Introduction to Programming (C/C++)

07: STL & Modern C++

Huanchen Zhang

清華大學
Tsinghua University

交叉信息研究院
Institute for Interdisciplinary
Information Sciences

Agenda

- ➜ Operator Overloading
- ➜ Standard Template Library (STL)
- ➜ Type Deduction
- ➜ Lambdas
- ➜ Move Semantics
- ➜ Smart Pointers

Agenda

- Operator Overloading
- Standard Template Library (STL)
- Type Deduction
- Lambdas
- Move Semantics
- Smart Pointers

Modern C++

Agenda

- → **Operator Overloading**
- → Standard Template Library (STL)
- → Type Deduction
- → Lambdas
- → Move Semantics
- → Smart Pointers

Operator Overloading

- ➜ Redefine built-in operators to work on your data types

Operator Overloading

→ Redefine built-in operators to work on your data types

```
int main() {
  int a = 3;
  int b = 5;
  int c = a + b;

  Vec a_vec(1, 2);
  Vec a_vec(2, 3);
  Vec c_vec = a_vec + b_vec;
}
```

Operator Overloading

➜ Redefine built-in operators to work on your data types

```
int main() {
  int a = 3;
  int b = 5;
  int c = a + b;

  Vec a_vec(1, 2);
  Vec a_vec(2, 3);
  Vec c_vec = a_vec + b_vec;
}
```

Overloaded

Operator Overloading

```
class Vec {
 public:
  Vec(int x, int y) : x_(x), y_(y) {}
  int GetX() const { return x_; }
  int GetY() const { return y_; }
 private:
  int x_, y_;
};
```

```
int main() {
  Vec a_vec(1, 2);
  Vec a_vec(2, 3);
  Vec c_vec = a_vec + b_vec;
}
```

Operator Overloading

```
class Vec {
 public:
  Vec(int x, int y) : x_(x), y_(y) {}
  int GetX() const { return x_; }
  int GetY() const { return y_; }
 private:
  int x_, y_;
};

Vec operator+(const Vec &a, const Vec &b) {
  return Vec(a.GetX() + b.GetX(), a.GetY() + b.GetY());
}
```

```
int main() {
  Vec a_vec(1, 2);
  Vec a_vec(2, 3);
  Vec c_vec = a_vec + b_vec;
}
```

Operator Overloading

```
class Vec {
 public:
  Vec(int x, int y) : x_(x), y_(y) {}
   ?

 private:
  int x_, y_;
};

Vec operator+(const Vec &a, const Vec &b) {
  return Vec(a.GetX() + b.GetX(), a.GetY() + b.GetY());
}
```

Operator Overloading

```
class Vec {
 public:
  Vec(int x, int y) : x_(x), y_(y) {}
  friend Vec operator+(const Vec &a, const Vec &b);

 private:
  int x_, y_;
};

Vec operator+(const Vec &a, const Vec &b) {
  return Vec(a.x_ + b.x_, a.y_ + b.y_);
}
```

Operator Overloading

```
class Vec {
 public:
  Vec(int x, int y) : x_(x), y_(y) {}
  friend Vec operator+(const Vec &a, int val);

 private:
  int x_, y_;
};

Vec operator+(const Vec &a, int val) {
  return Vec(a.x_ + val, a.y_);
}
```

```
int main() {
  Vec a_vec(1, 2);
  Vec c_vec = a_vec + 3;
}
```

Overloading I/O Operators

```
class Vec {
 public:
  Vec(int x, int y) : x_(x), y_(y) {}


 private:
  int x_, y_;
};
```

```
int main() {
  Vec a_vec(1, 2);
  std::cout << a_vec << “\n”;
}
```

```
>> (1, 2)
```

Overloading I/O Operators

```
class Vec {
 public:
  Vec(int x, int y) : x_(x), y_(y) {}
  friend std::ostream &operator<<(
       std::ostream &out, const Vec &v);
 private:
  int x_, y_;
};

std::ostream &operator<<(std::ostream &out, const Vec &v) {
  out << "(" << x_ << ", " << y_ << ")";
  return out;
}
```

```
int main() {
  Vec a_vec(1, 2);
  std::cout << a_vec << "\n";
}
```

```
>> (1, 2)
```

Overloading Using Member Functions

```
class Vec {
 public:
  Vec(int x, int y) : x_(x), y_(y) {}
  bool operator!() const;

 private:
  int x_, y_;
};

bool Vec::operator!() const {
  return (!x_ && !y_);
}
```

```
int main() {
  Vec a_vec(1, 2);
  if (!a_vec)
    ...
}
```

Overloading ++/--

```
class Vec {
 public:
  Vec(int x, int y) : x_(x), y_(y) {}
  Vec &operator++() {



  }

 private:
  int x_, y_;
};
```

```
int main() {
  Vec a_vec(1, 2);
  Vec b_vec = ++a_vec;
}
```

Overloading ++/--

```
class Vec {
 public:
  Vec(int x, int y) : x_(x), y_(y) {}
  Vec &operator++() {



  }

 private:
  int x_, y_;
};
```

Pretfix Increment

```
int main() {
  Vec a_vec(1, 2);
  Vec b_vec = ++a_vec;
}
```

Overloading ++/--

```
class Vec {
 public:
  Vec(int x, int y) : x_(x), y_(y) {}
  Vec &operator++() {
    x_++; y_++;
    return *this;
  }

 private:
  int x_, y_;
};
```

Pretfix Increment

```
int main() {
  Vec a_vec(1, 2);
  Vec b_vec = ++a_vec;
}
```

Overloading ++/--

```
class Vec {
 public:
  Vec(int x, int y) : x_(x), y_(y) {}
  Vec &operator++() {
    x_++; y_++;
    return *this;
  }

 private:
  int x_, y_;
};
```

Pretfix Increment

```
int main() {
  Vec a_vec(1, 2);
  Vec b_vec = ++a_vec;
}
```

Overloading ++/--

```
class Vec {
 public:
  Vec(int x, int y) : x_(x), y_(y) {}
  Vec &operator++() {
    x_++; y_++;
    return *this;
  }

 private:
  int x_, y_;
};
```

```
int main() {
  Vec a_vec(1, 2);
  Vec b_vec = a_vec++;
}
```

Overloading ++/--

```
class Vec {
 public:
  Vec(int x, int y) : x_(x), y_(y) {}
  Vec operator++(int) {




  }
 private:
  int x_, y_;
};
```

Postfix Increment

```
int main() {
  Vec a_vec(1, 2);
  Vec b_vec = a_vec++;
}
```

Overloading ++/--

```
class Vec {
 public:
  Vec(int x, int y) : x_(x), y_(y) {}
  Vec operator++(int) {
    Vec tmp(x_, y_);
    x_++; y_++;
    return tmp;
  }
 private:
  int x_, y_;
};
```

Postfix Increment

```
int main() {
  Vec a_vec(1, 2);
  Vec b_vec = a_vec++;
}
```

Overloading ++/--

```
class Vec {
 public:
  Vec(int x, int y) : x_(x), y_(y) {}
  Vec operator++(int) {
    Vec tmp(x_, y_);
    x_++; y_++;
    return tmp;
  }
 private:
  int x_, y_;
};
```

Postfix Increment

(Typically Less Efficient)

```
int main() {
  Vec a_vec(1, 2);
  Vec b_vec = a_vec++;
}
```

Overloading [ ]

```
class SimpleList {
 public:
  SimpleList(int size) : size_(size) {
    list_ = new int[size_];
  }
  int Get(int pos) const { return list_[pos]; }
  void Set(int pos, int val) { list_[pos] = val; }

 private:
  int size_, *list_;
};
```

```
int main() {
  SimpleList list(100);
  list.Set(1, 5);
  int val = list.Get(1);
}
```

Overloading [ ]

```
class SimpleList {
 public:
  SimpleList(int size) : size_(size) {
    list_ = new int[size_];
  }
  int &operator[](int pos) {
    return list_[pos];
  }

 private:
  int size_, *list_;
};
```

```
int main() {
  SimpleList list(100);
  list[1] = 5;
  int val = list[1];
}
```

Overloading [ ]

```
class SimpleList {
 public:
  SimpleList(int size) : size_(size) {
    list_ = new int[size_];
  }
  int &operator[](int pos) {
    return list_[pos];
  }

 private:
  int size_, *list_;
};
```

```
int main() {
  SimpleList list(100);
  list[1] = 5;
  int val = list[1];
}
```

Overloading [ ]

```
class SimpleList {
 public:
  SimpleList(int size) : size_(size) {
    list_ = new int[size_];
  }
  int &operator[](int pos) {
    assert(pos >= 0 && pos < size_);
    return list_[pos];
  }
 private:
  int size_, *list_;
};
```

#include <cassert>

```
int main() {
  SimpleList list(100);
  list[1] = 5;
  int val = list[1];
}
```

Agenda

- → Operator Overloading
- → **Standard Template Library (STL)**
- → Type Deduction
- → Lambdas
- → Move Semantics
- → Smart Pointers

Standard Template Library

- ➜ Part of the C++ Standard Library
- ➜ Implements common data structures and algorithms
  - Containers
  - Iterators
  - Algorithms
- ➜ Uses templates heavily

STL Containers

- ➜ A class designed to hold and organize instances of another type
- ➜ Typical Interface
  - Create an empty container
  - Insert a new object
  - Remove an object
  - Access to an object
  - Number of objects stored
  - Clear the container
  - …

std::vector<T>

➜ **Dynamic Array** (ArrayList)

```
#include <vector>
int main() {
  std::vector<int> v;









}
```

std::vector<T>

➔ **Dynamic Array** (ArrayList)

```
#include <vector>
int main() {
  std::vector<int> v;
  for (int i = 0; i < 100; i++) {
    v.push_back(i);
  }





}
```

std::vector<T>

➔ **Dynamic Array** (ArrayList)

```cpp
#include <vector>
int main() {
  std::vector<int> v;
  for (int i = 0; i < 100; i++) {
    v.push_back(i);
  }
  for (int i = 0; i < (int)v.size(); i++) {
    v[i] = v[i] + 1;
  }


}
```

std::vector<T>

➔ **Dynamic Array** (ArrayList)

```cpp
#include <vector>
int main() {
  std::vector<int> v;
  for (int i = 0; i < 100; i++) {
    v.push_back(i);
  }
  for (int i = 0; i < (int)v.size(); i++) {
    v[i] = v[i] + 1;
  }
  if (!v.empty()) v.pop_back();
  v.clear();
}
```

std::vector<T>

➔ **Dynamic Array** (ArrayList)

```
#include <vector>
int main() {
  std::vector<int> v;
  for (int i = 0; i < 100; i++) {
    v.push_back(i);
  }
  for (int i = 0; i < (int)v.size(); i++) {
    v[i] = v[i] + 1;
  }
  if (!v.empty()) v.pop_back();
  v.clear();
}
```

std::**stack**<T>

std::**queue**<T>

std::**priority_queue**<T>

std::string

➔ if you are tired of manipulating C-Strings, use **std::string**!

```
#include <string>
int main() {
  std::string s = “Hairy Potter”;







}
```

std::string

- ➜ if you are tired of manipulating C-Strings, use **`std::string`**!

```
#include <string>
int main() {
  std::string s = “Hairy Potter”;
  s[2] = ‘r’;




}
```

std::string

- if you are tired of manipulating C-Strings, use **std::string**!

```
#include <string>
int main() {
  std::string s = "Hairy Potter";
  s[2] = 'r';
  s += " is a genius";




}
```

std::string

- ➜ if you are tired of manipulating C-Strings, use **std::string**!

```
#include <string>
int main() {
  std::string s = “Hairy Potter”;
  s[2] = ‘r’;
  s += “ is a genius”;
  int len = s.length();




}
```

std::string

➜ if you are tired of manipulating C-Strings, use **std::string**!

```
#include <string>
int main() {
  std::string s = “Hairy Potter”;
  s[2] = ‘r’;
  s += “ is a genius”;
  int len = s.length();   ←  No end-of-string NULL



}
```

std::string

➔ if you are tired of manipulating C-Strings, use **std::string**!

```
#include <string>
int main() {
  std::string s = “Hairy Potter”;
  s[2] = ‘r’;
  s += “ is a genius”;
  int len = s.length();
  std::string sub_s = s.substr(6, 3);


}
```

std::string

➔ if you are tired of manipulating C-Strings, use **std::string**!

```
#include <string>
int main() {
  std::string s = “Hairy Potter”;
  s[2] = ‘r’;
  s += “ is a genius”;
  int len = s.length();
  std::string sub_s = s.substr(6, 3);

}
```

Length

Start position

std::string

➔ if you are tired of manipulating C-Strings, use **std::string**!

```
#include <string>
int main() {
  std::string s = “Hairy Potter”;
  s[2] = ‘r’;
  s += “ is a genius”;
  int len = s.length();
  std::string sub_s = s.substr(6, 3);     “Pot”
}
```

Length

Start position

std::string

→ if you are tired of manipulating C-Strings, use **std::string**!

```
#include <string>
int main() {
  std::string s = "Hairy Potter";
  s[2] = 'r';
  s += " is a genius";
  int len = s.length();
  std::string sub_s = s.substr(6, 3);
  int pos = s.find("Potter");

}
```

std::string

- ➜ if you are tired of manipulating C-Strings, use **std::string**!

```
#include <string>
int main() {
  std::string s = “Hairy Potter”;
  s[2] = ‘r’;
  s += “ is a genius”;
  int len = s.length();
  std::string sub_s = s.substr(6, 3);
  int pos = s.find(“Potter”);
```

← Returns the start position of the first occurrence

```
}
```

std::string

➜ if you are tired of manipulating C-Strings, use **std::string**!

```
#include <string>
int main() {
  std::string s = “Hairy Potter”;
  s[2] = ‘r’;
  s += “ is a genius”;
  int len = s.length();
  std::string sub_s = s.substr(6, 3);
  int pos = s.find(“Potter”);
  s.replace(pos, 6, “Hotter”);
}
```

std::set<T>

➜ Collection of **unique** values, **sorted**

```
#include <set>
int main() {
  std::set<int> int_set;
  for (int i = 0; i < 100; i++) {
    int_set.insert(i);
  }

}
```

std::set<T>

➜ Collection of **unique** values, **sorted**

```
#include <set>
int main() {
  std::set<int> int_set;
  for (int i = 0; i < 100; i++) {
    int_set.insert(i);
  }
  // how to go through all the values in the set?
}
```

STL Iterators

- ➜ An object used to traverse a container class
  - Kind of like a pointer to an element in the container

STL Iterators

- ➜ An object used to traverse a container class
  - Kind of like a pointer to an element in the container
- ➜ Each container provides at least:
  - `container::iterator`
  - `container::const_iterator` (read-only)

STL Iterators

- An object used to traverse a container class
  - Kind of like a pointer to an element in the container
- Each container provides at least:
  - `container::iterator`
  - `container::const_iterator` (read-only)
- Typical Interface (overloaded operators)
  - `Operator*`: dereference the iterator
  - `Operator++`: move to the next element in the container
  - `Operator=`: assign the iterator to a new "position"
  - `Operator==/!=`: compare the iterator "position"
  - ...

std::set<T> Traversal

→ Collection of **unique** values, **sorted**

```
#include <set>
int main() {
  std::set<int> int_set;
  for (int i = 0; i < 100; i++)
    int_set.insert(i);
  int sum = 0;





}
```

std::set<T> Traversal

➔ Collection of **unique** values, **sorted**

```
#include <set>
int main() {
  std::set<int> int_set;
  for (int i = 0; i < 100; i++)
    int_set.insert(i);
  int sum = 0;
  std::set<int>::const_iterator iter;



}
```

Declares a read-only iterator

std::set<T> Traversal

→ Collection of **unique** values, **sorted**

```
#include <set>
int main() {
  std::set<int> int_set;
  for (int i = 0; i < 100; i++)
    int_set.insert(i);
  int sum = 0;
  std::set<int>::const_iterator iter;
  iter = int_set.cbegin();



}
```

Declares a read-only iterator

Assign the iterator to the first element in the set

`std::set<T>` Traversal

➜ Collection of **unique** values, **sorted**

```
#include <set>
int main() {
  std::set<int> int_set;
  for (int i = 0; i < 100; i++)
    int_set.insert(i);
  int sum = 0;
  std::set<int>::const_iterator iter;      ← Declares a read-only iterator
  iter = int_set.cbegin();      ← Assign the iterator to the first element in the set
  while (iter != int_set.cend()) {      ← cend() returns the "(last + 1) position"



}
```

std::set<T> Traversal

➜ Collection of **unique** values, **sorted**

```
#include <set>
int main() {
  std::set<int> int_set;
  for (int i = 0; i < 100; i++)
    int_set.insert(i);
  int sum = 0;
  std::set<int>::const_iterator iter;
  iter = int_set.cbegin();
  while (iter != int_set.cend()) {
    sum += (*iter);


}
```

← Declares an read-only iterator

← Assign the iterator to the first element in the set

← `cend()` returns the "(last + 1) position"

← Dereference the iterator to get the value

`std::set<T>` Traversal

➜ Collection of **unique** values, **sorted**

```
#include <set>
int main() {
  std::set<int> int_set;
  for (int i = 0; i < 100; i++)
    int_set.insert(i);
  int sum = 0;
  std::set<int>::const_iterator iter;
  iter = int_set.cbegin();
  while (iter != int_set.cend()) {
    sum += (*iter);
    ++iter;
  }
}
```

Declares an read-only iterator

Assign the iterator to the first element in the set

`cend()` returns the “(last + 1) position”

Dereference the iterator to get the value

Move to next

std::set<T> Find

- Collection of **unique** values, **sorted**

```
#include <set>
int main() {
  std::set<int> int_set;
  for (int i = 0; i < 100; i++)
    int_set.insert(i);

  std::set<int>::iterator iter = int_set.find(16);


}
```

std::set<T> Find

➜ Collection of **unique** values, **sorted**

```
#include <set>
int main() {
  std::set<int> int_set;
  for (int i = 0; i < 100; i++)
    int_set.insert(i);

  std::set<int>::iterator iter = int_set.find(16);


}
```

If found, points to the element

If NOT found, points to **end()**

std::set<T> Find

➜ Collection of **unique** values, **sorted**

```
#include <set>
int main() {
  std::set<int> int_set;
  for (int i = 0; i < 100; i++)
    int_set.insert(i);

  std::set<int>::iterator iter = int_set.find(16);
  if (iter != int_set.end()) {
    int_set.erase(iter);
}
```

If found, points to the element

If NOT found, points to **end()**

std::map<T>

➜ Collection of **key-value** pairs, keys are **unique**, **sorted** by key

```
#include <map>
int main() {
  std::map<std::string, int> prices;
  prices["apple"] = 10;
  prices["milk"] = 20;
  prices.insert(std::make_pair("water", 5))
  prices["milk"] = 25;




}
```

std::map<T>

➔ Collection of **key-value** pairs, keys are **unique**, **sorted** by key

```
#include <map>
int main() {
  std::map<std::string, int> prices;
  prices["apple"] = 10;
  prices["milk"] = 20;
  prices.insert(std::make_pair("water", 5))
  prices["milk"] = 25;
```

Insert if key does not exist

Update if key does not exist

```
}
```

std::map<T>

➜ Collection of **key-value** pairs, keys are **unique**, **sorted** by key

```
#include <map>
int main() {
  std::map<std::string, int> prices;
  prices[“apple”] = 10;
  prices[“milk”] = 20;
  prices.insert(std::make_pair(“water”, 5))
  prices[“milk”] = 25;
  std::map<std::string, int>::iterator iter = prices.begin();



}
```

std::map<T>

➜ Collection of **key-value** pairs, keys are **unique**, **sorted** by key

```
#include <map>
int main() {
  std::map<std::string, int> prices;
  prices[“apple”] = 10;
  prices[“milk”] = 20;
  prices.insert(std::make_pair(“water”, 5))
  prices[“milk”] = 25;
  std::map<std::string, int>::iterator iter = prices.begin();
  while (iter != prices.end()) {
    std::cout << “(” << iter->first << “, ” << iter->second << “) ”;
    ++iter;
  }
}
```

std::map<T>

- Collection of **key-value** pairs, keys are **unique**, **sorted** by key

```
#include <map>
int main() {
  std::map<std::string, int> prices;
  prices[“apple”] = 10;
  prices[“milk”] = 20;
  prices.insert(std::make_pair(“water”, 5))
  prices[“milk”] = 25;
  std::map<std::string, int>::iterator iter = prices.begin();
  while (iter != prices.end()) {
    std::cout << “(” << iter->first << “, ” << iter->second << “) ”;
    ++iter;
  }
}
```

Map Family

	Unique	Not Unique
Sorted	`map`	`multimap`
Unsorted	`unordered_map`	`unordered_multimap`

Map Family

	Unique	Not Unique
Sorted	`map`	`multimap`
Unsorted	`unordered_map`	`unordered_multimap`

Balanced Binary Tree

(i.e., red-black tree)

Map Family

	Unique	Not Unique	
Sorted	`map`	`multimap`	**Balanced Binary Tree** (i.e., red-black tree)
Unsorted	`unordered_map`	`unordered_multimap`	**Hash Table**

Map Family

	Unique	Not Unique	
Sorted	`map`	`multimap`	**Balanced Binary Tree** (i.e., red-black tree)
Unsorted	`unordered_map`	`unordered_multimap`	**Hash Table** **Faster!**

STL Algorithms

- A number of generic algorithms to apply on the containers
- Examples
  - `std::min_element/max_element`
  - `std::find`
  - `std::sort, std::reverse, std::shuffle`
  - `...`

STL Algorithms

```
#include <algorithm>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  int min_elem = *std::min_element(v.begin(), v.end());




}
```

STL Algorithms

```
#include <algorithm>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  int min_elem = *std::min_element(v.begin(), v.end());
  std::vector<int>::iterator iter = std::find(v.begin(), v.end(), 6);




}
```

STL Algorithms

```
#include <algorithm>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  int min_elem = *std::min_element(v.begin(), v.end());
  std::vector<int>::iterator iter = std::find(v.begin(), v.end(), 6);
  if (iter != v.end())
    v.insert(iter, 9);




}
```

STL Algorithms

```
#include <algorithm>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  int min_elem = *std::min_element(v.begin(), v.end());
  std::vector<int>::iterator iter = std::find(v.begin(), v.end(), 6);
  if (iter != v.end())
    v.insert(iter, 9);   ←   {3, 5, 8, 9, 6, 1, 4, 7, 2}



}
```

STL Algorithms

```
#include <algorithm>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  int min_elem = *std::min_element(v.begin(), v.end());
  std::vector<int>::iterator iter = std::find(v.begin(), v.end(), 6);
  if (iter != v.end())
    v.insert(iter, 9); ←  {3, 5, 8, 9, 6, 1, 4, 7, 2}

  std::sort(v.begin(), v.end());

}
```

STL Algorithms

```
#include <algorithm>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  int min_elem = *std::min_element(v.begin(), v.end());
  std::vector<int>::iterator iter = std::find(v.begin(), v.end(), 6);
  if (iter != v.end())
    v.insert(iter, 9);  ←  {3, 5, 8, 9, 6, 1, 4, 7, 2}

  std::sort(v.begin(), v.end());  ←  {1, 2, 3, 4, 5, 6, 7, 8, 9}

}
```

STL Algorithms

```cpp
#include <algorithm>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  int min_elem = *std::min_element(v.begin(), v.end());
  std::vector<int>::iterator iter = std::find(v.begin(), v.end(), 6);
  if (iter != v.end())
    v.insert(iter, 9);  ←  {3, 5, 8, 9, 6, 1, 4, 7, 2}

  std::sort(v.begin(), v.end());  ←  {1, 2, 3, 4, 5, 6, 7, 8, 9}
  std::reverse(v.begin(), v.end());
}
```

STL Algorithms

```
#include <algorithm>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  int min_elem = *std::min_element(v.begin(), v.end());
  std::vector<int>::iterator iter = std::find(v.begin(), v.end(), 6);
  if (iter != v.end())
    v.insert(iter, 9); ←  {3, 5, 8, 9, 6, 1, 4, 7, 2}

  std::sort(v.begin(), v.end()); ←  {1, 2, 3, 4, 5, 6, 7, 8, 9}
  std::reverse(v.begin(), v.end()); ←  {9, 8, 7, 6, 5, 4, 3, 2, 1}
}
```

STL Algorithms

```
bool isEven(int x) {
  return ((i % 2) == 0);
}

#include <algorithm>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  std::vector<int>::iterator iter
      = std::find_if(v.begin(), v.end(), isEven);
}
```

STL Algorithms

```
bool isEven(int x) {
  return ((i % 2) == 0);
}

#include <algorithm>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  std::vector<int>::iterator iter
      = std::find_if(v.begin(), v.end(), isEven);
}
```

Exercise

➜ Count the number of occurrence of each name:

Input File

```
Tsinghua
CMU
MIT

Tsinghua
Tsinghua
CMU
```

Output

```
Tsinghua: 3
MIT: 1
CMU: 2
```

Agenda

- → Operator Overloading
- → Standard Template Library (STL)
- → **Type Deduction**
- → Lambdas
- → Move Semantics
- → Smart Pointers

Type Deduction

```
int main() {
  double x = 3.14;
  int a = 1;
  double y = a + x;
  int *p = new int[10];
}
```

➜ Let the compiler deduce the type for us

Type Deduction

```
int main() {
  auto x = 3.14;
  auto a = 1;
  auto y = a + x;
  auto p = new int[10];
}
```

➜ Let the compiler deduce the type for us

Type Deduction

```cpp
int main() {
  auto x = 3.14;
  auto a = 1;
  auto y = a + x;
  auto p = new int[10];
  Vec *p_vec = new Vec(1, 2);
}
```

➜ Let the compiler deduce the type for us

Type Deduction

```
int main() {
  auto x = 3.14;
  auto a = 1;
  auto y = a + x;
  auto p = new int[10];
  auto p_vec = new Vec(1, 2);
}
```

➔ Let the compiler deduce the type for us

Type Deduction

```cpp
int main() {
  auto x = 3.14;
  auto a = 1;
  auto y = a + x;
  auto p = new int[10];
  auto p_vec = new Vec(1, 2);
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  std::vector<int>::iterator iter = std::find(v.begin(), v.end(), 6);
}
```

➜ Let the compiler deduce the type for us

Type Deduction

```
int main() {
  auto x = 3.14;
  auto a = 1;
  auto y = a + x;
  auto p = new int[10];
  auto p_vec = new Vec(1, 2);
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  auto iter = std::find(v.begin(), v.end(), 6);
}
```

➜ Let the compiler deduce the type for us

Type Deduction

```
int main() {
  auto x = 3.14;
  auto a = 1;
  auto y = a + x;
  auto p = new int[10];
  auto p_vec = new Vec(1, 2);
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  auto iter = std::find(v.begin(), v.end(), 6);
}
```

➜ Let the compiler deduce the type for us

- Variable initialization
- Function return type

Type Deduction

```cpp
int main() {
  auto x = 3.14;
  auto a = 1;
  auto y = a + x;
  auto p = new int[10];
  auto p_vec = new Vec(1, 2);
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  auto iter = std::find(v.begin(), v.end(), 6);
  auto var;
}
```

➜ Let the compiler deduce the type for us

- Variable initialization
- Function return type

Type Deduction

```
int main() {
  auto x = 3.14;
  auto a = 1;
  auto y = a + x;
  auto p = new int[10];
  auto p_vec = new Vec(1, 2);
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  auto iter = std::find(v.begin(), v.end(), 6);
  auto var;
  auto s1 = "Tea Garden";
}
```

➜ Let the compiler deduce the type for us

- Variable initialization
- Function return type

Type Deduction

```
int main() {
  auto x = 3.14;
  auto a = 1;
  auto y = a + x;
  auto p = new int[10];
  auto p_vec = new Vec(1, 2);
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  auto iter = std::find(v.begin(), v.end(), 6);
  auto var; ✗
  auto s1 = "Tea Garden"; ←  s1 is (const char *) instead of std::string
}
```

➜ Let the compiler deduce the type for us

- Variable initialization
- Function return type

Type Deduction

```
int main() {
  auto x = 3.14;
  auto a = 1;
  auto y = a + x;
  auto p = new int[10];
  auto p_vec = new Vec(1, 2);
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  auto iter = std::find(v.begin(), v.end(), 6);
  auto var; ❌
  auto s1 = "Tea Garden"; ←
  auto s2 = s1;
}
```

s1 is (`const char *`) instead of `std::string`

➜ Let the compiler deduce the type for us

- Variable initialization
- Function return type

Type Deduction

```
int main() {
  auto x = 3.14;
  auto a = 1;
  auto y = a + x;
  auto p = new int[10];
  auto p_vec = new Vec(1, 2);
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  auto iter = std::find(v.begin(), v.end(), 6);
  auto var;
  auto s1 = "Tea Garden";    s1 is (const char *) instead of std::string
  auto s2 = s1;    s2 is (char *)
}
```

➜ Let the compiler deduce the type for us

- Variable initialization
- Function return type

Type Deduction

```cpp
int main() {
  auto x = 3.14;
  auto a = 1;
  auto y = a + x;
  auto p = new int[10];
  auto p_vec = new Vec(1, 2);
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  auto iter = std::find(v.begin(), v.end(), 6);
  auto var;
  auto s1 = "Tea Garden";
  const auto s2 = s1;
}
```

➜ Let the compiler deduce the type for us

- Variable initialization
- Function return type

← s1 is (`const char *`) instead of `std::string`

Type Deduction

```
int max(int a, int b) {
  return (a > b) ? a : b;
}
```

➜ Let the compiler deduce the type for us

- Variable initialization
- Function return type

Type Deduction

```
auto max(int a, int b) {
  return (a > b) ? a : b;
}
```

➜ Let the compiler deduce the type for us

- Variable initialization
- Function return type

Type Deduction

```
auto max(int a, int b) {
  return (a > b) ? a : b;
}
```

Compile with `-std=c++14`

➜ Let the compiler deduce the type for us

- Variable initialization
- Function return type

Type Deduction

```cpp
auto max(int a, int b) {
  return (a > b) ? a : b;
}

auto func(int a, int b) {
  if (a > b)
    return 1;
  else
    return 2.0;
}
```

➜ Let the compiler deduce the type for us

- Variable initialization
- Function return type

Type Deduction

```cpp
auto max(int a, int b) {
  return (a > b) ? a : b;
}

auto func(int a, int b) {
  if (a > b)
    return 1;
  else
    return 2.0;
}
```

➜ Let the compiler deduce the type for us

- Variable initialization
- Function return type

Best Practice

- Use **auto** for variable initialization whenever it can make code more readable

Type Deduction

```cpp
auto max(int a, int b) {
  return (a > b) ? a : b;
}

auto func(int a, int b) {
  if (a > b)
    return 1;
  else
    return 2.0;
}
```

➜ Let the compiler deduce the type for us

- Variable initialization
- Function return type

Best Practice

- Use **auto** for variable initialization whenever it can make code more readable
- Favor **explicit** return type for normal functions

Type Deduction

```cpp
template <typename T, typename U>
T max(T a, U b) {
  return (a > b) ? a : b;
}
```

```cpp
int main() {
  max(2.5, 3);
}
```

Type Deduction

```
template <typename T, typename U>
T max(T a, U b) {
  return (a > b) ? a : b;
}
```

```
int main() {
  max(2.5, 3);
  max(3, 2.5);
}
```

Type Deduction

```cpp
template <typename T, typename U>
T max(T a, U b) {
  return (a > b) ? a : b;
}

template <typename T, typename U>
U max(T a, U b) {
  return (a > b) ? a : b;
}
```

```cpp
int main() {
  max(2.5, 3);
  max(3, 2.5);
}
```

Type Deduction

```
template <typename T, typename U>
auto max(T a, U b) {
  return (a > b) ? a : b;
}
```

```
int main() {
  max(2.5, 3);
  max(3, 2.5);
}
```

Agenda

- ➔ Operator Overloading
- ➔ Standard Template Library (STL)
- ➔ Type Deduction
- ➔ **Lambdas**
- ➔ Move Semantics
- ➔ Smart Pointers

Lambdas (Anonymous Functions)

```
bool isEven(int x) {
  return ((i % 2) == 0);
}

#include <algorithm>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  std::vector<int>::iterator iter
      = std::find_if(v.begin(), v.end(), isEven);
}
```

Lambdas (Anonymous Functions)

```
bool isEven(int x) {
  return ((i % 2) == 0);
}

#include <algorithm>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  auto iter = std::find_if(v.begin(), v.end(), isEven);
}
```

Lambdas (Anonymous Functions)

```
#include <algorithm>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  auto iter = std::find_if(v.begin(), v.end()
                          [](int i){ return ((i % 2) == 0); });
}
```

Lambdas (Anonymous Functions)

```
#include <algorithm>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  auto iter = std::find_if(v.begin(), v.end()
                               [](int i){ return ((i % 2) == 0); });
}
```

➜ Define an anonymous function inside another function

Lambdas (Anonymous Functions)

```
#include <algorithm>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  auto iter = std::find_if(v.begin(), v.end()
                           [](int i){ return ((i % 2) == 0); });
}
```

➜ Define an anonymous function inside another function

- Avoid namespace pollution
- Define function as close to where it is used

Lambdas (Anonymous Functions)

```
#include <algorithm>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  auto iter = std::find_if(v.begin(), v.end()
                           [](int i){ return ((i % 2) == 0); });
}
```

➜ Define an anonymous function inside another function

- Avoid namespace pollution
- Define function as close to where it is used

➜ **Best Practice**: use lambdas for **trivial** and/or **non-reusable** cases

Defining Lambdas

```
[capture] (arguments) -> return_type {
  statements;
}
```

Defining Lambdas

```
[capture] (arguments) -> return_type {
  statements;
}
```

Assume **auto** if unspecified

Lambda Captures

```cpp
#include <algorithm>
#include <vector>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  int num_cmp = 0;
  std::sort(v.begin(), v.end(),
            [](const int &a, const int &b) {
              num_cmp++;
              return (std::abs(a) > std::abs(b));
            });
  std::cout << num_cmp << std::endl;
}
```

Lambda Captures

```
#include <algorithm>
#include <vector>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  int num_cmp = 0;
  std::sort(v.begin(), v.end(),
            [](const int &a, const int &b) {
              num_cmp++;
              return (std::abs(a) > std::abs(b));
            });
  std::cout << num_cmp << std::endl;
}
```

Lambda Captures

```
#include <algorithm>
#include <vector>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  int num_cmp = 0;
  std::sort(v.begin(), v.end(),
            [](const int &a, const int &b) {
              num_cmp++;
              return (std::abs(a) > std::abs(b));
            });
  std::cout << num_cmp << std::endl;
}
```

Not allowed to access variables in the outer scope!

Lambda Captures

```
#include <algorithm>
#include <vector>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  int num_cmp = 0;
  std::sort(v.begin(), v.end(),
            [num_cmp](const int &a, const int &b) {
              num_cmp++;
              return (std::abs(a) > std::abs(b));
            });
  std::cout << num_cmp << std::endl;
}
```

Lambda Under the Hood

```
[capture] (arguments) -> return_type {
  statements;
}
```

Lambda Under the Hood

```
[capture] (arguments) -> return_type {
  statements;
}
```

➜ Creates an object, called **functors**,

Lambda Under the Hood

```
[capture] (arguments) -> return_type {
  statements;
}
```

- Creates an object, called **functors**,
- Overloads `operator()`
  - so that if can be called like functions

Lambda Under the Hood

```
[capture] (arguments) -> return_type {
  statements;
}
```

- ➔ Creates an object, called **functors**,
- ➔ Overloads `operator()`
  - so that if can be called like functions
- ➔ What's in the `[capture]` are **data members** of the object

Lambda Under the Hood

```
[capture] (arguments) -> return_type {
  statements;
}
```

```
public:
return_type operator()(arguments) {
  statements;
}

private:
captured_variables
```

Lambda Captures

```
#include <algorithm>
#include <vector>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  int num_cmp = 0;
  std::sort(v.begin(), v.end(),
            [num_cmp](const int &a, const int &b) {
              num_cmp++;
              return (std::abs(a) > std::abs(b));
            });
  std::cout << num_cmp << std::endl;
}
```

Lambda Captures

```
#include <algorithm>
#include <vector>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  int num_cmp = 0;
  std::sort(v.begin(), v.end(),
            [num_cmp](const int &a, const int &b) {
              num_cmp++;
              return (std::abs(a) > std::abs(b));
            });
  std::cout << num_cmp << std::endl;
}
```

Lambda Captures

```
#include <algorithm>
#include <vector>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  int num_cmp = 0;
  std::sort(v.begin(), v.end(),
            [num_cmp](const int &a, const int &b) {
              num_cmp++;
              return (std::abs(a) > std::abs(b));
            });
  std::cout << num_cmp << std::endl;
}
```

```
const int num_cmp;
```

Lambda Captures

```
#include <algorithm>
#include <vector>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  int num_cmp = 0;
  std::sort(v.begin(), v.end(),
            [num_cmp](const int &a, const int &b) {
              num_cmp++;
              return (std::abs(a) > std::abs(b));
            });
  std::cout << num_cmp << std::endl;
}
```

Constant by default

```
const int num_cmp;
```

Lambda Captures

```
#include <algorithm>
#include <vector>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  int num_cmp = 0;
  std::sort(v.begin(), v.end(),
            [num_cmp](const int &a, const int &b) mutable
              num_cmp++;
              return (std::abs(a) > std::abs(b));
            });
  std::cout << num_cmp << std::endl;
}
```

```
int num_cmp;
```

Lambda Captures

```
#include <algorithm>
#include <vector>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  int num_cmp = 0;
  std::sort(v.begin(), v.end(),
            [num_cmp](const int &a, const int &b) mutable
              num_cmp++;
              return (std::abs(a) > std::abs(b));
            });
  std::cout << num_cmp << std::endl;
}
```

```
Functor(int arg1) {
  num_cmp = arg1;
}



int num_cmp;
```

Lambda Captures

```
#include <algorithm>
#include <vector>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  int num_cmp = 0;
  std::sort(v.begin(), v.end(),
            [num_cmp](const int &a, const int &b) mutable
              num_cmp++;
              return (std::abs(a) > std::abs(b));
            });
  std::cout << num_cmp << std::endl;
}
```

```
Functor(int arg1) {
  num_cmp = arg1;
}
auto operator()(...) {
  num_cmp++;
  ...
}
int num_cmp;
```

Lambda Captures

```
#include <algorithm>
#include <vector>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  int num_cmp = 0;
  std::sort(v.begin(), v.end(),
            [num_cmp](const int &a, const int &b) mutable
              num_cmp++;
              return (std::abs(a) > std::abs(b));
            });
  std::cout << num_cmp << std::endl;
}
```

Captured by **value**

```
Functor(int arg1) {
  num_cmp = arg1;
}
auto operator()(...) {
  num_cmp++;
  ...
}
int num_cmp;
```

Lambda Captures

```
#include <algorithm>
#include <vector>
int main() {
  std::vector<int> v = {3, 5, 8, 6, 1, 4, 7, 2};
  int num_cmp = 0;
  std::sort(v.begin(), v.end(),
            [&num_cmp](const int &a, const int &b) {
              num_cmp++;
              return (std::abs(a) > std::abs(b));
            });
  std::cout << num_cmp << std::endl;
}
```

Captured by **reference**

```
Functor(int arg1) {
  num_cmp = arg1;
}
auto operator()(...) {
  num_cmp++;
  ...
}
int num_cmp;
```

Agenda

- Operator Overloading
- Standard Template Library (STL)
- Type Deduction
- Lambdas
- **Move Semantics**
- Smart Pointers

rvalue Reference

```
int &r = 666;
const int &r = 666;
```

rvalue Reference

```
int &r = 666;
```

```
const int &r = 666;
```

But you can't alter the value

rvalue Reference

```
int &r = 666;
```

```
const int &r = 666;
```

But you can't alter the value

```
int &&r = 666;
```

rvalue Reference

```
int &r = 666;
```

```
const int &r = 666;
```

But you can't alter the value

```
int &&r = 666;
```

rvalue Reference

```
int &r = 666;
```

```
const int &r = 666;
```

But you can't alter the value

```
int &&r = 666;

r += 888;
```

Move Semantics

➜ Transfer resource ownership (or move resource around) without unnecessary copies of temporary objects

Move Semantics

```
class Data {
 public:
  Data(int size) : size_(size) {
    data_ = new int[size];
  }
  ~Data() { delete[] data_; }

 private:
  int size_;
  int *data_;
};
```

Move Semantics

```
class Data {
 public:
  Data(int size) : size_(size) {
    data_ = new int[size];
  }
  ~Data() { delete[] data_; }

 private:
  int size_;
  int *data_;
};
```

```
int main() {
  Data d1(10000);
  Data d2(20000);

}
```

Move Semantics

```
class Data {
 public:
  Data(int size) : size_(size) {
    data_ = new int[size];
  }
  ~Data() { delete[] data_; }

 private:
  int size_;
  int *data_;
};
```

```
int main() {
  Data d1(10000);
  Data d2(20000);
  Data d3(d1);
}
```

Move Semantics

```
// copy constructor
Data::Data(const Data &other) {
  size_ = other.size_;
  data_ = new int[size_];
  std::memcpy(data_, other.data_, size_);
}
```

```
int main() {
  Data d1(10000);
  Data d2(20000);
  Data d3(d1);
}
```

Move Semantics

```
// copy constructor
Data::Data(const Data &other) {
  size_ = other.size_;
  data_ = new int[size_];
  std::memcpy(data_, other.data_, size_);
}
```

```
int main() {
  Data d1(10000);
  Data d2(20000);
  Data d3(d1);
  d3 = d2;
}
```

Move Semantics

```
// overloading assignment
Data &Data::operator=(const Data &other) {




}
```

```
int main() {
  Data d1(10000);
  Data d2(20000);
  Data d3(d1);
  d3 = d2;
}
```

Move Semantics

```
// overloading assignment
Data &Data::operator=(const Data &other) {
  if (this == &other) return *this;
  delete[] data_;
  size_ = other.size_;
  data_ = new int[size_];
  std::memcpy(data_, other.data_, size_);
  return *this;
}
```

```
int main() {
  Data d1(10000);
  Data d2(20000);
  Data d3(d1);
  d3 = d2;
}
```

Move Semantics

```
Data CreateData(int size) {
  return Data(size);
}

int main() {
  Data d1(10000);
  Data d2(d1);
  Data d3 = CreateData(20000);
}
```

```
// copy constructor
Data::Data(const Data &other) {
  size_ = other.size_;
  data_ = new int[size_];
  std::memcpy(data_, other.data_, size_);
}
```

Move Semantics

```
Data CreateData(int size) {
  return Data(size);
}

int main() {
  Data d1(10000);
  Data d2(d1);
  Data d3 = CreateData(20000);
}
```

```
// copy constructor
Data::Data(const Data &other) {
  size_ = other.size_;
  data_ = new int[size_];
  std::memcpy(data_, other.data_, size_);
}

// move constructor
Data::Data(Data &&other) {
  size_ = other.size_;
  data_ = other.data_;
  other.size_ = 0;
  other.data_ = nullptr;
}
```

Move Semantics

```
Data CreateData(int size) {
  return Data(size);
}

int main() {
  Data d1(10000);
  Data d2(d1);
  Data d3 = CreateData(20000);
}
```

1

```
// copy constructor
Data::Data(const Data &other) {
  size_ = other.size_;
  data_ = new int[size_];
  std::memcpy(data_, other.data_, size_);
}
```

2

```
// move constructor
Data::Data(Data &&other) {
  size_ = other.size_;
  data_ = other.data_;
  other.size_ = 0;
  other.data_ = nullptr;
}
```

Move Semantics

```
Data CreateData(int size) {
  return Data(size);
}

int main() {
  Data d1(10000);
1 Data d2(d1);
  Data d3 = CreateData(20000);
}
```

```
1 // copy constructor
Data::Data(const Data &other) {
  size_ = other.size_;
  data_ = new int[size_];
  std::memcpy(data_, other.data_, size_);
}

2 // move constructor
Data::Data(Data &&other) {
  size_ = other.size_;
  data_ = other.data_;
  other.size_ = 0;
  other.data_ = nullptr;
}
```

Move Semantics

```
Data CreateData(int size) {
  return Data(size);
}

int main() {
  Data d1(10000);
1 Data d2(d1);
2 Data d3 = CreateData(20000);
}
```

```
1 // copy constructor
Data::Data(const Data &other) {
  size_ = other.size_;
  data_ = new int[size_];
  std::memcpy(data_, other.data_, size_);
}

2 // move constructor
Data::Data(Data &&other) {
  size_ = other.size_;
  data_ = other.data_;
  other.size_ = 0;
  other.data_ = nullptr;
}
```

Move Semantics

```
Data CreateData(int size) {
  return Data(size);
}

int main() {
  Data d1(10000);
1 Data d2((Data &&)d1);
2 Data d3 = CreateData(20000);
}
```

```
1 // copy constructor
Data::Data(const Data &other) {
  size_ = other.size_;
  data_ = new int[size_];
  std::memcpy(data_, other.data_, size_);
}

2 // move constructor
Data::Data(Data &&other) {
  size_ = other.size_;
  data_ = other.data_;
  other.size_ = 0;
  other.data_ = nullptr;
}
```

Move Semantics

```
Data CreateData(int size) {
  return Data(size);
}

int main() {
  Data d1(10000);
2 Data d2((Data &&)d1);
2 Data d3 = CreateData(20000);
}
```

```
1 // copy constructor
Data::Data(const Data &other) {
  size_ = other.size_;
  data_ = new int[size_];
  std::memcpy(data_, other.data_, size_);
}
```

```
2 // move constructor
Data::Data(Data &&other) {
  size_ = other.size_;
  data_ = other.data_;
  other.size_ = 0;
  other.data_ = nullptr;
}
```

Move Semantics

```
Data CreateData(int size) {
  return Data(size);
}

int main() {
  Data d1(10000);
  Data d2(std::move(d1));
  Data d3 = CreateData(20000);
}
```

1

```
// copy constructor
Data::Data(const Data &other) {
  size_ = other.size_;
  data_ = new int[size_];
  std::memcpy(data_, other.data_, size_);
}
```

2

```
// move constructor
Data::Data(Data &&other) {
  size_ = other.size_;
  data_ = other.data_;
  other.size_ = 0;
  other.data_ = nullptr;
}
```

Move Semantics

```
Data CreateData(int size) {
  return Data(size);
}

int main() {
  Data d1(10000);
  Data d2(std::move(d1));
  Data d3 = CreateData(20000);
  d2 = d3;
}
```

```
// assignment operator
Data &Data::operator=(const Data &other) {
  if (this == &other) return *this;
  delete[] data_;
  size_ = other.size_;
  data_ = new int[size_];
  std::memcpy(data_, other.data_, size_);
  return *this;
}
```

Move Semantics

```
Data CreateData(int size) {
  return Data(size);
}

int main() {
  Data d1(10000);
  Data d2(std::move(d1));
  Data d3 = CreateData(20000);
  d2 = d3;
  d2 = CreateData(30000)
}
```

```
// assignment operator
Data &Data::operator=(const Data &other) {
  if (this == &other) return *this;
  delete[] data_;
  size_ = other.size_;
  data_ = new int[size_];
  std::memcpy(data_, other.data_, size_);
  return *this;
}
```

Move Semantics

```
Data CreateData(int size) {
  return Data(size);
}

int main() {
  Data d1(10000);
  Data d2(std::move(d1));
  Data d3 = CreateData(20000);
  d2 = d3;
  d2 = CreateData(30000)
}
```

```
// move assignment operator
Data &Data::operator=(Data &&other) {
  if (this == &other) return *this;
  delete[] data_;
  size_ = other.size_;
  data_ = other.data_;
  other.size_ = 0;
  other.data_ = nullptr;
  return *this;
}
```

Move Semantics

```
class Entity {
 public:
  Entity(const Data &payload)
      : payload_(payload) {}

 private:
  Data payload_;
};
```

```
int main() {
  Entity e1(Data(10000));
}
```

Move Semantics

```
class Entity {
 public:
  Entity(const Data &payload)
      : payload_(payload) {}

 private:
  Data payload_;
};
```

```
int main() {
  Entity e1(Data(10000));
}
```

```
// copy constructor
Data::Data(const Data &other) {
  size_ = other.size_;
  data_ = new int[size_];
  std::memcpy(data_, other.data_, size_);
}
```

Move Semantics

```cpp
class Entity {
 public:
  Entity(const Data &payload)
      : payload_(payload) {}

  Entity(Data &&payload)
      : payload_(payload) {}

 private:
  Data payload_;
};
```

```cpp
int main() {
  Entity e1(Data(10000));
}


// move constructor
Data::Data(Data &&other) {
  size_ = other.size_;
  data_ = other.data_;
  other.size_ = 0;
  other.data_ = nullptr;
}
```

Move Semantics

```
class Entity {
 public:
  Entity(const Data &payload)
      : payload_(payload) {}

  Entity(Data &&payload)
      : payload_(payload) {}
```

lvalue

```
 private:
  Data payload_;
};
```

```
int main() {
  Entity e1(Data(10000));
}
```

```
// move constructor
Data::Data(Data &&other) {
  size_ = other.size_;
  data_ = other.data_;
  other.size_ = 0;
  other.data_ = nullptr;
}
```

Move Semantics

```cpp
class Entity {
 public:
  Entity(const Data &payload)
      : payload_(payload) {}

  Entity(Data &&payload)
      : payload_(std::move(payload)) {}

 private:
  Data payload_;
};
```

```cpp
int main() {
  Entity e1(Data(10000));
}
```

```cpp
// move constructor
Data::Data(Data &&other) {
  size_ = other.size_;
  data_ = other.data_;
  other.size_ = 0;
  other.data_ = nullptr;
}
```

Move Semantics

```
std::vector<std::string> a;
std::string str { “bye” };

a.push_back(str);
std::cout << a[0] << “\n”;
std::cout << str << “\n”;

a.push_back(std::move(str));
std::cout << a[0] << a[1] << “\n”;
std::cout << str << “\n”;
```

Move Semantics

```
std::vector<std::string> a;
std::string str { “bye” };

a.push_back(str);
std::cout << a[0] << “\n”;            ← bye
std::cout << str << “\n”;

a.push_back(std::move(str));
std::cout << a[0] << a[1] << “\n”;
std::cout << str << “\n”;
```

Move Semantics

```
std::vector<std::string> a;
std::string str { “bye” };

a.push_back(str);
std::cout << a[0] << “\n”;
std::cout << str << “\n”;

a.push_back(std::move(str));
std::cout << a[0] << a[1] << “\n”;
std::cout << str << “\n”;
```

bye

bye

Move Semantics

```
std::vector<std::string> a;
std::string str { “bye” };

a.push_back(str);
std::cout << a[0] << “\n”;           ← bye
std::cout << str << “\n”;            ← bye

a.push_back(std::move(str));
std::cout << a[0] << a[1] << “\n”;   ← byebye
std::cout << str << “\n”;
```

Move Semantics

```
std::vector<std::string> a;
std::string str { “bye” };

a.push_back(str);
std::cout << a[0] << “\n”;              ⟵  bye
std::cout << str << “\n”;               ⟵  bye

a.push_back(std::move(str));
std::cout << a[0] << a[1] << “\n”;      ⟵  byebye
std::cout << str << “\n”;               ⟵  Unspecified State
```

Agenda

- ➔ Operator Overloading
- ➔ Standard Template Library (STL)
- ➔ Type Deduction
- ➔ Lambdas
- ➔ Move Semantics
- ➔ **Smart Pointers**

Smart Pointers

```
void Func(int x) {
  int *ptr = new int[10000];
  ...
  delete[] ptr;
}
```

Smart Pointers

```
void Func(int x) {
  int *ptr = new int[10000];
  ...
  if (x == 0) return;
  ...
  if (x < 0) return;
  ...
  delete[] ptr;
}
```

Smart Pointers

```
void Func(int x) {
  int *ptr = new int[10000];
  ...
  if (x == 0) return;
  ...
  if (x < 0) return;
  ...
  delete[] ptr;
}
```

```
template <typename T>
class Pointer {
 public:
  Pointer(T *ptr) : ptr_(ptr) {}
  ~Pointer() { delete ptr_; }
  T &operator*() const { return *ptr_; }
  T *operator->() const { return ptr_; }

 private:
  T *ptr_;
}
```

Smart Pointers

```
void Func(int x) {
  Pointer<int> ptr(new int[10000]);
  ...
  if (x == 0) return;
  ...
  if (x < 0) return;
  ...
  delete[] ptr;
}
```

```
template <typename T>
class Pointer {
 public:
  Pointer(T *ptr) : ptr_(ptr) {}
  ~Pointer() { delete ptr_; }
  T &operator*() const { return *ptr_; }
  T *operator->() const { return ptr_; }

 private:
  T *ptr_;
}
```

Smart Pointers

```
void Func(int x) {
  Pointer<int> ptr(new int[10000]);
  ...
  if (x == 0) return;
  ...
  if (x < 0) return;
  ...

}
```

```
template <typename T>
class Pointer {
 public:
  Pointer(T *ptr) : ptr_(ptr) {}
  ~Pointer() { delete ptr_; }
  T &operator*() const { return *ptr_; }
  T *operator->() const { return ptr_; }

 private:
  T *ptr_;
}
```

Smart Pointers

```
void Func(int x) {
  Pointer<int> ptr(new int[10000]);
  ...
  if (x == 0) return;
  ...
  if (x < 0) return;
  ...

}
```

```
template <typename T>
class Pointer {
 public:
  Pointer(T *ptr) : ptr_(ptr) {}
  ~Pointer() { delete ptr_; }
  T &operator*() const { return *ptr_; }
  T *operator->() const { return ptr_; }

 private:
  T *ptr_;
}
```

Resource Acquisition Is Initialization (RAII)

Smart Pointers

```
void Func(int x) {
  Pointer<int> ptr(new int[10000]);
  Pointer<int> ptr2 = ptr;
  ...
}
```

```
template <typename T>
class Pointer {
 public:
  Pointer(T *ptr) : ptr_(ptr) {}
  ~Pointer() { delete ptr_; }
  T &operator*() const { return *ptr_; }
  T *operator->() const { return ptr_; }

 private:
  T *ptr_;
}
```

Smart Pointers

```
void Func(int x) {
  Pointer<int> ptr(new int[10000]);
  Pointer<int> ptr2 = ptr;
  ...
}
```

Double Free

```
template <typename T>
class Pointer {
 public:
  Pointer(T *ptr) : ptr_(ptr) {}
  ~Pointer() { delete ptr_; }
  T &operator*() const { return *ptr_; }
  T *operator->() const { return ptr_; }

 private:
  T *ptr_;
}
```

Smart Pointers

```
void Func(int x) {
  Pointer<int> ptr(new int[10000]);
  Pointer<int> ptr2 = ptr;
  ...
}
```

Double Free

1. Disable copy semantics, only allow move semantics

```
template <typename T>
class Pointer {
 public:
  Pointer(T *ptr) : ptr_(ptr) {}
  ~Pointer() { delete ptr_; }
  T &operator*() const { return *ptr_; }
  T *operator->() const { return ptr_; }

 private:
  T *ptr_;
}
```

Smart Pointers

```
void Func(int x) {
  Pointer<int> ptr(new int[10000]);
  Pointer<int> ptr2 = ptr;
  ...
}
```

Double Free

1. Disable copy semantics, only allow move semantics

   `std::unique_ptr`

```
template <typename T>
class Pointer {
 public:
  Pointer(T *ptr) : ptr_(ptr) {}
  ~Pointer() { delete ptr_; }
  T &operator*() const { return *ptr_; }
  T *operator->() const { return ptr_; }

 private:
  T *ptr_;
}
```

Smart Pointers

```
void Func(int x) {
  Pointer<int> ptr(new int[10000]);
  Pointer<int> ptr2 = ptr;
  ...
}
```

Double Free

1. Disable copy semantics, only allow move semantics

   `std::unique_ptr`

2. Add reference counter

```
template <typename T>
class Pointer {
 public:
  Pointer(T *ptr) : ptr_(ptr) {}
  ~Pointer() { delete ptr_; }
  T &operator*() const { return *ptr_; }
  T *operator->() const { return ptr_; }

 private:
  T *ptr_;
}
```

Smart Pointers

```
void Func(int x) {
  Pointer<int> ptr(new int[10000]);
  Pointer<int> ptr2 = ptr;
  ...
}
```

Double Free

1. Disable copy semantics, only allow move semantics

   `std::unique_ptr`

2. Add reference counter

   `std::shared_ptr`

```
template <typename T>
class Pointer {
 public:
  Pointer(T *ptr) : ptr_(ptr) {}
  ~Pointer() { delete ptr_; }
  T &operator*() const { return *ptr_; }
  T *operator->() const { return ptr_; }

 private:
  T *ptr_;
}
```

std::unique_ptr

```
#include <memory>
#include <utility>

int main() {
  std::unique_ptr<int> ptr(new int[1000]);



}
```

std::unique_ptr

```
#include <memory>
#include <utility>

int main() {
  std::unique_ptr<int> ptr(new int[1000]);
  std::unique_ptr<Data> d1_ptr(new Data(10000));


}
```

std::unique_ptr

```
#include <memory>
#include <utility>

int main() {
  std::unique_ptr<int> ptr(new int[1000]);
  std::unique_ptr<Data> d1_ptr(new Data(10000));
  std::unique_ptr<Data> d2_ptr;

}
```

std::unique_ptr

```cpp
#include <memory>
#include <utility>

int main() {
  std::unique_ptr<int> ptr(new int[1000]);
  std::unique_ptr<Data> d1_ptr(new Data(10000));
  std::unique_ptr<Data> d2_ptr;
  d2_ptr = d1_ptr;
}
```

std::unique_ptr

```
#include <memory>
#include <utility>

int main() {
  std::unique_ptr<int> ptr(new int[1000]);
  std::unique_ptr<Data> d1_ptr(new Data(10000));
  std::unique_ptr<Data> d2_ptr;
  d2_ptr = d1_ptr;
}
```

std::unique_ptr

```
#include <memory>
#include <utility>

int main() {
  std::unique_ptr<int> ptr(new int[1000]);
  std::unique_ptr<Data> d1_ptr(new Data(10000));
  std::unique_ptr<Data> d2_ptr;
  d2_ptr = std::move(d1_ptr);
}
```

std::unique_ptr

```
#include <memory>
#include <utility>

int main() {
  std::unique_ptr<int> ptr(new int[1000]);
  std::unique_ptr<Data> d1_ptr(new Data(10000));
  std::unique_ptr<Data> d2_ptr;
  d2_ptr = std::move(d1_ptr);   ←  Ownership transferred
}
```

std::unique_ptr

```
#include <memory>
#include <utility>

int main() {
  Data *d = new Data(10000);
  std::unique_ptr<Data> d1_ptr(d);
  std::unique_ptr<Data> d2_ptr(d);
}
```

std::unique_ptr

```
#include <memory>
#include <utility>

int main() {
  Data *d = new Data(10000);
  std::unique_ptr<Data> d1_ptr(d);
  std::unique_ptr<Data> d2_ptr(d);
}
```

std::unique_ptr

```cpp
#include <memory>
#include <utility>

int main() {
  Data *d = new Data(10000);
  std::unique_ptr<Data> d1_ptr(d);
  delete d;
}
```

std::unique_ptr

```
#include <memory>
#include <utility>

int main() {
  Data *d = new Data(10000);
  std::unique_ptr<Data> d1_ptr(d);
  delete d;
}
```

std::unique_ptr

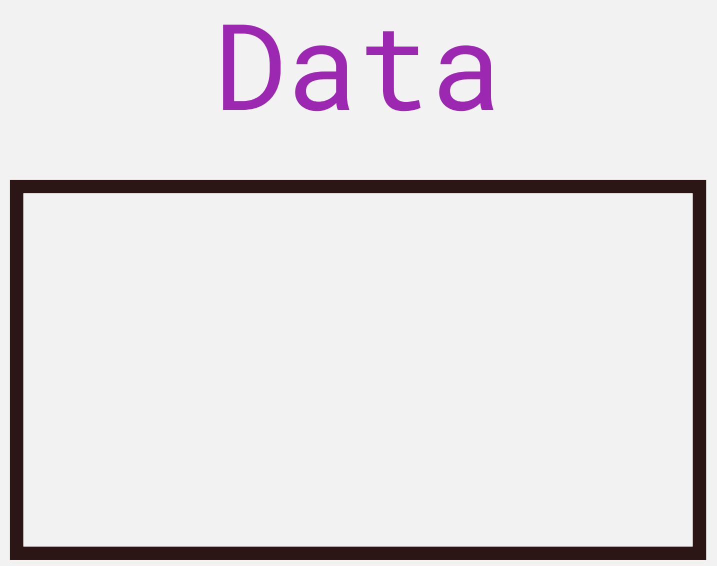
```
#include <memory>
#include <utility>

int main() {
  auto d1_ptr(std::make_unique<Data>(10000));
  ...
}
```

std::shared_ptr

```
#include <memory>

int main() {
  Data *d = new Data(100);
  std::shared_ptr<Data> d1_ptr(d);
  {
    std::shared_ptr<Data> d2_ptr(d1_ptr);
  }
  std::cout << d1_ptr->GetSize() << “\n”;
}
```

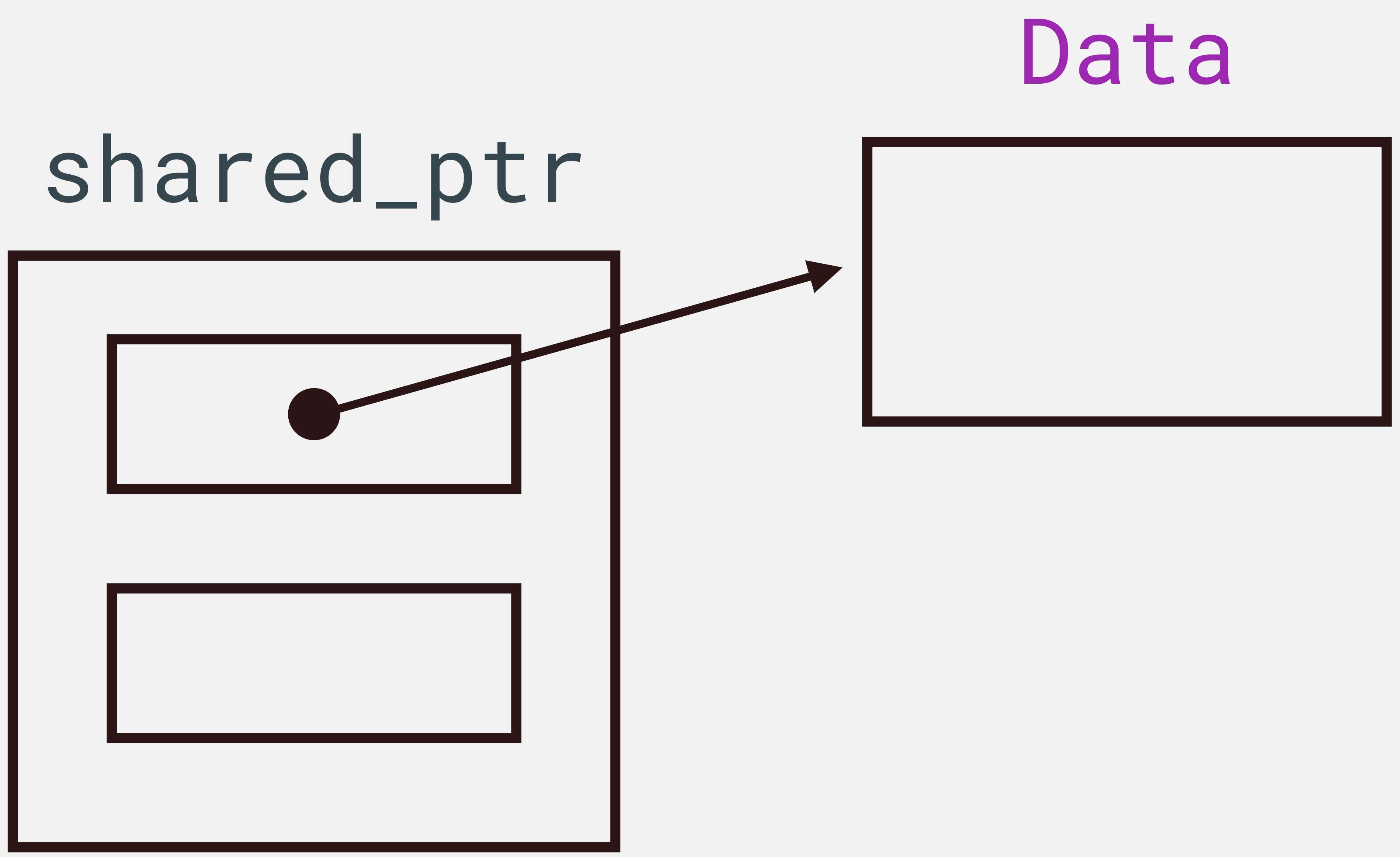
std::shared_ptr

```cpp
#include <memory>

int main() {
  Data *d = new Data(100);
  std::shared_ptr<Data> d1_ptr(d);
  {
    std::shared_ptr<Data> d2_ptr(d1_ptr);
  }
  std::cout << d1_ptr->GetSize() << “\n”;
}
```



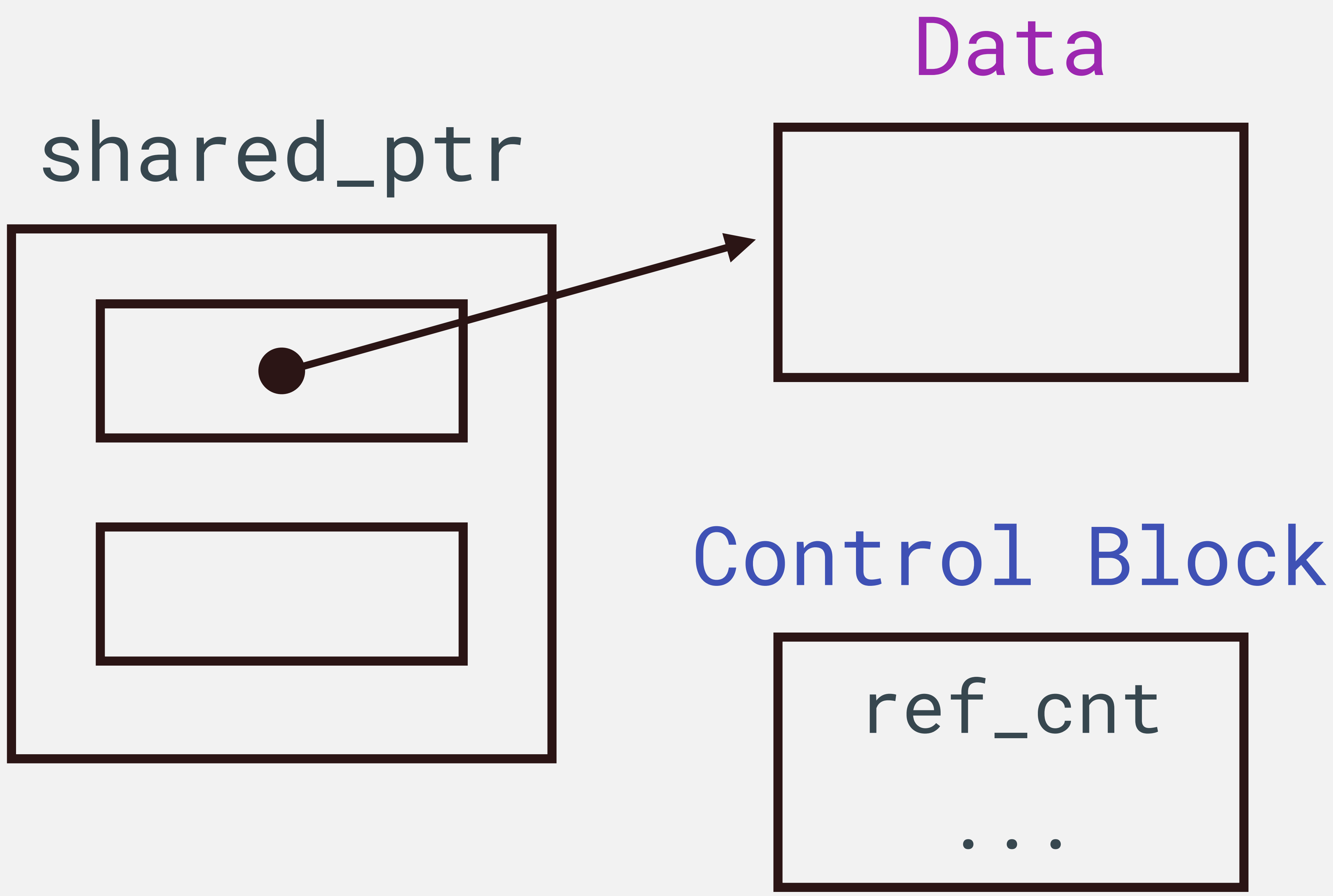
std::shared_ptr

```
#include <memory>

int main() {
  Data *d = new Data(100);
  std::shared_ptr<Data> d1_ptr(d);
  {
    std::shared_ptr<Data> d2_ptr(d1_ptr);
  }
  std::cout << d1_ptr->GetSize() << “\n”;
}
```



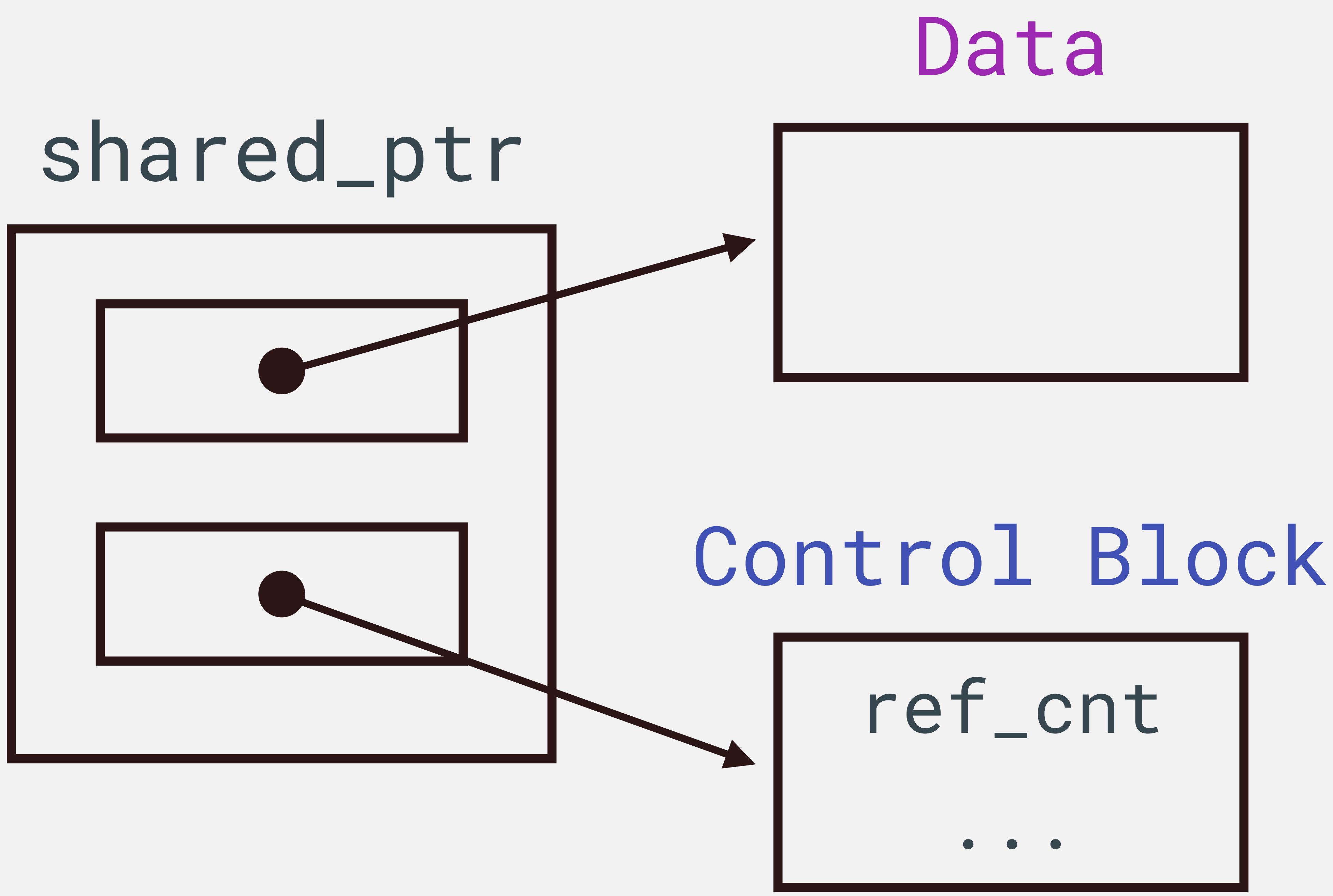
std::shared_ptr

```cpp
#include <memory>

int main() {
  Data *d = new Data(100);
  std::shared_ptr<Data> d1_ptr(d);
  {
    std::shared_ptr<Data> d2_ptr(d1_ptr);
  }
  std::cout << d1_ptr->GetSize() << “\n”;
}
```



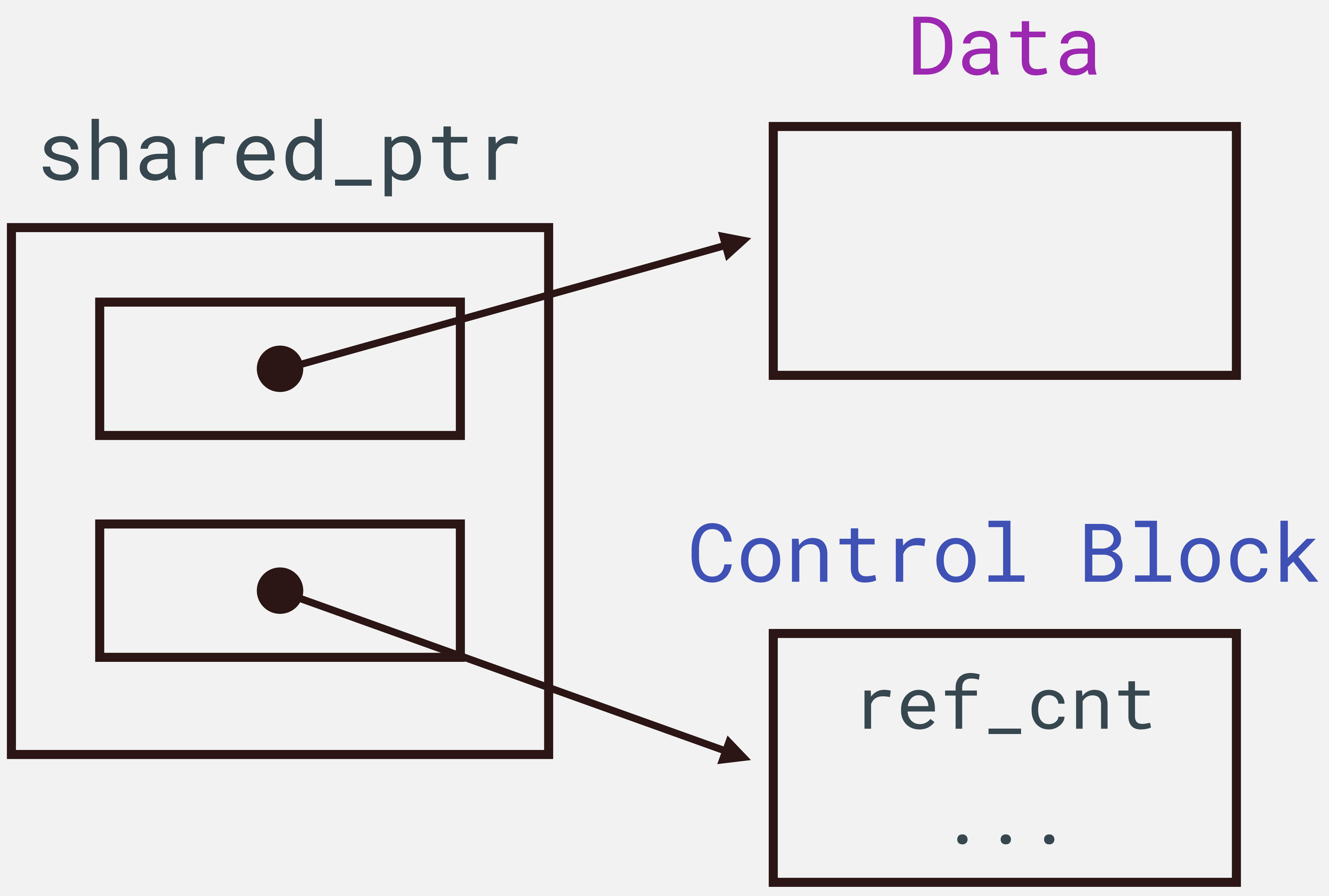
std::shared_ptr

```
#include <memory>

int main() {
  Data *d = new Data(100);
  std::shared_ptr<Data> d1_ptr(d);
  {
    std::shared_ptr<Data> d2_ptr(d1_ptr);
  }
  std::cout << d1_ptr->GetSize() << “\n”;
}
```



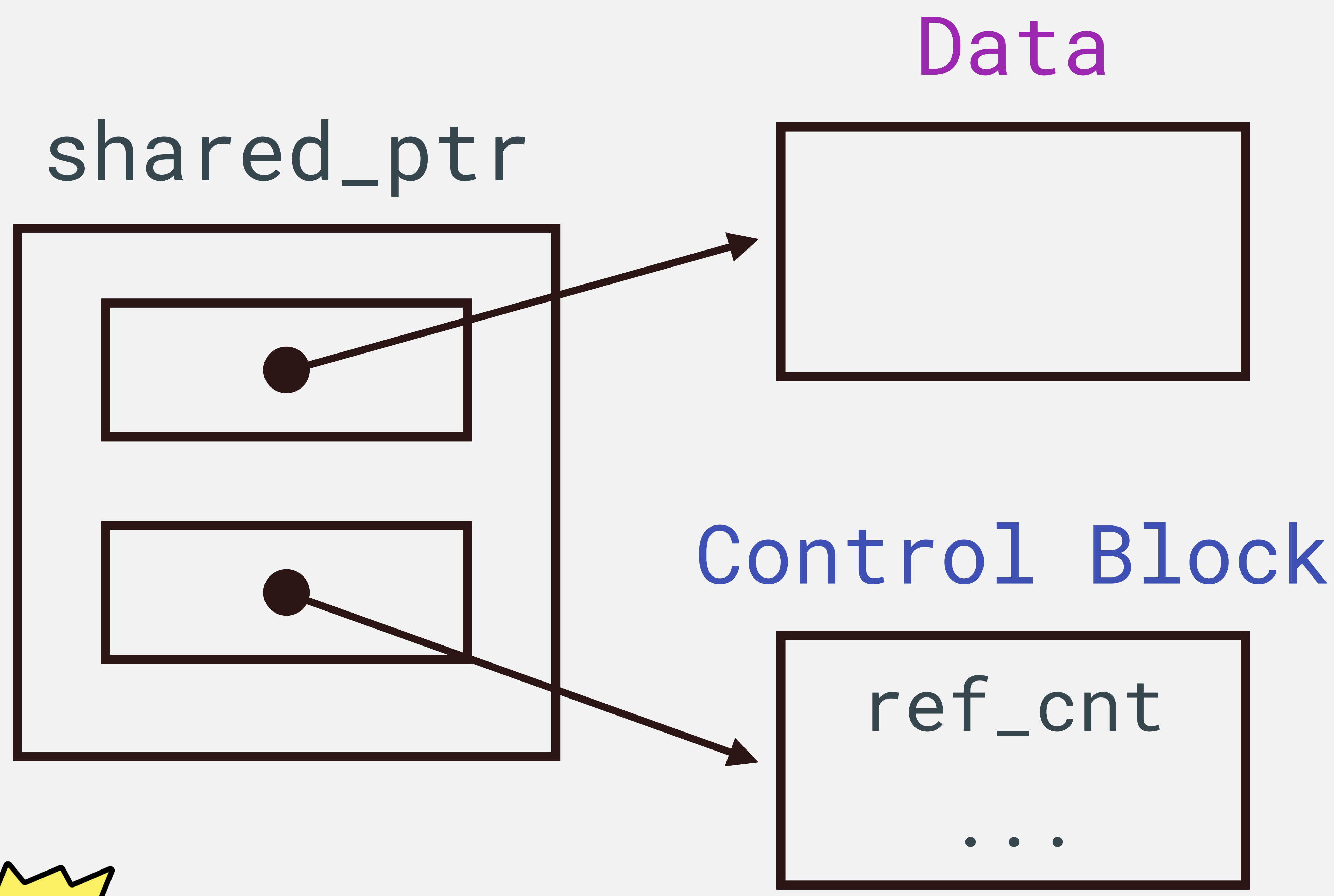
std::shared_ptr

```
#include <memory>

int main() {
  Data *d = new Data(100);
  std::shared_ptr<Data> d1_ptr(d);
  {
    std::shared_ptr<Data> d2_ptr(d);
  }
  std::cout << d1_ptr->GetSize() << "\n";
}
```



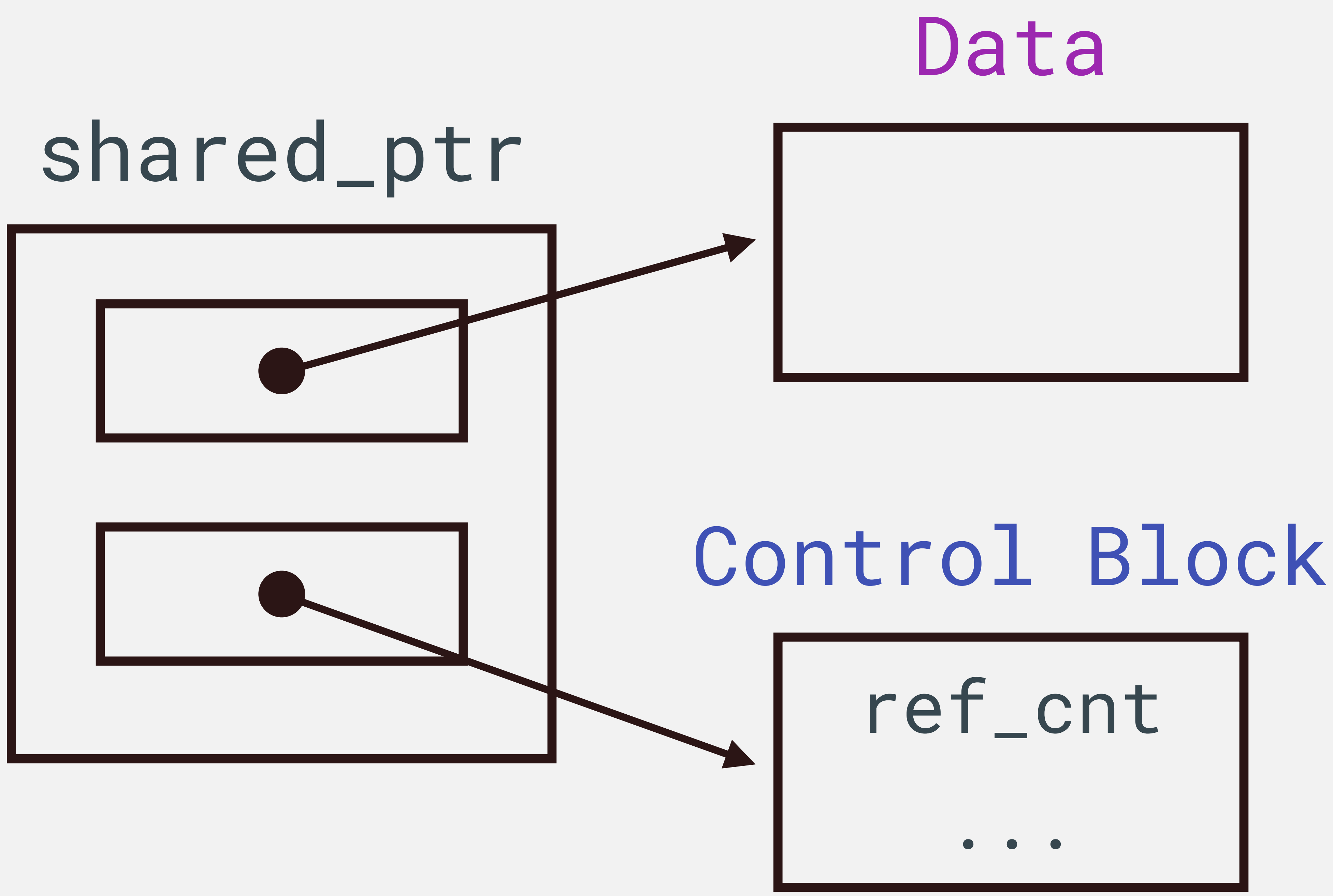
std::shared_ptr

```
#include <memory>

int main() {
  Data *d = new Data(100);
  std::shared_ptr<Data> d1_ptr(d);
  {
    std::shared_ptr<Data> d2_ptr(d);
  }
  std::cout << d1_ptr->GetSize() << "\n";
}
```



std::shared_ptr

```
#include <memory>

int main() {
  auto d1_ptr(std::make_shared<Data>(100));
  {
    auto d2_ptr(d1_ptr);
  }
  std::cout << d1_ptr->GetSize() << "\n";
}
```



Recap

- ➜ Operator Overloading
- ➜ Standard Template Library (STL)
- ➜ Type Deduction
- ➜ Lambdas
- ➜ Move Semantics
- ➜ Smart Pointers