6.088 Intro to C/C++

Day 5: Inheritance & Polymorphism

Eunsuk Kang & Jean Yang

In the last lecture...

Objects: Characteristics & responsibilities

Declaring and defining classes in C++

Fields, methods, constructors, destructors

Creating & deleting objects on stack/heap

Representation invariant

Today’s topics

Inheritance

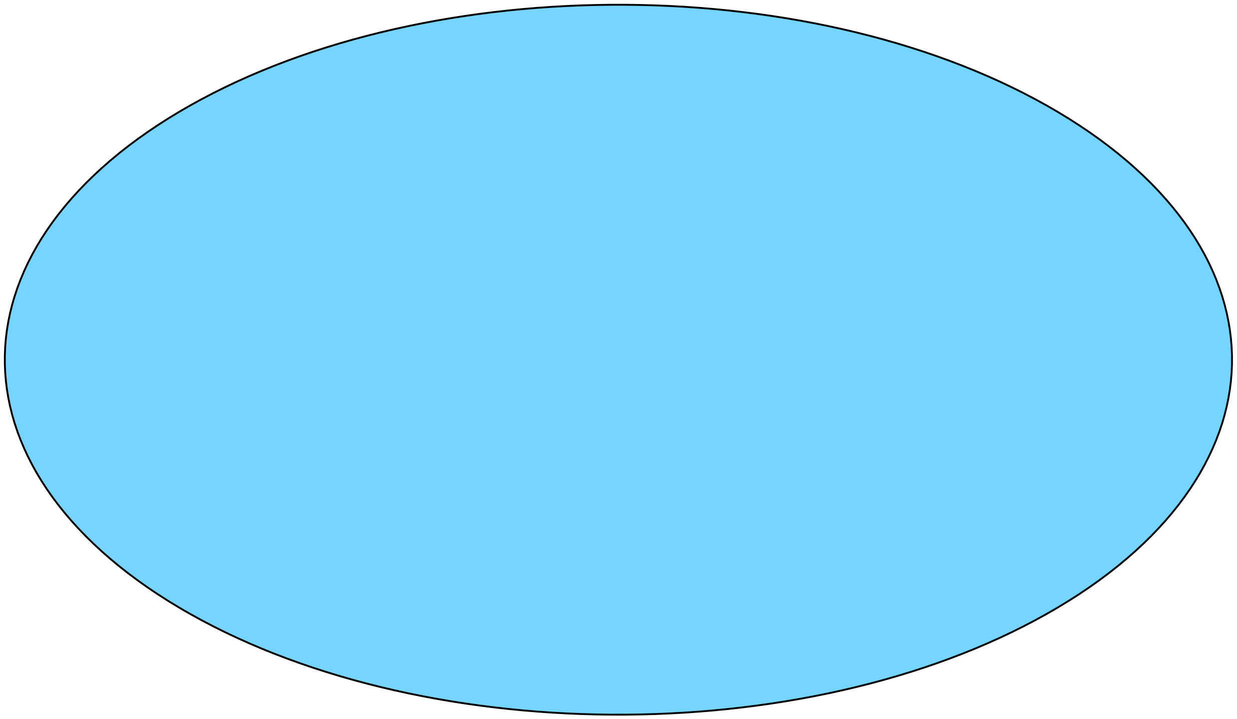
Polymorphism

Abstract base classes

Inheritance

Types

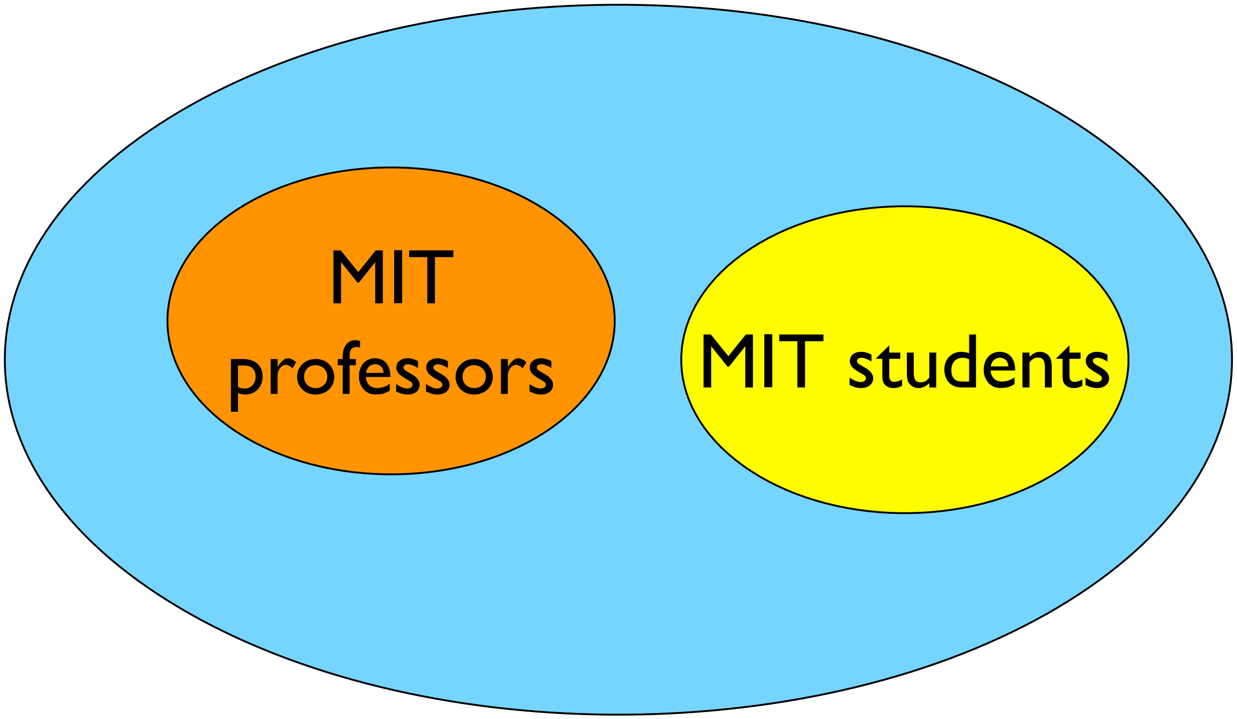
A class defines a set of objects, or a type



people at MIT

Types within a type

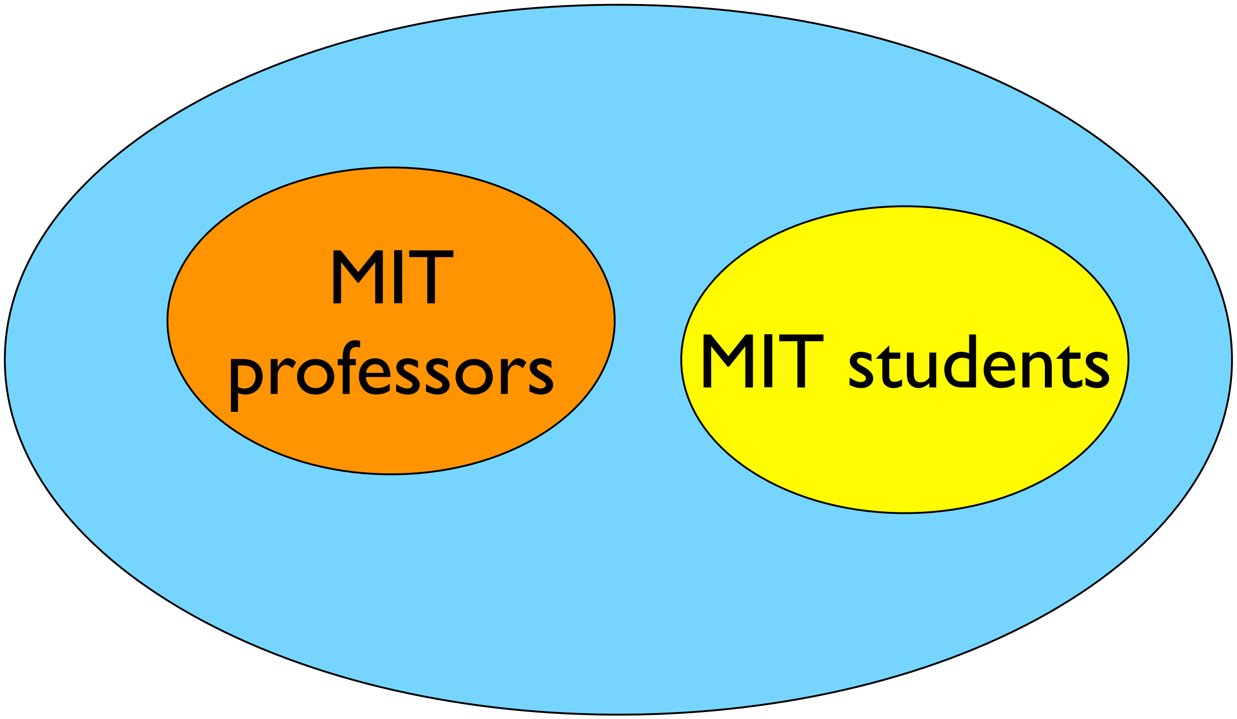
Some objects are distinct from others in some ways



people at MIT

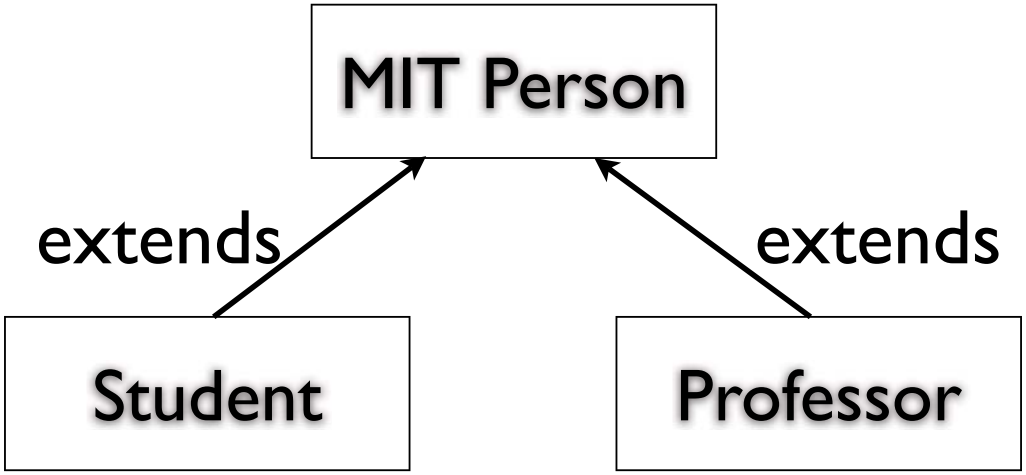
Subtype

MIT professor and student are subtypes of MIT people



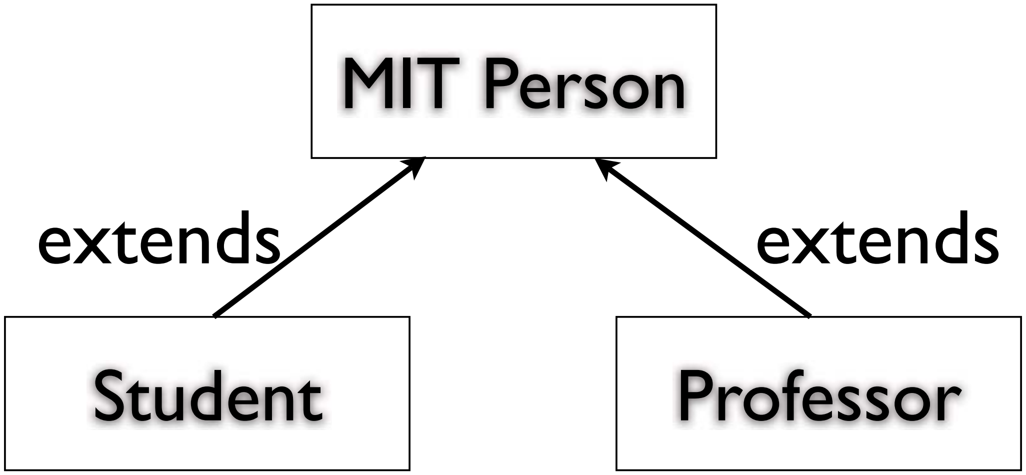
people at MIT

Type hierarchy



What characteristics/behaviors do people at MIT have in common?

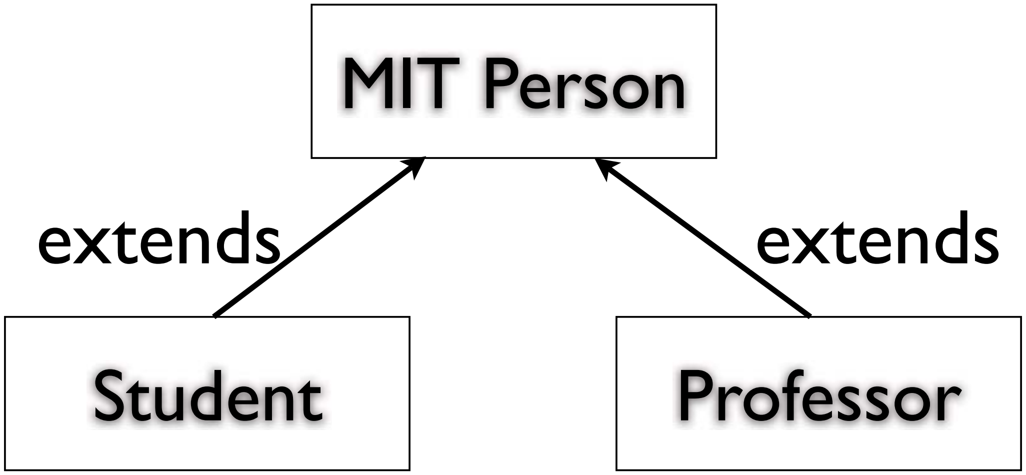
Type hierarchy



What characteristics/behaviors do people at MIT have in common?

- name, ID, address
- change address, display profile

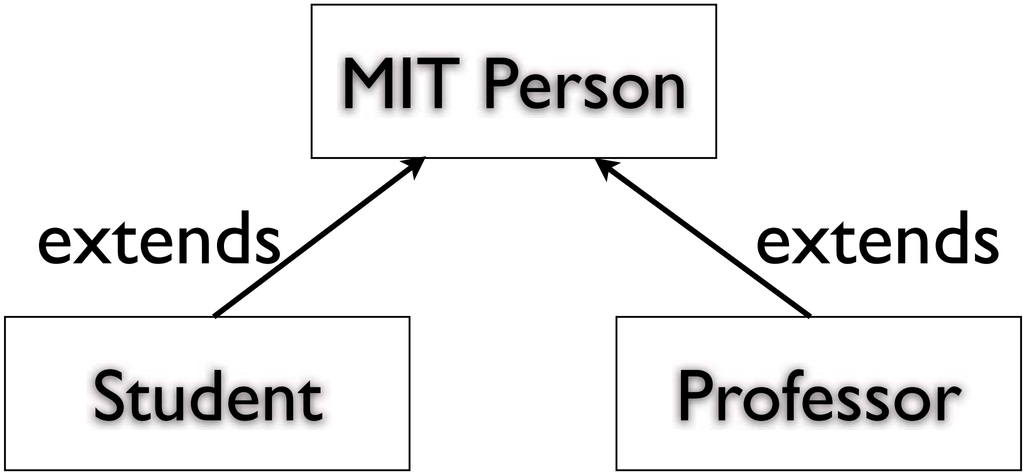
Type hierarchy



What things are special about students?

- course number, classes taken, year
- add a class taken, change course

Type hierarchy



What things are special about professors?

- course number, classes taught, rank (assistant, etc.)
- add a class taught, promote

Inheritance

A subtype inherits characteristics and behaviors of its base type.

e.g. Each MIT student has

Characteristics:

name
ID
address
course number
classes taken
year

Behaviors:

display profile
change address
add a class taken
change course

Base type: MITPerson

```
#include <string>

class MITPerson {

 protected:
  int id;
  std::string name;
  std::string address;

 public:

  MITPerson(int id, std::string name, std::string address);

  void displayProfile();
  void changeAddress(std::string newAddress);

};
```

Base type: MITPerson

```
#include <string>

class MITPerson {

 protected:
  int id;
  std::string name;
  std::string address;


public:
  MITPerson(int id, std::string name, std::string address);

  void displayProfile();
  void changeAddress(std::string newAddress);

};
```

namespace prefix

Base type: MITPerson

```
#include <string>

class MITPerson {

 protected:
  int id;
  std::string name;
  std::string address;

 public:

  MITPerson(int id, std::string name, std::string address);

  void displayProfile();
  void changeAddress(std::string newAddress);

};
```

access control

Access control

Public

accessible by anyone

Protected

accessible inside the class and by all of its subclasses

Private

accessible only inside the class, NOT including its subclasses

Subtype: Student

```
#include <iostream>
#include <vector>
#include "MITPerson.h"
#include "Class.h"

class Student : public MITPerson {

  int course;
  int year;     // 1 = freshman, 2 = sophomore, etc.
  std::vector<Class*> classesTaken;

 public:
  Student(int id, std::string name, std::string address,
         int course, int year);
  void displayProfile();
  void addClassTaken(Class* newClass);
  void changeCourse(int newCourse);

};
```

Subtype: Student

```
#include <iostream>
#include <vector>
#include "MITPerson.h"
#include "Class.h"

class Student : public MITPerson {

  int course;
  int year;      // 1 = freshman, 2 = sophomore, etc.
  std::vector<Class*> classesTaken;

 public:
  Student(int id, std::string name, std::string address,
         int course, int year);
  void displayProfile();
  void addClassTaken(Class* newClass);
  void changeCourse(int newCourse);

};
```

dynamic array,
part of C++ standard library

Subtype: Student

```
#include <iostream>
#include <vector>
#include "MITPerson.h"
#include "Class.h"

class Student : public MITPerson {

  int course;
  int year;     // 1 = freshman, 2 = sophomore, etc.
  std::vector<Class*> classesTaken;

 public:
  Student(int id, std::string name, std::string address,
         int course, int year);
  void displayProfile();
  void addClassTaken(Class* newClass);
  void changeCourse(int newCourse);

};
```

Constructing an object of subclass

```
#include <iostream>
#include <vector>
#include "MITPerson.h"
#include "Class.h"

class Student : public MITPerson {

  int course;
  int year;     // 1 = freshman, 2 = sophomore, etc.
  std::vector<Class*> classesTaken;

 public:
  Student(int id, std::string name, std::string address,
         int course, int year);
  void displayProfile();
  void addClassTaken(Class* newClass);
  void changeCourse(int newCourse);

};
```

Constructing an object of subclass

```
// in Student.cc
Student::Student(int id, std::string name, std::string address,
                 int course, int year) : MITPerson(id, name, address)
{
  this->course = course;
  this->year = year;
}
```

```
// in MITPerson.cc
MITPerson::MITPerson(int id, std::string name, std::string address){
  this->id = id;
  this->name = name;
  this->address = address;
}
```

Calling constructor of base class

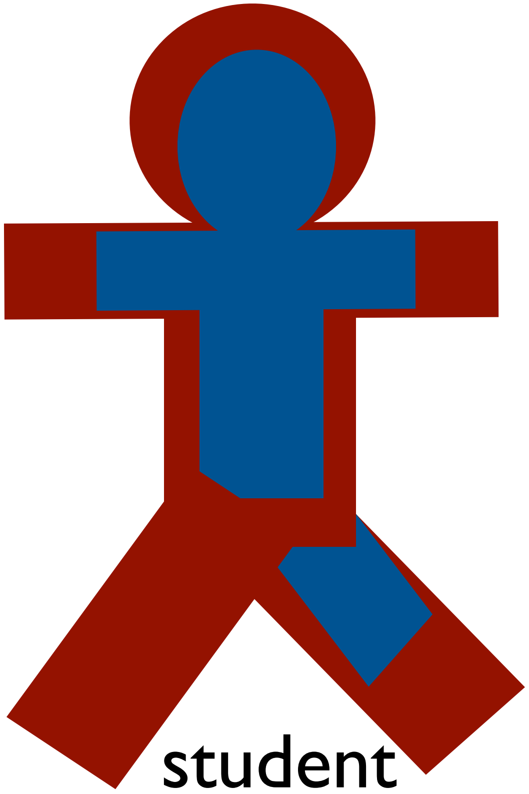
call to the base constructor

```
// in Student.cc
Student::Student(int id, std::string name, std::string address,
                 int course, int year) : MITPerson(id, name, address)
{
  this->course = course;
  this->year = year;
}
```

```
// in MITPerson.cc
MITPerson::MITPerson(int id, std::string name, std::string address){
  this->id = id;
  this->name = name;
  this->address = address;
}
```

Constructing an object of subclass

```
Student* james =
  new Student(971232, “James Lee”, “32 Vassar St.”, 6, 2);
```



name = “James Lee”
ID = 971232
address = “32 Vassar St.”
course number = 6
classes taken = none yet
year = 2

Overriding a method in base class

```
class MITPerson {
 protected:
  int id;
  std::string name;
  std::string address;
 public:
  MITPerson(int id, std::string name, std::string address);
  void displayProfile();
  void changeAddress(std::string newAddress);
};
```

```
class Student : public MITPerson {
  int course;
  int year;     // 1 = freshman, 2 = sophomore, etc.
  std::vector<Class*> classesTaken;
 public:
  Student(int id, std::string name, std::string address,
           int course, int year);
  void displayProfile();  // override the method to display course & classes
  void addClassTaken(Class* newClass);
  void changeCourse(int newCourse);
};
```

Overriding a method in base class

```
void MITPerson::displayProfile() {  // definition in MITPerson
  std::cout << "-----------------------------\n";
  std::cout << "Name: " << name << " ID: " << id
            << " Address: " << address << "\n";
  std::cout << "-----------------------------\n";
}
```

```
void Student::displayProfile(){  // definition in Student
  std::cout << "-----------------------------\n";
  std::cout << "Name: " << name << " ID: " << id
            << " Address: " << address << "\n";
  std::cout << "Course: " << course << "\n";
  std::vector<Class*>::iterator it;
  std::cout << "Classes taken:\n";
  for (it = classesTaken.begin(); it != classesTaken.end(); it++){
    Class* c = *it;
    std::cout << c->getName() << "\n";
  }
  std::cout << "-----------------------------\n";
}
```

Overriding a method in base class

```
MITPerson* john =
  new MITPerson(901289, “John Doe”, “500 Massachusetts Ave.”);
Student* james =
  new Student(971232, “James Lee”, “32 Vassar St.”, 6, 2);
Class* c1 = new Class(“6.088”);
james->addClassTaken(c1);
john->displayProfile();
james->displayProfile();
```

```
------------------------------
Name: John Doe ID: 901289 Address: 500 Massachusetts Ave.
------------------------------
------------------------------
Name: James Lee ID: 971232 Address: 32 Vassar St.
Course: 6
Classes taken:
6.088
------------------------------
```

Polymorphism

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 MITPerson 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)

```
MITPerson* john =
  new MITPerson(901289, “John Doe”, “500 Massachusetts Ave.”);
MITPerson* steve =
  new Student(911923, "Steve", "99 Cambridge St.", 18, 3);
```

What are the declare types of john and steve?
What about actual types?

Calling an overridden function

```
MITPerson* steve =
  new Student(911923, "Steve", "99 Cambridge St.", 18, 3);

steve->displayProfile();
```

Calling an overridden function

```
MITPerson* steve =
  new Student(911923, "Steve", "99 Cambridge St.", 18, 3);

steve->displayProfile();
```

```
-----------------------------
Name: Steve ID: 911923 Address: 99 Cambridge St.
-----------------------------
```

Why doesn’t it display the course number and classes taken?

Virtual functions

Declare overridden methods as virtual in the base

```
class MITPerson {

 protected:
  int id;
  std::string name;
  std::string address;

public:

  MITPerson(int id, std::string name, std::string address);
  virtual void displayProfile();
  virtual void changeAddress(std::string newAddress);
};
```

‘virtual’ keyword

What happens in other languages (Java, Python, etc.)?

Calling a virtual function

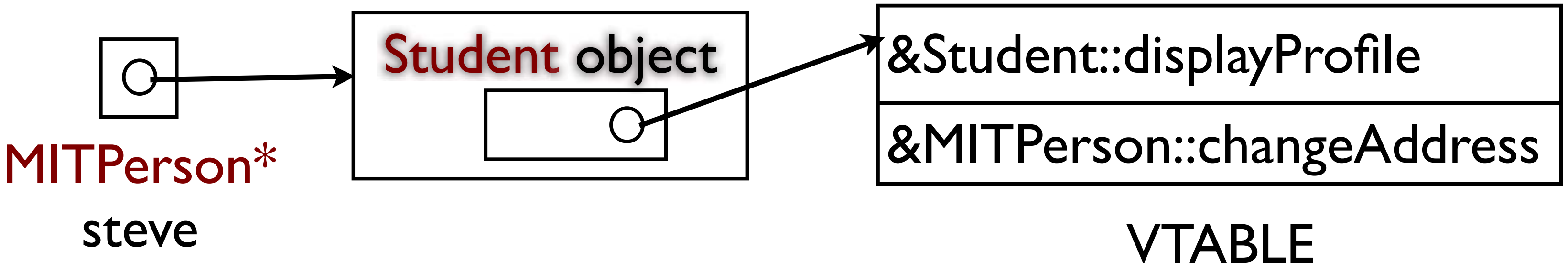
```
MITPerson* steve =
  new Student(911923, "Steve", "99 Cambridge St.", 18, 3);

steve->displayProfile();
```

```
------------------------------
Name: Steve ID: 911923 Address: 99 Cambridge St.
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



MITPerson*
steve

Student object

&Student::displayProfile

&MITPerson::changeAddress

VTABLE

Note "changeAddress" is declared virtual in but not overridden

Virtual destructor

Should destructors in a base class be declared as virtual? Why or why not?

Virtual destructor

Should destructors in a base class be declared as virtual? Why or why not?

Yes! We must always clean up the mess created in the subclass (otherwise, risks for memory leaks!)

Virtual destructor example

```cpp
class Base1 {
public:
  ~Base1() { std::cout << "~Base1()\n"; }
};
class Derived1 : public Base1 {
public:
  ~Derived1() { std::cout << "~Derived1()\n"; }
};
class Base2 {
public:
  virtual ~Base2() { std::cout << "~Base2()\n"; }
};
class Derived2 : public Base2 {
public:
  ~Derived2() { std::cout << "~Derived2()\n"; }
};

int main() {
  Base1* bp = new Derived1; // Upcast
  delete bp;
  Base2* b2p = new Derived2; // Upcast
  delete b2p;
}
```

Courtesy of Bruce Eckel, from Thinking in C++. Used with permission.

Virtual destructor in MITPerson

```
class MITPerson {

 protected:
  int id;
  std::string name;
  std::string address;

 public:

  MITPerson(int id, std::string name, std::string address);
  ~MITPerson();
  virtual void displayProfile();
  virtual void changeAddress(std::string newAddress);
};

MITPerson::~MITPerson() { }
```

Virtual constructor

Can we declare a constructor as virtual? Why or why not?

Virtual constructor

Can we declare a constructor as virtual? Why or why not?

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

```
MITPerson* steve =
  new Student(911923, "Steve", "99 Cambridge St.", 18, 3);
Class* c1 = new Class("6.088");

steve->addClassTaken(c1);
```

What will happen?

Type casting

```
MITPerson* steve =
  new Student(911923, "Steve", "99 Cambridge St.", 18, 3);
Class* c1 = new Class("6.088");

steve->addClassTaken(c1); ✗
```

Can only invoke methods of the declared type!

“addClassTaken” is not a member of MITPerson

Type casting

```
MITPerson* steve =
  new Student(911923, "Steve", "99 Cambridge St.", 18, 3);
Class* c1 = new Class("6.088");

Student* steve2 =
  dynamic_cast<Student>*(steve);

steve2->addClassTaken(c1);  // OK
```

Use "dynamic_cast<...>" to downcast the pointer

Static vs. dynamic casting

Can also use “static_cast<...>”

```
Student* steve2 =
  static_cast<Student>*(steve);
```

Cheaper but dangerous! No runtime check!

```
MITPerson* p = MITPerson(...);
Student* s1 = static_cast<Student>*(p);   // s1 is not checked! Bad!
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

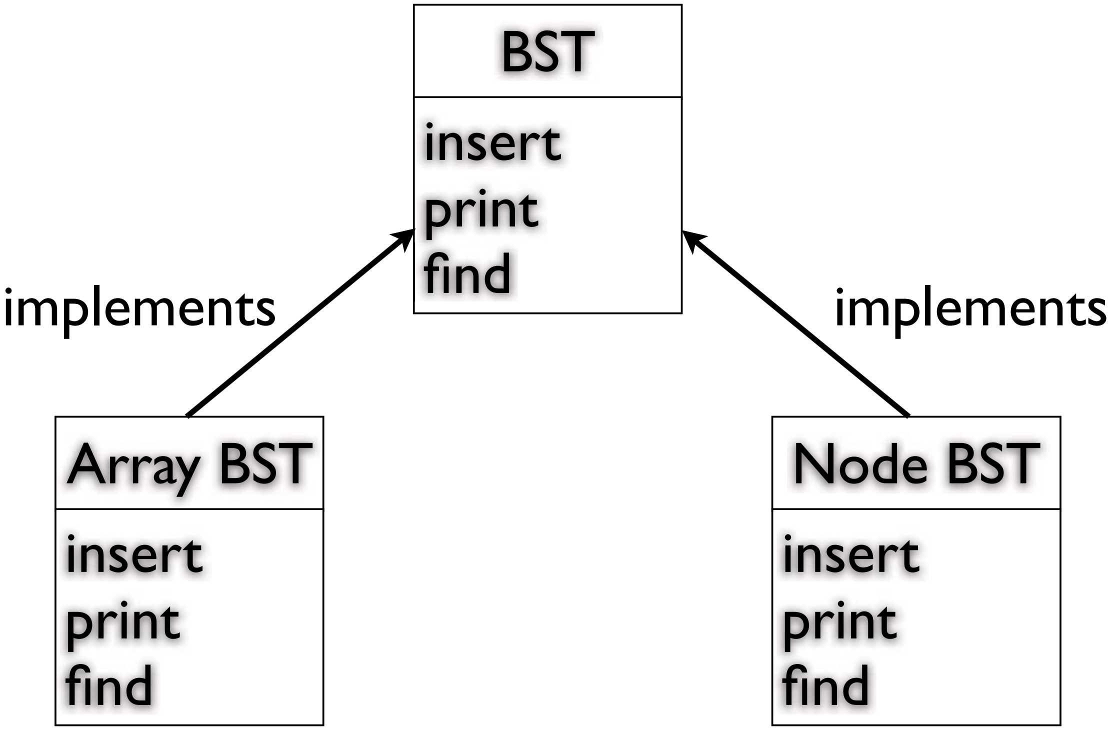
Sometimes you want to inherit only declarations, not definitions

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++

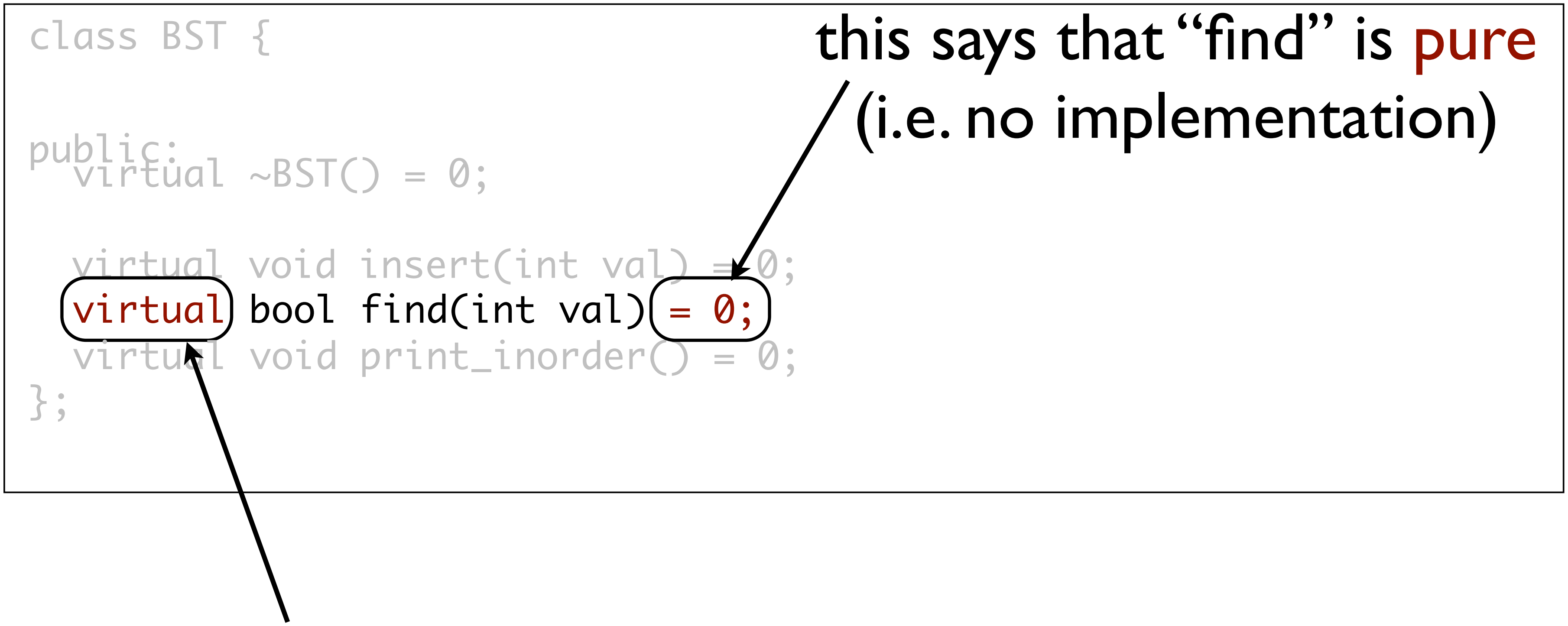
Use pure virtual functions

```
class BST {

 public:
  virtual ~BST() = 0;

  virtual void insert(int val) = 0;
  virtual bool find(int val) = 0;
  virtual void print_inorder() = 0;
};
```

(How would you do this in Java?)

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;
  virtual void print_inorder() = 0;
};
```

this says that “find” is pure (i.e. no implementation)

this says that “find” is virtual

Defining abstract methods in C++

Can we have non-virtual pure functions?

```
class BST {

 public:
  virtual ~BST() = 0;

  virtual void insert(int val) = 0;
  ~~virtual~~ bool find(int val) = 0;
  virtual void print_inorder() = 0;
};
```

Abstract classes in C++

Abstract base class

- a class with one or more pure virtual functions
- cannot be instantiated

```
BST bst = new BST();   // can’t do this!
```

- its subclass must implement the all of the pure virtual functions (or itself become an abstract class)

Extending an abstract base class

```
class NodeBST : public BST {

  Node* root;

 public:
  NodeBST();
  ~NodeBST();
  void insert(int val);
  bool find(int val);
  void print_inorder();
};
// implementation of the insert method using nodes
void NodeBST::insert(int val) {
  if (root == NULL) {
    root = new Node(val);
  } else {
    ...
  }
}
```

Constructors in abstract classes

Does it make sense to define a constructor?

The class will never be instantiated!

Constructors in abstract classes

Does it make sense to define a constructor?
The class will never be instantiated!

Yes! You should still create a constructor to initialize its members, since they will be inherited by its subclass.

Destructors in abstract classes

Does it make sense to define a destructor?

The class will never be created in the first place.

Destructors in abstract classes

Does it make sense to define a destructor?
The class will never be created in the first place.

Yes! Always define a virtual destructor in the base class, to make sure that the destructor of its subclass is called!

Pure virtual destructor

Can also define a destructor as pure.

```
class BST {

 public:
  virtual ~BST() = 0;

  virtual void insert(int val) = 0;
  virtual bool find(int val) = 0;
  virtual void print_inorder() = 0;
};
```

But must also provide a function body. Why?

```
BST::~BST() {}
```

Until next time...

Homework #5 (due 11:59 PM Tuesday)

- Designing & implementing type hierarchy for simple arithmetic expressions

Next lecture

- templates
- common C++ pitfalls
- C/C++ interview questions & tricks

References

Thinking in C++ (B. Eckel) Free online edition!

Essential C++ (S. Lippman)

Effective C++ (S. Meyers)

C++ Programming Language (B. Stroustrup)

Design Patterns (Gamma, Helm, Johnson, Vlissides)

Object-Oriented Analysis and Design with Applications (G. Booch, et. al)

Extra slides

Subtype: Student

```
#include <iostream>
#include <vector>
#include "MITPerson.h"
#include "Class.h"

class Student : public MITPerson {

  int course;
  int year;     // 1 = freshman, 2 = sophomore, etc.
  std::vector<Class*> classesTaken;

 public:
  Student(int id, std::string name, std::string address,
          int course, int year);
  void displayProfile();
  void addClassTaken(Class* newClass);
  void changeCourse(int newCourse);

};
```

what if this is private?

6.088 Introduction to C Memory Management and C++ Object-Oriented Programming
January IAP 2010