Lecture 10
Polymorphism

TIC1002 Introduction to Computing and Programming II

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

Polymorphism

Poly = Many; Morphism = Form

Interacting with C++ Objects

There are 3 ways to refer to objects in C++

– Object variable

```
BankAcct ba_var(1234, 100);
```

– Object pointer

```
BankAcct *ba_ptr;
ba_ptr = new BankAcct(5678, 200);
```

– Object reference

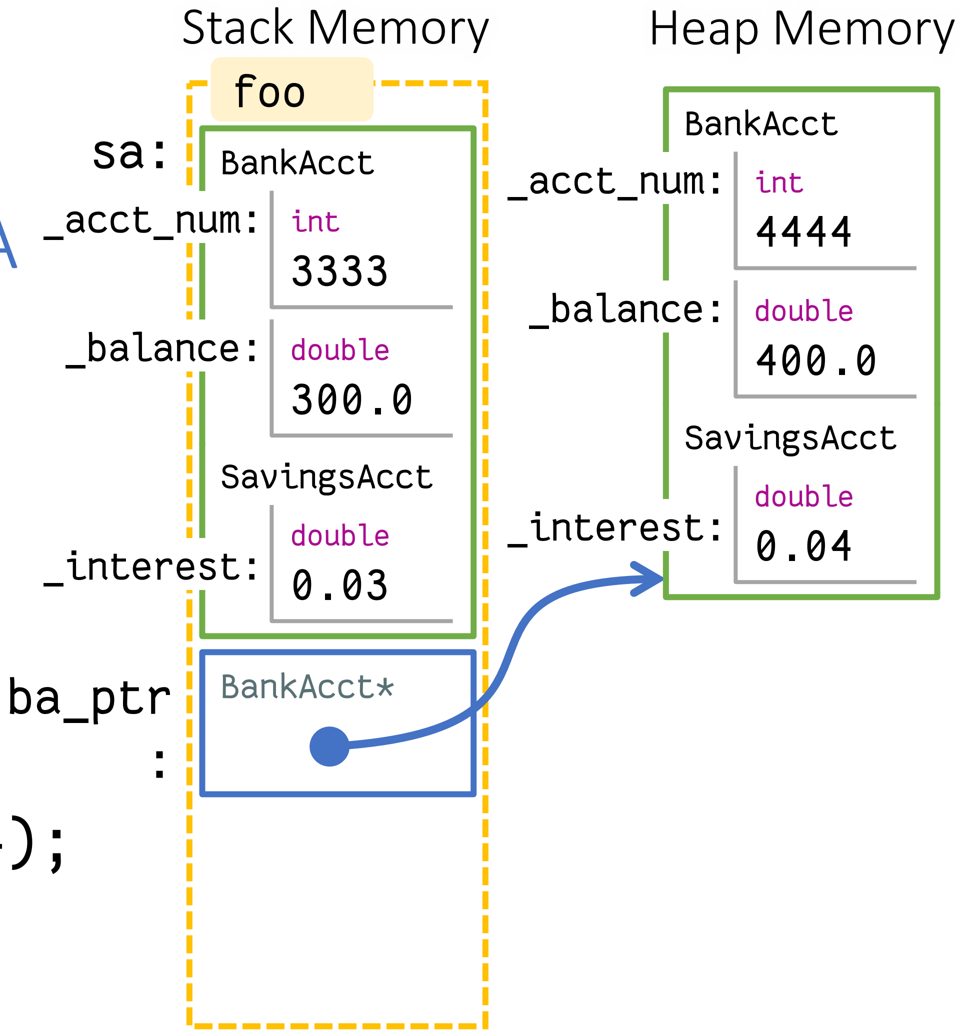
```
BankAcct& ba_ref = ba_var
```

Subclass Substitution

Object pointer and reference of class A

- can refer to objects of class A
- can refer to objects of a subclass of A

```
SavingsAcct sa(3333, 300, 0.03);

BankAcct *ba_ptr;
ba_ptr = new SavingsAcct(4444, 400, 0.04);

BankAcct& ba_ref = sa_var;
```

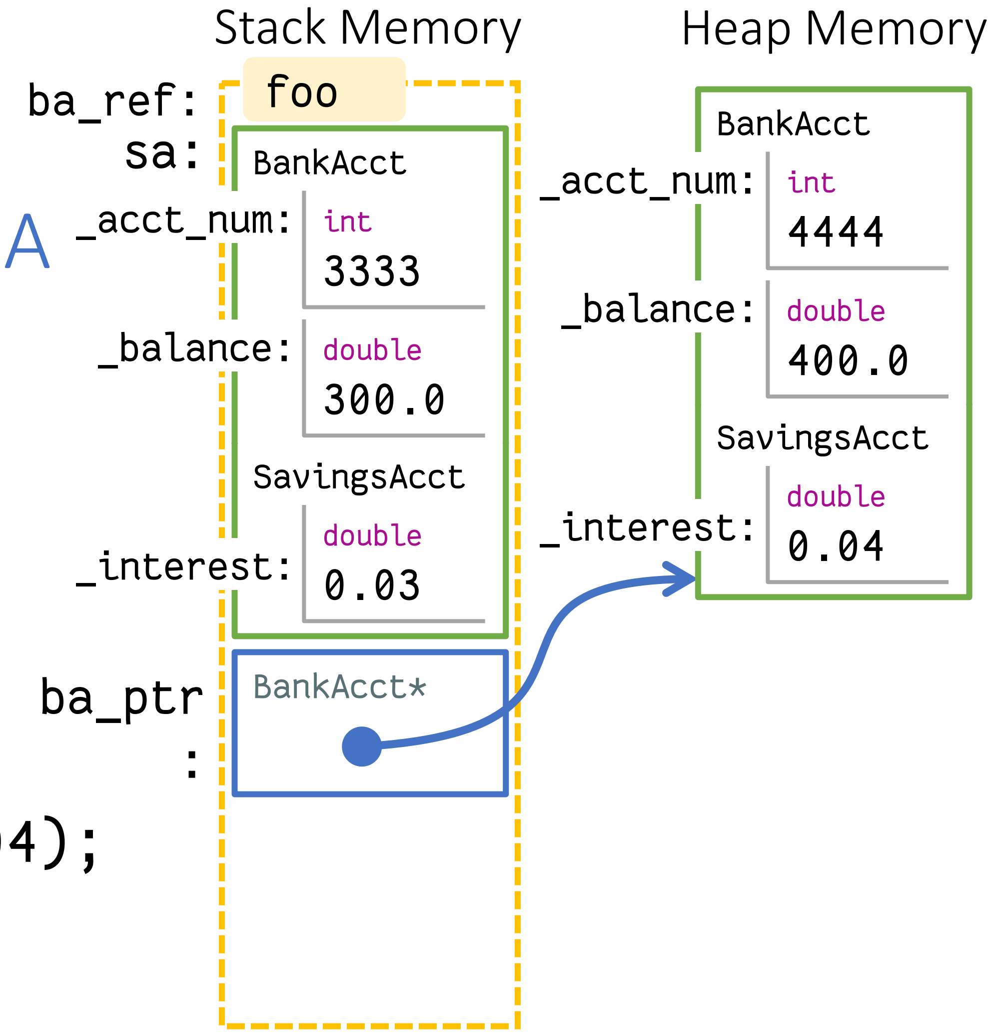


Subclass Substitution

Object pointer and reference of class A

- can refer to objects of class A
- can refer to objects of a subclass of A

```
SavingsAcct sa(3333, 300, 0.03);

BankAcct *ba_ptr;
ba_ptr = new SavingsAcct(4444, 400, 0.04);

BankAcct& ba_ref = sa_var;
```



Polymorphism: Basic Idea

Since we know

1. A superclass pointer/reference can refer to an object of subclass
2. A method implementation can be overridden in subclass, resulting in multiple versions

– Question: Which version of the method will be invoked?

C++ provides two possibilities

– Static binding (Not covered in class)
– Dynamic binding (aka polymorphism)

Most modern programming languages only provide dynamic binding

Static Binding

Not Tested
FYI only

Base on the class type defined by the pointer/ reference

– The information is know during compilation time

```
class BankAcct: {
  void print() {
    // prints acct number and balance
  }
};
```

```
class SavingsAcct: public BankAcct {
  void print() {
    // prints acct number and balance
    // and interest rate
  }
};
```

```
BankAcct *ba_ptr;
ba_ptr = new SavingsAcct(1, 100, 0.01);

ba-ptr->print();
```

This will call BankAcct's print.
Because the type of ba_ptr is BankAcct, so BankAcct's print is invoked.

Dynamic Binding

Not Tested
FYI only

Base on the **actual** class type of the object

– The information is only know during runtime

```
class BankAcct: {
  virtual void print() {
    // prints acct number and balance
  }
};
```

```
class SavingsAcct: {
  void print() {
    // prints acct number and balance
    // and interest rate
  }
};
```

```
BankAcct *ba_ptr;
ba_ptr = new SavingsAcct(1, 100, 0.01);

ba-ptr->print();
```

This will call SavingsAcct's print.
Because ba_ptr points to a SavingsAcct object, so SavingsAcct’s print is invoked.

Dynamic Binding Syntax

virtual keyword is added before method declaration

```
virtual return_type method_name ( parameters )
```

One a method is declared virtual

- It will remain virtual in all descendant classes
- No need to restate the virtual keyword

Because we will only discuss dynamic binding in class

- Just put virtual on all methods
- Assume all previously defined methods are virtual

Substitution Example

Recall function to transfer money

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

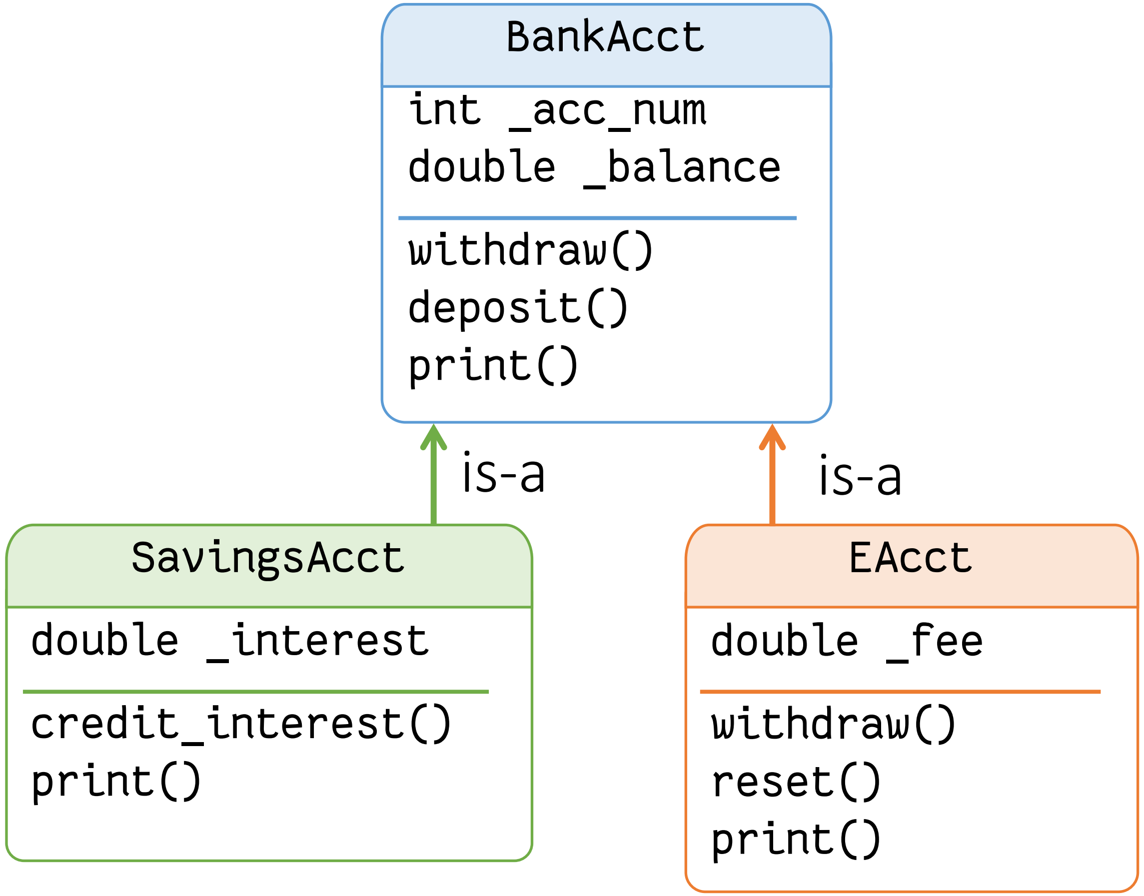
- The arguments from and to are `BankAcct` types

So which method does `from.withdraw()` call?

- With static binding, it will always call `BankAcct.withdraw`
- With dynamic binding, it will depend on what from actually is at runtime

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();
ea.reset()
```



Polymorphism Advantage

Makes code easier to reuse

- Code written to use virtual method of class A, can work with all future subclass of A with no modification
- New behaviour of subclass of A can be incorporated by overriding the virtual method implementation

For example

- Code that uses print in BankAcct can work with all subclasses of BankAcct even when print() is overridden
- Same with withdraw()

Common Mistake

Do not assume that the actual type of the object determines validity of method invocation

```
BankAcct *ba_ptr = SavingsAcct(1, 100, 0,01);
ba_ptr->credit_interest();
```

Type of pointer/reference determines the validity of method invocation

- ba_ptr is type BankAcct
- BankAcct does not have credit_interest() method

Abstract Class

Motivation for Abstract Class

Inheritance and polymorphism

- Gained ability to prepare for future expansion
- Code that works for base class, will work for future subclasses

New design philosophy

- Design a base class that contains all essential methods
- Sometimes this base class is substantial enough to be a normal class
- But, what if you want a base class that is just a placeholder for future subclasses?

Abstract Class Syntax

An abstract method is a method with no definition

- Intended to be overridden in future subclasses
- A.k.a pure virtual method

```
virtual return_type method_name(params) = 0;
```

An abstract class

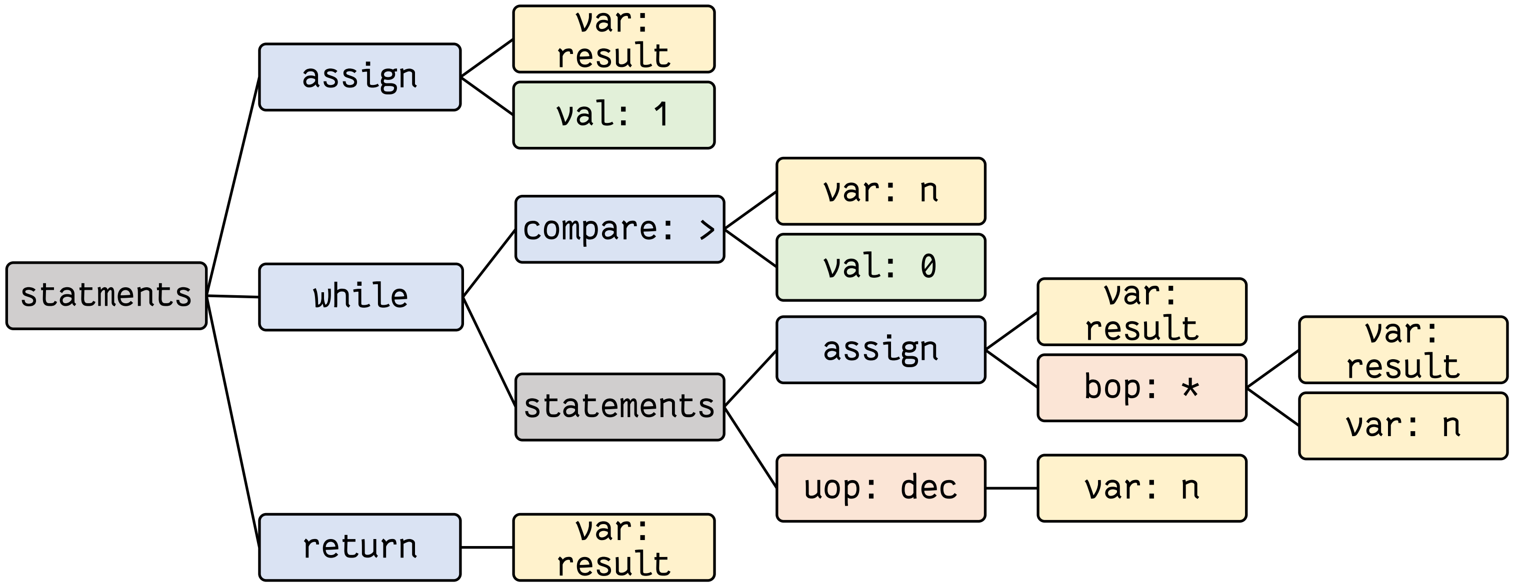
- Has at least one abstract method
- Cannot instantiate an object
- Otherwise similar to normal class definition

Example: Abstract Syntax Tree

What is an Abstract Syntax Tree (AST)

- An intermediate representation of a program/expression
- Using a tree structure

```
int factorial(int n) {
    int result = 1;
    while (n > 0) {
        result *= n;
        n--;
    }
    return result;
}
```



Arithmetic Expressions

A full fledged AST is very complicated

– Let’s limit ourselves to arithmetic expressions

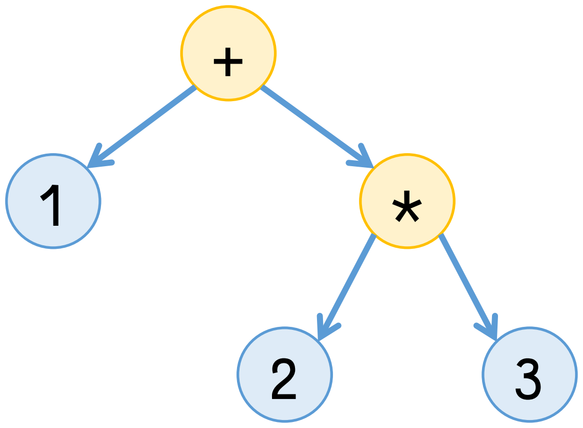
$$1 + 2 * 3$$

– AST for this expression

Different types of nodes

– Operators

– Operands



Representing an AST

Heterogenous Tree

- But C++ only allows homogenous containers
- Create a Token (Problem Set 3)

```
TreeNode<Token> one = {make_token(1)},
                two = {make_token(2)},
              three = {make_token(3)},
                mul = {make_token('*')},
               plus = {make_token('+')};
```

AST: Ye olde way

```
// Build our tree
add_child(one, plus);   //    +     //
add_child(mul, plus);   //   / \    //
                        // 1    *   //
add_child(two, mul);    //     / \  //
add_child(three, mul);  //    2   3 //

cout << evaluate(&plus) << endl;
```

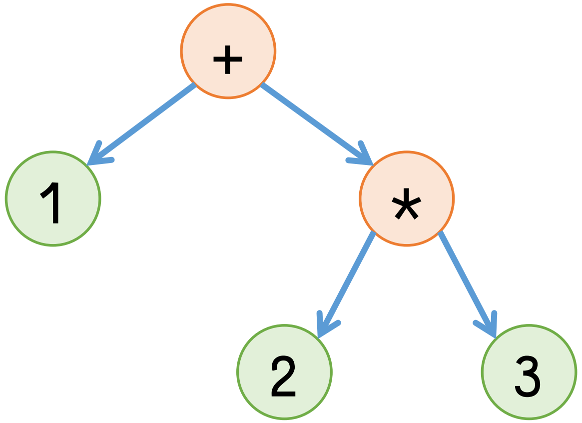
Evaluating ye olde AST

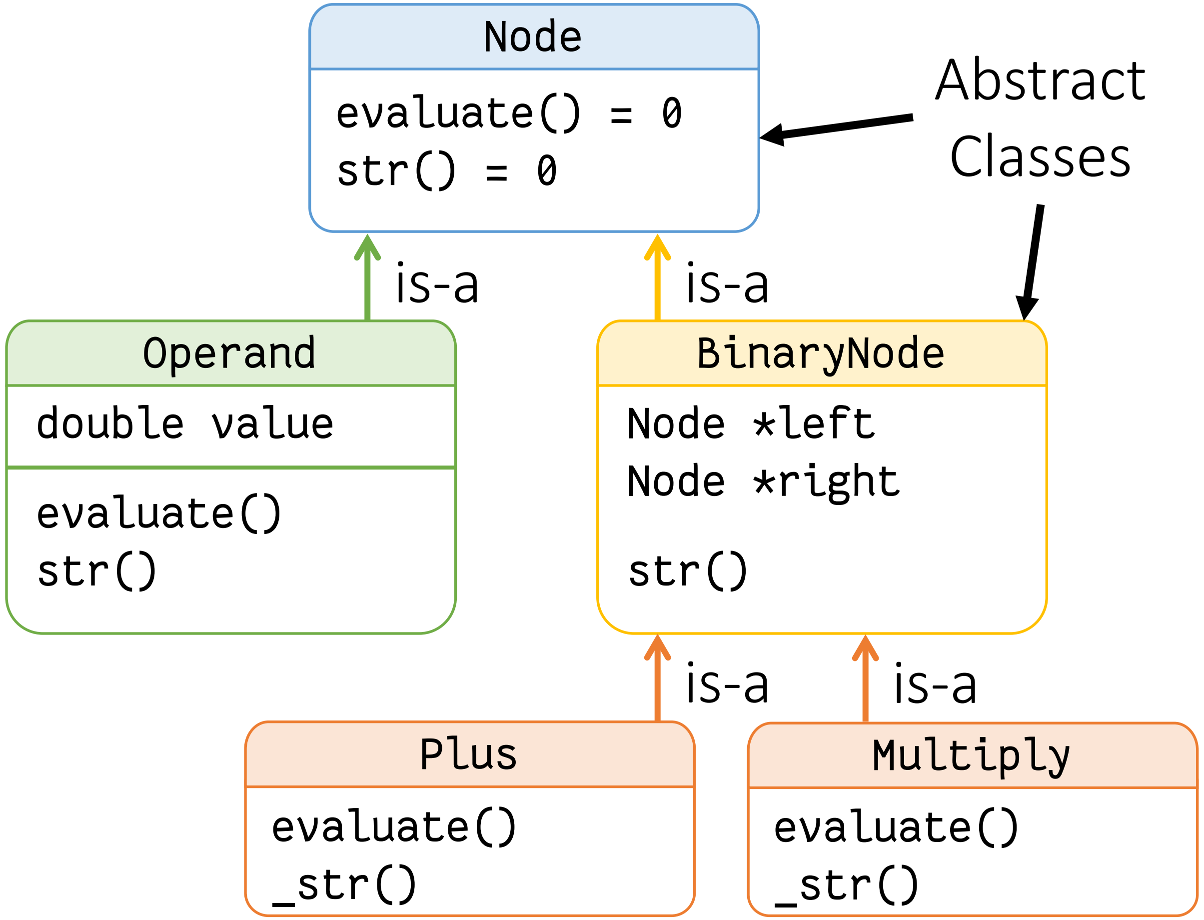
```
double evaluate(TreeNode<Token> *node) {
  Token token = node->item;
  if (is_opnd(token))
    return get_opnd(token);

  switch (get_optr(token)) {
  case '+':
    return evaluate(node->children[0]) + evaluate(node->children[1]);
  case '*':
    return evaluate(node->children[0]) * evaluate(node->children[1]);
  }
}
```

AST: The OOP way

Different types of nodes

- Use inheritance
- All nodes will be same base type





Abstract Base Class

```
class Node {
public:
    virtual double evaluate() = 0;
    virtual string str() = 0;
};
```

Operand Class

```
class Operand : public Node {
double value;

public:
    Operand(double value) {
        this->value = value;
    }

    double evaluate() { return value; }

    string str() { return to_string(value); }
};
```

Abstract Class: BinaryNode

```
class BinaryNode : public Node {
protected:
    Node *left, *right;
    virtual string _str() = 0;

public:
    BinaryNode(Node *left, Node *right) {
        this->left = left;
        this->right = right;
    }

    string str() {
        return "(" + left->str() + _str() + right->str() + ")";
    }
};
```

Operator Nodes

```cpp
class Plus : public BinaryNode {
public:
    // need to implement constructor in C++
    Plus(Node *left, Node *right) : BinaryNode(left, right) {}

    double evaluate() {
        return left->evaluate() + right->evaluate();
    }

    string _str() { return "+"; }
};
```

Operator Nodes

```
class Mulitply : public BinaryNode {
public:
  // need to implement constructor in C++
  Multiply(Node *left, Node *right) : BinaryNode(left, right) {}

  double evaluate() {
      return left->evaluate() * right->evaluate();
  }

  string _str() { return "*"; }
};
```

Using the AST

```
int main() {
    Node *root = new Plus(new Operand(1),
                          new Multiply(new Operand(-2),
                                       new Operand(3)));

    cout << root->evaluate() << endl;

    cout << root->str() << endl;
}
```

Multiple Inheritance

Because one is not enough

Multiple Inheritance

C++ allows classes to subclass more than one class

```
class C: public A, public B { ... }
```

- Order of superclass matters
- Constructors are called left-to-right
- Destructors are called right-to-left

Constructor Order

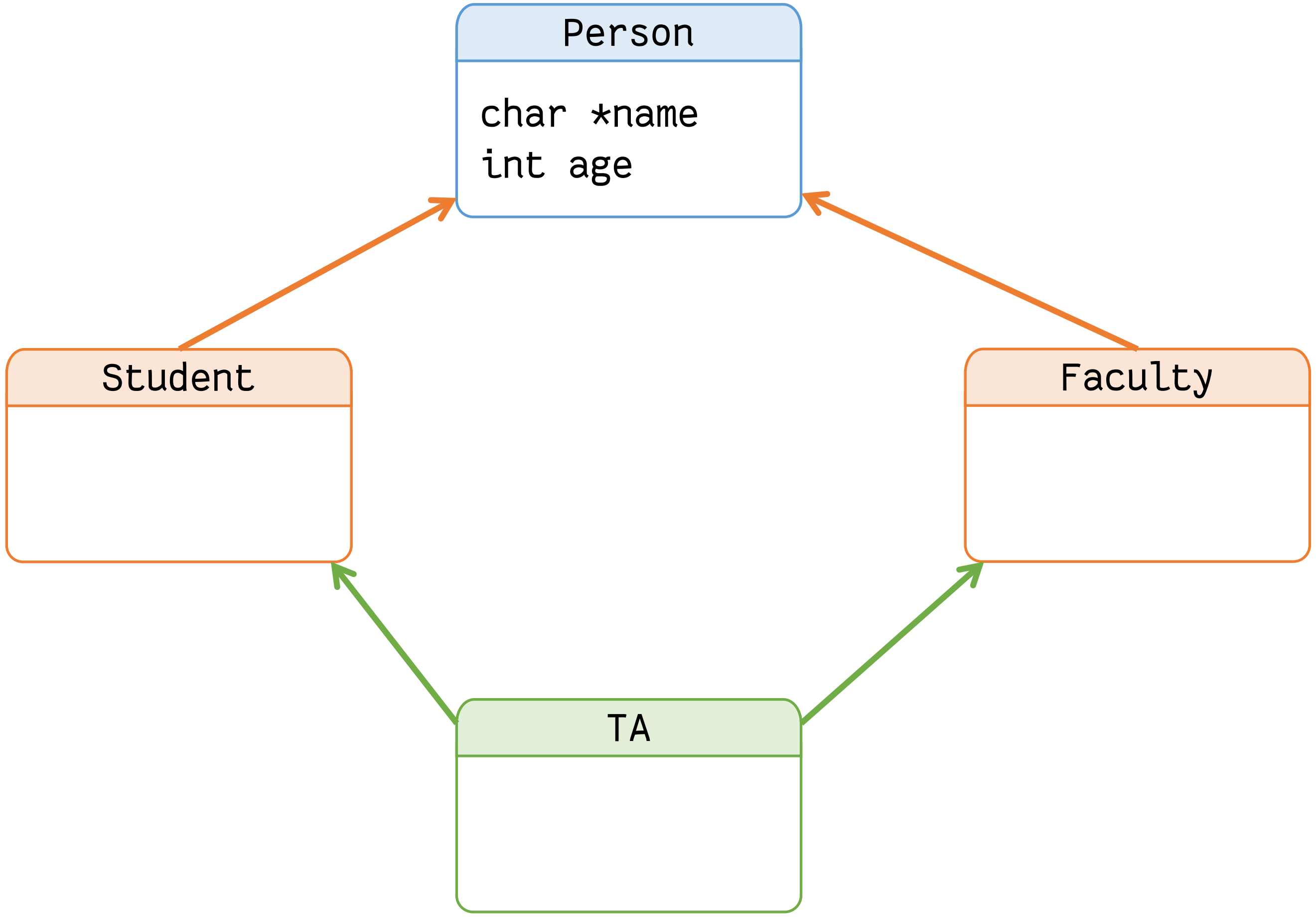
```
class A {
  public: A() { cout << "A const" << endl;
};

class B {
  public: B() { cout << "B const" << endl;
};

class C: public A, public B {
  public: C() { cout << "C const" << endl;
};

int main() {
  C c;
}
```

Output

```
A const
B const
C const
```

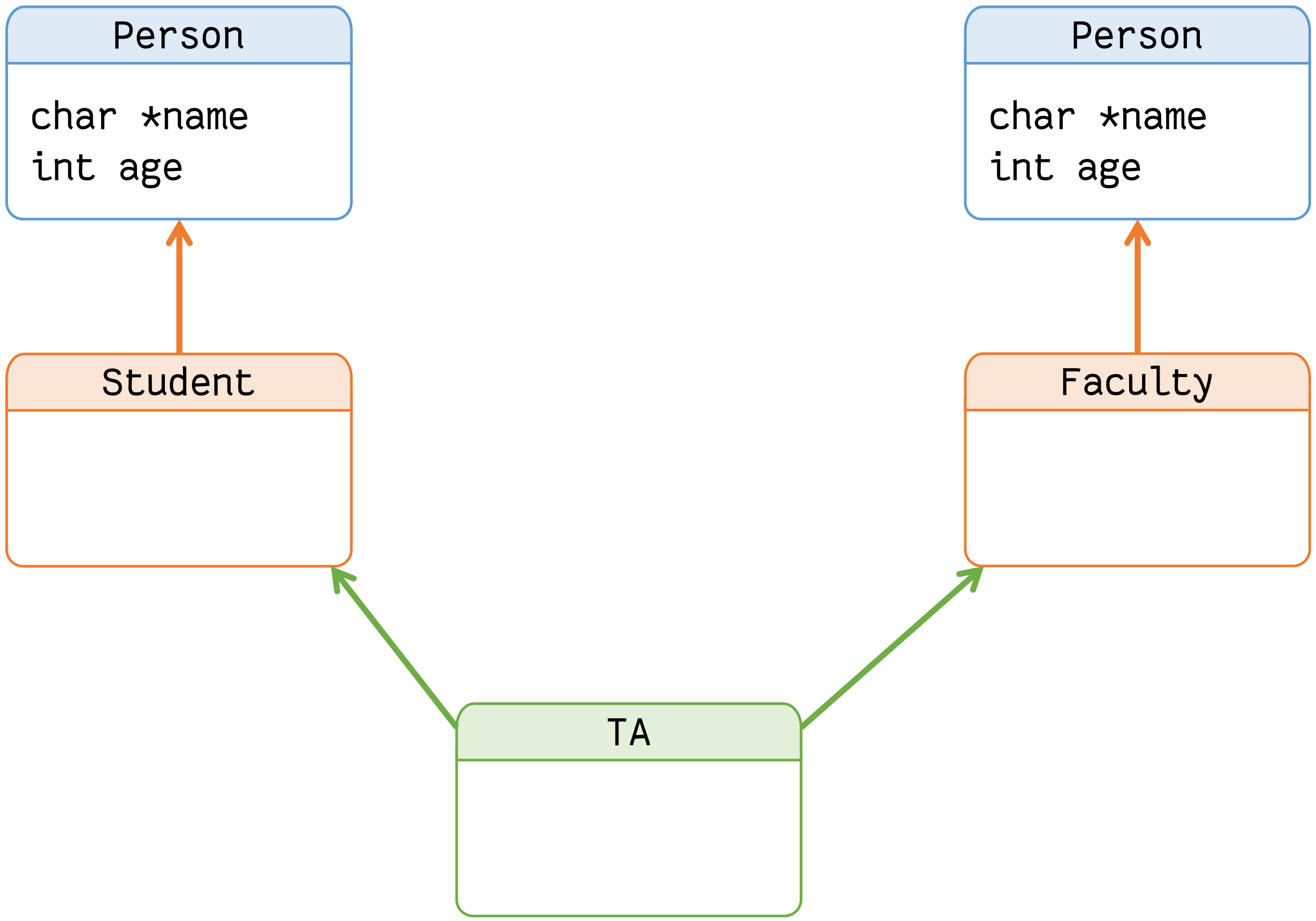
Diamond Problem

```
class TA: public Student, public Faculty { … };
```



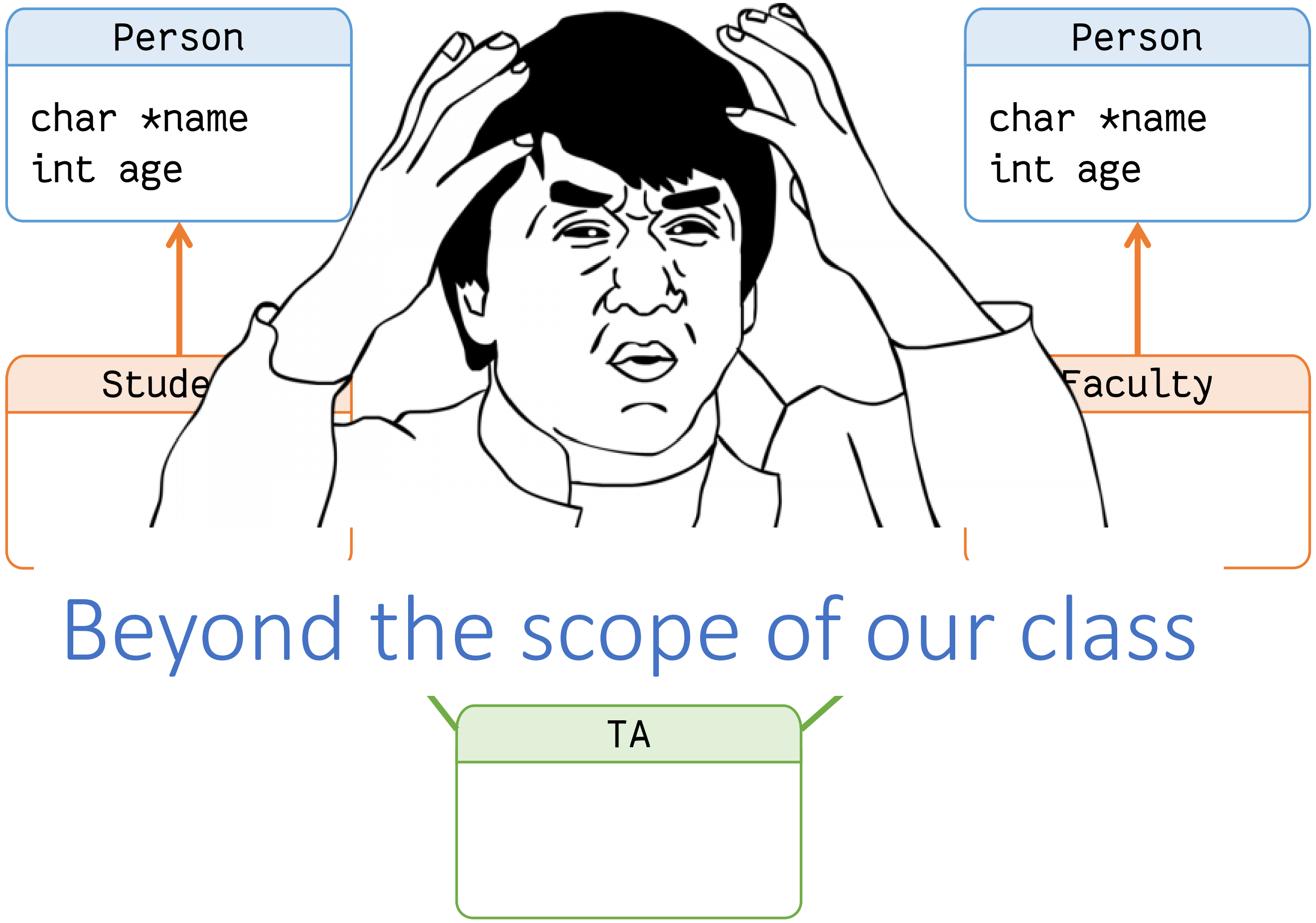
Diamond Problem

C++ will duplicate the superclass



Diamond Problem

C++ will duplicate the superclass



Beyond the scope of our class

Summary of OOP

Encapsulation

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

Inheritance

- Superclass and subclass
- Method overriding
- Subclass substitutability

Polymorphism

- Subclass substitution principle
- Static binding
- Dynamic binding

Supplementary Reading

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