333 Section 7 Solutions - C++ Casting and Inheritance

Welcome back to section! We're glad that you're here :)

Casting in C++

While in C++, we want to use casts that are more explicit in their behaviour. This gives us a better understanding of what happens when we read our code, because C-style casts can do many (sometimes unwanted) things. There are four types of casts we will use in C++:

`static_cast<to_type>(expression);`

- ★ Converts between pointers of related types.
  - Compiler error if not related.
- ★ Performs not pointer conversion (e.g. float to int conversion).

`dynamic_cast<to_type>(expression);`

- ★ Converts between pointers of related types.
  - Compiler error if not related.
  - Also checks at runtime to make sure it is a 'safe' conversion (returns `nullptr` if not).

`const_cast<to_type>(expression);`

- ★ Used to add or remove const-ness.

`reinterpret_cast<to_type>(expression);`

- ★ Casts between incompatible types *without changing the data*.
  - The types you are casting to and from must be the same size.
  - Will not let you convert between integer and floating point types.

Exercise 1

For each of the following snippets of code, fill in the blank with the most appropriate C++ style cast. Assume that we have the following classes defined:

```
class Base {
 public:
  int x;
};
```

```
class Derived : public Base {
 public:
  int y;
};
```

```
int64_t x = 0x7ffffffffe870;
char* str = reinterpret_cast<char*>(x);
```

```
void foo(Base* b) {
  Derived *d = dynamic_cast<Derived*>(b);
  // additional code omitted
}
```

```
Derived* d = new Derived;
```

```
Base* b = static_cast<Base*>(d);
```

```
double x = 64.382;
int64_t y = static_cast<int64_t>(x);
```

C++ Inheritance

Access Specifiers:

- `public:` visible to all other classes
- `protected:` visible to this class and its *derived* classes
- `private:` visible only to the current class

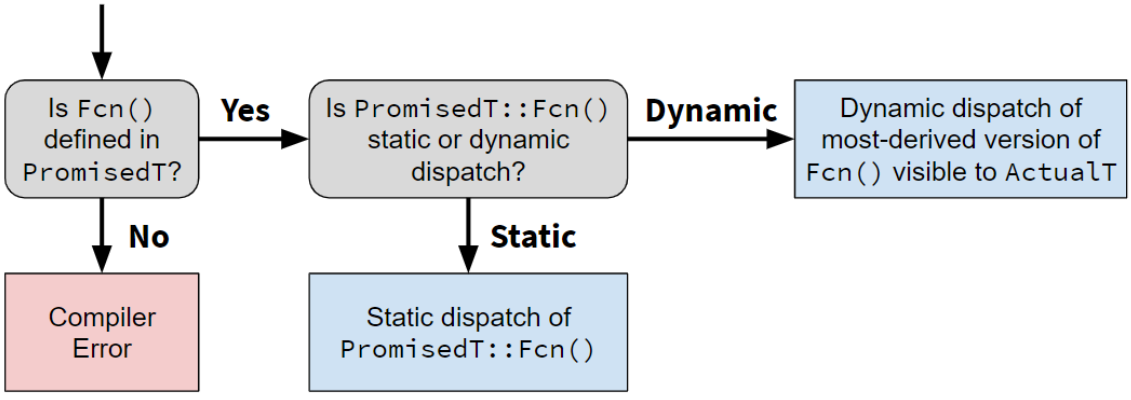
What's different in C++ (compared to Java)?

- *Static vs. dynamic dispatch* – in Java, all method calls are dynamic dispatch
- Pure virtual functions, abstract classes, why no Java "interfaces"
- Assignment slicing, using class hierarchies with STL

`virtual` keyword: Prefix a member function's declaration with this to use dynamic dispatch. Note: derived (child) functions don't need to repeat the virtual keyword, but it traditionally often do.

`override` keyword (C++11): Postfix a member function's declaration with this to tell the compiler that this method should be overriding an inherited virtual function – good to use if available.

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



Exercise:

2) Inheritance & Virtual Function

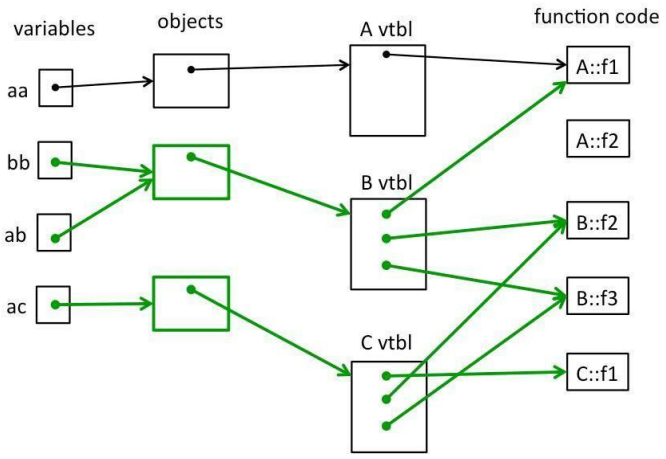
Consider the program on the following page, which does compile and execute with no errors, except that it leaks memory (which doesn't matter for this question).

(a) Complete the diagram on the next page by adding the remaining objects and all of the additional pointers needed to link variables, objects, virtual function tables, and function bodies. Be sure that the order of pointers in the virtual function tables is clear (i.e., which one is first, then next, etc.). One of the objects and a couple of the pointers are already included to help you get started.

(b) Write the output produced when this program is executed. If the output doesn't fit in one column in the space provided, write multiple vertical columns showing the output going from top to bottom, then successive columns to the right

```
#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; }
};
```



```
int main() {
  A* aa = new A();
  B* bb = new B();
  A* ab = bb;
  A* ac = new C();
  aa->f1();
  cout << "---" << endl;
  bb->f1();
  cout << "---" << endl;
  bb->f2();
  cout << "---" << endl;
  ab->f2();
  cout << "---" << endl;
  bb->f3();
  cout << "---" << endl;
  ac->f1();
  return EXIT_SUCCESS;
}
```

```
Output:
A::f2
A::f1
---
A::f2
A::f1
---
B::f2
---
A::f2
---
A::f2
A::f1
B::f3
---
B::f2
C::f1
```