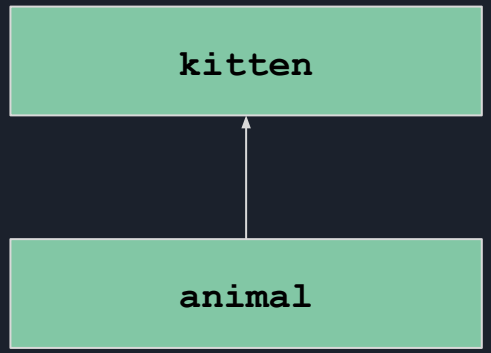
CS202 Lab 3

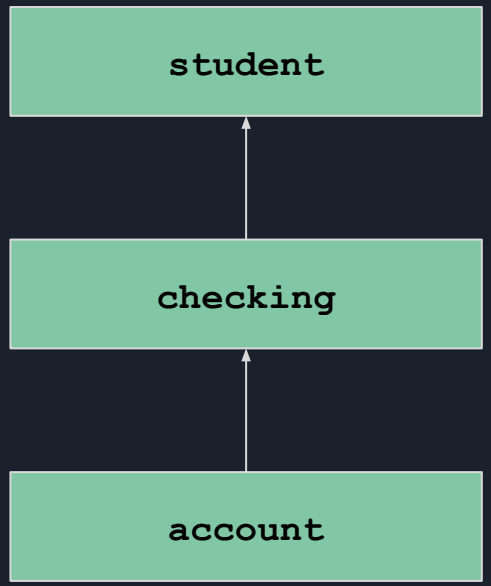
Dynamic Binding

Upcasting




```
animal * k = new kitten;

kitten k_obj;
animal & k_obj_upcasted = k_obj;
// OR
animal * k_obj_upcasted = &
k_obj;

animal * c = new kitten;
```

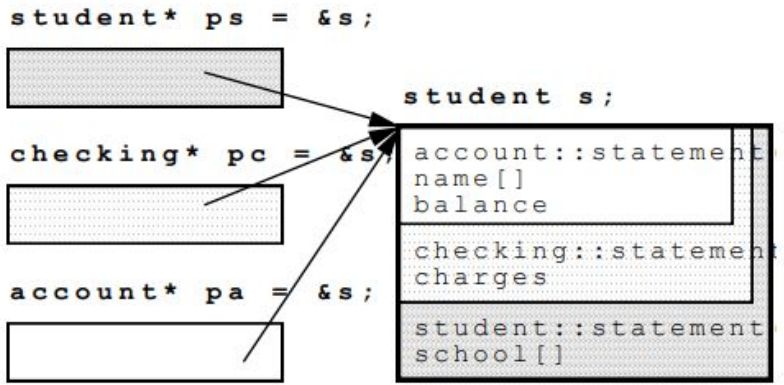
Upcasting




```
account * c = new checking;

checking c_obj;
account & c_obj_upcasted = c_obj;
// OR
account * c_obj_upcasted = & c_obj;

account * c = new student;

checking s_obj;
account & s_obj_upcasted = s_obj;
// OR
account * s_obj_upcasted = & s_obj;
```

Upcasting

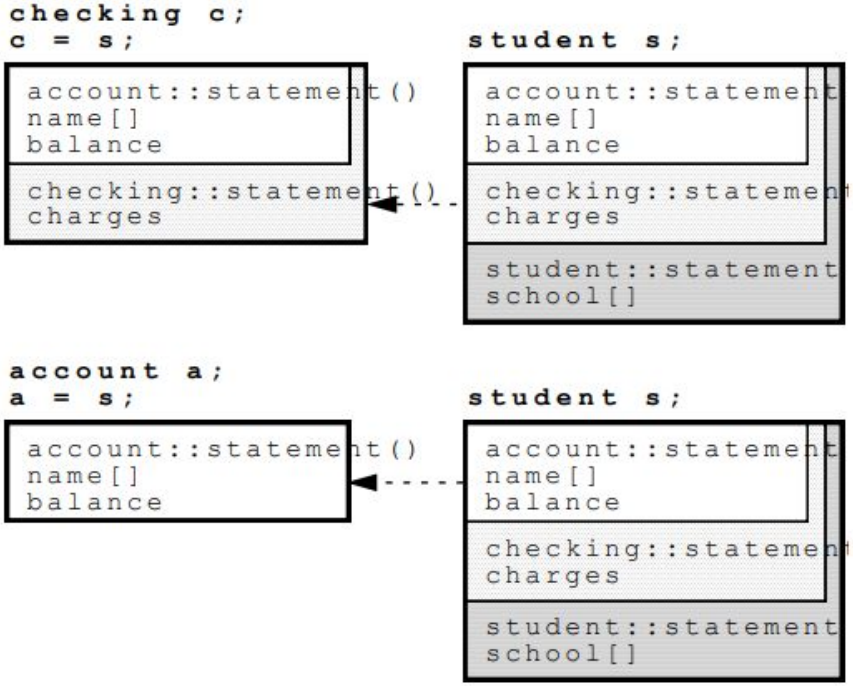
- Notice that when we have a pointer to an account object and we initialize or assign it to the address of a student or checking object, that we are still actually pointing to the student or checking object.
- This is the only time in C++ when it is allowed to assign a pointer of one type to another without an explicit cast operation.
- This is because a pointer to a derived class object points to a direct or indirect base class object as well!



Upcasting

```
void print_account(account* p) { //pointer
  p->statement();
}

void print_account(account &r) { //reference
  r.statement();
}

int main() {
  student smith("Joe Smith", 5000, "UT");
  student* ps = &smith;        ps->statement();

  checking* pc = &smith;       pc->statement();

  account* pa = &smith;        pa->statement();

  print_account(&smith); //pass by pointer
  print_account(smith);  //pass by reference
}
```

Static Binding



Rules of Dynamic Binding

- Function must have the virtual keyword
- Function signature and return type must be the same for all implementations
- Only need virtual keyword in base class
- Don't have to have function defined in derived classes if behavior isn't different
- Once a member function is declared virtual, it remains virtual for all derived classes

Rules of Dynamic Binding

- Virtual functions cannot be static member functions.
- Second, the signature and return type must be the same for all implementations of the virtual function.
- Third, while the function must be defined as a virtual function within a direct or indirect base class, it need not be defined in those derived classes where the inherited behavior does not need to differ.
- And finally, the keyword virtual is only required within the base class itself; derived class implementations of the overridden function do not need to repeat the use of that keyword.
- Once a member function is declared to be virtual, it remains virtual for all derived classes.

Function Terminology

- **Overloading:**
  - Same scope
  - Different signatures
- **Hiding:**
  - Static binding (compile time)
  - Different scope
  - No signature requirements
- **Overriding:**
  - Dynamic binding (runtime)
  - Different Scope

```
void display();
void display(node * root);



void animal::display();
void kitten::display(char *);


virtual void
animal::display();
void kitten::display();
```

Overriding...What is it?

- Overriding:
  - Defining a function to be virtual in a base class and then implementing that function in a derived class using exactly the same signature and return type.
- The selection of which function to use depends on the dynamic type of the object when accessed through a direct or indirect base class pointer or reference at run time.

Overriding vs. Overloading

- There are two major differences between overloading and overriding.
  - Overloading requires unique signatures whereas overriding requires the same signature and return type.
  - Second, overloading requires that each overloaded version of the function be specified within the same scope whereas overriding requires each overridden version be specified within the scope of each derived class.

Overriding vs. Hiding

- There are two major differences between hiding and overriding.
  - Hiding has no requirements on the signatures whereas overriding requires exactly the same signature and return type.
  - Second, hiding uses the static type of the object at compile time to determine which member function to bind whereas overriding uses the dynamic type of the object at run time to determine which member function to bind.

Abstract Base Classes

- Why?
  - To provide a common interface
  - Forces derived classes to implement pure virtual functions.
- Pure virtual function in base class
  - virtual void display() = 0;
- You can't create a pure virtual class object (can still use upcasting)

Abstract Base Classes

- An abstract class is a class that can only be derived from; no objects can be instantiated it.
- Its purpose is to define an interface and provide a common base class for derived classes.
- A base class becomes an abstract class by making its constructor(s) protected or by declaring a virtual function pure: `virtual void statement()=0;`
- Derived classes must implement all pure virtual functions. If a derived class does not implement these functions, then it becomes an abstract class as well.
- Abstract classes are not required to implement their pure virtual functions.

Abstract Base Classes

- The purpose of declaring a function to be pure is to force the derived classes to implement it.
- A virtual function is a contract with a derived class indicating the name, signature, and return type for the function.
- Making the virtual function pure forces the contract to be fulfilled.

RTTI

- Run Time Type Identification (RTTI) uses type information stored in objects by the compiler to determine at run time the actual type of an object pointed or referred to.
- RTTI can only be used when at least one function has been declared to be **virtual** in the class or in a direct or indirect base class.
- For the full benefits of dynamic binding, applications must write code independent of the type of object being pointed or referred to.
- RTTI provides a way for client applications to determine the type of an object without having to compromise their use of dynamic binding.

Downcasting and RTTI

- We might need to know what an upcasted object is during run time. (What is the animal pointer? Is it a kitten, turtle, owl?...)
- To do this, we use run time type identification (RTTI)
- In c++ we accomplish this with static or dynamic casting.
- We try to avoid RTTI because it's harder to maintain and may indicate poor design.

Dynamic Casting

```
int zoo::to_add(animal * animal_ptr)
{
     animal * to_add = NULL;

     kitten * k_ptr = dynamic_cast<kitten *>(animal_ptr);
     if(k_ptr)
     {
          //animal_ptr of base class animal is actually a kitten pointer

          to_add = new kitten(*k_ptr); // allocate memory
     }
     else if(ptr == NULL)
          //animal_ptr was not a kitten pointer

     owl * o_ptr = dynamic_cast<owl *>(animal_ptr);
     if(o_ptr)
     {
          //animal_ptr was a pointer to an owl!
          to_add = new owl(*o_ptr);
     }
     Else
          //not a pointer to an owl…
}
```

Dynamic Casting Const Object

```
int zoo::add_animal( const animal & animal_obj)
{
     const kitten * k_ptr = dynamic_cast< const kitten *>(&animal_obj);
     if(k_ptr)
     {
          //It was a kitten object!
          animal_arr[index] = new kitten(*k_ptr);
     }
     Else
          //not a kitten…
     }
}
```

Commonalities & Differences

Animals

"Fly"

"Roll around"

"Swim"

Animals

Move();

Speak();

"Fly"

"Roll around"

"Swim"