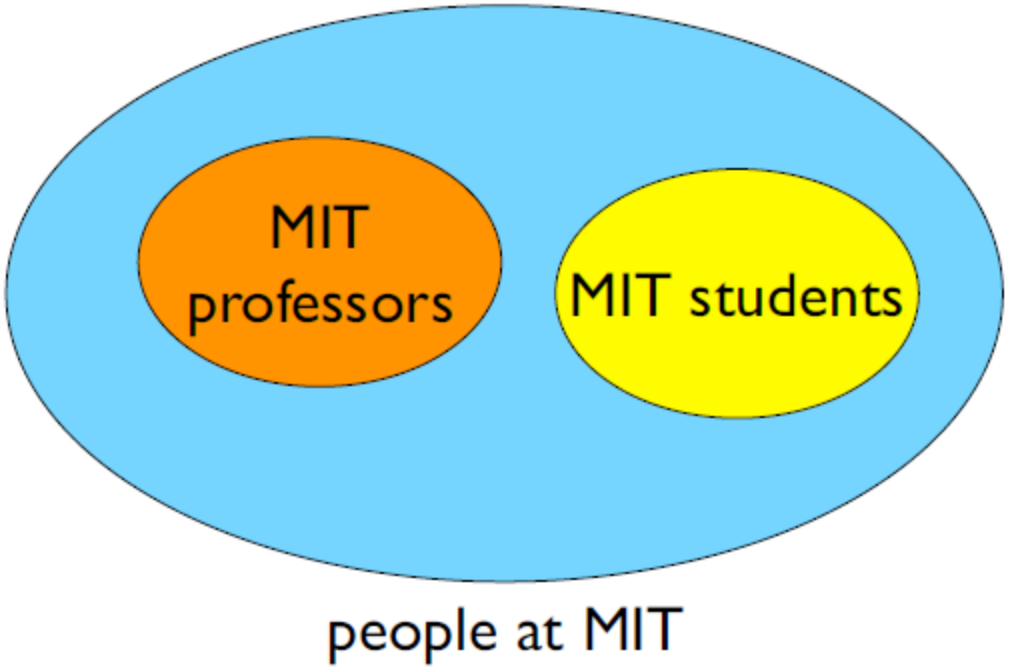
Inheritance & Polymorphism

금융공학 프로그래밍

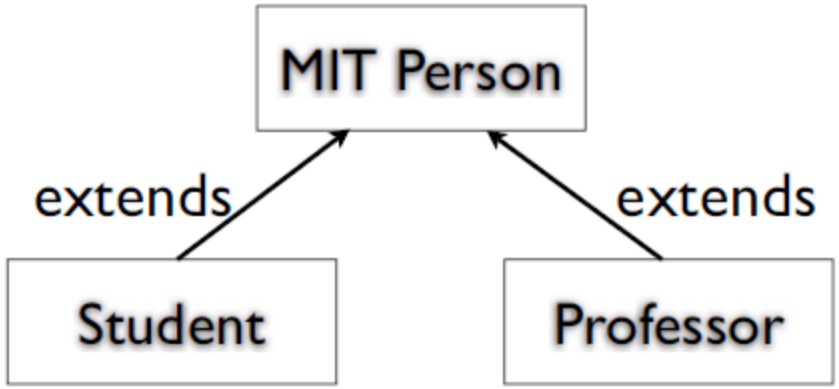
본 강의노트는 MIT opencourseware 에 게시된 Eunsuk Kang & Jean Yang의 강의노트에 기반하여 작성되었습니다.
출처: http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-088-introduction-to-c-memory-management-and-c-object-oriented-programming-january-iap-2010/

Types within a type

Some objects are distinct from others in some ways



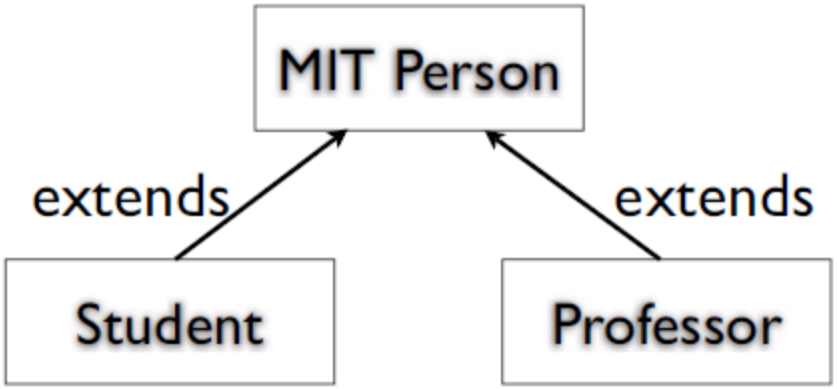
Type hierarchy



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

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

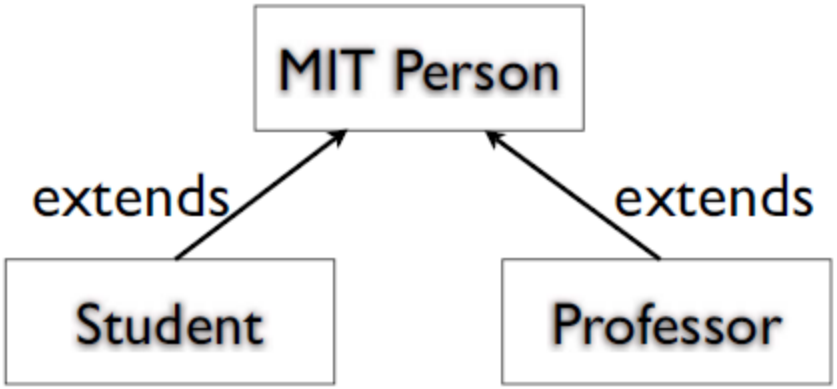
Type hierarchy



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:	Behaviors:
name	display profile
ID	change address
address	add a class taken
course number	change course
classes taken	
year	

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

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);

};
```

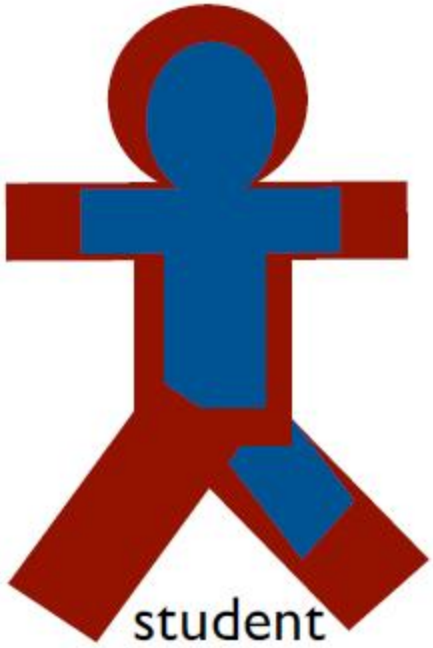
Constructing an object of subclass

```
// 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;
}
```

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

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();
```

```
----------------------------
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

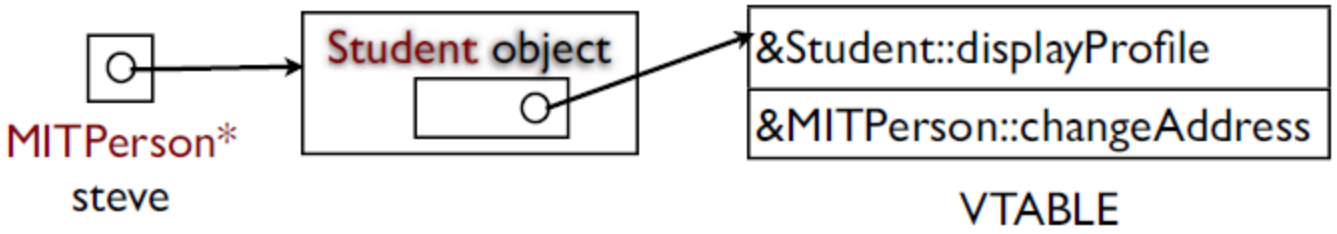
```
MITPerson* steve =
  new Student(911923, "Steve", "99 Cambridge St.", 18, 3);

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



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?

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

Abstract methods

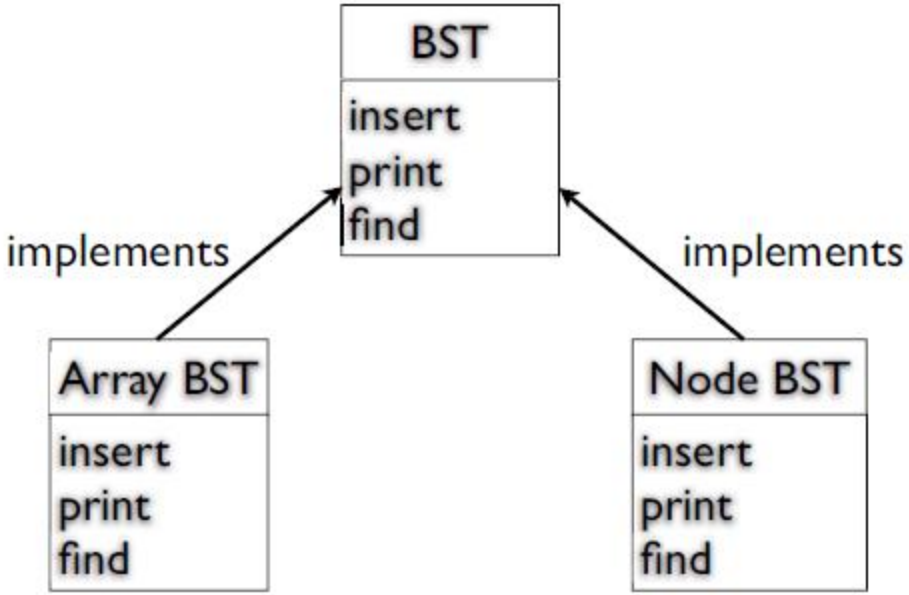
Sometimes you want to inherit only declarations, not definitions

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?)

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 {
    ...
  }
}
```

Q & A