CSE 333
Section 7

Templates, STL, and Inheritance

UNIVERSITY of WASHINGTON

Inheritance

- **Derived** class inherits from the **base** class
  - In 333, we always use *public* inheritance
  - Inherits all *non-private* member variables
  - Inherits all *non-private* member functions
    except for ctor, cctor, dtor, op=
- Access specifiers revisited:
  - **Private** members cannot be accessed by derived classes
  - **Protected** members are available to base & derived

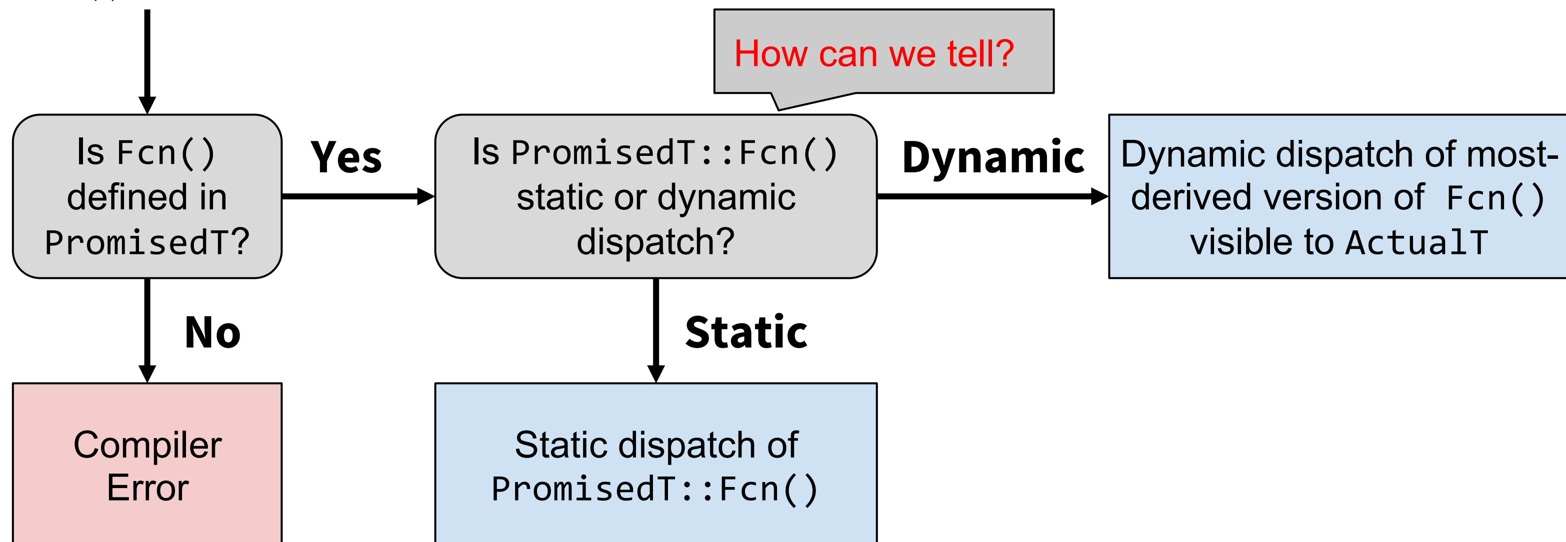
Static vs. Dynamic Dispatch

How to resolve invoking a method via a polymorphic pointer:

1. Static dispatch
   - Default behavior in C++
   - More to come in Friday's lecture!
2. Dynamic dispatch
   - Which implementation is determined *at runtime* via lookup
   - Compiler generates code that accesses function pointers added to the class

Dispatch Decision Tree

```
PromisedT *ptr = new ActualT();
ptr->Fcn();  // which version is called?
```



Dispatch Keywords

- `virtual` – request dynamic dispatch
  - Is "sticky": overridden virtual method in derived class is still virtual with or without the keyword
- `override` – ensures that the function is virtual and is overriding a virtual function from a base class (`@override` in Java)
  - Generates a compiler error if conditions are not met
  - Catches overloading vs. overriding bugs at compile time

Practice: static, dynamic, or error?

```
class Base {
  void Foo();             //
  void Bar();             //
  virtual void Baz();     //
};

class Derived : public Base {
  virtual void Foo();     //
  void Bar() override;    //
  void Baz();             //
};
```

Practice: static, dynamic, or error?

```
class Base {
  void Foo();             // static dispatch
  void Bar();             // static dispatch
  virtual void Baz();     // dynamic dispatch
};

class Derived : public Base {
  virtual void Foo();     //
  void Bar() override;    //
  void Baz();             //
};
```

Practice: static, dynamic, or error?

```
class Base {
  void Foo();              // static dispatch
  void Bar();              // static dispatch
  virtual void Baz();      // dynamic dispatch
};

class Derived : public Base {
  virtual void Foo();      // now dynamic (for more derived)
  void Bar() override;     //
  void Baz();              //
};
```

Practice: static, dynamic, or error?

```
class Base {
  void Foo();                // static dispatch
  void Bar();                // static dispatch
  virtual void Baz();        // dynamic dispatch
};

class Derived : public Base {
  virtual void Foo();        // now dynamic (for more derived)
  //void Bar() override;// compiler error
  void Bar();                // static dispatch
  void Baz();                //
};
```

Practice: static, dynamic, or error?

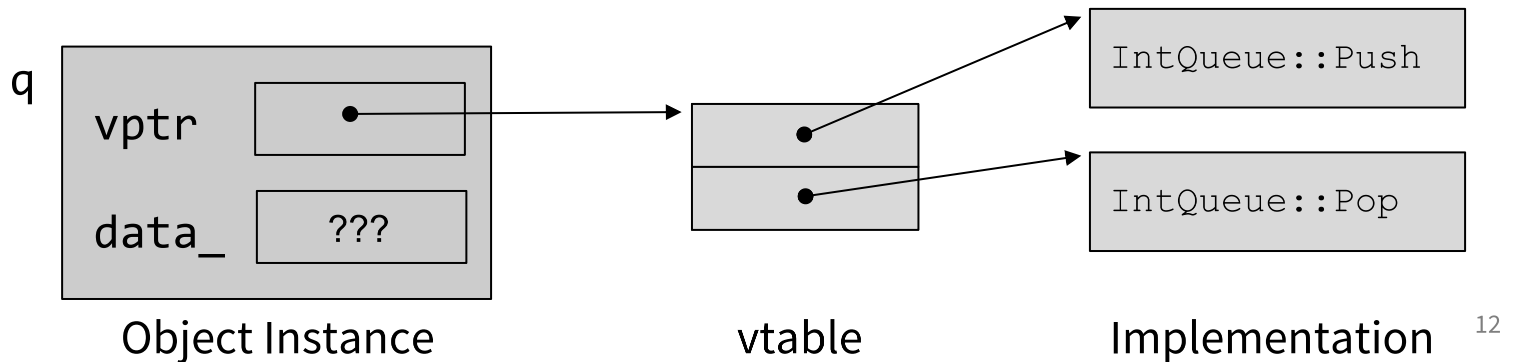
```
class Base {
  void Foo();             // static dispatch
  void Bar();             // static dispatch
  virtual void Baz();     // dynamic dispatch
};

class Derived : public Base {
  virtual void Foo();     // now dynamic (for more derived)
  //void Bar() override;// compiler error
  void Bar();             // static dispatch
  void Baz();             // still dynamic (sticky!)
};
```

Vtable (Virtual Function Table) & Vptr (Vtable pointer)

- vtable: An array of function pointers defined for each class that has at least one virtual method to enable dynamic dispatch
  - One per class
- vptr: Each class object instance has a pointer to that vtable
  - One per object instance

Vtable Diagrams

```
class IntQueue {
 public:
  virtual void Push(int x);
  virtual int Pop();
 private:
  vector<int> data_;
};
IntQueue q;
```

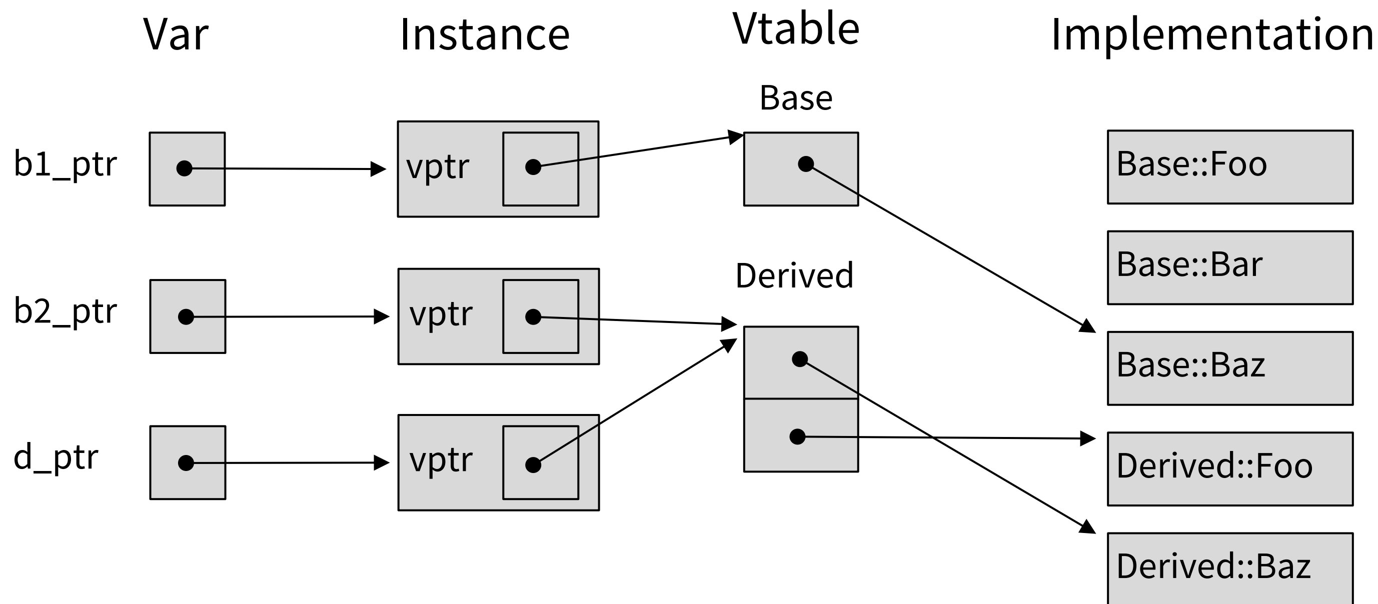


Vtable Diagrams

```
Base *b1_ptr = new Base;
Base *b2_ptr = new Derived;
Derived *d_ptr = new Derived;
```

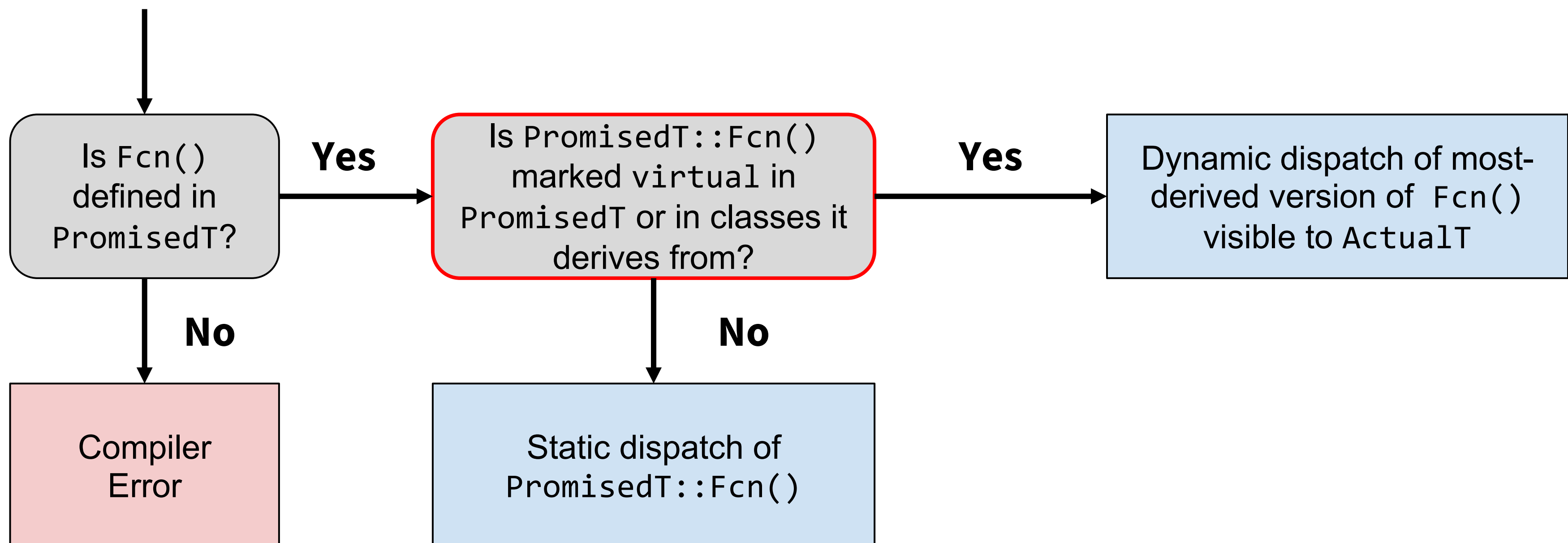
```
class Base {
  void Foo();
  void Bar();
  virtual void Baz();
};
```

```
class Derived :
public Base {
  virtual void Foo();
  void Baz();
};
```



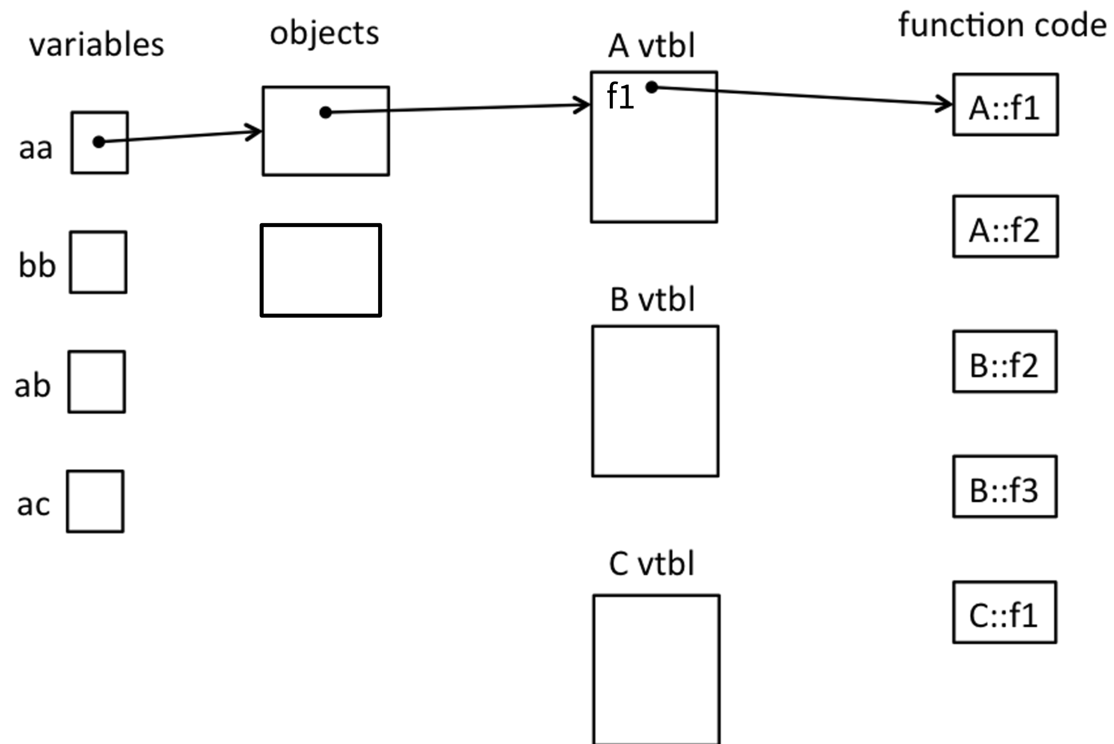
Updated Dispatch Decision Tree

```
PromisedT *ptr = new ActualT();
ptr->Fcn();  // which version is called?
```

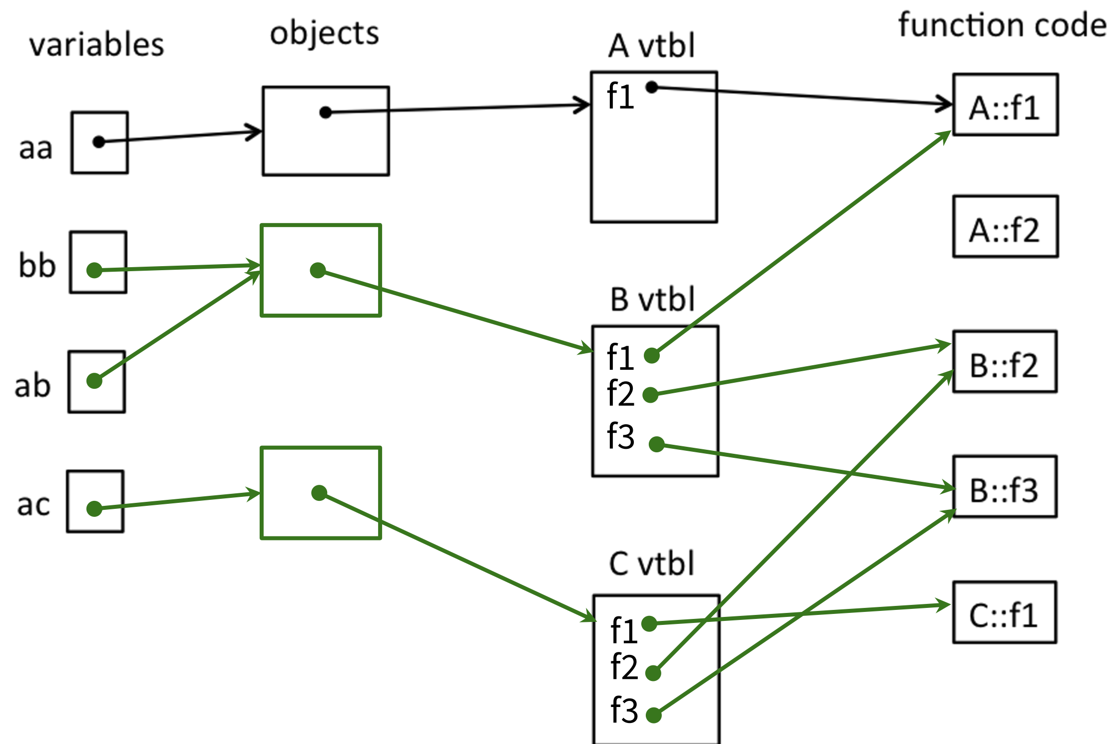


Exercise 1!

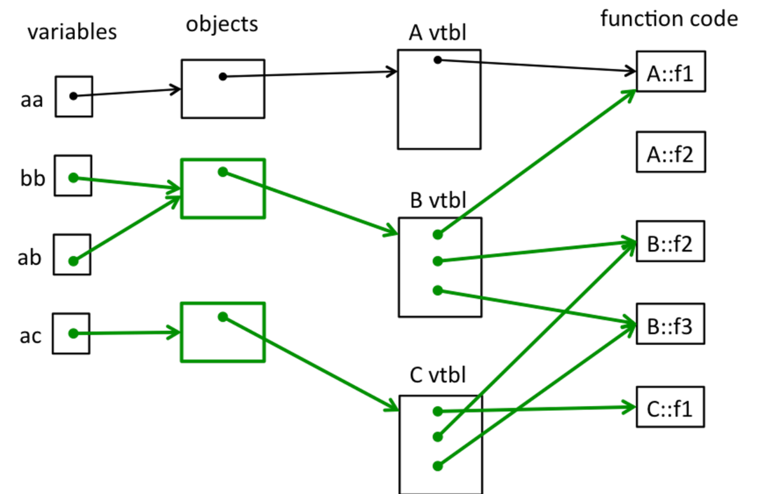
Exercise 1



Exercise 1 Solution (pointers)



Exercise 1 Solution (output)

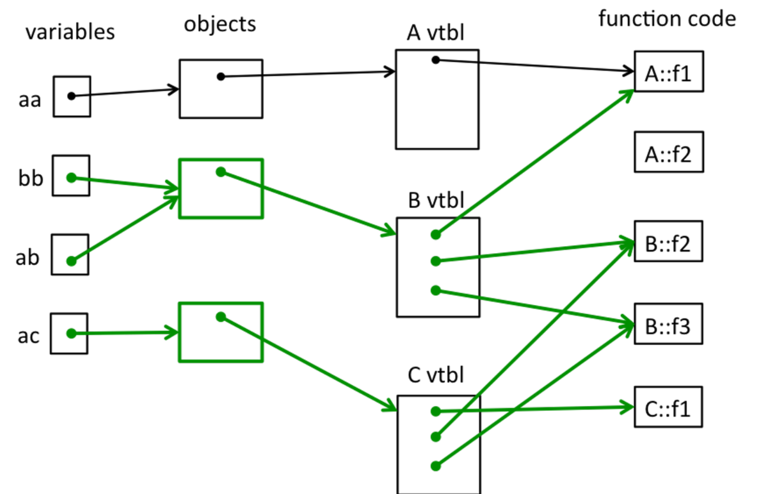



```
#include <iostream>
using namespace std;

class A {
 public:
  virtual void f1() { f2(); cout << "A::f1" << endl; }
  void f2() { cout << "A::f2" << endl; }
};

class B: public A {
 public:
  virtual void f3() { f1(); cout << "B::f3" << endl; }
  virtual void f2() { cout << "B::f2" << endl; }
};

class C: public B {
 public:
  void f1() { f2(); cout << "C::f1" << endl; }
};
```

```
A* aa = new A();

aa->f1();
```

```
A::f2
A::f1
```

Exercise 1 Solution (output)

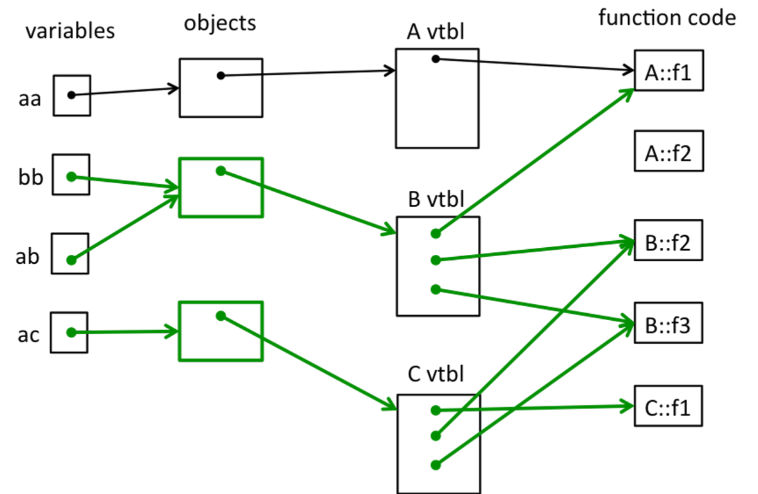



```
#include <iostream>
using namespace std;

class A {
 public:
  virtual void f1() { f2(); cout << "A::f1" << endl; }
  void f2() { cout << "A::f2" << endl; }
};

class B: public A {
 public:
  virtual void f3() { f1(); cout << "B::f3" << endl; }
  virtual void f2() { cout << "B::f2" << endl; }
};

class C: public B {
 public:
  void f1() { f2(); cout << "C::f1" << endl; }
};
```

```
B* bb = new B();

bb->f1();
```

```
A::f2
A::f1
```

Exercise 1 Solution (output)

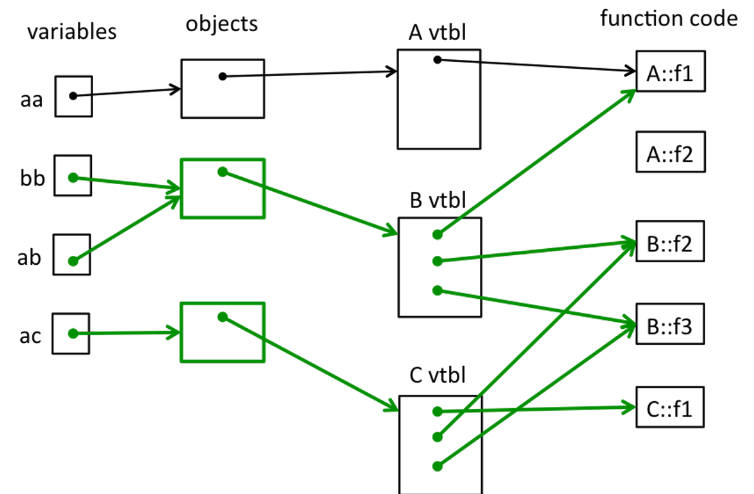



```
#include <iostream>
using namespace std;

class A {
 public:
  virtual void f1() { f2(); cout << "A::f1" << endl; }
  void f2() { cout << "A::f2" << endl; }
};

class B: public A {
 public:
  virtual void f3() { f1(); cout << "B::f3" << endl; }
  virtual void f2() { cout << "B::f2" << endl; }
};

class C: public B {
 public:
  void f1() { f2(); cout << "C::f1" << endl; }
};
```

```
B* bb = new B();
A* ab = bb;

bb->f2();
cout << "----" << endl;
ab->f2();
```

```
B::f2
----
A::f2
```

Exercise 1 Solution (output)

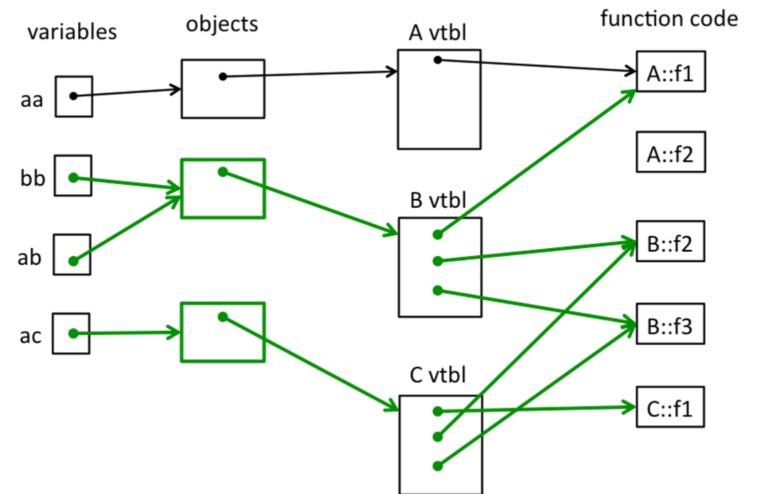



```
#include <iostream>
using namespace std;

class A {
 public:
  virtual void f1() { f2(); cout << "A::f1" << endl; }
  void f2() { cout << "A::f2" << endl; }
};

class B: public A {
 public:
  virtual void f3() { f1(); cout << "B::f3" << endl; }
  virtual void f2() { cout << "B::f2" << endl; }
};

class C: public B {
 public:
  void f1() { f2(); cout << "C::f1" << endl; }
};
```

```
B* bb = new B();

bb->f3();
```

```
A::f2
A::f1
B::f3
```

Exercise 1 Solution (output)




```
#include <iostream>
using namespace std;

class A {
 public:
  virtual void f1() { f2(); cout << "A::f1" << endl; }
  void f2() { cout << "A::f2" << endl; }
};

class B: public A {
 public:
  virtual void f3() { f1(); cout << "B::f3" << endl; }
  virtual void f2() { cout << "B::f2" << endl; }
};

class C: public B {
 public:
  void f1() { f2(); cout << "C::f1" << endl; }
};
```

```
A* ac = new C();

ac->f1();
```

```
B::f2
C::f1
```

Templates

- C++ syntax to generate code that works with *generic types*.
- Generates a new implementation in assembly for every type it is used with:
    - *e.g.* calls to `foo<int>()` and `foo<double>()` generate two implementations
    - *e.g.* calls to `foo<int>()` and another `foo<int>()` requires only one implementation
    - *e.g.* if `foo` is never used, zero implementations are generated

Template Syntax & Practice

```
template<typename T>
T add3(T arg) {
        T result = arg + 3;
        return result;
}
```

Results?

```
add3<int>(3);           // uses add3<int>, returns 6
add3(5.5);
add3<char*>("a str");
add3<string>("a str");
```

Template Syntax & Practice

```
template<typename T>
T add3(T arg) {
       T result = arg + 3;
       return result;
}
```

Results?

```
add3<int>(3);             // uses add3<int>, returns 6
add3(5.5);                // uses add3<double>, returns 8.5
add3<char*>("a str");
add3<string>("a str");
```

Template Syntax & Practice

```
template<typename T>
T add3(T arg) {
        T result = arg + 3;
        return result;
}
```

Results?

```
add3<int>(3);            // uses add3<int>, returns 6
add3(5.5);               // uses add3<double>, returns 8.5
add3<char*>("a str");    // uses add3<char*>, return ->"tr"
add3<string>("a str");
```

Template Syntax & Practice

```
template<typename T>
T add3(T arg) {
        T result = arg + 3;
        return result;
}
```

Results?

```
add3<int>(3);            // uses add3<int>, returns 6
add3(5.5);               // uses add3<double>, returns 8.5
add3<char*>("a str");    // uses add3<char*>, return ->"tr"
add3<string>("a str");   // Compiler error! No `+` for string
                         //                 and num
```

Templates Get Fancier

```
template<typename T, int N = 2> // Templatize values
T modulo(T arg) {
	T result = arg % N;
	return result;
}

modulo(5) == 1  (=5%2)
modulo<int, 5>(17) == 2  (=17%5)

// C++ template system is very powerful
// Simple type-substitution is enough for most programs
```

Exercise 3!

Exercise 3 Solution

```
#include <iostream>
using namespace std;


class List {
 public:
  // construct empty list
  List() : head_(nullptr) { }

  // add new node with value n to the front of the list
  virtual void add(int n) {
    Link *p = new Link(n, head_);
    head_ = p;
  }

...
```

Exercise 3 Solution

```
#include <iostream>
using namespace std;

template <typename T>
class List {
 public:
  // construct empty list
  List() : head_(nullptr) { }

  // add new node with value n to the front of the list
  virtual void add(int n) {
    Link *p = new Link(n, head_);
    head_ = p;
  }

...
```

Exercise 3 Solution

```
#include <iostream>
using namespace std;

template <typename T>
class List {
 public:
  // construct empty list
  List() : head_(nullptr) { }

  // add new node with value n to the front of the list
  virtual void add(int T n) {
    Link *p = new Link(n, head_);
    head_ = p;
  }

...
```

Exercise 3 Solution

```
...
private:
  struct Link {  // nodes for the linked list
    int val;
    Link * next;
    Link(int n, Link* nxt): val(n), next(nxt) { }
  };
  // List instance variable
  Link * head_;  // head of list or nullptr if list is empty
};  // end of List class

int main() {
  List nums;
  nums.add(1);
  nums.add(2);
  return EXIT_SUCCESS;
}
```

Exercise 3 Solution

```
...
private:
  struct Link {  // nodes for the linked list
    int T val;
    Link * next;
    Link(int T n, Link* nxt): val(n), next(nxt) { }
  };
  // List instance variable
  Link * head_;  // head of list or nullptr if list is empty
};  // end of List class

int main() {
  List nums;
  nums.add(1);
  nums.add(2);
  return EXIT_SUCCESS;
}
```

Exercise 3 Solution

```
...
private:
  struct Link {  // nodes for the linked list
    int T val;
    Link * next;
    Link(int T n, Link* nxt): val(n), next(nxt) { }
  };
  // List instance variable
  Link * head_;  // head of list or nullptr if list is empty
};  // end of List class

int main() {
  List<int> nums;
  nums.add(1);
  nums.add(2);
  return EXIT_SUCCESS;
}
```

C++ standard lib is built around templates

- Containers
  - Store data
  - Define iterators to go over that data
- Iterators
  - Different flavors (random access, bidirectional, etc)
  - Common interface to containers
- Algorithms
  - Use the common interface of iterators to do things
    - Different algorithms require different ‘complexities’ of iterators

Common STL Data Structures

- map<Key, Value, Order=std::Less<Key>>
  - Store key -> value pairs where we can use a key to get the value (TreeMap)
- set<Item, Order=std::Less<Item>>
  - An unindexed collection of items
  - When you care most about "do I have this"
- vector<Item>
  - Resizable array (ArrayList, in Java)
- unordered_map<Key, Value, Hash=std::Hash<Key>>
  - HashMap → use hash of key to order. Usually faster than map
- Assorted others (queue, linkedlist, etc.)

Now what's that 'std::less'? *//Out of scope*

```
std::less<T>(const T& lhs, const T& rhs) {
    return lhs < rhs;
}
```

- Much like in Java, some structures require ordering elements
  - E.g. set is implemented as a binary tree
- Want to let users store custom types.
  - Java uses Comparable, C++ uses operator< (in std::less)
- However, maybe you want to use a different ordering
  - Ordering is templated function so you can substitute
  - E.g. set<int, std::greater<int>> or set<int, myIntCompare>

Exercise 4!

Exercise 4!

Exercises:

2) Standard Template Library

Complete the function `ChangeWords` below. This function has as inputs a vector of strings, and a map of <string, string> key-value pairs. The function should return a new vector<string> value (not a pointer) that is a copy of the original vector except that every string in the original vector that is found as a key in the map should be replaced by the corresponding value from that key-value pair.

Example: if vector `words` is `{"the", "secret", "number", "is", "xlii"}` and map `subs` is `{{"secret", "magic"}, {"xlii", "42"}}`, then `ChangeWords(words, subs)` should return a new vector `{"the", "magic", "number", "is", "42"}`.

Hint: Remember that if `m` is a map, then referencing `m[k]` will insert a new key-value pair into the map if `k` is not already a key in the map. You need to be sure your code doesn't alter the map by adding any new key-value pairs. (Technical nit: `subs` is not a const parameter because you might want to use its `operator[]` in your solution, and `[]` is not a const function. It's fine to use `[]` as long as you don't actually change the contents of the map `subs`.)

Write your code below. Assume that all necessary headers have already been written for you.

```
using namespace std;
vector<string> ChangeWords(const vector<string> &words,
                           map<string,string> &subs) {








}
```

Write your code below. Assume that all necessary headers have already been written for you.

```
using namespace std;
vector<string> ChangeWords(const vector<string> &words,
                           map<string,string> &subs) {
    vector<string> result;
    for (auto &word : words) {
      if (subs.find(word) != subs.end()) {
        result.push_back(subs[word]);
      } else {
        result.push_back(word);
      }
    }
    return result;

}
```

Exercise 5!

Here is a little program that has a small class Thing and main function
(assume that necessary #includes and using namespace std; are included).

```
class Thing {
 public:
  Thing(int n): n_(n) { }
  int getThing() const { return n_; }
  void setThing(int n) { n_ = n; }
 private:
  int n_;
};
```

```
int main() {
  Thing t(17);
  vector<Thing> v;
  v.push_back(t);
}
```

This code compiled and worked as expected, but then we added the following two lines of code (plus the appropriate #include <set>):

```
set<Thing> s;
s.insert(t);
```

The second line (s.insert(t)) failed to compile and produced dozens of spectacular compiler error messages, all of which looked more-or-less like this (edited to save space):

In file included from string:48:0, from bits/locale_classes.h:40, from bits/ios_base.h:41,from ios:42,from ostream:38, from /iostream:39,from thing.cc:3: bits/stl_function.h: In instantiation of 'bool std::less<_Tp>::operator()(const _Tp&, const _Tp&) const [with _Tp = Thing]': <<*many similar lines omitted*> > thing.cc:37:13: required from here bits/stl_function.h:
387:20: error: no match for 'operator<' (operand types are 'const Thing' and 'const Thing') { return __x < __y; }

What on earth is wrong? Somehow class Thing doesn't work with set<Thing> even though insert is the correct function to use here. (a) What is the most likely reason, and (b) what would be needed to fix the problem? (Be brief but precise – you don't need to write code in your answer, but you can if that helps make your explanation clear.)

Here is a little program that has a small class Thing and main function
(assume that necessary #includes and using namespace std; are included).

```
class Thing {
 public:
  Thing(int n): n_(n) { }
  int getThing() const { return n_; }
  void setThing(int n) { n_ = n; }
 private:
  int n_;
};
```

```
int main() {
  Thing t(17);
  vector<Thing> v;
  v.push_back(t);
}
```

The second line (s.insert(t)) failed to compile and produced dozens of spectacular compiler error messages, all of which looked more-or-less like this (edited to save space):

In file included from string:48:0, from bits/locale_classes.h:40, from bits/ios_base.h:41,from ios:42,from ostream:38, from /iostream:39,from thing.cc:3: bits/stl_function.h: In instantiation of 'bool std::less<_Tp>::operator()(const _Tp&, const _Tp&) const [with _Tp = Thing]': <*<many similar lines omitted*> > thing.cc:37:13: required from here bits/stl_function.h:
387:20: error: no match for 'operator<' (operand types are 'const Thing' and 'const Thing') { return __x < __y; }

What on earth is wrong? Somehow class Thing doesn't work with set<Thing> even though insert is the correct function to use here. (a) What is the most likely reason, and (b) what would be needed to fix the problem? (Be brief but precise – you don't need to write code in your answer, but you can if that helps make your explanation clear.)

STL has to compare them using operator<. Add an appropriate operator< as either a member function in Thing, or as a free-standing function that compares two Thing& parameters.