Polymorphism

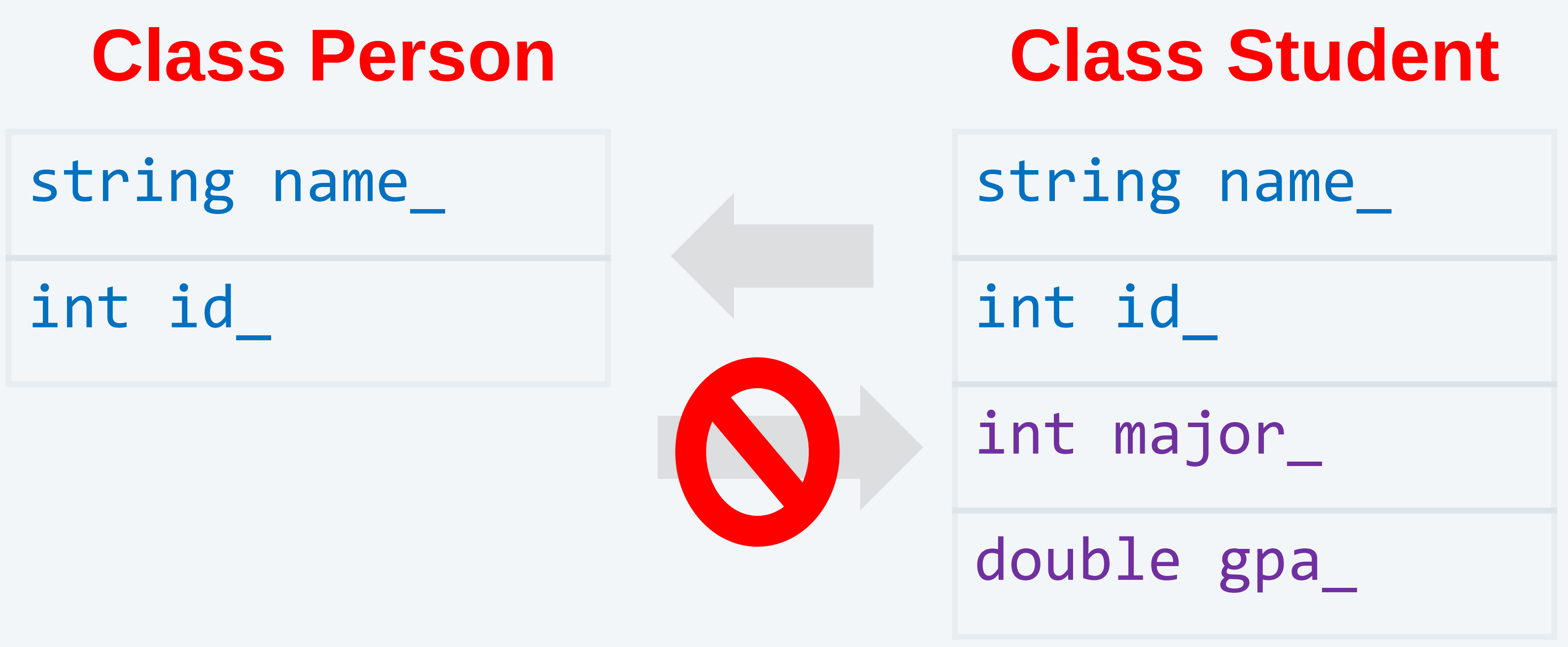
Sandra Batista, Mark Redekopp, and David Kempe

1.1-1.2

Assignment of Derived Classes to Base cases

A class derived via public inheritance can be assigned to its base class.

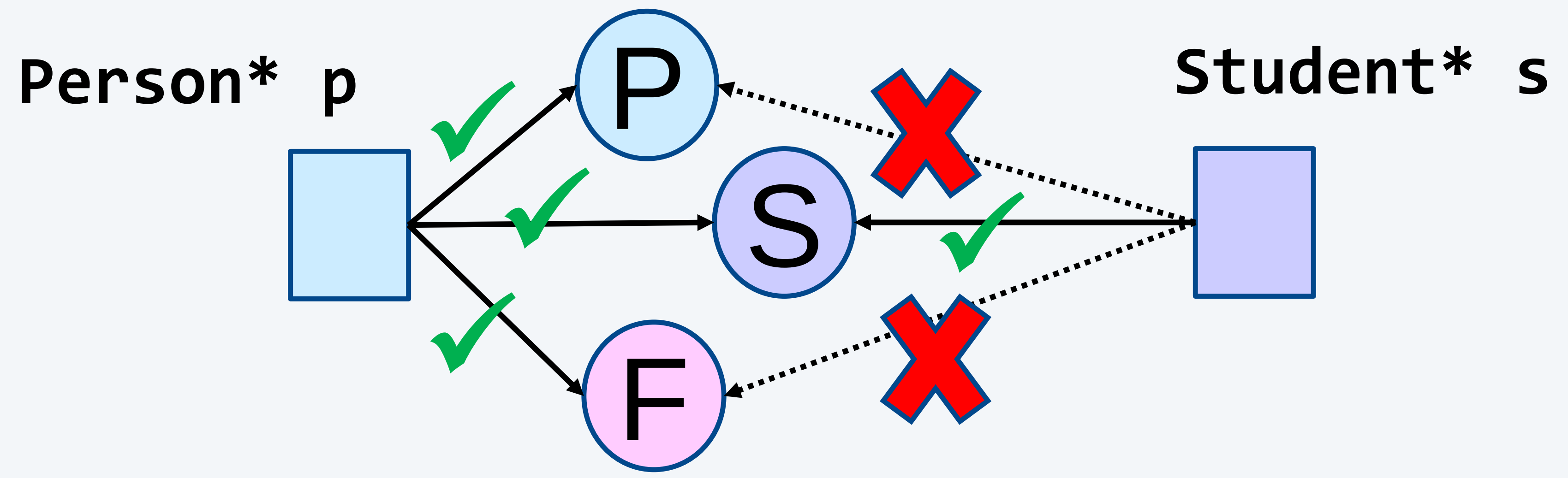
- p = s; // Base = Derived… ?
- s = p; // Derived = Base…?




```
class Person {
 public:
 Person(string n, int ident): name(), id(ident) {}
 Person():name(""), id(0) {}
  void print_info() { cout << name << " " << id << endl;}
  private:
  string name;
   int id;
};
class Student : public Person {
      public:
      Student(): mjr(0), gpa(0){}
       Student(string n, int ident, int m): Person(n,ident), mjr(m) {}
      void print_info() {
       Person::print_info();
       cout << mjr << " " << gpa << endl;
     }
     private:
     int mjr;
      double gpa;
};
int main(){
        Person p("Bill",1);
        Student s("Joe",2,5);
         p = s;
         //s = p;
}
```

Inheritance and Type-compatibility

A pointer or reference to a publicly derived class object is **type-compatible** with its base class type.



Base pointer CAN point at any publicly derived object.

Derived pointer CANNOT point at base or "sibling" objects

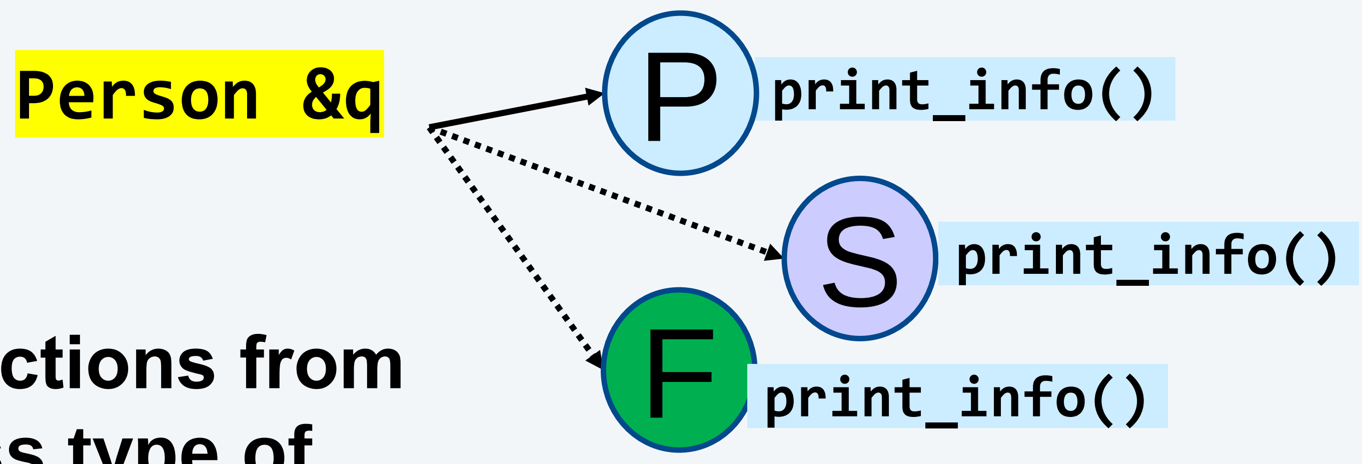
```
class Faculty : public Person {
     public:
     Faculty(): tenure(false){}
     Faculty(string n, int ident, bool t):
     Person(n,ident), tenure(t) {}
     void print_info() {
     Person::print_info();
     (tenure)? cout << "Tenured" << endl :
      cout << "Not tenured" << endl;
     }
     private:
     bool tenure;
};
int main(){
    Person p("Bill",1);
     Student s("Joe",2,5);
     Faculty f("Brian", 3, true);
     Person &q = p;
      q = s;
      q = f;
      Student &r = s;
     // r = p;
     // r = f;
      return 0;
}
```

STATIC VS. DYNAMIC BINDING

Inheritance and Static Binding

Static binding is when the version of the function called is based on the static **type of the pointer or reference used**

1 **No VIRTUAL declaration…**



2 **…only functions from the class type of reference are used**

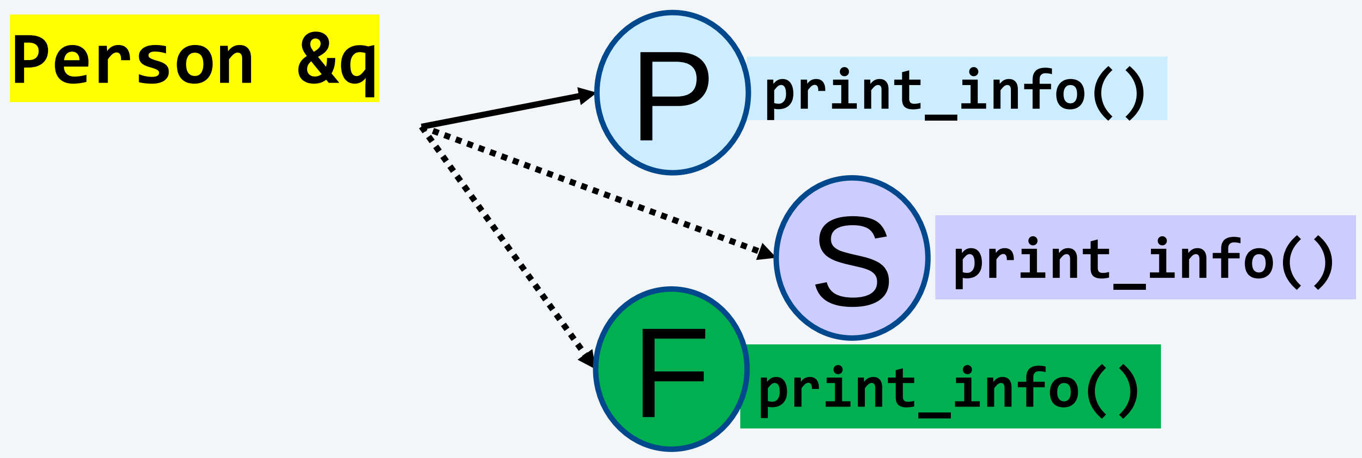
```
class Person {
 public:
  void print_info() const; // print name, ID
  //string name; int id;
};

class Student : public Person {
 public:
  void print_info() const; // print major too
  //int major; double gpa;
};

class Faculty : public Person {
 public:
  void print_info() const; // print tenured
  //bool tenure;
};

int main(){
  Person p = new Person("Bill",1);
  Student s = new Student("Joe",2,5);
  Faculty f = new Faculty(“Brian",3,true);
  Person *q = &p;
  q->print_info();
  q = &s; q->print_info();
  q = &f; q->print_info();
} // calls
```

Bill 1
Joe 2
Brian 3

Virtual Functions and Dynamic Binding

`virtual` keyword specifies that the version of a member function called is determined by the **type of object referenced** at runtime.

This is called dynamic binding




```
class Person {
 public:
  virtual void print_info() const; // print name, ID
  //string name; int id;
};

class Student : public Person {
 public:
  void print_info() const; // print major too
  //int major; double gpa;
};

class Faculty : public Person {
 public:
  void print_info() const; // print tenured
  //bool tenure;
};

int main(){
  Person p = new Person("Bill",1);
  Student s = new Student("Joe",2,5);
  Faculty f = new Faculty(“Brian",3,true);
  Person *q = &p;
  q->print_info();
  q = &s; q->print_info();
  q = &f; q->print_info();
} // calls
```

Bill 1
Joe 2
5 0
Brian 3
Tenured

Polymorphism

Use base class pointers to point at derived types and use virtual functions for different behaviors for each derived type

Polymorphism via virtual functions allows one set of code to operate appropriately on all derived types of objects

```
int main()
{
  unique_ptr<Person> people[3];
  people[0] = make_unique<Person>("Bill",1);
  people[1] = make_unique<Student>("Joe",2,5);
  people[2] = make_unique<Faculty>("Brian", 3, true);
  for (size_t i = 0; i < 3; i++){
      people[i]->print_info();
  }
  return 0;
}
```

Bill 1
Joe 2
5 0
Brian 3
Tenured

Pointers, References, and Objects

To allow dynamic binding and polymorphism you use a base class

- **Pointer**
- **Reference**

Copying a derived object to a base object makes a copy and so no polymorphic behavior is possible

```
void f1(Person* p)
{
   p->print_info();
   // calls Student::print_info()
}


void f2(const Person& p)
{
   p.print_info();
   // calls Student::print_info()
}

void f3(Person p)
{
   p.print_info();
   // calls Person::print_info() on the copy
}

int main(){
  Student s("Joe",2,5);
  f1(&s);
  f2(s);
  f3(s);
  return 0;
}
```

```
Joe  2
5 0
Joe 2
5 0
Joe 2
```

Virtual Destructors: Old School Memory Problems

```
class Student{
 ~Student() {  }
 string major();
 ...
}
class StudentWithGrades : public Student
{
 public:
  StudentWithGrades(...)
  { grades = new int[10]; }
  ~StudentWithGrades { delete [] grades; }
  int *grades;
}
int main()
{
  Student *s = new StudentWithGrades(...);
  ...
  delete s; // Which destructor gets called?
  return 0;
}
```

Due to static binding (no virtual decl.) ~Student() gets called and doesn't delete grades array

```
class Student{
 virtual ~Student() {  }
 string major();
 ...
}
class StudentWithGrades : public Student
{
 public:
  StudentWithGrades(...)
  { grades = new int[10]; }
  ~StudentWithGrades { delete [] grades; }
  int *grades;
}
int main()
{
  Student *s = new StudentWithGrades(...);
  ...
  delete s; // Which destructor gets called?
  return 0;
}
```

Due to dynamic binding (virtual decl.) ~StudentWithGrades() gets called and does delete grades array

Classes that will be used as a base class should have a virtual destructor

(http://www.parashift.com/c++-faq-lite/virtual-functions.html#faq-20.7)

Summary

No virtual declaration:

- Member function that is called is based on the ***type of the pointer or reference***
- Static binding

With virtual declaration:

- Member function that is called is based on the ***type of the object referenced***
- Dynamic Binding

ABSTRACT CLASSES

Abstract Classes & Pure Virtual Functions

A class with at least one pure virtual functions is called an **abstract base class**. They serve as an interface future derived classes.

- Prototype only
- Make function body `" = 0; "`

Objects of the abstract class type MAY NOT be instantiated

```
class Shape
{
 public:
  virtual ~Shape() { }
  virtual double getArea() = 0;
  virtual double getPerimeter() = 0;
  virtual string getType() = 0;
};


int main(){
   Shape s; // Cannot instantiate abstract class

   s.getArea(); // WILL NOT COMPILE!!



}
```

Abstract Classes as Interfaces

- Functions that are not implemented by the abstract base class but **must be implemented by the derived class to be instantiated**

- **Exercise: Can you write a Square class that is publicly derived from the Rectangle class?**

```
class Shape
{
 public:
  virtual ~Shape() { }
  virtual double getArea() = 0;
  virtual double getPerimeter() = 0;
  virtual string getType() = 0;
};

class Rectangle :     public Shape
{
public:
  Rectangle(double b, double h)
     : _b(b),_h(h){}
  ~Rectangle() { }
  double getArea() { return _b * _h;  }
  double getPerimeter() { return   2*_b+2*_h;  ; }
  string getType() { return "Rectangle"; }
private:
   double _b, _h;
};


int main(){
   Rectangle r(3,4);
   Shape &s = r;

   cout << s.getArea() << endl;


}
```

Abstract Classes with implementation

An abstract base class can have functions and data members defined

These data and function members are inherited by derived classes.

```
class Animal {
  public:
  Animal(string c): color(c) { }
  virtual ~Animal() {}
  string get_color() { return color; }
  virtual void make_sound() = 0;
  protected:
  string color;
};

class Dog : public Animal {
  public:
  Dog(string c): Animal(c){}
  void make_sound() { cout << "Bark"; }
};

class Cat : public Animal {
  public:
  Cat(string c): Animal(c){}
  void make_sound() { cout << "Meow"; }
};

int main(){
   Animal* a[3];
   //a[0] = new Animal;   // WON'T COMPILE...abstract class
   a[1] = new Dog("brown");
   a[2] = new Cat("calico");
   cout << a[1]->get_color() << endl;
   a[2]->make_sound();
}
```

Output:
brown
meow

A Queue Interface

This abstract Queue class specifies the Queue ADT operations

Any derived implementation must implement these public member functions to be instantiated

```
class IntQueue {

    public:
    virtual int front() =  0;
    virtual bool empty() = 0;
    virtual size_t size() = 0;
    virtual void push_back(int v) = 0;
    virtual void pop_front() = 0;
};
```

A STL List Queue Class

This class uses STL List to implement a queue.

It is an example of implementing a queue with a doubly linked list.

It is basically a wrapper class for calls directly to STL List.

```
// This class implements the queue ADT interface using STL list.
// Remember STL list is a doubly linked list implementation.

class ListIntQueue : public IntQueue {
    public:
    int front() {
        // Add proper error handling
        if (queue.empty()) return -1;
        return queue.front();
    }
    bool empty() {    return queue.empty();    }
    size_t size() {    return queue.size();    }
    void push_back(int value) {    queue.push_back(value);    }
    void pop_front() {
        if (!empty()) {
            queue.pop_front();
        }
    }

    private:
    std::list<int> queue;

};
```

A STL Vector Queue Class

This class uses STL Vector to implement a queue.

It is an example of implementing a queue with a dynamically sized array.

It is basically a wrapper class for calls directly to STL Vector.

```
// This class implements the queue ADT interface using STL vector.
// Remember STL vector is a dynamically sized array



class  VectorIntQueue : public IntQueue {

        public:
            int front() {
                    // add appropriate error handling
                    if (this->empty()) { return -1;}
                    return queue.front();
             }

             bool empty() {       return queue.empty();    }
             size_t size() {       return queue.size();    }
             void push_back(int value) {       queue.push_back(v);    }
             void pop_front() {
                         if (!empty()) {
                             queue.erase(queue.begin());
                         }
             }

            private:
            std::vector<int> queue;
};
```

The Benefits of Polymorphism

By using references to the abstract class, we can use the same code to test any derived implementation of it.

```
#include "queue.h“
#include "listqueue.h“
#include "vectorqueue.h“

#include <iostream>

#include <chrono>using namespace std;

int main(){

 // Change this for whatever queue you are testing
list_IntQueue test_queue;
vector_IntQueue vt_queue;
//   AbstractIntQueue &q = vt_queue;
 AbstractIntQueue &q = test_queue;

 // Is there any difference between the STL List and Queue implementations
// for adding to the end of the queue? Run tests increasing the size...
 // What measurements can you take? Time for timing experiments.
  for (size_t i = 0; i < 100000; i++){
         q.push_back(i);
  }

  // Can you think of how to test pop_front?
}
```

Polymorphism & Private Inheritance

Warning: If private or protected inheritance is used, the **derived class is no longer type-compatible with base class**

```
class Person {
 public:
  virtual void print_info();
  //string name; int id;
};
class Student : public Person {
 public:
  void print_info(); // print major too
  //int major; double gpa;
};
// if we use private inheritance
// for some reason
class Faculty : private Person {
 public:
  void print_info(); // print tenured
  //bool tenure;
};

int main(){
  Person p = new Person("Bill",1);
  Faculty f = new Faculty(“Brian",3,true);
  Person *q = &p;
  q->print_info();
  // q = &f; q->print_info(); X No longer works

  f.print_info();


}
```