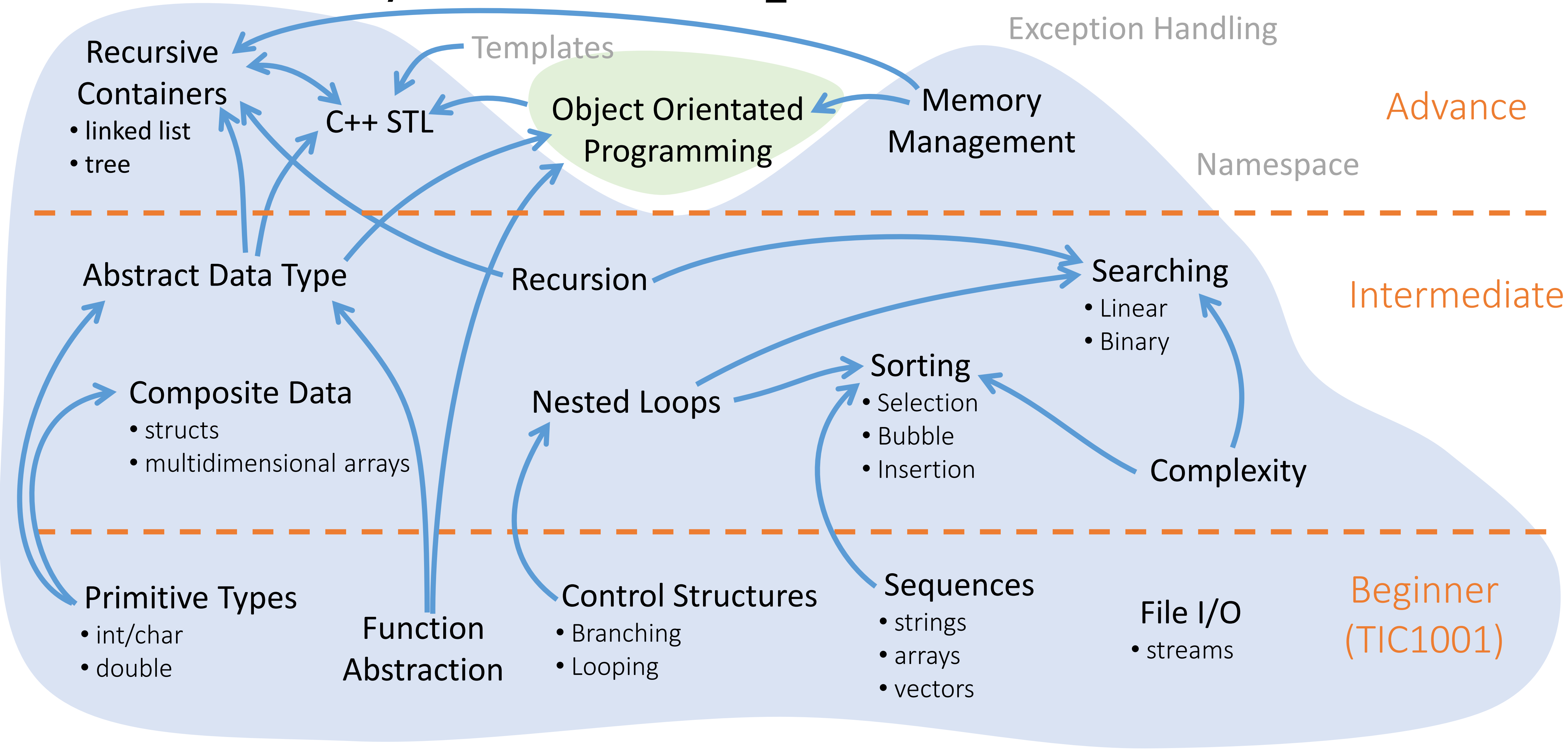
Lecture 9
Inheritance

TIC1002 Introduction to Computing and Programming II

TIC1001/2 Roadmap



Advance

- Recursive Containers
  - linked list
  - tree
- C++ STL
- Templates
- Object Orientated Programming
- Memory Management
- Exception Handling
- Namespace

Intermediate

- Abstract Data Type
- Recursion
- Searching
  - Linear
  - Binary
- Composite Data
  - structs
  - multidimensional arrays
- Nested Loops
- Sorting
  - Selection
  - Bubble
  - Insertion
- Complexity

Beginner (TIC1001)

- Primitive Types
  - int/char
  - double
- Function Abstraction
- Control Structures
  - Branching
  - Looping
- Sequences
  - strings
  - arrays
  - vectors
- File I/O
  - streams

Course Schedule

Week	Topic(s)		
7	Midterm Test		
8	Abstract Data Type & C++ STL		
9	Working with Collections	Problem Set 3	
10	Object Oriented Programming		Problem Set 4
11	OOP: Inheritance		
12	OOP: Polymorphism		
13	Revision	Practical Exam 2	
Reading			
Exam	Final Exam (Tue 27 Apr)		

Practical Exam 2

Saturday, 17 April, 9:00 am

- Same seating plan as PE1
- Topics everything until OOP (encapsulation)

Makeup PE2

- Afternoon of final exam
- Tue 27 Apr

Object Oriented Languages

✓ Encapsulation

- Group data and function together
- Internal details hidden/abstracted

Inheritance

- Extend current implementation
- Logical relationship between entities

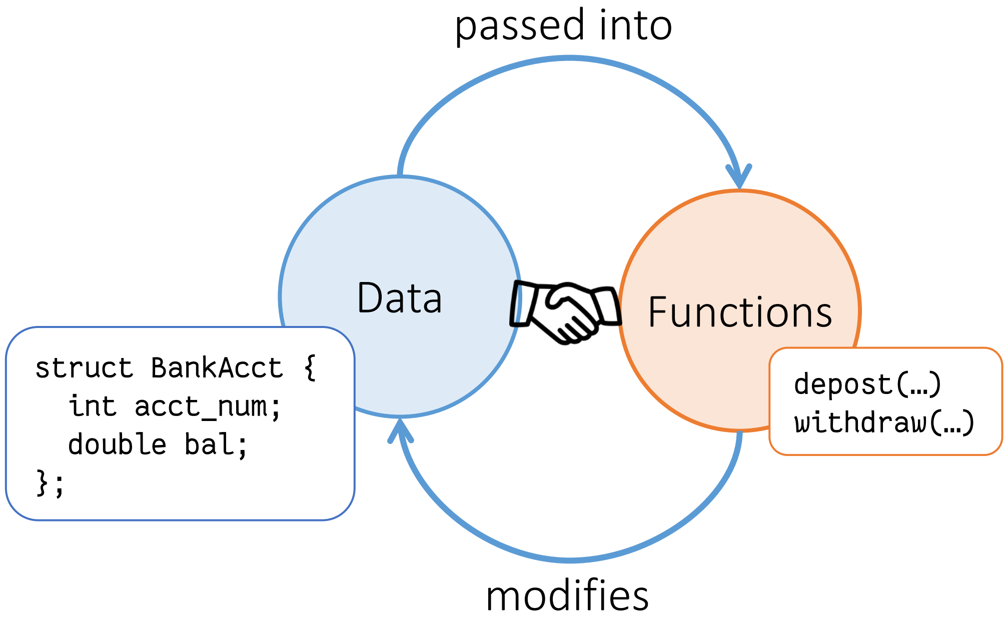
Polymorphism

- Behaviour changes according to actual data type
- Abstract classes

Comparing Programming Paradigms

Procedural Model

- Data (struct) and process (functions) are separate entities

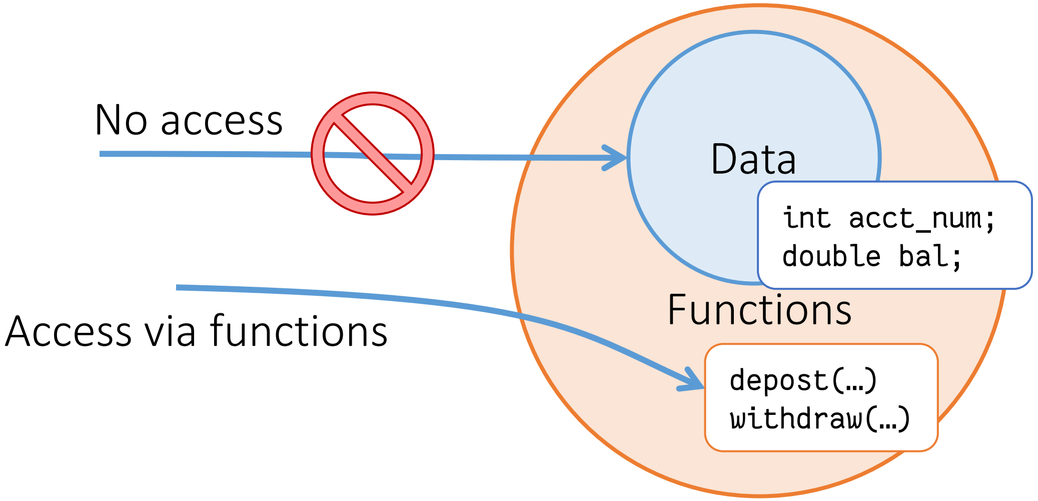


```
struct BankAcct {
  int acct_num;
  double bal;
};
```

```
depost(…)
withdraw(…)
```

Object Oriented Model

- Data is encapsulated in functions
- No direct access to data
- Only access using exposed functions



```
int acct_num;
double bal;
```

```
depost(…)
withdraw(…)
```

BankAcct Class

Constructor(s)

```
BankAcct(int acct_num)
BankAcct(int acct_num, double amt)
```

Accessors

```
virtual int get_acct_num()
virtual double get_balance()
```

Mutators

```
virtual void deposit(double amt)
virtual bool withdraw(double amt)
```

Destructors

```
~BankAcct()
```

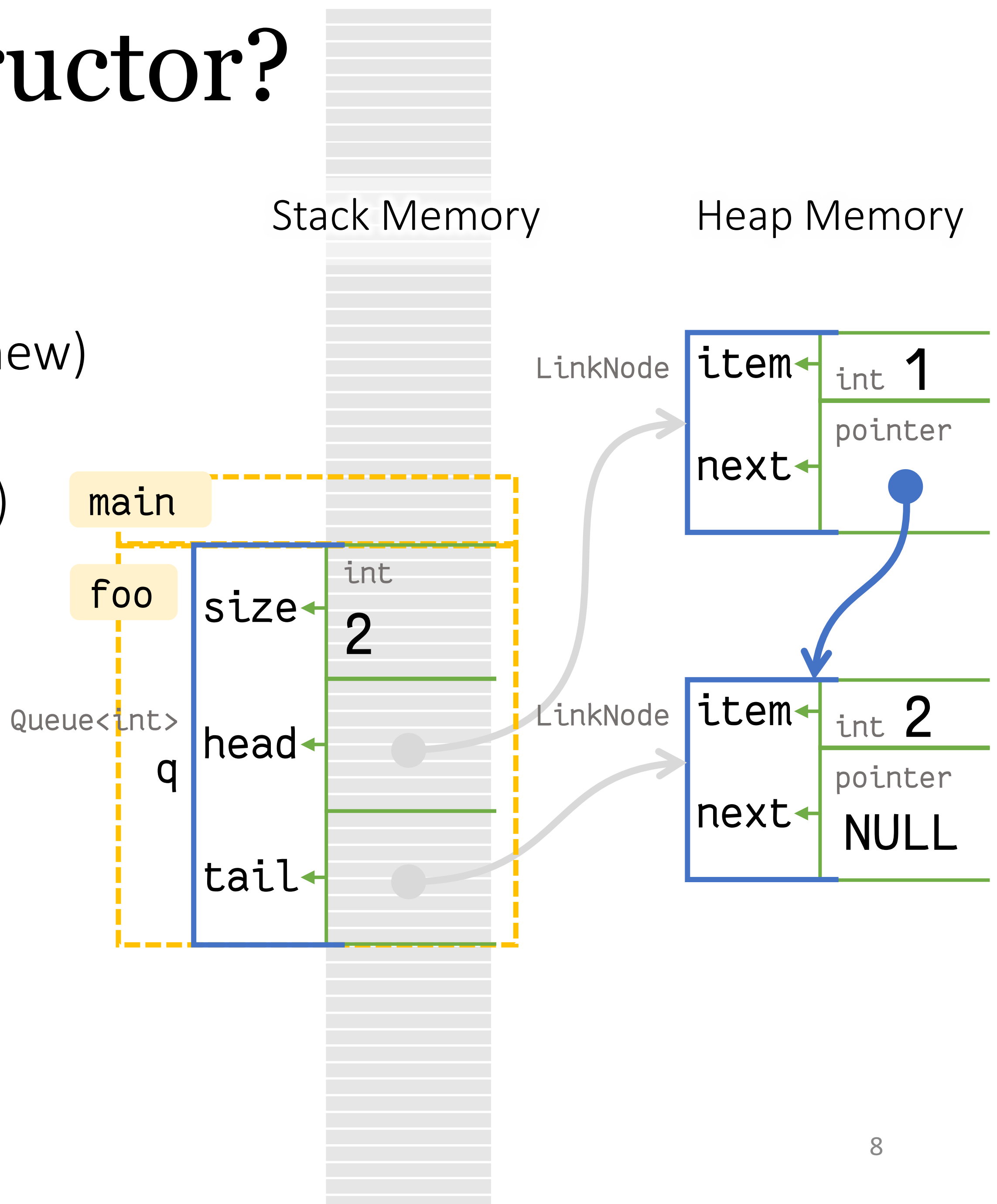
Why do we need Destructor?

Example: Queue ADT

- Because ADT instantiated new objects
- It allocated memory from the heap (i.e. new)
- Thus, when ADT is begin deleted
- It needs to deallocate objects (i.e. delete)

```
void foo() {
    Queue<int> q;
    enqueue(q, 1);
    enqueue(q, 2);
}

int main() {
    foo();
    // memory leak. dequeue was never called
}
```

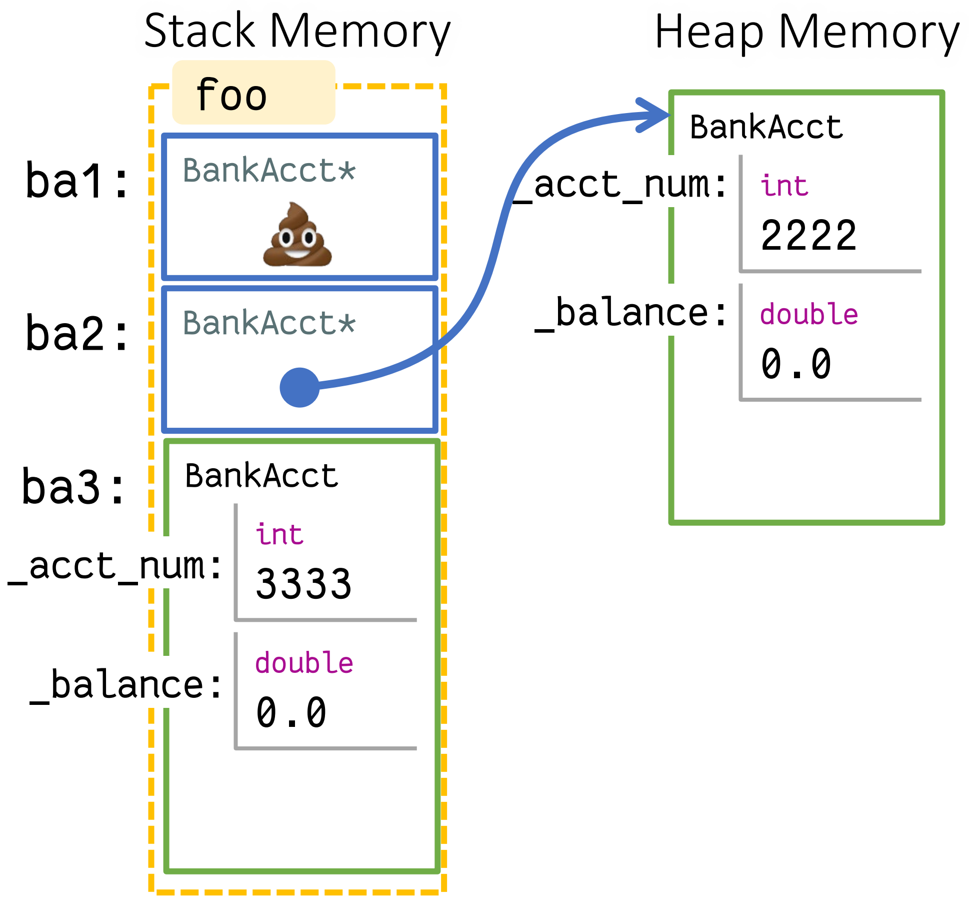


https://bit.ly/2FZ0brx

Ways to Instantiate Class

What's the difference in the following ways?

```
void foo() {
  BankAcct *ba1;
  BankAcct *ba2 = new BankAcct(2222);
  BankAcct ba3(3333);
}
```



Ways to Instantiate Class

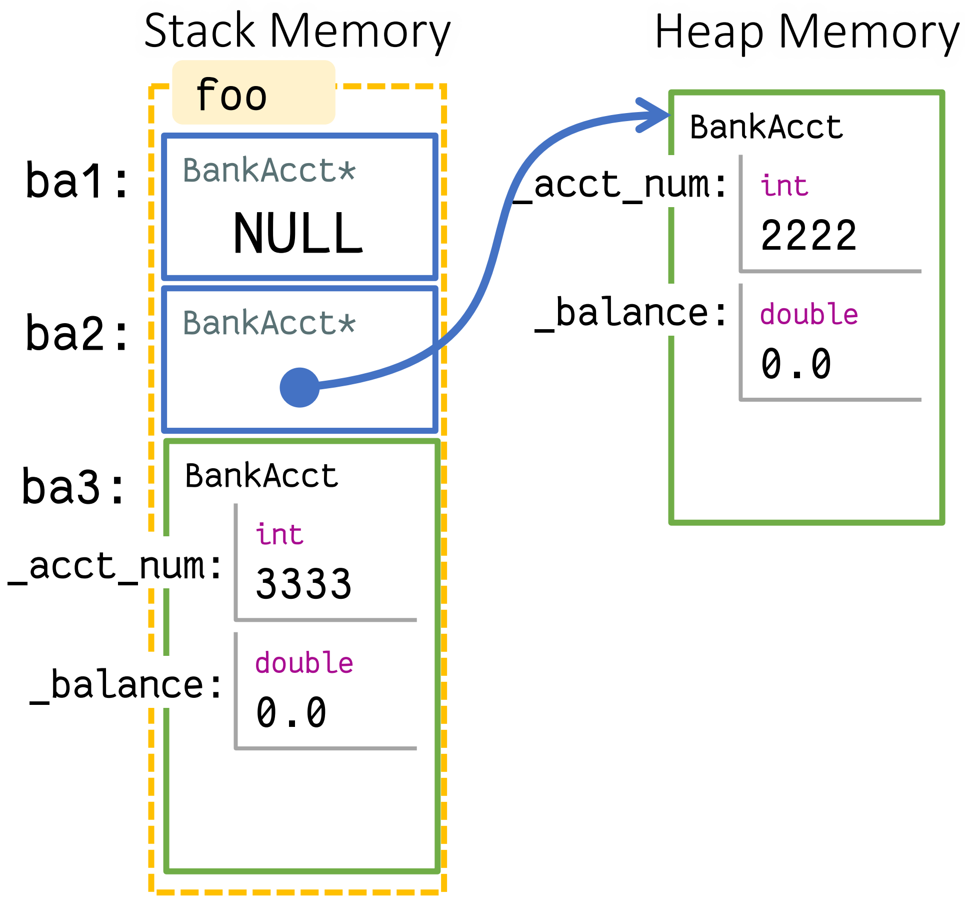
What's the difference in the following ways?

```
void foo() {
   BankAcct *ba1 = NULL;
   BankAcct *ba2 = new BankAcct(2222);
   BankAcct ba3(3333);
   ...



}
```

✓ good practice to set pointer to NULL



Stack Memory

Heap Memory

foo

ba1: BankAcct* NULL

ba2: BankAcct*

ba3: BankAcct

_acct_num: int 3333

_balance: double 0.0

BankAcct

_acct_num: int 2222

_balance: double 0.0

Which method should you use?

Ways to Instantiate Class

What's the difference in the following ways?

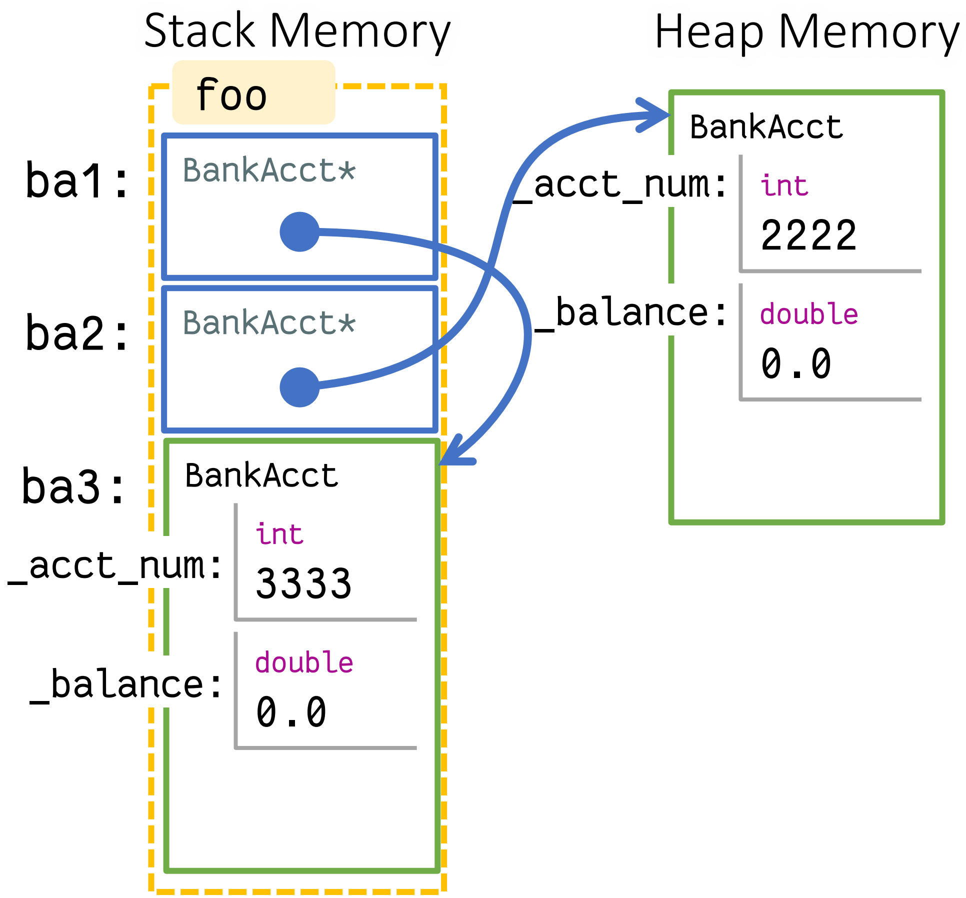
```
void foo() {
   BankAcct *ba1 = NULL;
   BankAcct *ba2 = new BankAcct(2222);
   BankAcct ba3(3333);
   ...
   ba1 = &ba3;

   delete ba2;
}
```

✓ good practice to set pointer to NULL

Pointer can be assigned later

Remember to delete if no longer needed



Which method should you use?

Ways to Instantiate Class

Allocate on stack

- When you do not need object to persist outside of current scope

Allocate on heap

- If you need it to persist beyond current scope
- Be mindful of ownership

- *Note: C++11 has “smart pointers” that take care of deallocation. But that is beyond the scope of the class*

Inheritance

Like father, like son

Inheritance: Motivation

Let's define a Savings Account class

- Data
  - account number
  - balance
  - interest
- Operations
  - deposit
  - withdraw
  - credit_interest

Savings Account

```
class BankAcct SavingsAcct {
private:
  int _acct_num;
  double _balance = 0;
  double _interest = 0;

public:
  // Constructors
  ...
  // Mutators
  virtual bool withdraw(double amt) {
    if (_balance < amt) return false;
      _balance -= amt;
      return true;
  }
```

```
virtual void deposit(double amt) {
  _balance += amt;
}

virtual void credit_interest() {
  _balance *= 1 + _interest;
}

// Accessors
...
```

Savings Account shares > 50% code with Bank Account

– Cut and paste code? 😱 🙅‍♀️

Cut & Paste

DRY principle (Don't Repeat Yourself)

- Hard to maintain
- Need to synchronize all copies (updates or bugfix)

But if classes are independent

- functions that work on one class cannot work on another

```
void transfer(BankAcct &from, BankAcct& to, double amt);
```

- will not work on SavingAcct objects

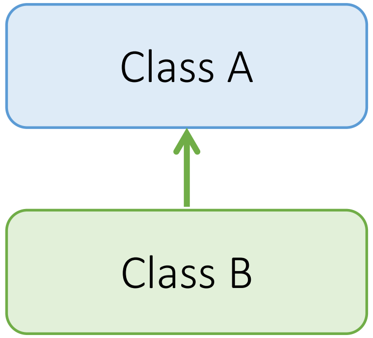
Inheritance

OO langauges allow inheritance

- Classes can be derived from another class
- New class inherits the attributes and methods of the other class

Terminology

- Class B derives from class A
- B is the subclass of A
- A is the superclass of B



Defining a Subclass

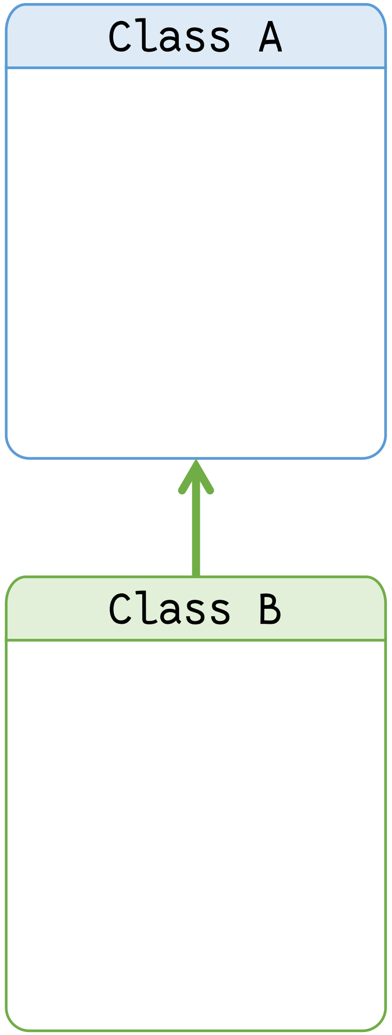
Syntax is as follows

```
class B : public A {
   // class definitions
   ...
}
```

- Indicates that class B will be a subclass of class A

public indicates that this subclassing is made public

- You could also have protected or private subclassing
- But that is beyond the scope of this class



Savings Account with Inheritance

```
class SavingsAcct: public BankAcct {
private:
  double _interest;

public:
  // TODO: constructor

  // TODO: Mutators
};
```

“publicly” indicate inheritance (from superclass)

no declaration for account number and balance

they are “inherited” from superclass

What exactly are we inheriting?

Basically all `public` and `protected`

- Attributes/properties (variables)
- Methods (functions)

Exception: Constructors are not inherited

- Subclass have to define constructor
- Can call the superclass constructor using *initialization list*

```
SavingsAcct(int acct_num, double bal, double interest)
    : BankAcct(acct_num, bal) {
...
}
```

Savings Account with Inheritance

```
class SavingsAcct: public BankAcct {
private:
  double _interest;

public:
  SavingsAcct(int a_num, double bal, double interest)
    : BankAcct(a_num, bal) {
    _interest = interest;
  }

  void credit_interest() {
    _balance += _balance * _interest;
  }
};
```

“publicly” indicate inheritance (from superclass)

no declaration for account number and balance

they are “inherited” from superclass

call superclass constructor using initialization list

Compile Error: ‘double BankAcct::_balance’ is private

Accessibility

public

- Anyone can access
- Typically for methods only

private

- Only instances of the same class can access
- Recommended for all attributes

protected

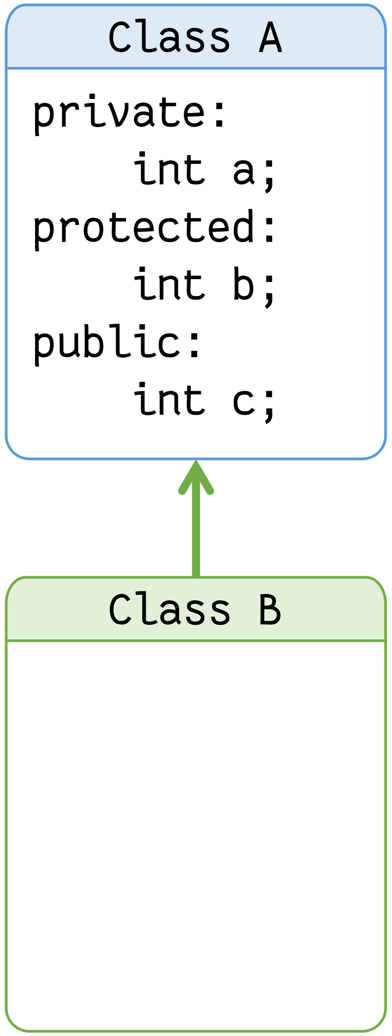
- Only instances of the same class or subclass can access
- For attributes/methods common in a family

Accessibility Example

```
class A {
private:
  int a = 0;
protected:
  int b = 1;
public:
  int c = 2;
};
```

```
class B : public A {
public:
  void f() {
    cout << a;
    cout << b;
    cout << c;
  }
};
```

Error. ‘a’ is private



```
Class A
private:
    int a;
protected:
    int b;
public:
    int c;
```

Accessibility Example

```
class A {
private:
  int a = 0;
protected:
  int b = 1;
public:
  int c = 2;
};
```

```
class B : public A {
public:
  void f()
    cout <<
    cout << b;
    cout << c;
  }
};
```

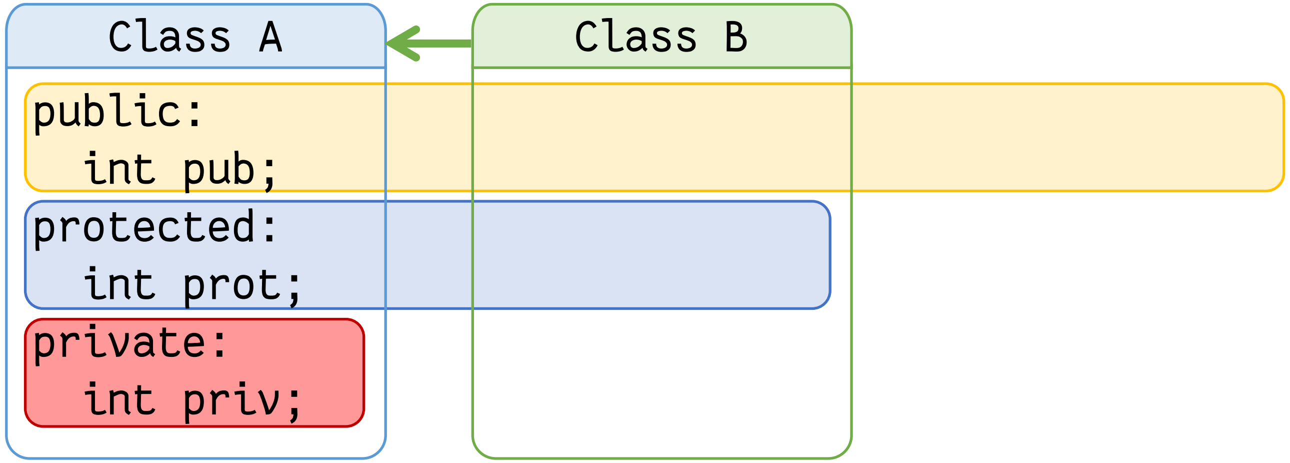
Error. ‘a’ is private

Error. ‘b’ is protected

```
int main() {
  B b;
  cout << b.a;
  cout << b.b;
  cout << b.c;
}
```

https://bit.ly/2UglDkv

Accessibility



public

- Anyone can access
- Typically for methods only

private

- Only instances of the same class can access
- Recommended for all attributes

protected

- Only instances of the same class or subclass can access
- For attributes/methods common in a family

Savings Account with Inheritance

We need to modify BankAcct

```
class BankAcct {
protected:
  int _acc_num;
  double _balance;
  ...
};
```

changed from `private`

- Private properties have to be made protected

Savings Account with Inheritance

Alternatively

```
class SavingsAcct: public BankAcct {
private:
  double _interest;

public:
  ...
  void credit_interest() {
    deposit(get_balance() * _interest);
  }
};
```

calls superclass mutator and accessor

Which way should we use? Protected properties or mutators/accessors

Observations

Inheritance reduces the amount of redundant code

- No redefinition of account number and balance
- No redefinition of withdraw() and deposit()

Access to properties

- Can be given directly using protected
- Or publicly through mutators/accessors

Improved maintainability

- Code in BankAcct remains untouched
- Other programs using BankAcct are not affected
- If for example, withdraw in BankAcct needs to be modified, no change in SavingsAcct is needed

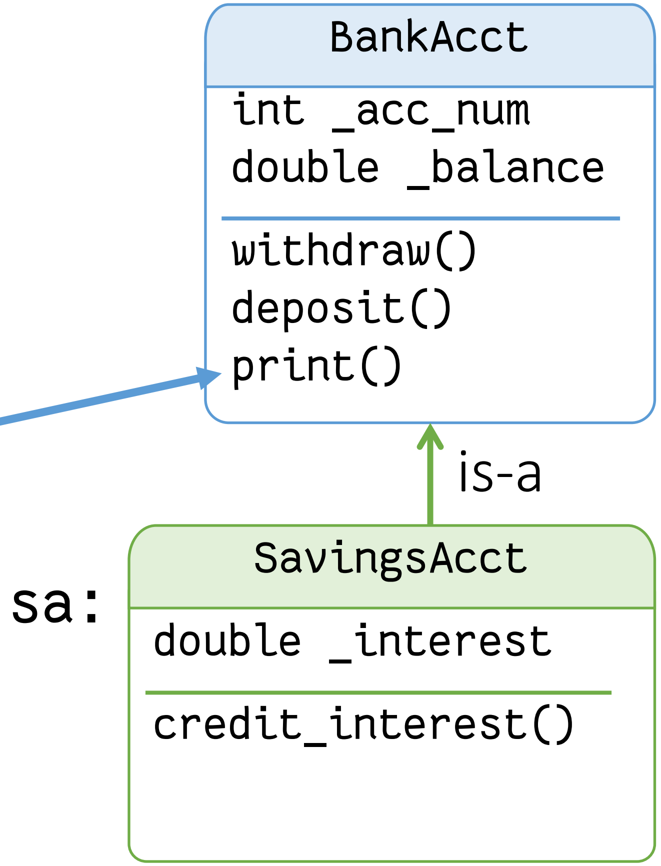
Savings Account: Usage

```
int main() {
  BackAcct ba(1234, 500);
  SavingsAcct sa(8888, 1000, 0.025);

  sa.print();
  sa.deposit(1000);

  sa.credit_interest();
}
```

Assume a method **print()** that displays summary of account



BankAcct

int _acc_num
double _balance

withdraw()
deposit()
print()

is-a

sa:

SavingsAcct

double _interest

credit_interest()

“is-a” relationship

- sa is also a Bank account
- sa has properties and methods of BankAcct class

Method Overriding

Sometimes we want to modify the inherited method

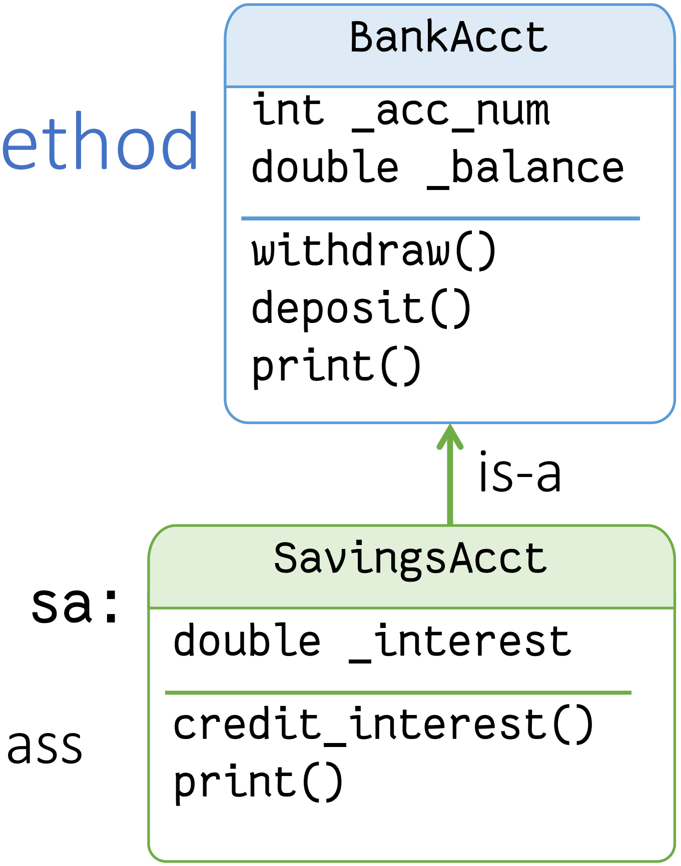
- To change/extend functionality
- This is know as method overriding

Example: Savings Account

- print() method should also print out interest rate

To override

- define a method with the same method header in the subclass
- method header = signature (name + params)



Method Overriding: Example

```
class BankAcct {
  ...
public:
  virtual void print() {
    cout << "Account Number: "
         << _acct_num << endl;
    cout << "Balance: "
         << _bal << endl;
  }
};
```

```
class SavingsAcct : public BankAcct {
  ...
public:
  virtual void print() {
    cout << "Account Number: "
         << _acct_num << endl;
    cout << "Balance: "
         << _bal << endl;
    cout << "Interest: "
         << _interest << endl;
  }
};
```

SavingsAcct's `print()` will be called instead

Duplicate code! 😱🙅

- Can we reuse **BankAcct**'s print()?

Calling Superclass Method

```
class BankAcct {
  ...
public:
  virtual void print() {
    cout << "Account Number: "
         << _acct_num << endl;
    cout << "Balance: "
         << _bal << endl;
  }
};
```

```
class SavingsAcct : public BankAcct {
  ...
public:
  virtual void print() {
    BankAcct::print();
    cout << "Interest: "
         << _interest << endl;
  }
};
```

This will call superclass method 👍

– Yes we can reuse **BankAcct**'s print()?

What happens if we just call `print()` without `BankAcct::`?

Calling Superclass Method

- A non-private superclass method can be called by any subclass
  - Useful for overridden methods
- Syntax

```
superclass_name::method( parameters )
```

Online E-Account

- Charge a fee for withdraws

What properties do we need?

- Fee
- First two withdraws per month free? Need a counter

What methods do we need?

- reset counter?

Who should we inherit?

- BankAcct? SavingsAcct?

E-Account

```
class EAcct
  : public BankAcct {
private:
  double _fee;
  int _counter = 0;

public:
  EAcct(int acct_num,
        double fee)
    : BankAcct(acct_num) {
    _fee = fee;
  }
```

```
bool withdraw(double amt) {
  if (_counter > 1)
    amt += _fee;
  if (BankAcct::withdraw(amt)) {
    _counter += 1;
    return true;
  }
  return false;
}

void reset() { _counter = 0; }
```

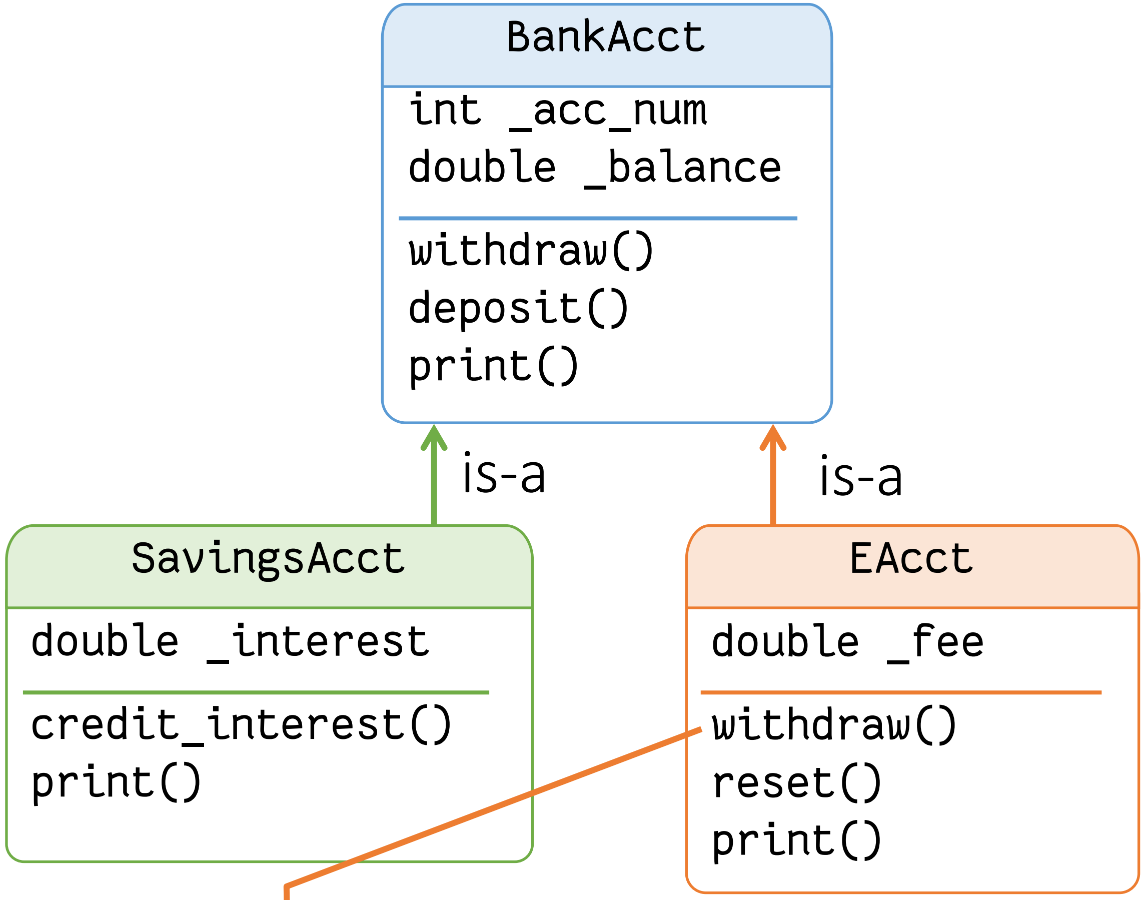
What we have so far

```
EAcct ea(1111, 1.50);

ea.deposit(1000);

ea.withdraw(200);
ea.withdraw(200);
ea.withdraw(200);
ea.print();

ea.withdraw(398.5);
ea.print();
```



```
bool withdraw(double amt) {
  if (_counter > 1)
    amt += _fee;
  if (BankAcct::withdraw(amt)) {
    _counter += 1;
    return true;
  }
  return false;
}
```

Subclass Substitution

When a superclass object is expected

- A subclass is an acceptable substitution
- The converse is NOT true
- Hence, all functions that work with superclass objects, now work with subclass objects, with no modifications!

Analogy

- I have license to drive class 3 vehicles
- A Honda S2000 is-a class 3 vehicle
- Thus, I can drive a Honda S2000

Substitution Example

```
void transfer(BankAcct &from, BankAcct &to, double amt) {
  if (from.withdraw(amt))
    to.deposit(amt);
}

SavingsAcct sa(1234, 1000, 0.01);
BankAcct ba(4321, 1000);

transfer(sa, ba, 500);
```

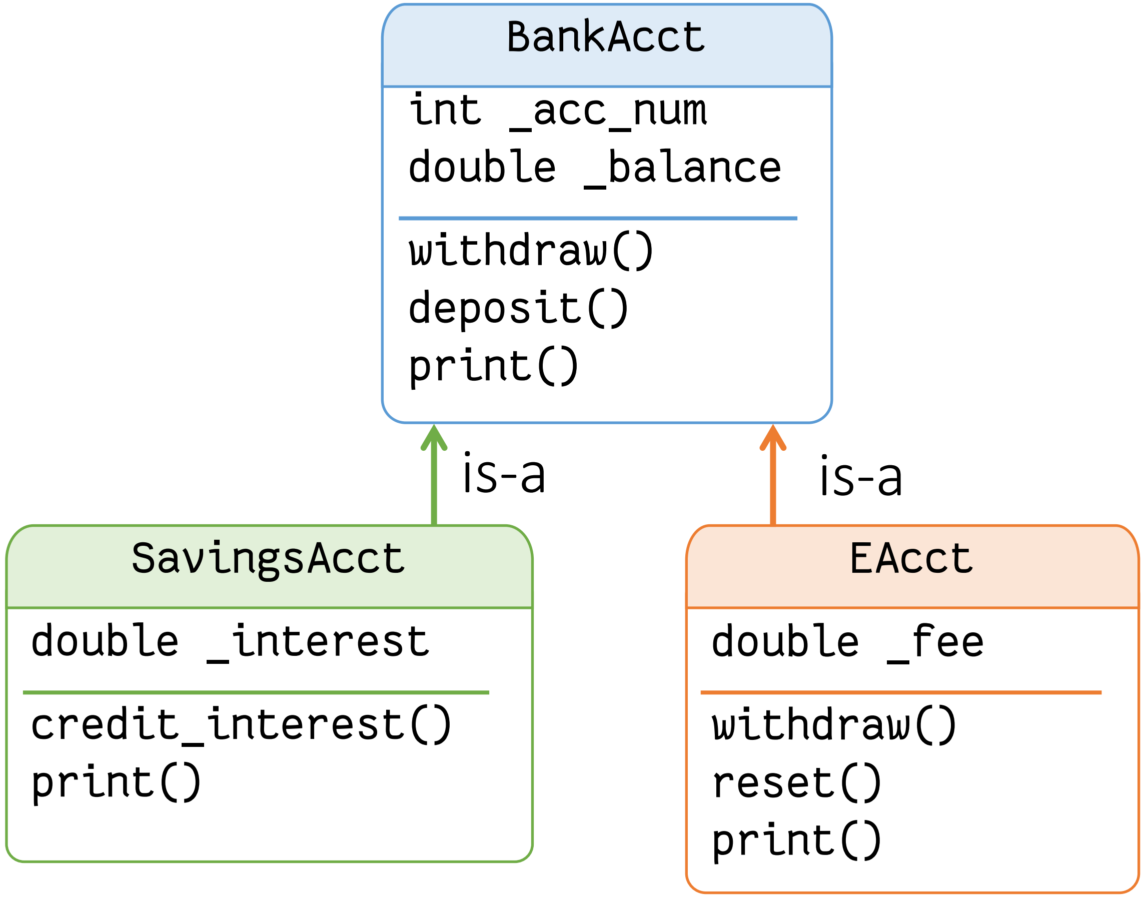
- Transfer accepts SavingsAcct and EAcct as well
- Because inheritance guarantees whatever BankAcct has, its subclasses will have

Examples

```
EAcct ea(1111, 1.50);
SavingsAcct sa(2222);

BankAcct *ba = &ea;
ba->deposit(1000);
ba->withdraw(200);
ba->withdraw(200);

transfer(*ba, sa, 200);
ba.reset();
```



Pitfalls and Rules of Thumb

Beware of

- overusing inheritance
- overusing protected
- make sure it is something inherent for future subclassing

To determine if inheritance is the correct thing to use

- Use the "is-a" rule of thumb
  - If B is-a A sounds right, then B is a subclass of A
- Frequently confused with "has-a" rule
  - If A has-a B sounds right, then A should have a B attribute

Inheritance vs Composition

Inheritance

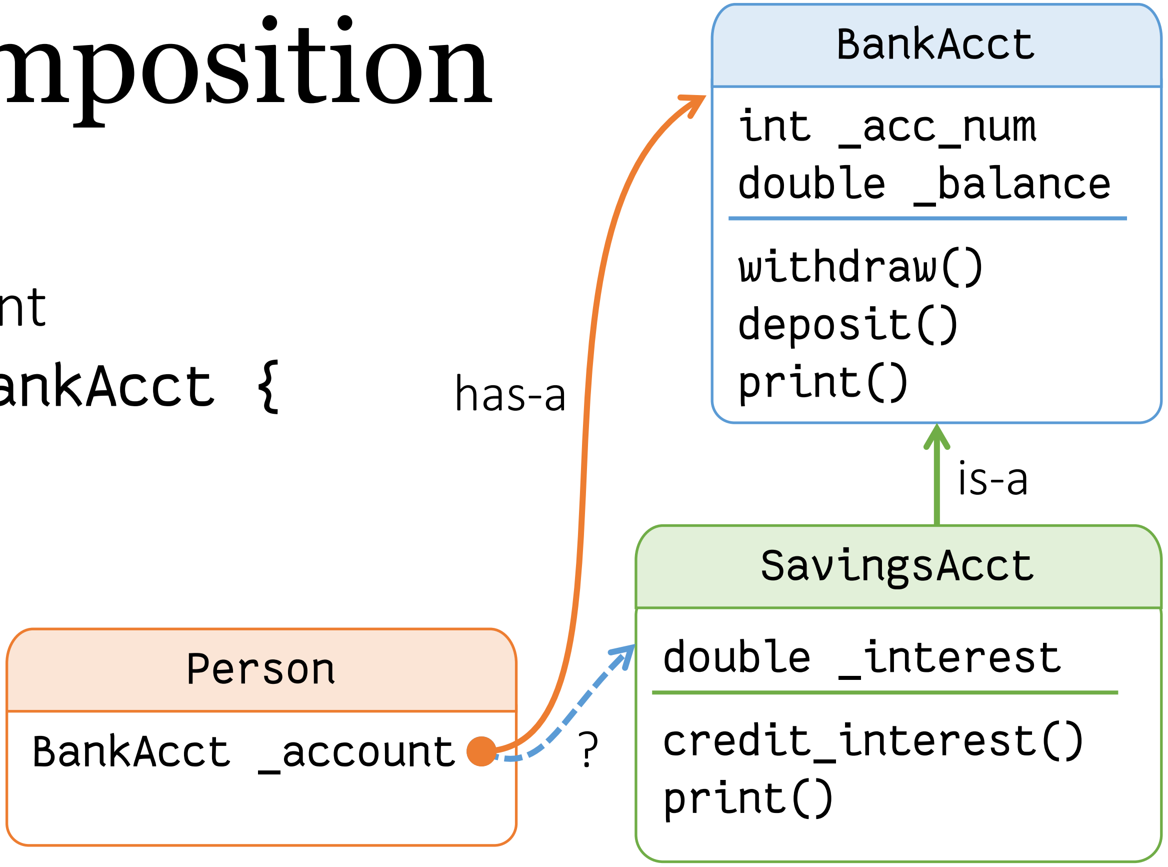
– Savings Account *is a* Bank Account

```
class SavingsAcct: public BankAcct {
  ...
};
```

Composition

– Person *has a* Bank Account

```
class Person {
    BankAcct _account;
};
```



BankAcct

int _acc_num
double _balance

withdraw()
deposit()
print()

has-a

is-a

SavingsAcct

double _interest

credit_interest()
print()

Person

BankAcct _account

?

Can Person have a SavingsAcct instead?

Summary

Inheritance

- Superclass and subclass
- Method overriding
- Subclass substitutability

Supplementary Reading

- Carrano’s Book
  - **C++ Interlude 1** — C++ Classes
  - **C++ Interlude 2** — Pointers, Polymorphism, and Memory Allocation