Lecture 11

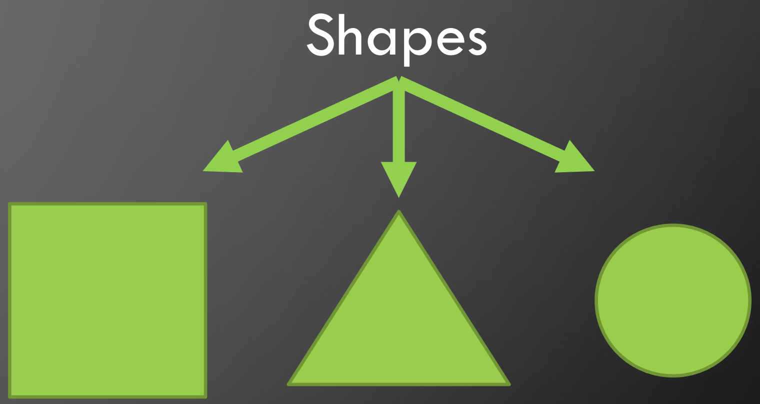
CLASS RELATIONSHIPS

Outline

- Inheritance
- Dynamic Polymorphism
- Composition
- Operator Overloading
- Examples

Inheritance: Building one class from another

- Represents an "is-a" relationship.
- Examples:
  - A circle ***is-a*** shape
  - ***Call Of Duty (COD) is-a*** videogame franchise
  - ECC-1 (Error-Correcting-Codes) ***is-a*** memory error-correcting scheme
  - A bulldog ***is-a*** dog.

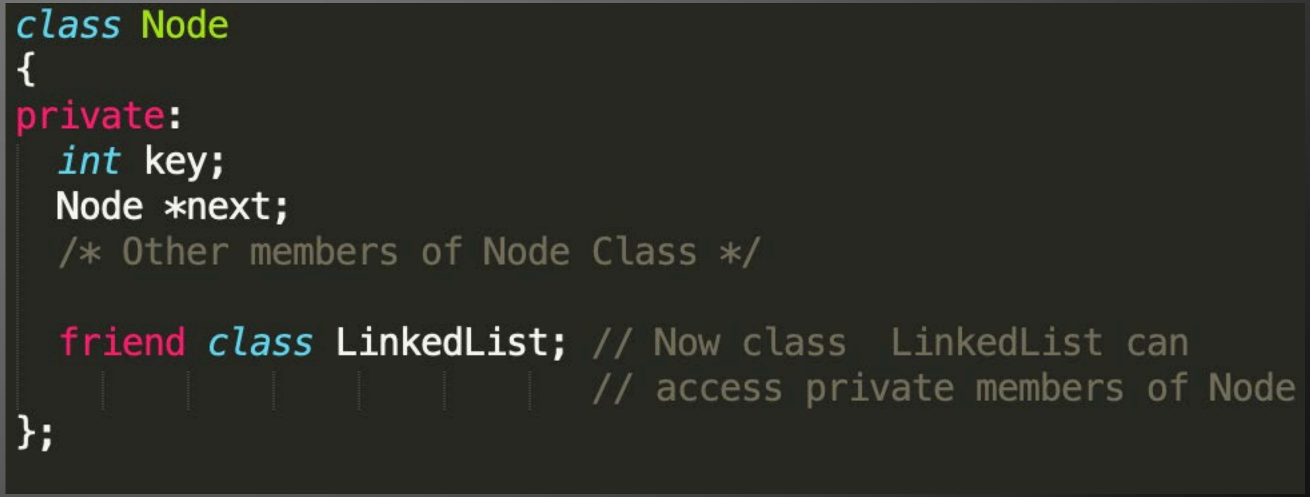


- Syntax:

```
class BaseClass {};

class DerivedClass: public BaseClass {};
```

Friend Class

- Other classes declared as friends can access private members of a class
- Useful for deeply connected classes, such as LinkedList and Nodes



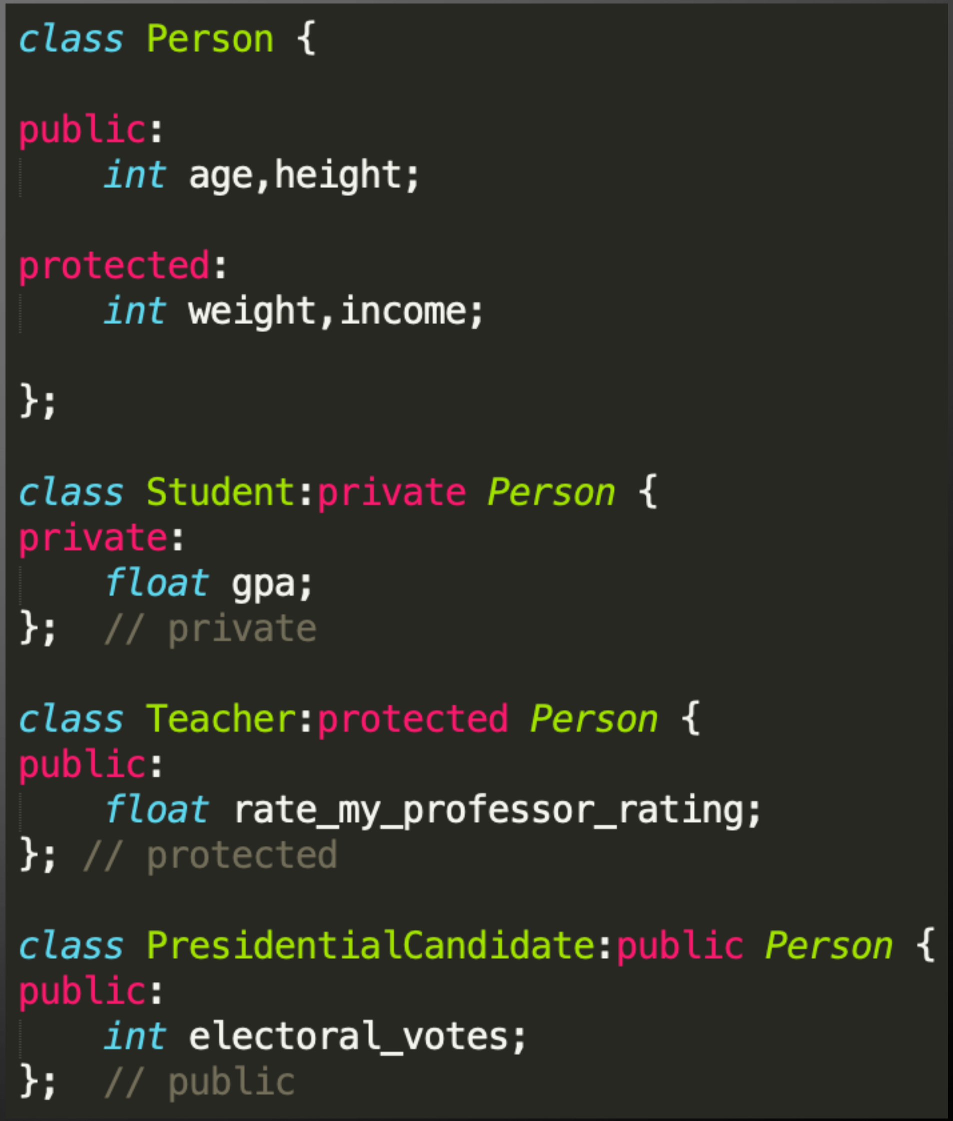
```cpp
class Node
{
private:
  int key;
  Node *next;
  /* Other members of Node Class */

  friend class LinkedList; // Now class  LinkedList can
                           // access private members of Node
};
```

Access: Members (Excluding Friend)

- Private Members
  - Can only be accessed by the original class itself
- Protected Members
  - Can only be accessed in the original class and those derived from it
- Public Members
  - Can be accessed by anything

Kinds of Inheritance

- Public Inheritance
  - Base class member access level doesn’t change in the derived class
- Protected Inheritance
  - Base class public/protected members appear protected when accessing derived class
- Private Inheritance
  - Base class public/protected members appear private when accessing derived class

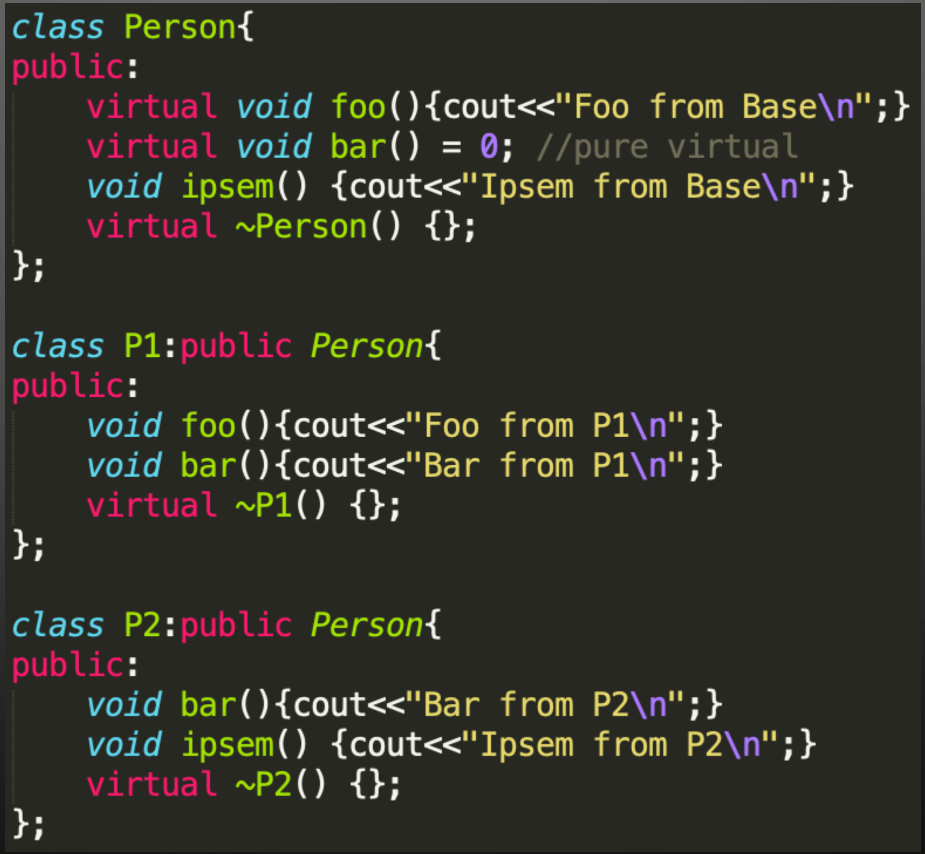


```
class Person {

public:
    int age,height;

protected:
    int weight,income;

};

class Student:private Person {
private:
    float gpa;
};  // private

class Teacher:protected Person {
public:
    float rate_my_professor_rating;
}; // protected

class PresidentialCandidate:public Person {
public:
    int electoral_votes;
};  // public
```

Regardless of the kind of inheritance, a derived class can access all of the base class’s public and protected members, but not its private members.

Virtual Functions

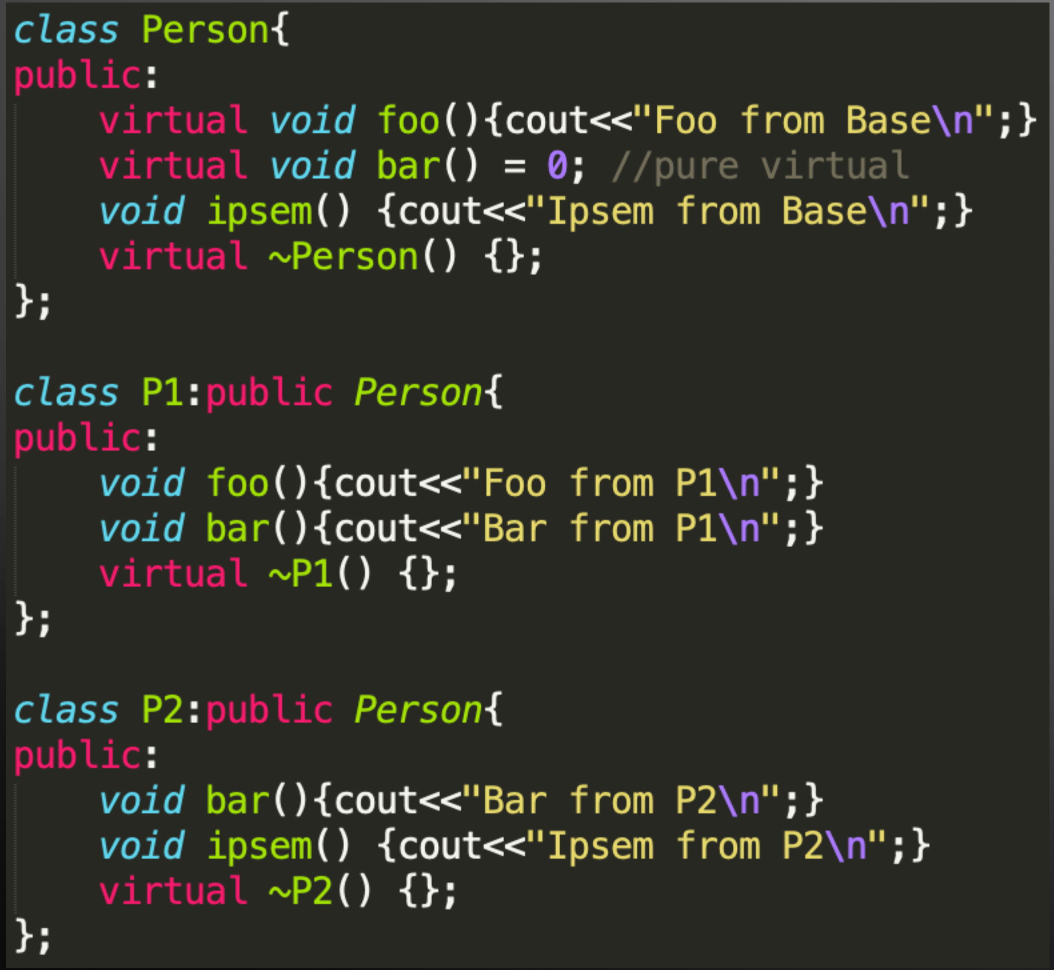
- Virtual functions are called from the base class.
- There are two types of virtual functions:
  - Standard Virtual: *May* be overwritten by derived class
  - Pure Virtual: *Must* be overwritten by derived class



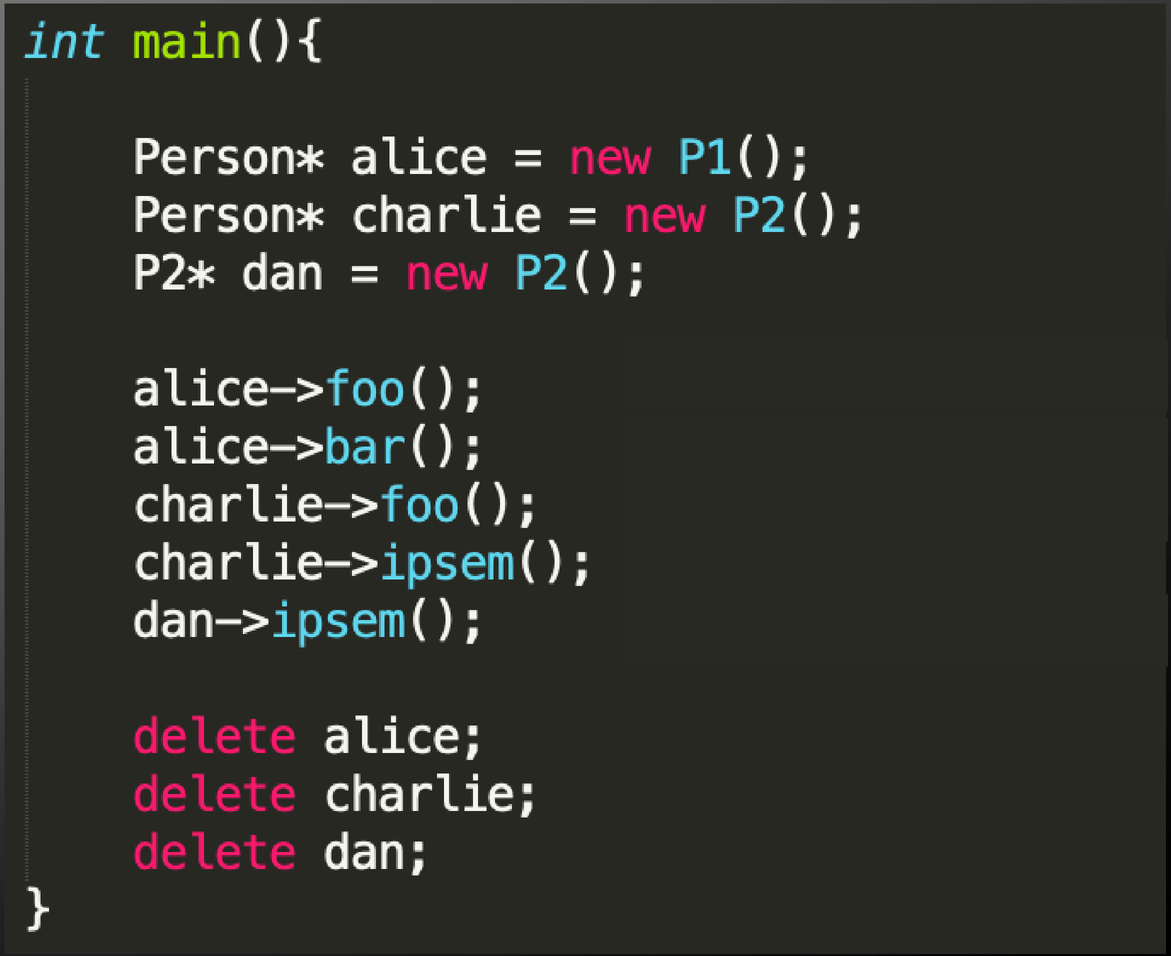
```
class Person{
public:
    virtual void foo(){cout<<"Foo from Base\n";}
    virtual void bar() = 0; //pure virtual
    void ipsem() {cout<<"Ipsem from Base\n";}
    virtual ~Person() {};
};

class P1:public Person{
public:
    void foo(){cout<<"Foo from P1\n";}
    void bar(){cout<<"Bar from P1\n";}
    virtual ~P1() {};
};

class P2:public Person{
public:
    void bar(){cout<<"Bar from P2\n";}
    void ipsem() {cout<<"Ipsem from P2\n";}
    virtual ~P2() {};
};
```

Virtual Functions: Example

- What do each of these functions in the right example print?



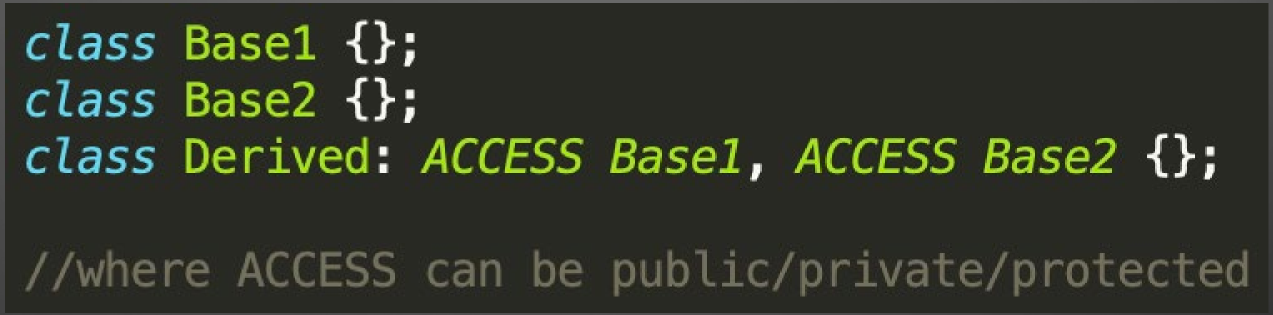
```cpp
class Person{
public:
    virtual void foo(){cout<<"Foo from Base\n";}
    virtual void bar() = 0; //pure virtual
    void ipsem() {cout<<"Ipsem from Base\n";}
    virtual ~Person() {};
};

class P1:public Person{
public:
    void foo(){cout<<"Foo from P1\n";}
    void bar(){cout<<"Bar from P1\n";}
    virtual ~P1() {};
};

class P2:public Person{
public:
    void bar(){cout<<"Bar from P2\n";}
    void ipsem() {cout<<"Ipsem from P2\n";}
    virtual ~P2() {};
};
```



```cpp
int main(){

    Person* alice = new P1();
    Person* charlie = new P2();
    P2* dan = new P2();

    alice->foo();
    alice->bar();
    charlie->foo();
    charlie->ipsem();
    dan->ipsem();

    delete alice;
    delete charlie;
    delete dan;
}
```

Multiple Inheritance

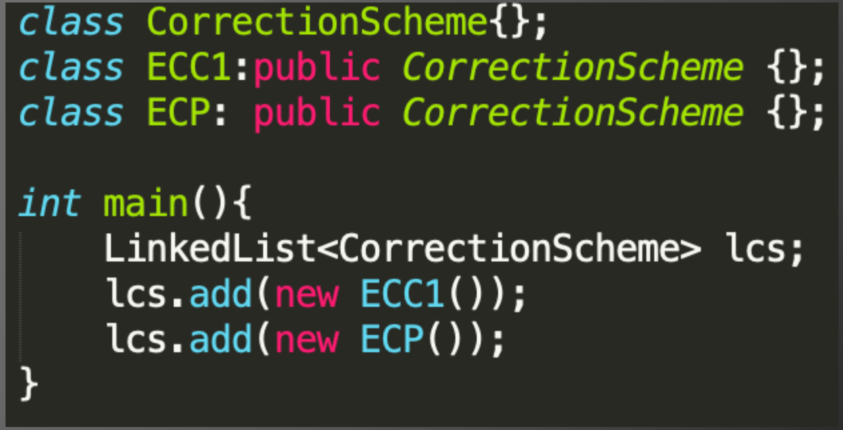
- Can inherit from multiple classes
  - Can cause issues with conflicts in function/member names
  - Most useful when using one or all base classes as an *abstract interface*



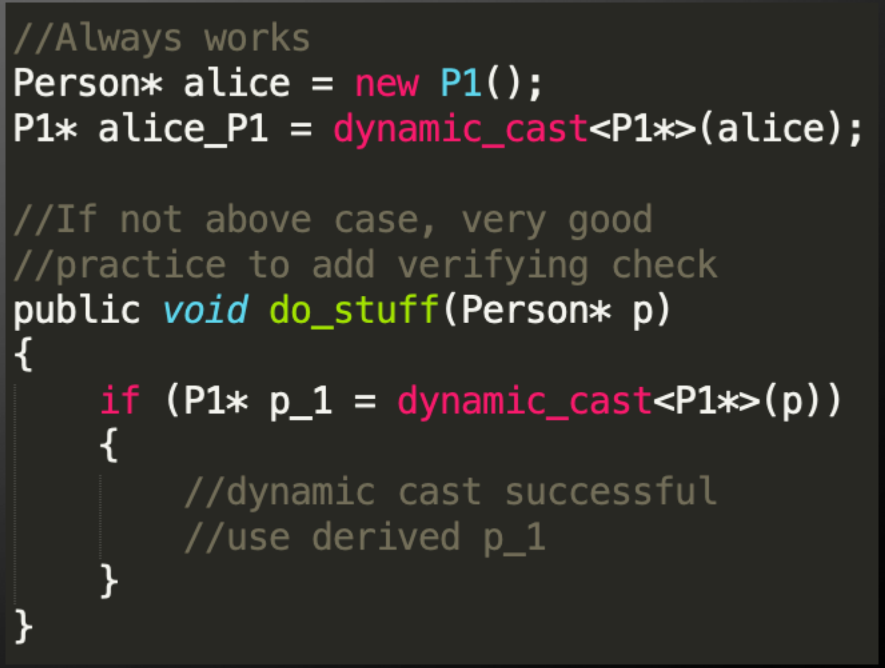
```
class Base1 {};
class Base2 {};
class Derived: ACCESS Base1, ACCESS Base2 {};

//where ACCESS can be public/private/protected
```

Good Uses of Inheritance

- Heterogeneous Collections
  - Have containers of mixed type which share same base class
  - Also useful to add future derived class implementations
- Dynamic Casting
  - Used to convert from base class to derived class, or from one derived class to another (from the same base class)
  - Very useful for hierarchical data



```
class CorrectionScheme{};
class ECC1:public CorrectionScheme {};
class ECP: public CorrectionScheme {};

int main(){
    LinkedList<CorrectionScheme> lcs;
    lcs.add(new ECC1());
    lcs.add(new ECP());
}
```



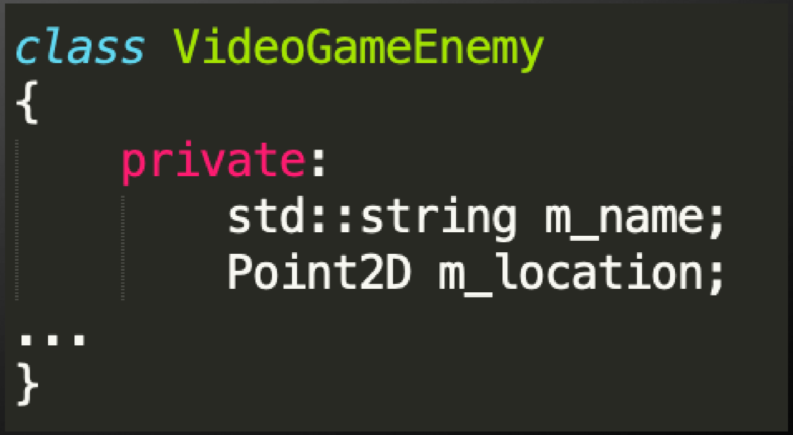
```
//Always works
Person* alice = new P1();
P1* alice_P1 = dynamic_cast<P1*>(alice);

//If not above case, very good
//practice to add verifying check
public void do_stuff(Person* p)
{
    if (P1* p_1 = dynamic_cast<P1*>(p))
    {
        //dynamic cast successful
        //use derived p_1
    }
}
```

Dynamic Polymorphism Pitfalls

- Make sure your inheritance structure is clean
    - Nebulous inheritance often leads to unintended consequences
- Do you really need inheritance in your code?
    - Does list derive from bag? Does list derive from node? Does tree derive from list?
        - These things seem related, but maybe they’re not sufficiently for creating inheritance relations
- Excessive casting
    - Are you constantly casting between base and derived classes?
    - Why do you need to do this?

Composition

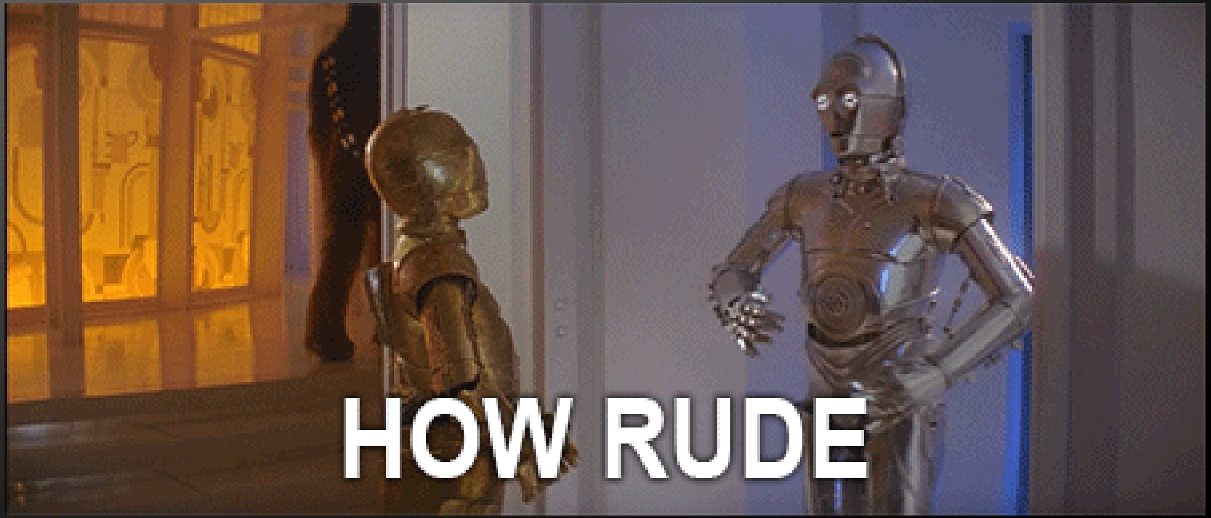
- Represents the “has-a” relationship
  - A person has-a name
  - A person has-a(n) id
  - A person has-a(n) age
- Compared to inheritance, composition is:
  - Looser coupling
  - Shorter compile times
- Example:
  - VideoGameEnemy *has-a* Point2D



```
class VideoGameEnemy
{
    private:
        std::string m_name;
        Point2D m_location;
...
}
```

Composition Example: People and Employees

- A Person has a:
  - Name
  - Age
  - Address
- An Employee is a Person and has a:
  - ID
  - Role
  - Salary
- What about a customer?
  - Is a customer always a person?



What is Operator Overloading?

- Recall copy assignment operator (aka, "operator=“).
- For some type T:
  - T a, b;
  - b = a;
- This is translated by the compiler to b.operator=(a);

What can be overloaded?

- Arithmetic operators: =, +, -, *, /, ++, --, %
- Comparison/relational operators: ==, !=, >, <, >=, <=
- Logical operators: ! && ||
- Bitwise operators: ~, &, |, ^, <<, >>
- Compound assignment operators: +=, -=, *=, etc
- Member/Pointer Operators: [], dereferences, address operator
- Function calls, new, delete, many many others

What should be overloaded? -> only things that make sense for your class/struct/enum

Common Sense for Operator Overloading

1. The overload should do something that makes sense for someone else reading the code.
2. One of the most common uses is to be compatible with existing functions which use common arithmetic and comparison operators (for example, sorting libraries)
3. For comparison operators, if you define one, you should define all of them (==, !=, >, <, >=, <=)

Internal vs External Overloading

Assume you have classes A, B, T. You can define B = T + A using either:

1. Internal Overloading:
    - B T::operator+(A rhs)
2. External Overloading:
    - B operator+(T lhs, A rhs)

Internal overloading can use private member variables, while external overloading cannot (unless declared as a friend).

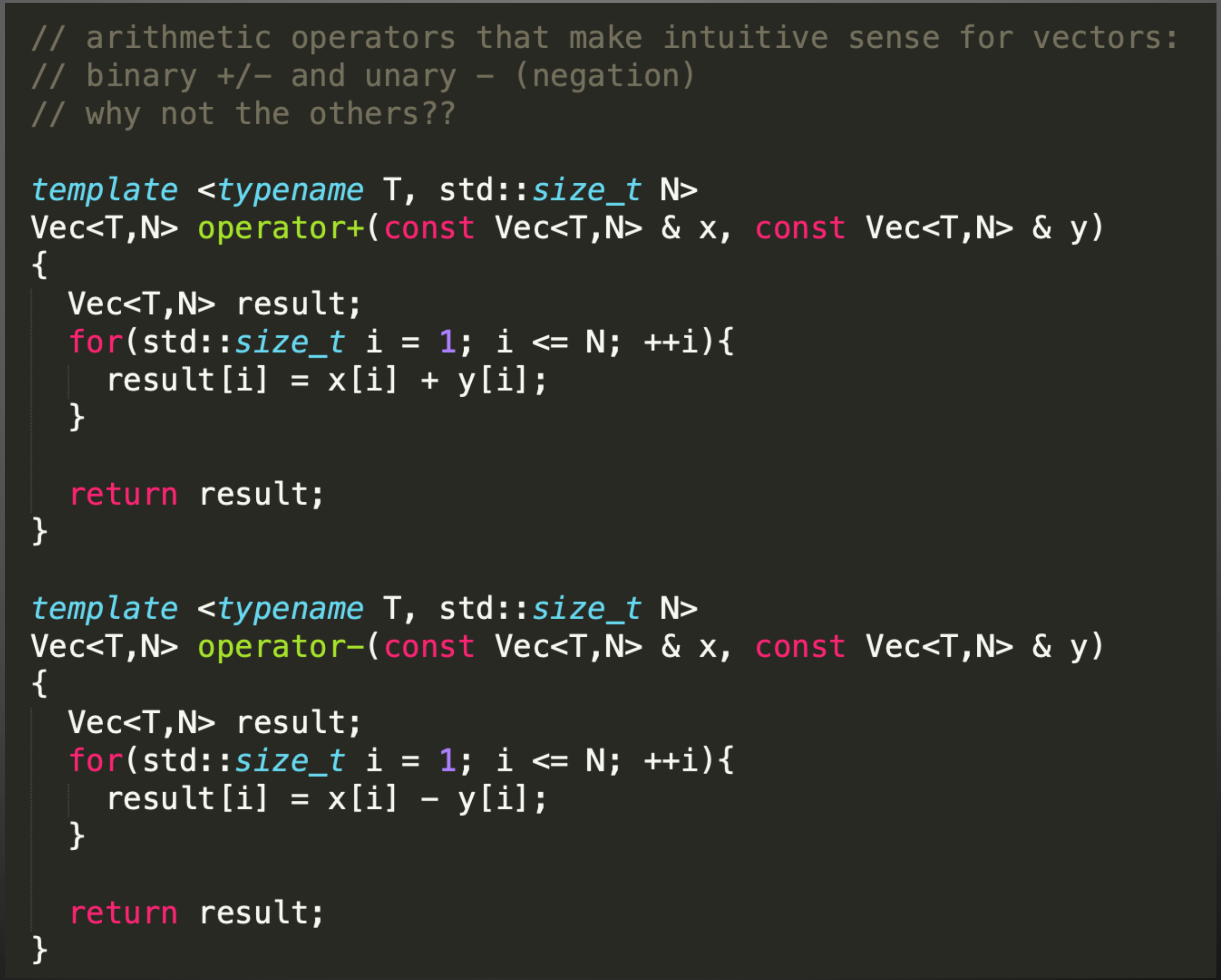
Example:

Students s; Professors p;

People peeps = s + p;

- This calls either an internal or externally overloaded operator+

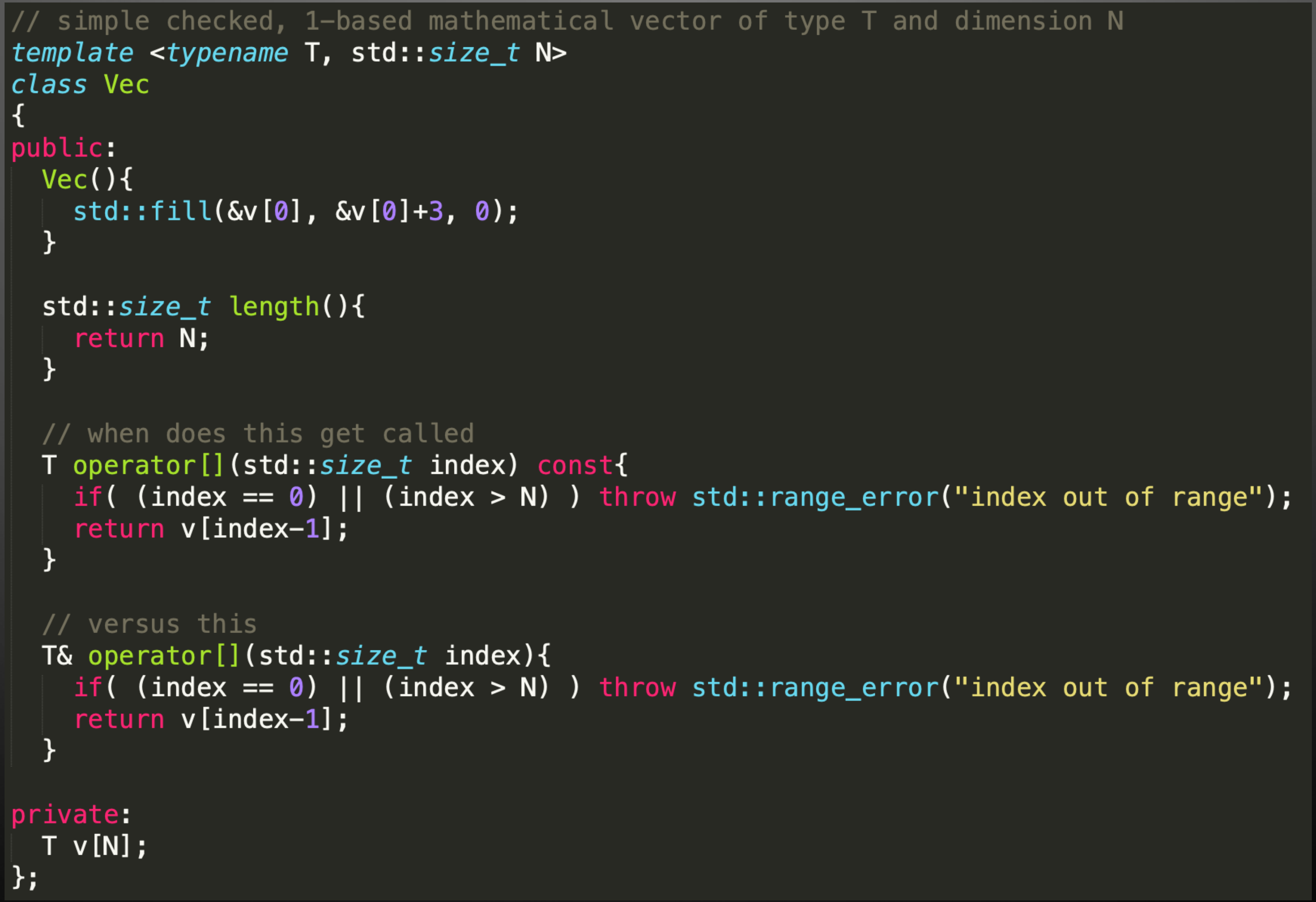
Arithmetic Overloading Example



```
// arithmetic operators that make intuitive sense for vectors:
// binary +/- and unary - (negation)
// why not the others??

template <typename T, std::size_t N>
Vec<T,N> operator+(const Vec<T,N> & x, const Vec<T,N> & y)
{
  Vec<T,N> result;
  for(std::size_t i = 1; i <= N; ++i){
    result[i] = x[i] + y[i];
  }

  return result;
}

template <typename T, std::size_t N>
Vec<T,N> operator-(const Vec<T,N> & x, const Vec<T,N> & y)
{
  Vec<T,N> result;
  for(std::size_t i = 1; i <= N; ++i){
    result[i] = x[i] - y[i];
  }

  return result;
}
```

Subscript Operator

- Useful for classes which represent indexable items, such as lists



```cpp
// simple checked, 1-based mathematical vector of type T and dimension N
template <typename T, std::size_t N>
class Vec
{
public:
  Vec(){
    std::fill(&v[0], &v[0]+3, 0);
  }

  std::size_t length(){
    return N;
  }

  // when does this get called
  T operator[](std::size_t index) const{
    if( (index == 0) || (index > N) ) throw std::range_error("index out of range");
    return v[index-1];
  }

  // versus this
  T& operator[](std::size_t index){
    if( (index == 0) || (index > N) ) throw std::range_error("index out of range");
    return v[index-1];
  }

private:
  T v[N];
};
```

Assignment/Homework

- HW4 due on Today
- Midterm this Thursday during class.