# Review

Inheritance captures the “is a “ relationship

- A class is defined as a derived class of a base class.

  class CSFSUstudent: public FSUstudent {

  …

  }

- The derived class inherits all public interface of the base class.
  - Have access to “protected” members of the base class
  - Can redefine the functions in the public interface of the base class.
- What is the order of invoking sequence for constructors/destructor of base/derived types?

# Polymorphism and virtual functions

# Motivating virtual functions

Consider the student example in sample5.cpp

- Each type of students prints a different report cards.

```
Student s
Grad g;
Undergrad u;
```

– What if we want to iterate through all students and print a report card for each student

```
Student list[30000];
……
For (i=0; i<size; i++) list[i].GradeReport();
```

– Two problems with this code:
  - List is a homogeneous array, we have to use the most common type (the base type). The base class does not have all the information about the derived Grad and Undergrad classes to produce the report.
  - The GradeReport called is the Student version (the most generic report), not the desired customized routine for each derived class.

# Motivation

- Iterate through all students and print a report card for each student

```
Student list[30000];
……
For (i=0; i<size; i++) list[i].GradeReport();
```

- One potential solution to this: use pointers to create a **heterogeneous** list
  - A pointer can only point at one type
    - A pointer to a base class type **can** be pointed at an object derived from that base class (an object of the derived class "is-an" object of the base class) !
      - Similarly, a base class type reference variable can refer to an object derived from that base class

```
Student * list[30000]; // an array of Student pointers
list[0] = &g; // using the earlier Grad object
list[1] = &u; // undergrad object
list[2] = new Grad; // dynamic Grad object
…
```

  - This creates a heterogeneous list that points to different types of objects.

# Motivation

- Iterate through all students and print a report card for each student

- One potential solution: use pointers to create a heterogeneous list

```
Student *list[30000];
…
for (int i = 0; i < size; i++) list[i]->GradeReport();
```

- List[i] points to different objects, but which version of the GradeReport() is invoked when we do list[i]->GradeReport()?

# Motivation

```
Student *list[30000];
…
for (int i = 0; i < size; i++) list[i]->GradeReport();
```

- List[i] points to different objects, but which version of the GradeReport() is invoked when we do list[i]->GradeReport()?
  - When compiler compiles this program, it must make a choice as to which routine to call. At compile time, what is the logical choice?
  - Since list[i] is of Student * type, the GradeReport() in the Student class will be invoked.
  - This is due to the binding requirement: the compiler must decide which version of the routine is bound before the program executes (static binding), the only logical choice is to do the binding based on the object type (in this case Student * type) if nothing special is done.

  - To resolve our problem, we need something that makes the binding dynamic at runtime – a function that can bind to different routines
    - Virtual function is the mechanism in C++ for this purpose.

# Virtual function

When a function is declared as a “virtual” function, it means that the function can be bound to different routines dynamically at runtime.

- Polyphorphism refers to the ability to associate many meanings to one (virtual) function.
- Example:

  class Student { public: virtual void GradeReport(); ... };

- The GradeReport() funcation can mapped to different functions in the derived class (the program looks for the target of the object to decide which functions to be invoked. See sample1.cpp.

```
Student *sp1, *sp2, *sp3;
Student s;
Grad g;
Undergrad u;
sp1 = &s;
sp2 = &g;
sp3 = &u;
sp1->GradeReport(); // runs Student's version
sp2->GradeReport(); // runs Grad's version
sp3->GradeReport(); // runs Undergrad's version
```

# Solution to the motivation problem

- Create a heterogeneous list, use virtual function to access different GradeReport() routines in different derived classes.

```
class Student { public: virtual void GradeReport(); ... };
…
Student *list[30000];
……// assigning the pointer to the right place
for(  i=0; i<size; i++) list[i] ->GradeReport();
```

# Pure virtual function

- Normal function members including virtual function members must have an implementation.
  - So the GradeReport() virtual function should normally have an implementation in the base class.
  - This is not always necessary (or possible) since there may not have much to do in such as function – the functionality should be implemented in the derived class when more information is available.
  - C++ allows a virtual function to be without implementation – this is known as pure virtual function with the actual implementation done in the derived class
  - Pure virtual function syntax with “=0” in the function declaration:
    - virtual void GradeReport() = 0;

# Pure virtual function and Classes

- When an object is declared to be of a normal class type, we know everything about the object – what data members, and how each function member behaves – these are required to define an object.
- What  about an object declaring to be of a class with a pure virtual function?
  - The object is not well defined.
    - The behave of the pure virtual function is NOT defined – so we don't know everything about the object.
    - C++ does not allow this to happen.
  - C++ differentiates normal classes from the classes with pure virtual functions, which is called an abstract class.

# Abstract Classes

- An abstract class is a class with at least one pure virtual function
  - An abstract class cannot instantiate an object.
    - Student st;  // illegal, student is an abstract class, the object cannot be created.
  - An abstract class can be used to declare pointers.
    - A pointer does not require an object to be created
    - A pointer of a base class may point to an object of an derived object
    - Student *st; // legal, st may point to derived objects.
  - Check out sample2.cpp and sample3.cpp

# Case study

- Look at the employee example.