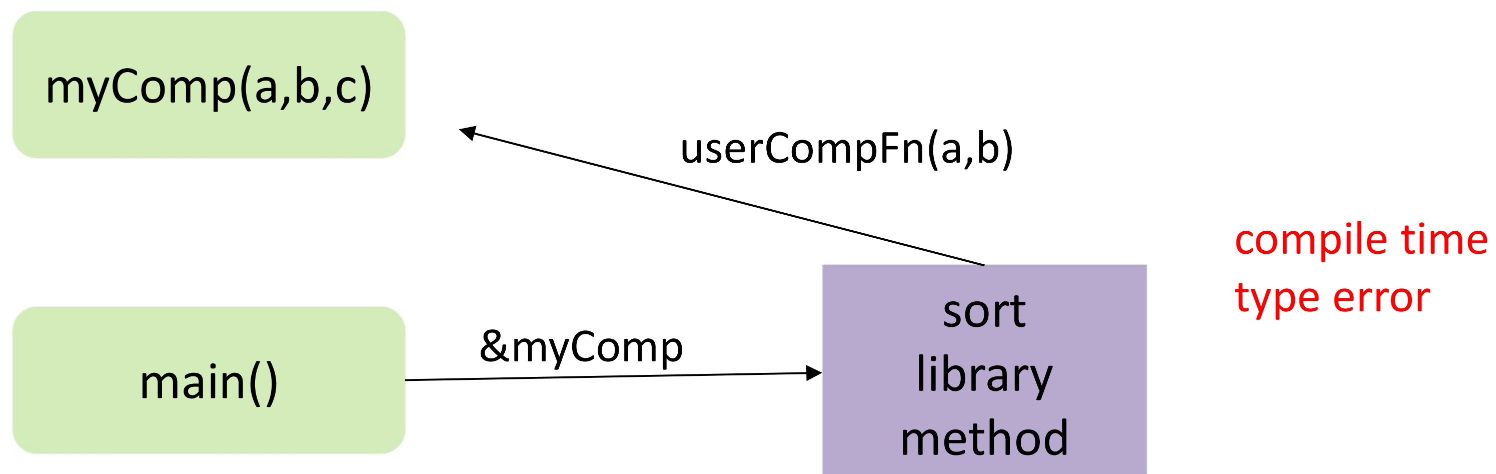
```
class MyObject {
public:
  ...
operator int() const {
   if ( pN_ ) return *pN_;
   else return 0;
  }
private:
  int * pN_;
};
```

```
int main(int argc, char *argv[])
{
  MyObject A(2);
  MyObject B(3);
  A = A * B;
  std::cout << "A = " << A << std::endl;
  return 0;
}
```

```
attu> g++ -std=c++17 -Wall -g test.cc
attu> ./a.out
Move assignment: 6
A = 6
```

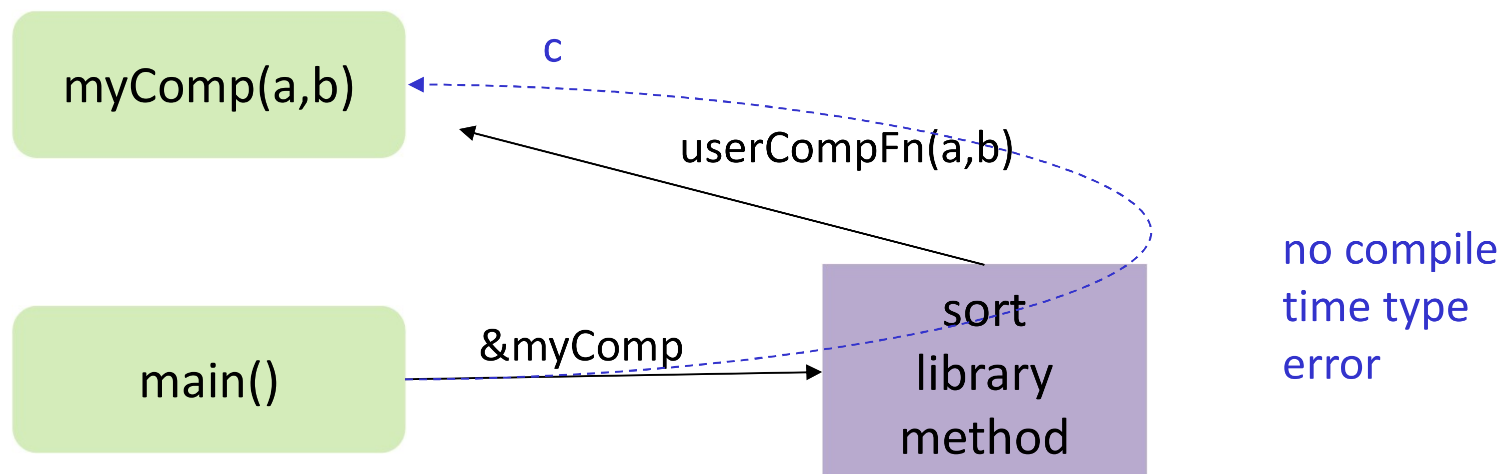
Function Objects

- Example issue
  - You're working with a library that allows customized operation by passing a pointer to a function you write that the library will invoke as a callback
  - The library defines a specific interface for that function
  - Your functionality requires additional arguments



Function Objects

- Example issue
  - You're working with a library that allows customized operation by passing a pointer to a function you write that the library will invoke
  - The library defines a specific interface for that function
  - Your functionality requires additional arguments



Function Object Example

- We're going to build to it slowly because it involves a number of C++ features you may not be entirely familiar with
- Steps:
  1. Use of Vector<T> and templated method to print it
  2. Use of std::sort to sort in default order
  3. Passing compare method to sort to sort in descending order
  4. Using templated compare method to sort in an odd order
     - Compile time function specialization
  5. Use of a function object to sort in an odd order
     - Execution time function specialization
  6. A different example use of stateful function objects
  7. Compiler support to create some kinds of function objects

Step 1: Vector<T> and Templated Function

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

template<class T>
void printVec(T& v)
{
  for (auto el : v)
    std::cout << " " << el;
  std::cout << std::endl;
}

int main(int argc, char *argv[])
{
  std::vector<int> test_vec;
  for ( int i=0; i<20; i++ )
    test_vec.push_back(rand() % 100);
  printVec(test_vec);

  return 0;
}
```

I'll be omitting #include line after this

```
attu> ./a.out
 83 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36
```

Step 2: std::sort in default order

```
template<class T>
void printVec(T& v)
{
  for (auto el : v)
    std::cout << " " << el;
  std::cout << std::endl;
}

int main(int argc, char *argv[])
{
  std::vector<int> test_vec;
  for ( int i=0; i<20; i++ )
    test_vec.push_back(rand() % 100);
  std::sort(test_vec.begin(), test_vec.end());

  printVec(test_vec);

  return 0;
}
```

I'll be omitting printVec() after this

```
attu> ./a.out
 15 21 26 26 27 35 36 40 49 59 62 63 72 77 83 86 86 90 92 93
```

Step 3: Customized Comparison

```
template<class T>
bool sort_descending(T a, T b)
{
  return (a > b);
}

int main(int argc, char *argv[])
{
  std::vector<int> test_vec;
  for ( int i=0; i<20; i++ )
    test_vec.push_back(rand() % 100);
  std::sort(test_vec.begin(), test_vec.end(), sort_descending<int>);

  printVec(test_vec);

  return 0;
}
```

```
attu> ./a.out
 93 92 90 86 86 83 77 72 63 62 59 49 40 36 35 27 26 26 21 15
```

Step 4: Compile Time Function Specialization

- I want now to sort all elements that are evenly divisible by N before all other elements
- Example: N = 3
  93 90 72 63 36 27 21 15 92 86 86 83 77 62 59 49 40 35 26 26

- My implementation has a generic comparison method that achieves this partitioned sort. It should work for any N.
- I don't examine N at runtime. Instead, a version of the general method specialized specialize for a particular N is created at compile time
  - *The compiler may be able to apply optimizations to the specialized function that it would not be able to apply the unspecialized function*

Step 4: Compile-time function specialization

```
template<class T, int N>
bool sort_weird(T a, T b) {
  if ( a % N == 0) {
     if ( b % N != 0 ) return true;
     else return (a > b);
  }
  if ( b % N == 0)
    return false;
  return (a > b);
}
int main(int argc, char *argv[])
{
  std::vector<int> test_vec;
  for ( int i=0; i<20; i++ )
    test_vec.push_back(rand() % 100);
  std::sort(test_vec.begin(), test_vec.end(), sort_weird<int, 3>);

  printVec(test_vec);

  return 0;
}
```

```
attu> ./a.out
93 90 72 63 36 27 21 15 92 86 86 83 77 62 59 49 40 35 26 26
```

Next Step

- I'd like to specialize at runtime
  - May not know N until runtime
- It's not a problem writing a compare that takes two elements and an argument N
  - We basically just make the compile time parameter N to the template a run time parameter to the comparison method
- The problem is that sort() isn't prepared to invoke a three argument function
- We need to get the argument N from main() to the comparison function while passing through sort()
  - But how?

Step 5: Function Object

```
class SortObj {
public:
  SortObj(int factor) { factor_ = factor; }
  bool operator() (int a, int b)
  {
    if ( a % factor_ == 0)
     {
       if ( b % factor_ != 0 ) return true;
       else return (a > b);
     }
    if ( b % factor_ == 0)
      return false;
    return (a > b);
  }
private:
  int factor_;
};
```

```
int main(int argc, char *argv[])
{
  std::vector<int> test_vec;
  for ( int i=0; i<20; i++ )
    test_vec.push_back(rand() % 100);

  SortObj S(argc);
  std::sort(test_vec.begin(), test_vec.end(), S);

  printVec(test_vec);

  return 0;
}
```

Passing object as function

```
attu>  ./a.out two three four
 92 72 40 36 93 90 86 86 83 77 63 62 59 49 35 27 26 26 21 15
```

Step 6: Another Example Function Object

```
/*  (In non-example code,
    use std::max_element()) */
template<class T>
class MaxObj {
public:
  void operator() (T term)
  {
    if ( !initialized_ )
     {
       max_ = term;
       initialized_ = true;
     }
    else if ( max_ < term )
      max_ = term;
  }
  T value() { return max_; }
private:
  bool initialized_ = false;
  T max_;
};
```

```
int main(int argc, char *argv[])
{
  std::vector<int> test_vec;
  for ( int i=0; i<20; i++ )
    test_vec.push_back(rand() % 100);

  MaxObj<decltype (test_vec)::value_type> max_obj;
  max_obj = for_each ( test_vec.begin(), test_vec.end(), max_obj);

  std::cout << "Vector max is " << max_obj.value() << std::endl;
  printVec(test_vec);

  return 0;
}
```

```
Vector max is 93
 83 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36
```

Step 7: Creating Function Objects – bind()

```
bool compare_weird (int a, int b, int factor)
{
  if ( a % factor == 0)
    {
      if ( b % factor != 0 ) return true;
      else return (a > b);
    }
  if ( b % factor == 0)
    return false;
  return (a > b);
}
```

```
int main(int argc, char *argv[])
{
  std::vector<int> test_vec;
  for ( int i=0; i<20; i++ )
    test_vec.push_back(rand() % 100);

  auto bound_compare = std::bind(compare_weird, _1, _2, argc);
  std::sort(test_vec.begin(), test_vec.end(), bound_compare);

  printVec(test_vec);

  return 0;
}
```

```
attu> ./a.out two three four five six seven
 77 63 49 35 21 93 92 90 86 86 83 72 62 59 40 36 27 26 26 15
```