Intro to C++

# Lecture Topics

* Inheritance and polymorphism

These notes are taken from Eunsuk Kang & JeanYang @ MIT.

# Inheritance

* A class defines a set of objects, or a type, e.g., all *University people*
* Some objects are distinct from others in some ways, e.g., *University students* vs*. University professors*, but they all are still *University people*
  + *University* professor and student are subtypes of *University* people



* + What characteristics/behaviors do people at *University* have in common?
    - name, ID, address, …
    - change address, display profile, …
  + What things are special about students?
    - course number, classes taken, year, …
  + What things are special about professors?
    - course number, classes taught, rank (assistant, etc.), …
    - add a class taught, promote, …
* Inheritance means that a subtype inherits characteristics and behaviors of its base type
  + e.g. Each *University* student has
    - Characteristics that it inherits from *University* person: name, ID, address
    - Methods that it inherits from *University* person: display profile, etc.
* Base Type: Person

#include <string>

using namespace std;

class Person

{

protected:

int id;

string name;

string address;

public:

Person(int id, string name, string address);

~Person();

void displayProfile();

void changeAddress(string newAddress);

};

Person::Person(int id, string name, string address)

{

this->id = id;

this->name = name;

this->address = address;

}

Person::~Person() { }

void Person::displayProfile()

{

cout << "-----------------------------\n";

cout << "Name: " << name << " ID: " << id <<;

cout << " Address: " << address << "\n";

cout << "-----------------------------\n";

}

* Subtype: Student

class Student : public Person

{

protected:

int course;

int year; // 1 = freshman, 2 = sophomore, etc.

//vector<int\*> classesTaken; // dynamic array, part of

// C++ standard library

public:

Student(int id, string name, string address, int course, int year);

void displayProfile();

void updateYear(int newyear) { this->year = newyear; }

//void changeCourse(int newCourse);

};

* Constructing an object of subclass

Student::Student(int id, string name, string address, int course, int year) : Person(id, name, address)

// call to the base constructor

{

this->course = course;

this->year = year;

}

* Creating an object

Student\* james = new Student(971232, “James Lee”, “32 Lincoln Ave.”, 6, 2);

* + From base class
    - name = “James Lee”
    - ID = 971232 person
    - address = “32 Lincoln Ave.”
  + from derived class (subclass)
    - course number = 6
    - year = 2
* Overriding a method in base class
  + Both Person and Student have a method void displayProfile();
    - The method defined in Student will overwrite the method defined in Person

void Student::displayProfile()

{

cout << "--------------------------" << endl;

cout << "Name: " << name << ", ID: " << id;

cout << ", Address: " << address << endl;

cout << "Course: " << course << ", year: " << year << endl;

cout << "--------------------------" << endl;

}

Person\* john = new Person(901289, “John Doe”, “500 University Ave.”);

Student\* james = new Student(971232, “James Lee”, “32 Lincoln Ave.”, 6, 2);

james->addClassTaken(220);

john->displayProfile();

james->displayProfile();

# Polymorphism

* Ability of type A to appear as and be used like another type B
  + e.g., a Student object can be used in place of an Person object
* Actual type vs. declared type
  + Every variable has a *declared type* at compile-time
  + But during runtime, the variable may refer to an object with an *actual type* (either the same or a subclass of the declared type)

Person\* john = new **Person**(901289, “John Doe”, “500 University Ave.”);

Person\* steve = new **Student**(911923, "Steve", "99 Lincoln Ave.", 18, 3);

* + What are the declare types of john and steve?

steve->displayProfile();

Name: Steve ID: 911923 Address: 99 Lincoln Ave.

* + Why doesn’t it display the course number and classes taken?
    - Because steve ‘s declared class is Person and thus its Person::displayProfile is invoked.
    - To ensure that a function from the actual class is called, the overridden method must be declared as virtual.
* Virtual functions
  + Declare overridden methods as virtual in the base

class Person

{

…

**virtual** void displayProfile();

};

* + Calling a virtual function

Person\* steve = new Student(911923, "Steve", "99 Lincoln Ave.", 18, 3);

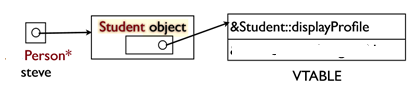
steve->displayProfile();

Name: Steve ID: 911923 Address: 99 Lincoln Ave.

Course: 18

Classes taken

* + What goes on under the hood?
    - Virtual table
      * stores pointers to all virtual functions
      * created per each class
      * lookup during the function call



* Should destructors in a base class be declared as virtual?
  + Yes, we must always clean up the mess created in the subclass (otherwise, risks for memory leaks!)
* Can we declare a constructor as virtual?
  + No, not in C++. To create an object, you must know its exact type.
  + The VPTR has not even been initialized at this point.
* Type casting
  + What will happen?

Person\* steve = new Student(911923, "Steve", "99 Lincoln Ave.", 18, 3);

steve-> updateYear(4); // will not work!

* + - Can only invoke methods of the declared type!
    - “updateYear” is not a member of Person
  + Use “dynamic\_cast<...>” to downcast the pointer

Person\* steve = new Student(911923, "Steve", "99 Lincoln Ave.", 18, 3);

Student\* steve2 = dynamic\_cast<Student\*>(steve);

steve2-> updateYear(4); // OK

* Static vs. dynamic casting
  + Can also use “static\_cast<...>”

Student\* steve2 = static\_cast<Student\*>(steve);

* + - Cheaper but dangerous because there is no runtime check

Person\* p = Person(...);

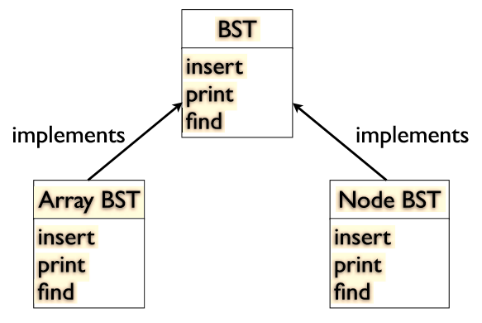
Student\* s1 = static\_cast<Student\*>(p); // s1 is not checked!

Student\* s2 = dynamic\_cast<Student\*>(p); // s2 is set to NULL

* + Use “static\_cast<...>” only if you know what you are doing!

# Abstract base class

* Abstract methods
  + Sometimes you want to inherit only declarations, not deﬁnitions
  + A method without an implementation is called an abstract method
  + Abstract methods are often used to create an interface
* Example: Binary search tree
  + Can provide multiple implementations to BST
  + Decouples the client from the implementations



* Deﬁning abstract methods in C++
  + Use pure virtual functions

class BST

{

public:

virtual ~BST() = 0;

virtual void insert(int val) = 0;

virtual bool find(int val) = 0; // “ﬁnd” is pure

virtual void print\_inorder() = 0; };

};

* + - Here virtual “says” that the methods are virtual and =0 “says” that they are pure, i.e., no implementation is provided at this point.
* Abstract base class in C++
  + A class with one or more pure virtual functions
  + Cannot be instantiated

int main()

{

BST \*bst = new BST(); // cannot do this

}

* + Its subclass must implement all of the pure virtual functions:

class NodeBST : public BST

{

protected:

Node \*root;

public:

NodeBST();

~NodeBST();

void insert(int val);

void print();

bool find(int val);

};

voind NodeBST:insert(int val)

{

if (root == NULL) { root = new Node(val); }

else { ... }

}

* + Does it make sense to define a constructor since the class will never be instantiated?
    - Yes, the constructor is still needed to initialize its members, since they will be inherited by its subclass.
  + Does it make sense to define a destructor since the class will never be created in the first place?
    - Yes, a destructor must be defined as virtual so that the destructor of its subclass is called.
    - Destructor can also be defined as pure, but its body must still be provided.