MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

06 C++ Class Templates and Exceptions

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

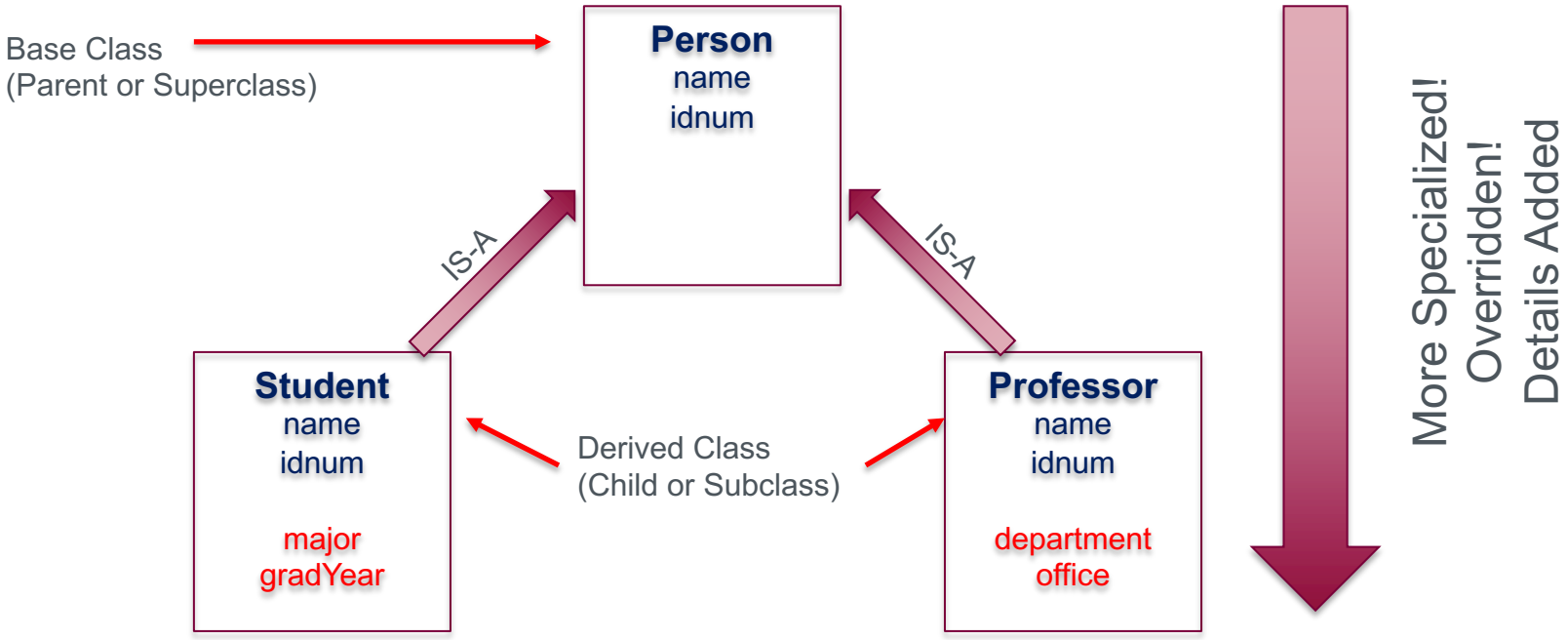
January 26, 2022

Review Inheritance

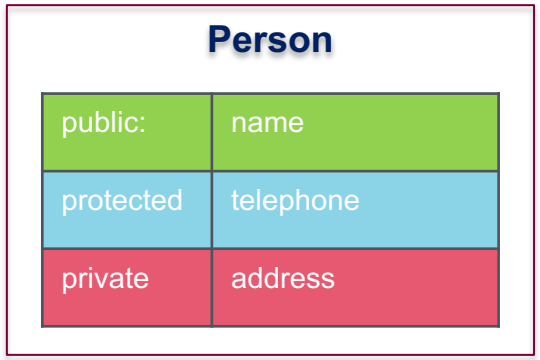
- And check the code

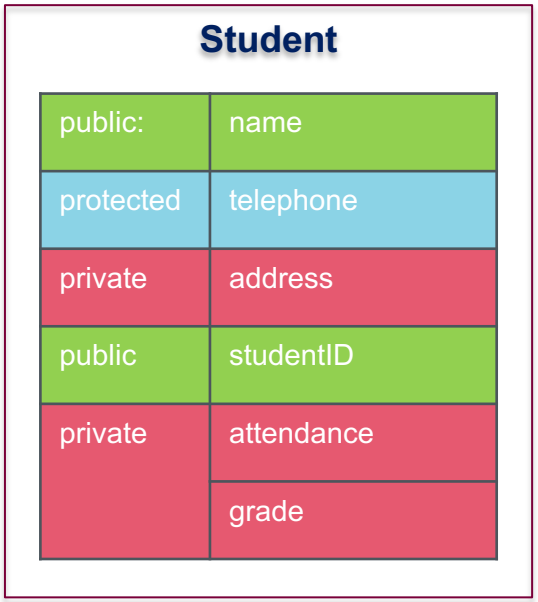
Inheritance

- Subclassing: define a class based on another class
  - Another class is parent class (or superclass)
  - New class is child class (subclass)
  - Hierarchical classification in a tree form
  - A way of "polymorphism" – we will discuss later!



Public Derivation

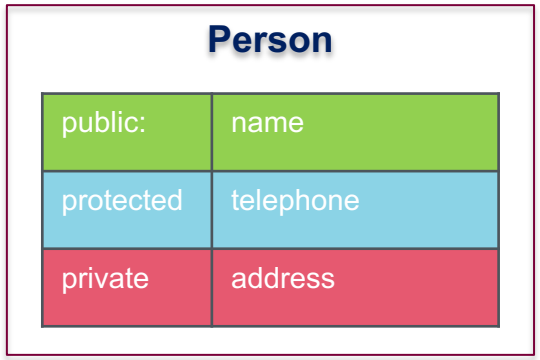


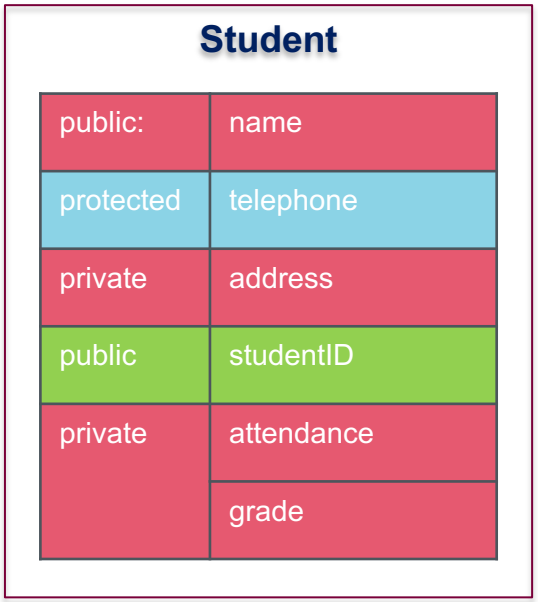



```
class Person {
  public:
    string name;
  protected:
    string telephone;
  private:
    string address;
};
```

```
class Student: public Person {
  public:
    int studentID;
  private:
    int attendance;
    double grade;
};
```

Private Derivation






```
class Person {
  public:
    string name;
  protected:
    string telephone;
  private:
    string address;
};
```

```
class Student: private Person {
  public:
    int studentID;
  private:
    int attendance;
    double grade;
};
```

Inheritance: A Mechanism for Reuse

```
class Person {  // Person (base class)
    private:
        string name;   // name
        string idNum;  // university ID number

    public:
        Person(const string& nm, const string& id);
        void print();        // print information
        string getName();  // retrieve name
};
```

```
class Student : public Person {  // Student (derived from Person)
    private:
        string major;  // major subject
        int gradYear;  // graduation year

    public:
        Student(const string& nm, const string& id, const string& maj, int year);
        void print();                                  // print information
        void changeMajor(const string& newMajor);  // change major

};
```

shared print()

Inheritance

```
class Person {  // Person (base class)
    private:
        string name;   // name
        string idNum;  // university ID number

    public:
        Person(const string& nm, const string& id);
        void print();      // print information
        string getName();  // retrieve name
};
```

```
Person::Person(const string& nm, const string& id)
    : name(nm), idNum(id)  // initialize name and id
    { }

void Person::print() { // definition of Person print
    cout << "Name " << name << ", " << "IDnum " << idNum << endl;
}

string Person::getName() { // definition of Person getName
    return name;
}
```

Inheritance

```
class Student : public Person {  // Student (derived from Person)
    private:
        string major;  // major subject
        int gradYear;  // graduation year

    public:
        Student(const string& nm, const string& id, const string& maj, int year);
        void print();                                  // print information
        void changeMajor(const string& newMajor);  // change major

};
```

Derived class must call base class's constructor

Base class's constructor must be in the initialization list!

```
Student::Student(const string& nm, const string& id, const string& maj, int year)
    : Person(nm, id), major(maj), gradYear(year)
    { }

void Student::print() {   // definition of Student print
    Person::print();      // first print Person information
    cout << "Major " << major << ", Year " << gradYear << endl;  // then student-specific info
}

void Student::changeMajor(const string& newMajor) {  // definition of Student print
    major = newMajor;
}
```

Inheritance

```
class Student : public Person {  // Student (derived from Person)
    private:
        string major;  // major subject
        int gradYear;  // graduation year

    public:
        Student(const string& nm, const string& id, const string& maj, int year);
        void print();                                  // print information
        void changeMajor(const string& newMajor);  // change major

};
```

Constructor order:
- base class => derived class

Destructor order:
- derived class => base class

```
Student::Student(const string& nm, const string& id, const string& maj, int year)
    : Person(nm, id), major(maj), gradYear(year)
    { }

void Student::print() {   // definition of Student print
    Person::print();      // first print Person information
    cout << "Major " << major << ", Year " << gradYear << endl;  // then student-specific info
}

void Student::changeMajor(const string& newMajor) {  // definition of Student print
    major = newMajor;
}
```

Inheritance

```
class Student : public Person {  // Student (derived from Person)
    private:
        string major;  // major subject
        int gradYear;  // graduation year

    public:
        Student(const string& nm, const string& id, const string& maj, int year);
        void print();                                  // print information
        void changeMajor(const string& newMajor);  // change major

};
```

calling parent's print()

```
Student::Student(const string& nm, const string& id, const string& maj, int year)
    : Person(nm, id), major(maj), gradYear(year)
    { }

void Student::print() {    // definition of Student print
    Person::print();       // first print Person information
    cout << "Major " << major << ", Year " << gradYear << endl;  // then student-specific info
}

void Student::changeMajor(const string& newMajor) {  // definition of Student print
    major = newMajor;
}
```

Inheritance

- Testing Inheritance

```
int main() {
    Person person("Mary", "12-345");  // declare a Person
    Student student("Bob", "98-764", "Math", 2012);  // declare a Student
    cout << student.getName() << endl;  // invokes Person::getName()
    person.print();   // invokes Person::print()
    student.Person::print();  // invokes studnet's parent's print() !!
    student.print();  // invokes Student::print()
    //person.changeMajor("Physics");   // ERROR!
    student.changeMajor("English");  // Okay
    student.print();

    return EXIT_SUCCESS;
}
```

Bob
Name Mary, IDnum 12-345
Name Bob, IDnum 98-764
Name Bob, IDnum 98-764
Major Math, Year 2012
Name Bob, IDnum 98-764
Major English, Year 2012

Static Binding

- When determining which member function to call, C++'s default action is to consider an object's declared type, not its actual type.

```
class Parent {
public:
    void print() {
        cout << "I am parent's print" << endl;
    }
};

class Child : public Parent {
public:
    void print() {
        cout << "I am child's print" << endl;
    }
};
```

```
int main() {
    Child *child = new Child( );
    child->print();

    Parent *father = child;
    father->print();

    delete child;

    return EXIT_SUCCESS;
}
```

Static binding

Output:
I am child's print
I am parent's print

Dynamic Binding

- Polymorphism (Ability to have many forms)
  - Objects with different internal structures can share the same external interface
  - virtual function and class derivation are means to realize polymorphism

```
class Parent {
public:
    virtual void print() {
        cout << "I am parent's print" << endl;
    }
};

class Child : public Parent {
public:
    void print() {
        cout << "I am child's print" << endl;
    }
};
```

```
int main() {
    Child *child = new Child( );
    child->print();

    Parent *father = child;
    father->print();

    delete child;

    return EXIT_SUCCESS;
}
```

Dynamic binding

Output:
I am child's print
I am child's print

Virtual vs Non-Virtual Functions

- Dynamic (run-time) binding vs Static (compile-time) binding

```cpp
class Parent {
public:
    virtual void vprint() {
        cout << "Virtual: I am parent's print" << endl;
    }
    void nvprint() {
        cout << "Non-Virtual: I am parent's print" << endl;
    }
};

class Child : public Parent {
public:
    void vprint() {
        cout << "Virtual: I am child's print" << endl;
    }

    void nvprint() {
        cout << "Non-Virtual: I am child's print" << endl;
    }
};
```

```cpp
int main() {
    Parent father;
    Child son;

    Parent *par_pt = &son;

    father.vprint();     // Virtual: I am parent's print
    father.nvprint();    // Non-Virtual: I am parent's print

    son.vprint();        // Virtual: I am child's print
    son.nvprint();       // Non-Virtual: I am child's print

    par_pt -> vprint();  // Virtual: I am child's print
    par_pt -> nvprint(); // Non-Virtual: I am parent's print


    return EXIT_SUCCESS;
}
```

Output:
Virtual: I am parent's print
Non-Virtual: I am parent's print
Virtual: I am child's print
Non-Virtual: I am child's print
Virtual: I am child's print
Non-Virtual: I am parent's print

Function Template

```
int integerMin(int a, int b)          // returns the minimum of a and b
  { return (a < b ? a : b); }
```

- Useful, but what about min of two doubles?
  - C-style answer: double doubleMin(double a, double b)

- Function template is a mechanism that enables this
  - Produces a generic function for an arbitrary type T.

```
template <typename T>
T genericMin(T a, T b) {              // returns the minimum of a and b
  return (a < b ? a : b);
}
```

Function Template

- Function template is a mechanism that enables this
  - Produces a generic function for an arbitrary type T.

```
template <typename T>
T genericMin(T a, T b) {                    // returns the minimum of a and b
  return (a < b ? a : b);
}

  cout << genericMin(3, 4) << ' '     // = genericMin<int>(3,4)
    << genericMin(1.1, 3.1) << ' '    // = genericMin<double>(1.1, 3.1)
    << genericMin('t', 'g') << endl; // = genericMin<char>('t','g')
```

Function Template

- Function overloading
  - Same function name, but different function prototypes
  - These functions do not have to have the same code
  - Does not help in code reuse, but helps in having a consistent name
- Function template
  - Same code piece, which applies to different types

```
int abs(int n) {
    return n >= 0 ? n : -n;
}

double abs(double n) {
    return (n >= 0 ? n : -n);
}

int main( ) {
    cout << "absolute value of " << -123;
    cout << " = " << abs(-123) << endl;
    cout << "absolute value of " << -1.23;
    cout << " = " << abs(-1.23) << endl;
}
```

Class Template

- In addition to function, we can define a generic template class
  - Example: BasicVector
    - Stores a vector of elements
    - Can access i-th element using [ ] just like an array (Vect class in this week's tutorial)

```
template <typename T>
class BasicVector {                    // a simple vector class
public:
  BasicVector(int capac = 10);         // constructor
  T& operator[](int i)                 // access element at index i
    { return a[i]; }
  // ... other public members omitted
private:
  T* a;                                // array storing the elements
  int capacity;                        // length of array a
};
```

Class Template

- BasicVector
  - Constructor code?

```
template <typename T>                    // constructor
BasicVector<T>::BasicVector(int capac) {
  capacity = capac;
  a = new T[capacity];                   // allocate array storage
}
```

- How to use?

```
BasicVector<int>     iv(5);     iv[3] = 8;
BasicVector<double>  dv(20);    dv[14] = 2.5;
BasicVector<string>  sv(10);    sv[7] = "hello";
```

Class Template

- The actual argument in the instantiation of a class template can itself be a templated type
- Example: Two-dimensional array of int

```
BasicVector<BasicVector<int> > xv(5); // a vector of vectors
// ...
xv[2][8] = 15;
```

- BasicVector consisting of 5 elements, each of which is a BasicVector consisting of 10 integers
  - In other words, 5 by 10 matrix

Questions?