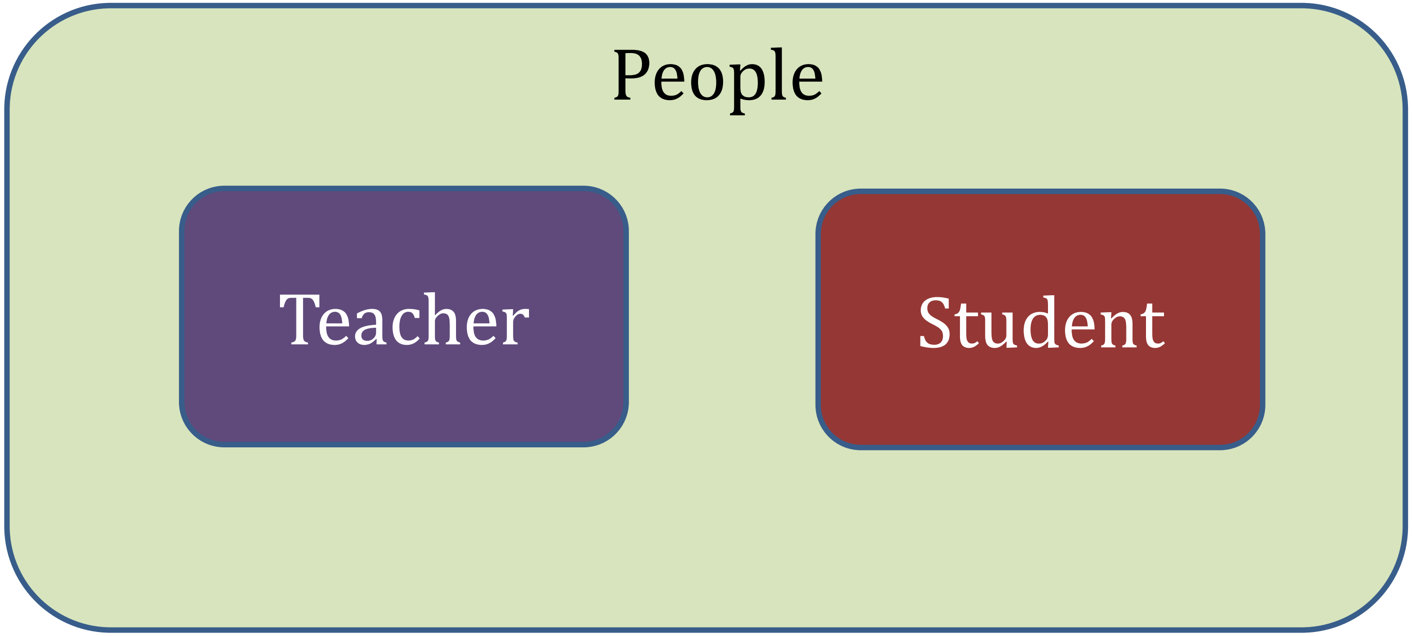
CSCI 3132 – Object Orientation and Generic Programming

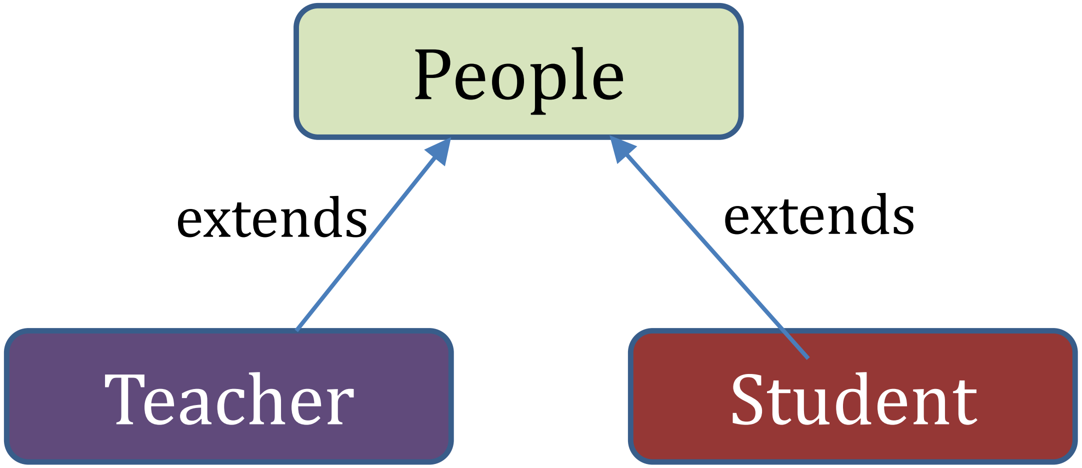
Week 6, 7 – Inheritance and Polymorphism

Types and Subtypes

- A class defines a type, or a set of objects
- Subtypes are types within a type
  - Teacher and Student are subtypes of People

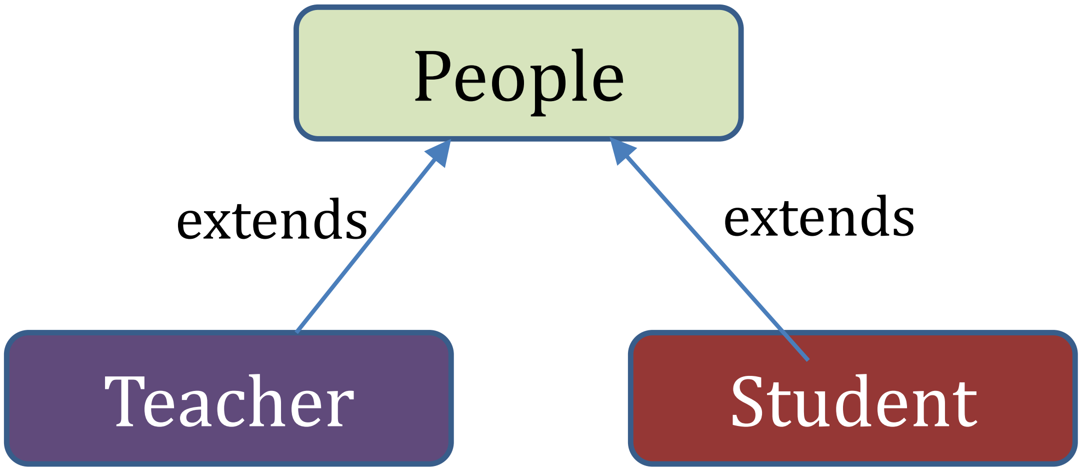


Type Hierarchy



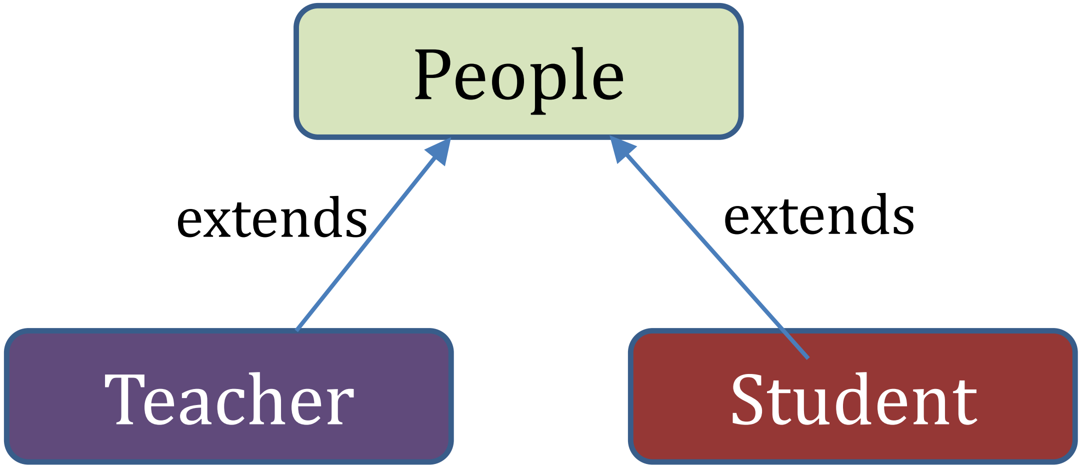
- What do people have in common?
  - Characteristics
    - Name, address, ID
  - Behaviours
    - Display profile, update address

Type Hierarchy



- What things are specific to students?
  - Characteristics
    - Program, courses, semester, grade
  - Behaviours
    - Add/drop course, add/remove classes, update grades

Type Hierarchy



- What things are specific to teachers?
  - Characteristics
    - Courses taught, classes taken, research projects
  - Behaviours
    - Add a class, add a course, add/update project

Inheritance

- A subtype inherits characteristics and behaviours of its base type
- Student has:
  - Characteristics
    - Name, address, id, program, courses, semester, grade
  - Behaviours
    - Display profile, update address, update program, add/drop course, update grade

Base Class – People

```
Class People {
protected:
        int id;
        string name;
        string address;
Public:
        People (int id, string name, string address);
        void DisplayProfile();
        void UpdateAddress(string newAddr);
};
```

//Access control

Access Control

- Private
  - No one from outside of the class can access it.
  - Even the subclasses cannot access it
- Protected
  - Only the class itself and all its subclasses can access it
- Public
  - Anyone can access it
- Friend
  - Friend class or function can access private members of the class it in which it is declared friend

Derived Class – Student

```
#include "People.h"
class Student : public People {
protected:
      int program;
      int semester;
      float grade;

Public:
      Student (int id, string name, string address,
      int program, int semester, float grade);
      void DisplayProfile();
      void UpdateGrade(float grade);
};
```

Constructing the Student Subclass

```
//in implementation of the derived class Student
Student::Student(int id, string name, string address,
            int program, int semester, float grade) :
            People(id, name, address){
      this->program = program;
      this->semester = semester;
      this->grade = grade;
}
//in implementation of the base class People
People::People(int id, string name, string address);
      this->id = id;
      this->name = name;
      this->address = address;
}
```

Overriding a Base Class Method

```
class People {
protected:
  int id;
  string name;
  string address;
public:
  People (int id, string name, string address);
  void DisplayProfile();
  void UpdateAddress(string newAddr);
};
```

```
#include "People.h"
class Student : public People {
protected:
   int program;
   int semester;
   float grade;
public:
   Student (int id, string name, string address, int program, int semester, 
float grade);
   void DisplayProfile();
   void UpdateGrade(float grade);
};
```

Overriding a Base Class Method

```
void People::DisplayProfile(){
      cout <<"ID: "<<id<<", Name: "<<name<<endl;
      cout <<"Addr: "<<address<<endl;
}

void Student::DisplayProfile(){
      cout <<"ID: "<<id<<", Name: "<<name<<endl;
      cout <<"Addr: "<<address<<endl;
      cout <<"Program: "<<program;
      cout <<", Semester: "<<semester<<endl;
      cout <<"GPA: "<<grade<<endl;
      cout<<endl;
}
```

Overriding a Base Class Method

```
People* john = new People(5542, "John Snow",
      "Somewhere on the wall");
Student* bran = new Student(7944, "Brandon Stark",
      "Somewhere beyond the wall", 62255, 6, 3.98);

john->DisplayProfile();

      ID: 5542, Name: John Snow
      Addr: Somewhere on the wall

bran->DisplayProfile();
      ID: 7944, Name: Brandon Stark
      Addr: Somewhere beyond the wall
      Program: 62255, Semester: 6
      GPA: 3.98
```

POLYMORPHISM

Polymorphism

Poly → Many   Morph → Forms

- Ability of a type X to appear and be used like another type Y
  - Example: Student object can be used in place of a People object
- Bjarne Stroustrup, creator of C++ defines polymorphism as:
  - Providing a single interface to entities of different types. Virtual functions provide dynamic (run-time) polymorphism through an interface provided by a base class. Overloaded functions and templates provide static (compile-time) polymorphism. TC++PL 12.2.6, 13.6.1, D&E 2.9.

Actual versus Declared Type

- At compile-time
  - Variables have declared types
- At runtime
  - Variable may refer to an object with an actual type
  - Actual type may be the same or a subclass of the declared type

```
People* john = new People(5542, “John Snow”,
      “Somewhere on the wall”);
Student* bran = new Student(7944, “Brandon Stark”,
      “Somewhere beyond the wall”, 62255, 6, 3.98);
```

- What are the declared and actual types of john and bran?

Calling an Overriden Function

```
People* bran = new Student(7944, "Brandon Stark",
      "Somewhere beyond the wall", 62255, 6, 3.98);

bran->DisplayProfile();
```

```
ID: 7944, Name: Brandon Stark
Addr: Somewhere beyond the wall
```

Why other details for the Student are not shown?

– Where are the program, semester and GPA?

Virtual Functions

- In base class
  - Declare overridden methods as virtual

```
Class People {
protected:
   int id;
   string name;
   string address;
Public:
   People (int id, string name, string address);
   virtual void DisplayProfile();
   virtual void UpdateAddress(string newAddr);
};
```

Calling a Virtual Function

```
People* bran = new Student(7944, "Brandon Stark",
      "Somewhere beyond the wall", 62255, 6, 3.98);

bran->DisplayProfile();
```

```
ID: 7944, Name: Brandon Stark
Addr: Somewhere beyond the wall
Program: 62255, Semester: 6
GPA: 3.98
Courses Taken:
```

Why the details for the Student are now shown?

Early versus Late Binding

- Without using virtual, we get early binding
  - Also known as static (or compile-time) binding
  - Which implementation of the method should be called is decided at compile time
    - Based on the type of the pointer used to make the call
- Using virtual, we get late binding
  - Also known as dynamic (or runtime) binding
  - Which implementation of the method should be called is decided at runtime
    - Based on the type of the pointed-to object

Early versus Late Binding

```
People* bran = new Student(7944, "Brandon Stark",
      "Somewhere beyond the wall", 62255, 6, 3.98);

bran->DisplayProfile();
```

```
ID: 7944, Name: Brandon Stark
Addr: Somewhere beyond the wall
Program: 62255, Semester: 6
GPA: 3.98
```

- Example of late binding.
  - bran gets bound to the type of pointed-to object i.e. Student

What goes on Behind the Scenes

- When a class contains virtual functions or overrides virtual functions from a parent class
  - The compiler builds a vtable for that class
    - vtable stores pointers to all virtual functions
    - One virtual table created for each class
    - Lookup performed during the function call
  - All objects of the same class share the same vtable
  - Virtual tables are not constructed for every class
    - Only classes with virtual functions or overriding functions

Virtual Pointers

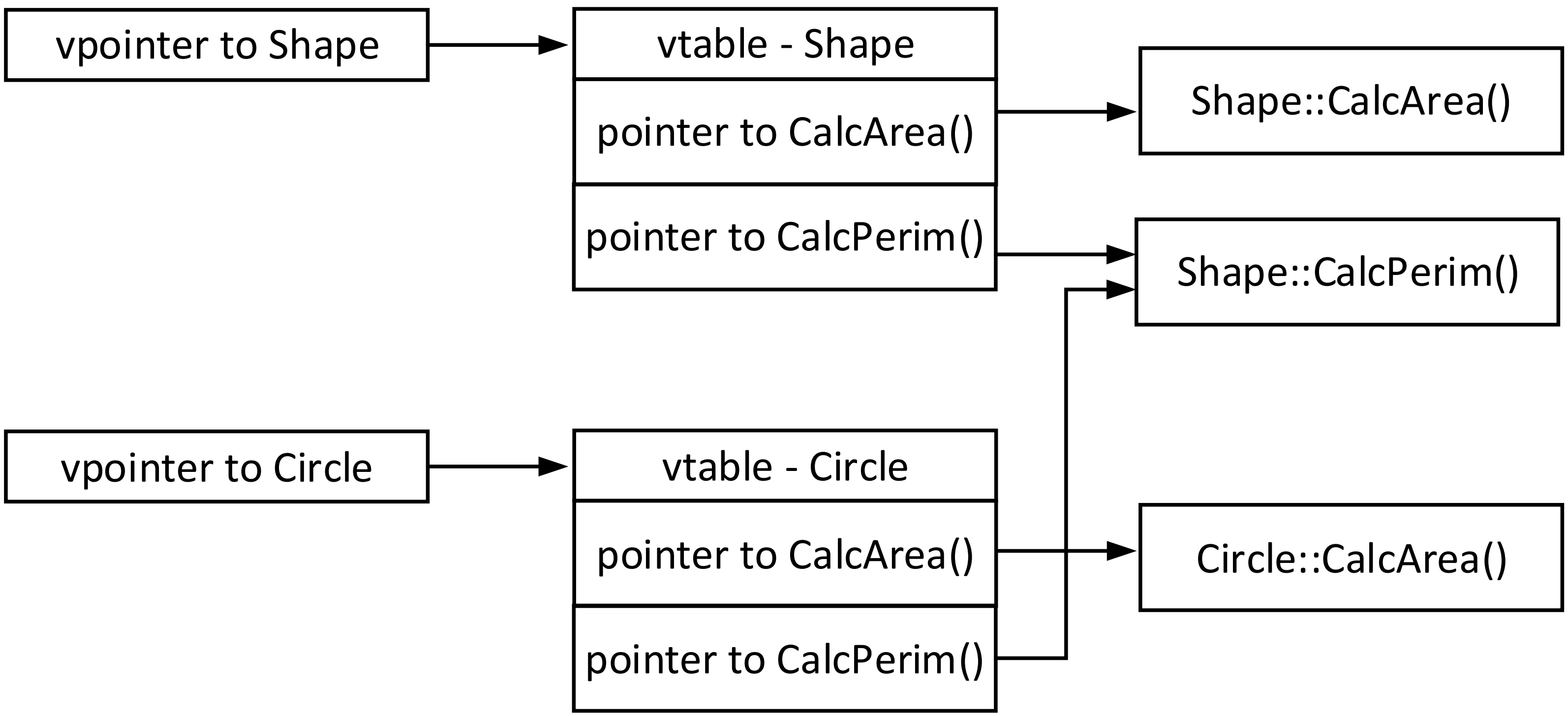
- vpointers associated with every vtable
  - Used to access functions inside the vtable
- For every class with a vtable
  - Compiler adds hidden code to the constructor of that class to initialize the vpointers of its objects to the address of the corresponding vtable
- How does the program know which function to invoke?

Example

```
#include <iostream>
using namespace std;
class shape{
public:
	virtual float CalcArea(float r) {return 0;}
	virtual float CalcPerim(float r) {return 0;}
};
class circle : public shape{
public:
	float CalcArea(float r) { return (3.14159*r*r);}
};
int main(){
	shape *s1 = new circle;
	cout<<s1->CalcArea(2);
}
```

//behind the scenes, s1 uses vpointer to the vtable of the circle class to access the correct CalcArea() function of circle class. i.e. s1 ---> vpointer ---> CalcArea() method for circle

Virtual Tables and Pointers



Virtual Destructor

- Destructors are called when an object is destroyed
  - Compiler generated public member if not specified
  - Automatically called
  - Used to free up resources acquired by the object
  - Syntax: ~ClassName();
- Base class destructors should be virtual
  - This ensures that correct destructor is called through late binding

Virtual Destructor – Example

```
#include <iostream>
using namespace std;

class shape{
public:
    virtual ~shape(){cout<<"Shape Destructor\n";}
};

class circle : public shape{
public:
    ~circle(){cout<<"Circle Destructor\n";}
};

int main(){
    shape *s1 = new shape;  //s1 points to base class object
    shape *s2 = new circle; //s2 points to derived class object
    delete s1;              //base class destructor called
    delete s2;              //derived class, then base class destructor called
}
```

```
Shape Destructor
Circle Destructor
Shape Destructor
```

Pure Virtual Functions and Abstract Classes

- A virtual function may be declared as a pure virtual function by adding =0 to its declaration
  - Providing the implementation is not required
  - Derived class MUST implement provide an implementation of the pure virtual function
  - Pure virtual function makes a class Abstract

- Abstract classes cannot be instantiated
  - Not implementing all pure virtual functions in a derived class makes that class Abstract too

Abstract class – Example

```
#include <iostream>
using namespace std;

class shape{
public:
    virtual float CalcArea(float r) = 0;    //pure virtual function
};

class circle : public shape{
public:
    float CalcArea(float r){ return (3.14159*r*r); }
};

int main(){
    //shape sh;                    //error - Cannot initialize Abstract class
    //shape *sh = new shape;       //error - Cannot initialize Abstract class
    shape *s1 = new circle;        //valid
    cout<<s1->CalcArea(2)<<"\n"; //calls derived class' CalcArea()
}
```