Computer Programming Course

From Inheritance to Polymorphism

Mahdi Abbasi

Applicant for the Position Senior lectureship in Computer Science at Karlstad University.

Today's topic

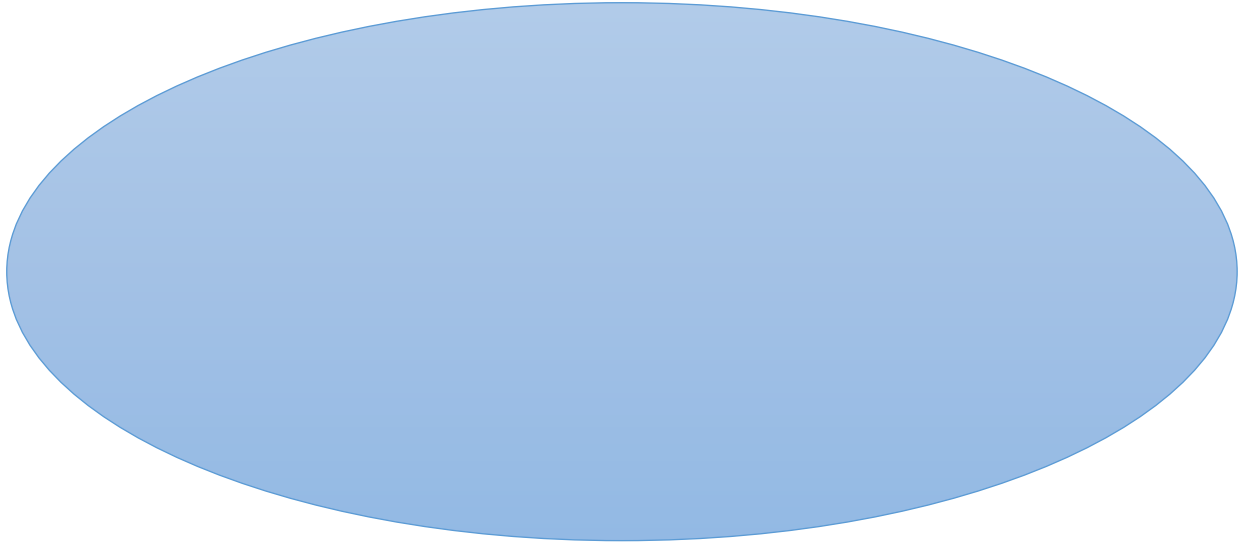
- Inheritance review
- Polymorphism

Slides adapted from:
Introduction To C Memory Management And C++ Object-Oriented Programming

Inheritance

Types

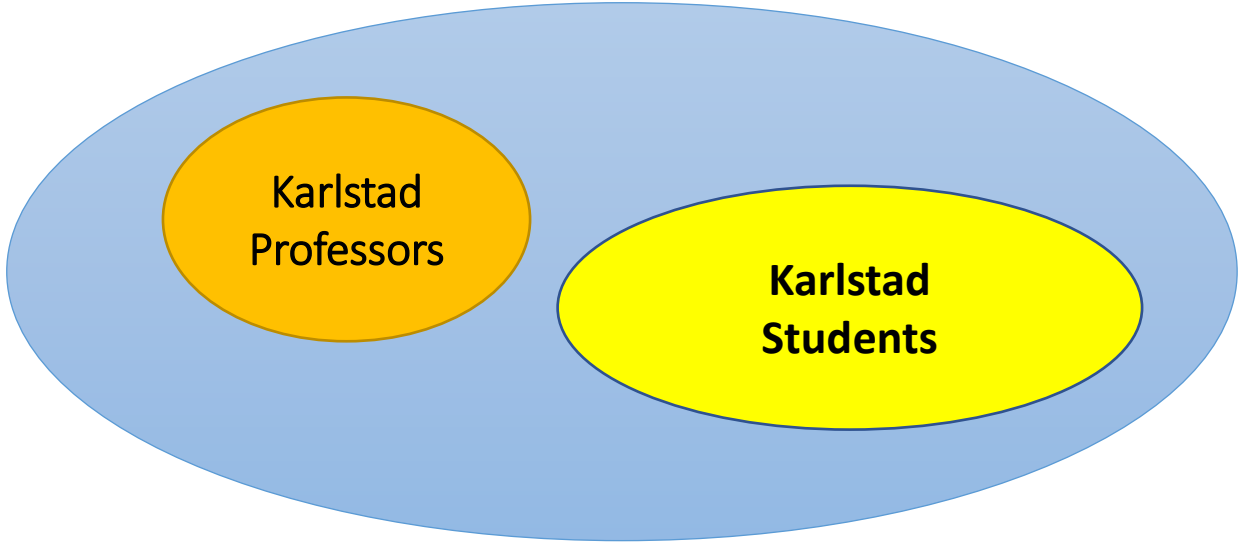
A Class defines a set of objects, or a type



People at Karlstad

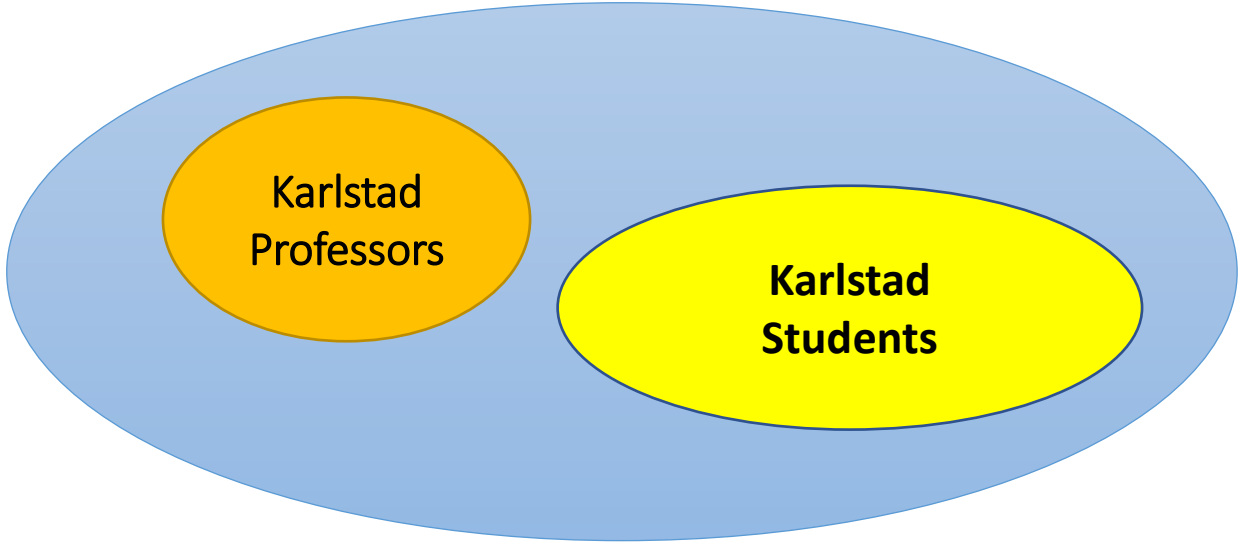
Types within a type

Some objects are distinct from others in some ways

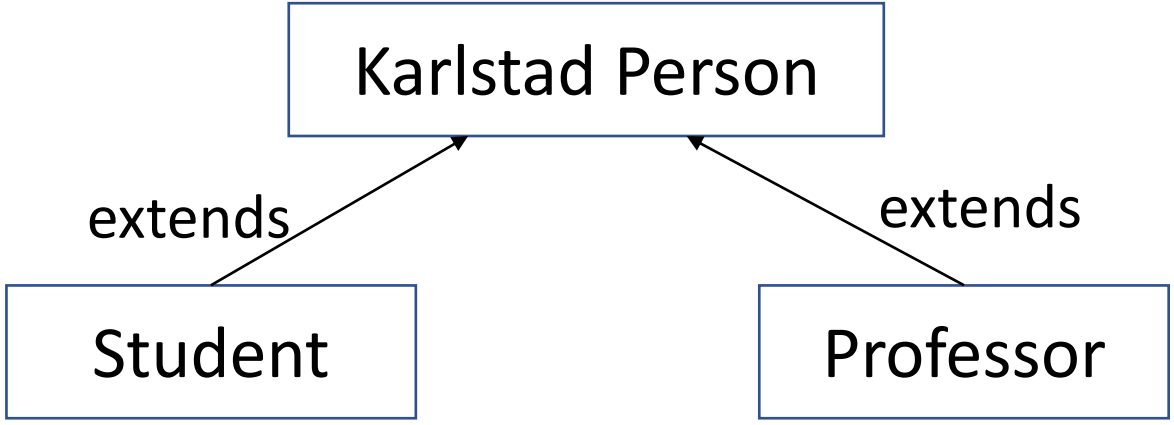


Subtype

Karlstad professor and student are subtypes of Karlstad people

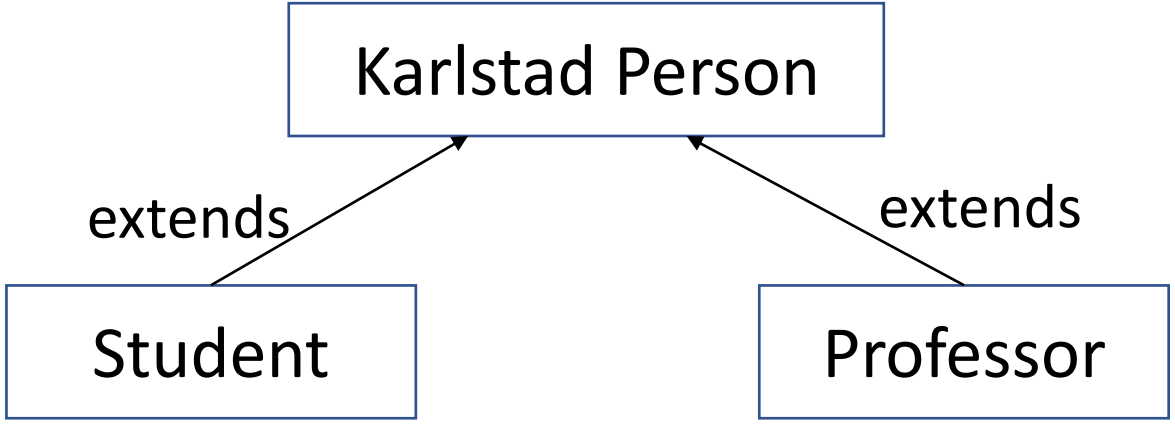


Type hierarchy



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

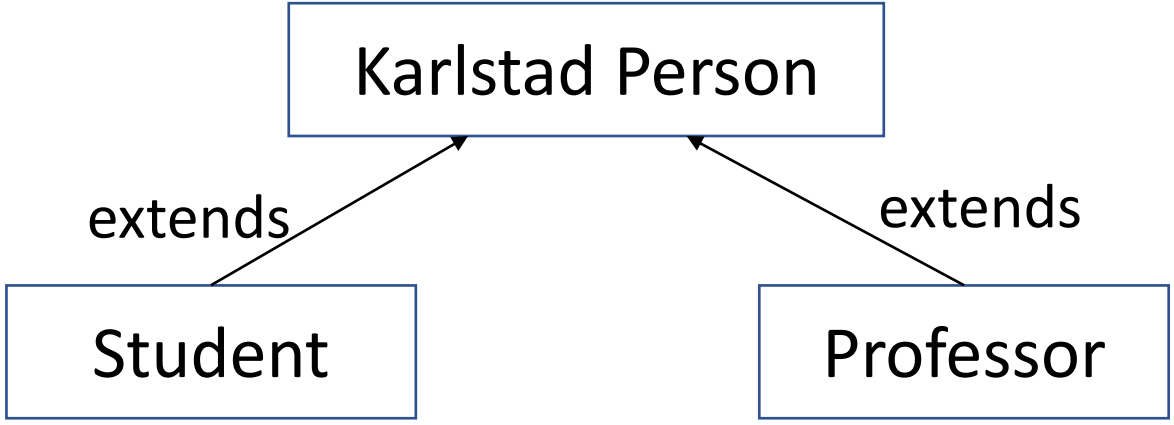
Type hierarchy



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

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

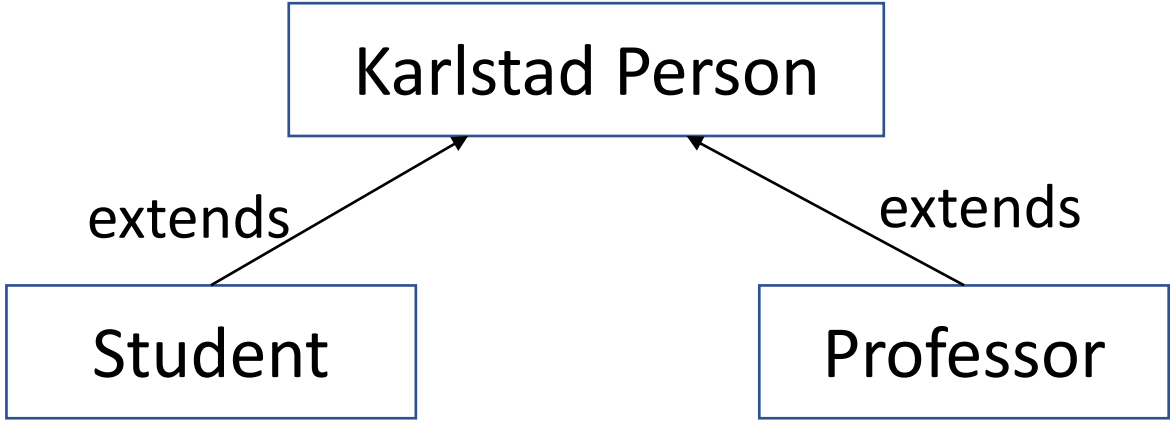
Type hierarchy



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 (lecturer, senior lecturer, etc.)
- add a class taught, promote

Inheritance

A subtype inherits characteristics and behaviors of its base type.

e.g. Each Karlstad student has

Characteristics:

name
ID
address
course number
classes taken
year

Behaviors:

display profile
change address
add a class taken
change course

Base type: KarlstadPerson

```
#include <string>

class KarlstadPerson {

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

 public:

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

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

};
```

Base type: KarlstadPerson

```
#include <string>

class KarlstadPerson {

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

 public:

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

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

};
```

namespace prefix

Base type: KarlstadPerson

```
#include <string>

class KarlstadPerson {

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

 public:

  KarlstadPerson(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 "KarlstadPerson.h"
#include "Class.h"

class Student : public KarlstadPerson {

  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 "KarlstadPerson.h"
#include "Class.h"

class Student : public KarlstadPerson {

  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 "KarlstadPerson.h"
#include "Class.h"

class Student : public KarlstadPerson {

  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);
};
```

base type

Constructing an object of subclass

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

class Student : public KarlstadPerson {

  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) : KarlstadPerson(id, name, address) {
  this->course = course;
  this->year = year;
```

```
// in KarlstadPerson.cc

KarlstadPerson::KarlstadPerson(int id, std::string name, std::string address){
  this->id = id;
  this->name = name;
  this->address = address;
}
```

Constructing an object of subclass

```
// in Student.cc

Student::Student(int id, std::string name, std::string address,
                 int course, int year) : KarlstadPerson(id, name, address) {
  this->course = course;
  this->year = year;
```

Call to the base class constructor

```
// in KarlstadPerson.cc

KarlstadPerson::KarlstadPerson(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 = non yet
Year = 2

Overriding a method in base class

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

```
class Student : public KarlstadPerson {
  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 courses & classes
  void addClassTaken(Class* newClass);
  void changeCourse(int newCourse);
};
```

Overriding a method in base class

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

```
void Student::displayProfile(){ // definition in Student.cc
  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

```
KarlstadPerson* john= new KarlstadPerson(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 a **KarlstadPerson** object

Actual type vs. declared type

Every variable has a declared type at compile-time

But during run-time, the variable may refer to an object with an actual type
(either the same or a subclass of the declared type)

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

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

Calling an overridden function

```
KarlstadPerson* steve = new Student(911923, "Steve", "99 Cambridge St.", 18, 3);
steve->displayProfile();
```

Calling an overridden function

```
KarlstadPerson* steve = new Student(911923, "Steve", "99 Cambridge St.", 18, 3);
steve->displayProfile();
```

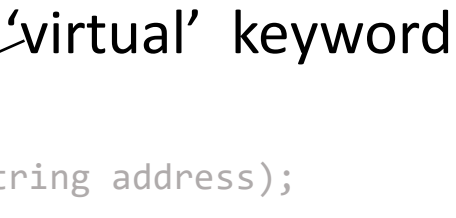
```
----------------------------
Name: Steve ID: 911923 Address: 99 Cambridge St.
----------------------------
```

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

Virtual functions

Declare overridden methods as virtual in the base

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



Calling a virtual function

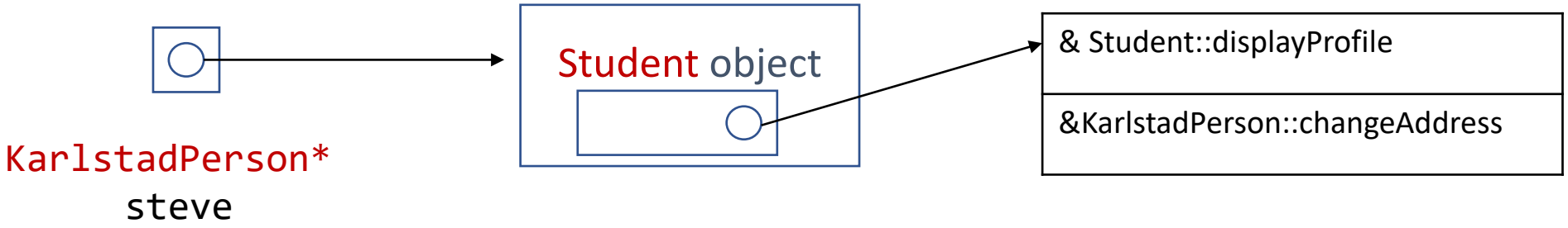
```
KarlstadPerson* steve = new Student(911923, "Steve", "99 Cambridge St.", 18, 3);
steve->displayProfile();
```

```
----------------------------
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
- Looking up during 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?

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!)

Please try the corresponding demo (available in course GitHub page)

Virtual destructor in KarlstadPerson

```
class KarlstadPerson {
 protected:
  int id;
  std::string name;
  std::string address;

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

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++. when constructor of a class is executed there is no VT in memory, means no virtual pointer defined yet. Hence the constructor should always be non-virtual.

Until next time…

Assignments

- Homework #5 (due 11.59 PM Friday, 9.12.2022)
- Lab assignment #5

  Title: Completing Karlstad classes source code for processing salaries of university employees
- You may access the *lecture slides*, *demo codes*, *homework*, and *lab assignment* at:
  The course gitub:

  https://github.com/Mahdi-abbasi1358/CProgramming

Next lecture

- Pure virtual functions
- Abstract base classes
- C++ tricks

References

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

Essential C++ (S. Lipman)

Effective C++ (S. Meyers)

C++ Programming Language (B. Stroustrup)

Extra slides

Type casting

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

What will happen?

Type casting

```
KarlstadPerson* steve = new Student(911923, "Steve", "99 Cambridge St.", 18, 3);
Class* c1 = new Class("6.088");
steve->addClassTaken(c1); X
```

Can only invoke methods of declared type!

“addClassTaken” is not a member of KarlstadPerson

Type casting

```
KarlstadPerson* 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);
```

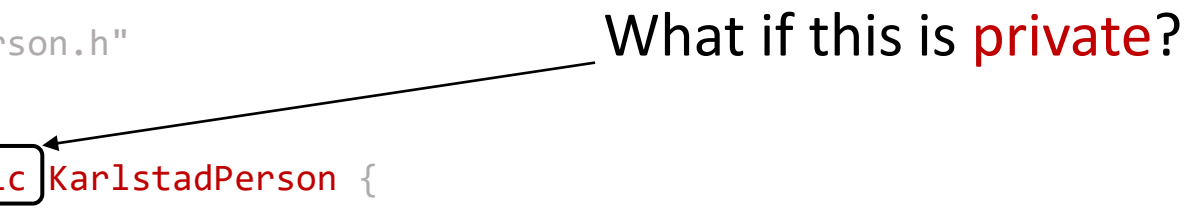
Cheaper but dangerous! No runtime check!

```
KarlstadPerson* p = KarlstadPerson(…);
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!

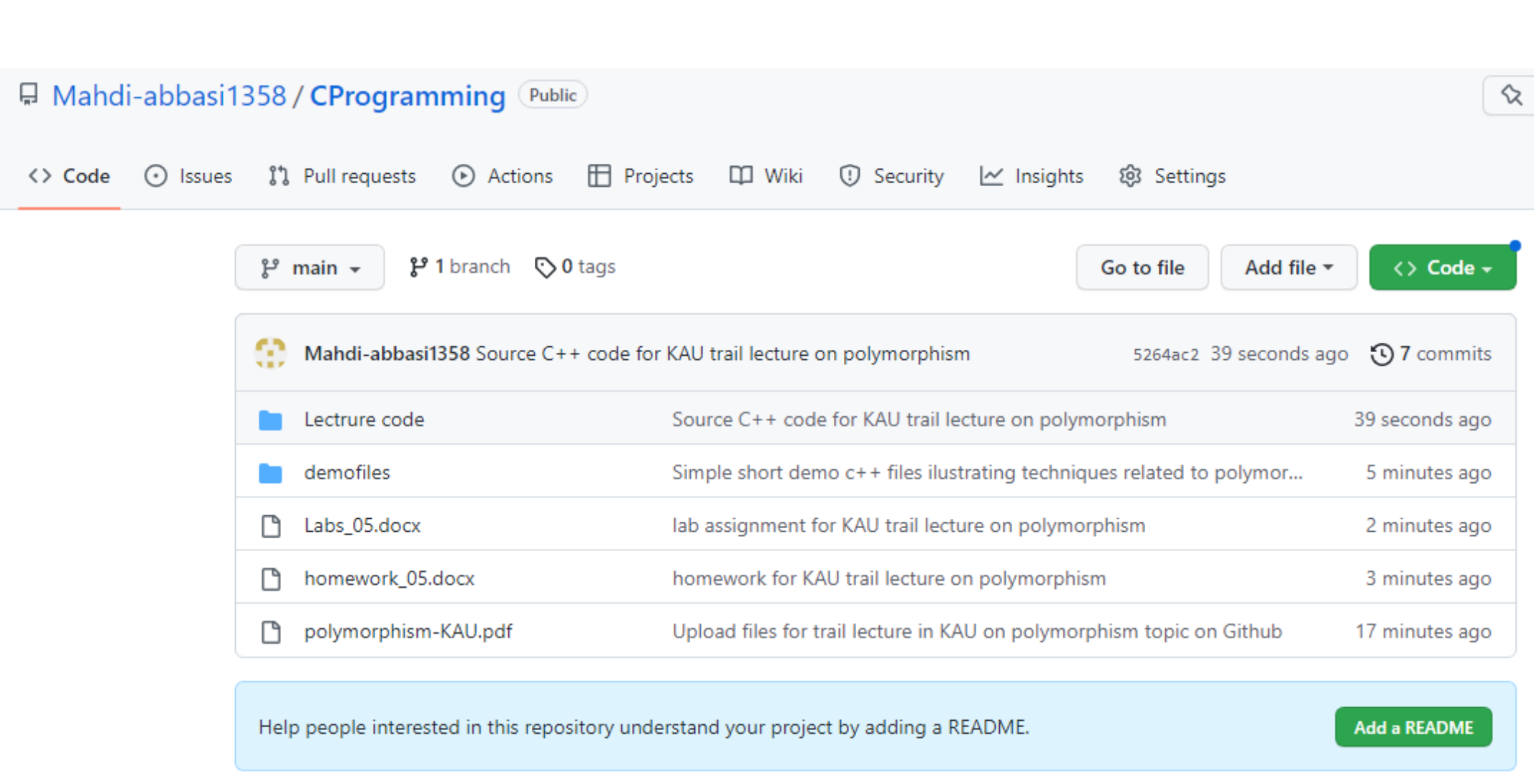
Subtype: Student

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

class Student : public KarlstadPerson {

  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);
};
```



Virtual destructor example

```
class Base1 {
    public:
           ~Base1() {std::cout << "~Base1()\n";}
};
class Derived1: public Base1 {
    public:
        ~Derived1() {cout<<" ~Derived1()\n“;};
};

class Base {
    public:
           virtual ~Base2() {std::cout << "~Base2()\n";}
};
class Derived2: public Base2 {
    public:
        ~Derived2() {cout<<" ~Derived2()\n“;};
}
int main(){
    Base1* bp= new Derived1; // Upcast
    delete bp;
    Base1* b2p= new Derived2; // Upcast
    delete b2p;
    return 0;
}
```


Mahdi-abbasi1358 / CProgramming
Public
Code
Issues
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
main
1 branch
0 tags
Go to file
Add file
Code
Mahdi-abbasi1358 Source C++ code for KAU trail lecture on polymorphism
5264ac2 39 seconds ago
7 commits
Lectrure code
Source C++ code for KAU trail lecture on polymorphism
39 seconds ago
demofiles
Simple short demo c++ files ilustrating techniques related to polymor...
5 minutes ago
Labs_05.docx
lab assignment for KAU trail lecture on polymorphism
2 minutes ago
homework_05.docx
homework for KAU trail lecture on polymorphism
3 minutes ago
polymorphism-KAU.pdf
Upload files for trail lecture in KAU on polymorphism topic on Github
17 minutes ago
Help people interested in this repository understand your project by adding a README.
Add a README