Not So Tiny Task №7 (2 points)

Implement a hierarchy of classes for symbolic differentiation of expressions.

- Base class: Expression;
- Derived classes: Binary, Unary, Add, Sub, Mult, Div, Exponent, Var, Val;
- Base class expression should contain pure virtual function Expression* diff(std::string var); (its implementations should return differentiation result for the expression by the given variable);
- Tests should be prepared as usual.

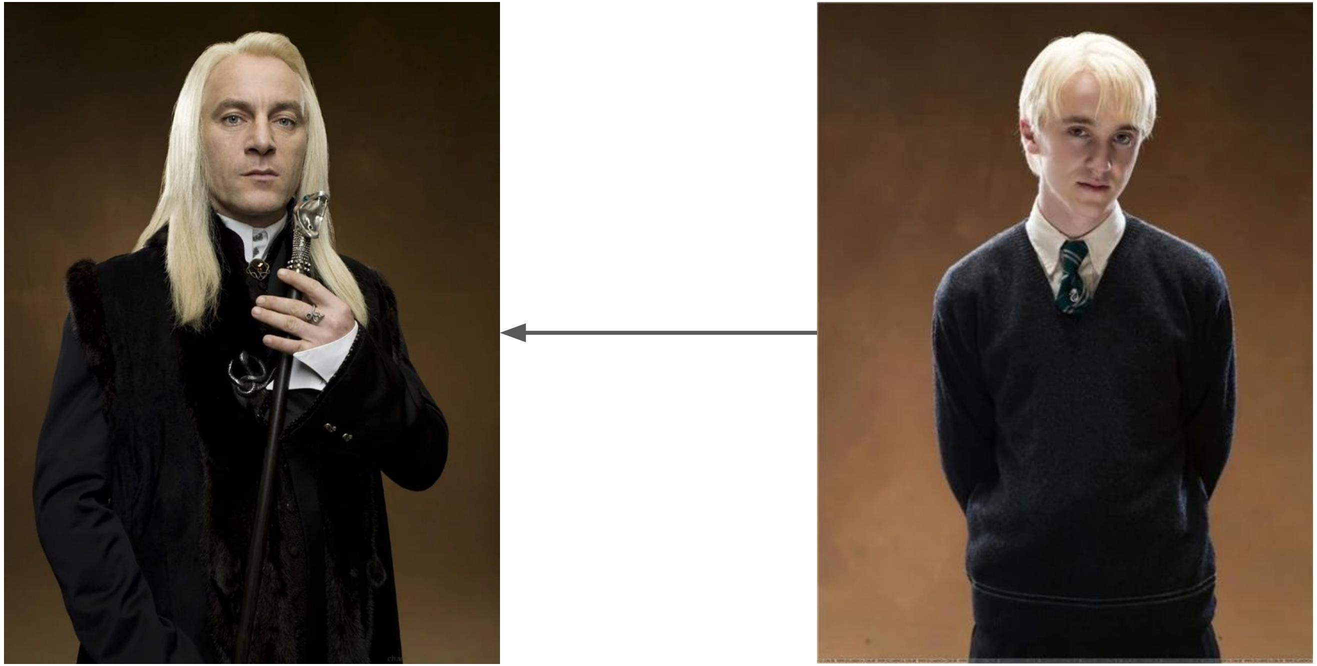
Not So Tiny Task №7 (2 points)

```
Example:

Expression* e = new Add(new Var("x"),
                        new Mult(new Val(10), new Var("y")));

// e = x + 10*y

Expression* res1 = e->diff("x");
// res1 = 1 + (0*y + 10*0) (it is ok to have non-simplified exprs)

Expression* res2 = e->diff("y");
// res2 = 0 + (0*y + 10*1) (it is ok to have non-simplified exprs)
```

System Programming with C++

Inheritance and virtual calls

Inheritance



Inheritance

```
class Person {
   const char* name;
   size_t age;
public:
   Person(): name("John Doe"), age(33) {}
   Person(const char* n, int a): name(n), age(a) {}

   const char* getName() const { return name; }
   size_t getAge() const { return age; }
   ...
};
```

Inheritance

```
class Person {
    const char* name;
    size_t age;
public:
    Person(): name("John Doe"), age(33) {}
    Person(const char* n, int a): name(n), age(a) {}

    const char* getName() const { return name; }
    size_t getAge() const { return age; }
    ...
};
```

```
Person:
  const char* name
  size_t age
```

Imagine we have one more class

```
Person:
  const char* name
  size_t age
```

```
Student:
  const char* name
  size_t age
  - - - - - - - - -
  size_t group
  size_t id
```

Imagine we have one more class

```
Person:
   const char* name
   size_t age
```

Common part

```
Student:
   const char* name
   size_t age
   ----------------
   size_t group
   size_t id
```

Similar fields and methods => code duplication!

Imagine we have one more class

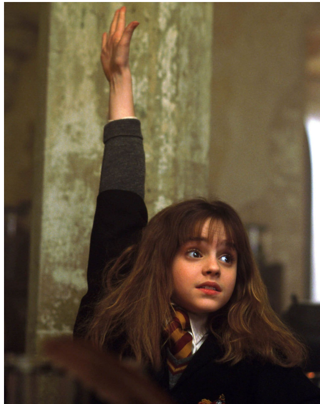
```
Person:
   const char* name
   size_t age
```

Common part

```
Student:
   const char* name
   size_t age
   - - - - - - - - - -
   size_t group
   size_t id
```

Similar fields and methods => code duplication!

How can we solve that?



Imagine we have one more class

```
Student:
  const char* name
  size_t age
  - - - - - - - - -
  size_t group
  size_t id
```

⟹

```
Student:
  Person base
  - - - - - - - - -
  size_t group
  size_t id
```

Imagine we have one more class

```
Student:
  const char* name
  size_t age
  - - - - - - - - - - - -
  size_t group
  size_t id
```

$\Longrightarrow$

```
Student:
  Person base
  - - - - - - - - - - - -
  size_t group
  size_t id
```

This is called composition.

Imagine we have one more class

```
Student:
  const char* name
  size_t age
- - - - - - - - - - -
  size_t group
  size_t id
```

⟹

```
Student:
  Person base
- - - - - - - - - - -
  size_t group
  size_t id
```

This is called composition.

There are some benefits of such approach, but it just doesn't look logical here (why Person should be a part of Student?)

Imagine we have one more class

```
Student:
  const char* name
  size_t age
  ----------------
  size_t group
  size_t id
```

$\Longrightarrow$

```
Student:
  Person base
  ----------------
  size_t group
  size_t id
```

This is called composition.

There are some benefits of such approach, but it just doesn't look logical here (why Person should be a part of Student?)

Also: how can I get a name from Student?

Imagine we have one more class

```
Student:
  const char* name
  size_t age
- - - - - - - - - - -
  size_t group
  size_t id
```

$\Longrightarrow$

```
Student:
  Person base
- - - - - - - - - - -
  size_t group
  size_t id
```

This is called composition.

There are some benefits of such approach, but it just doesn't look logical here (why Person should be a part of Student?)

Also: how can I get a name from Student? Some forwarding method to base => boilerplate code! Something we want to get rid of.

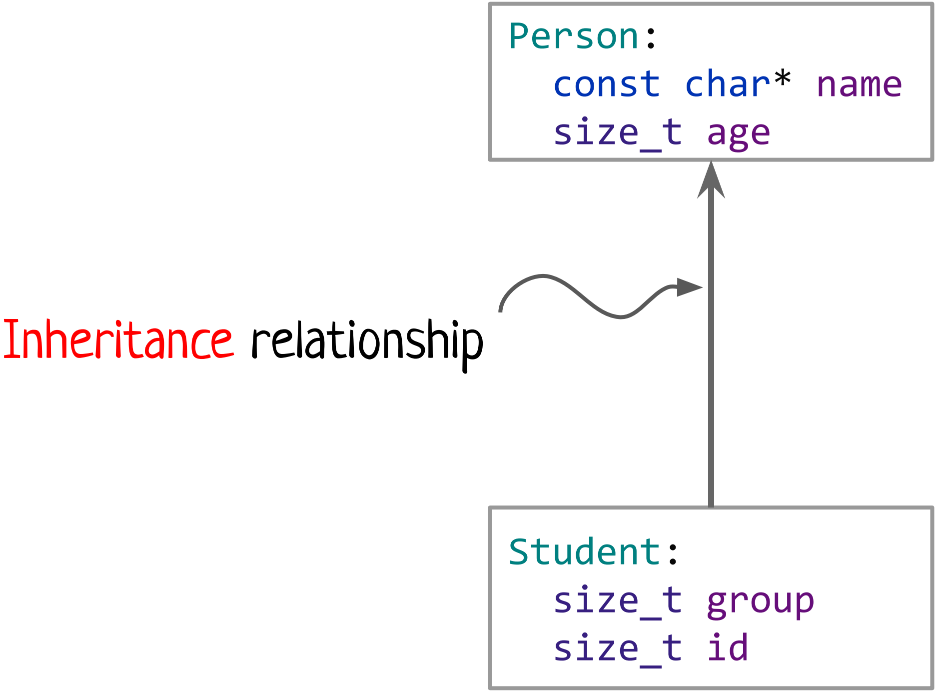
```
Person:
  const char* name
  size_t age
```

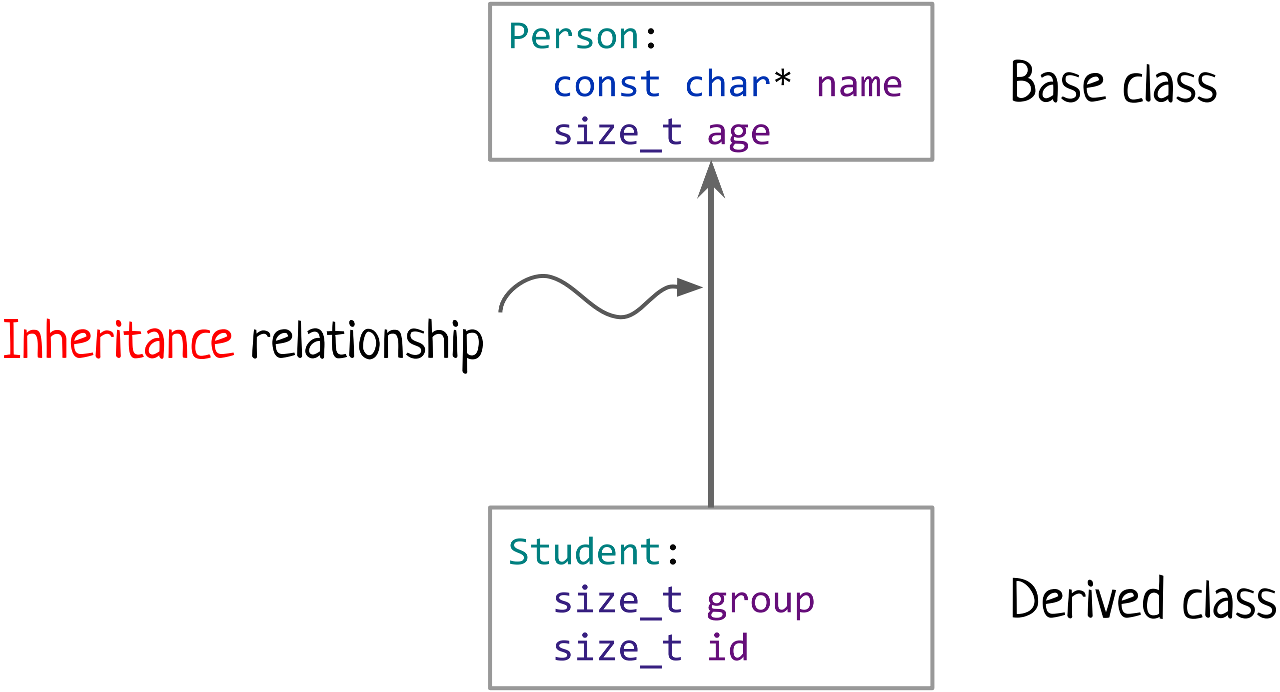
```
Person:
  const char* name
  size_t age
```

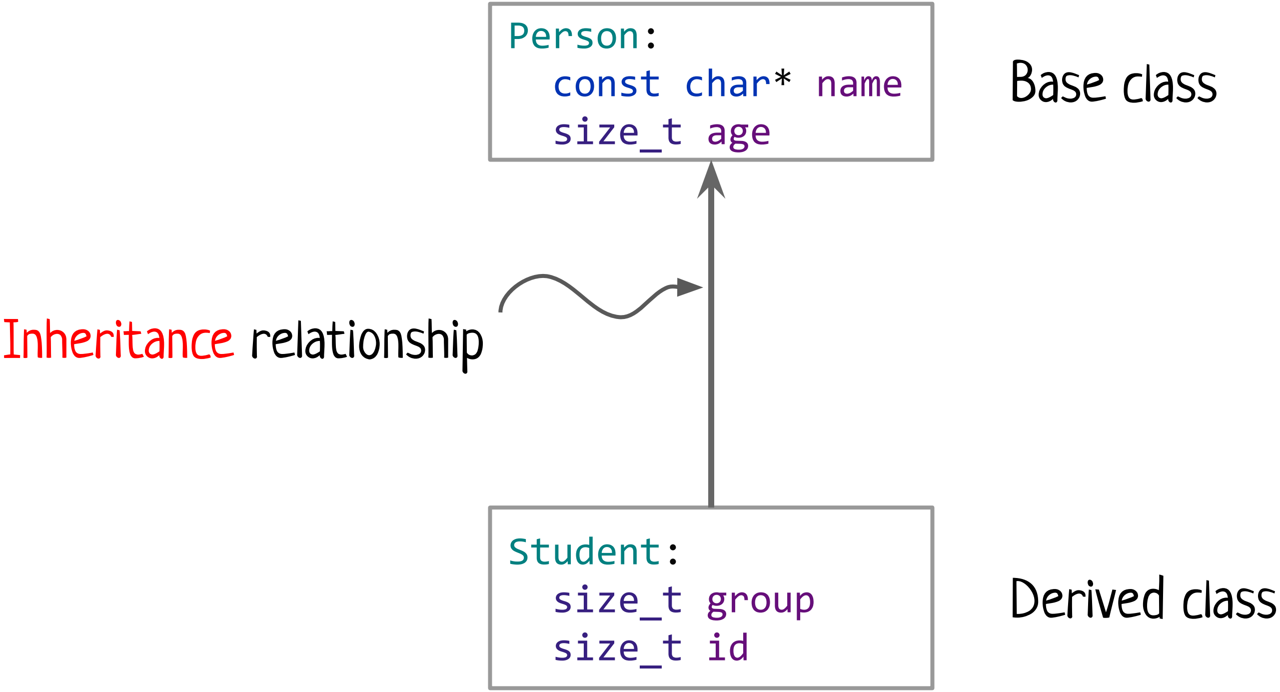
```
Student:
  size_t group
  size_t id
```

```
Person:
  const char* name
  size_t age
```

```
Student:
  size_t group
  size_t id
```

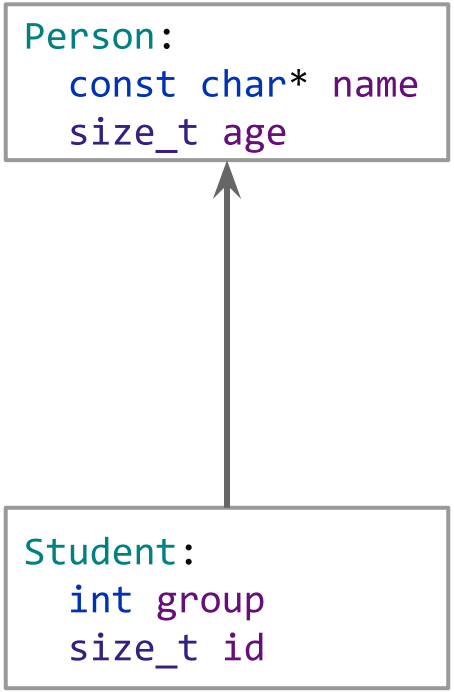

Person:
const char* name
size_t age
Inheritance relationship
Student:
size_t group
size_t id


Person:
const char* name
size_t age
Base class
Inheritance relationship
Student:
size_t group
size_t id
Derived class

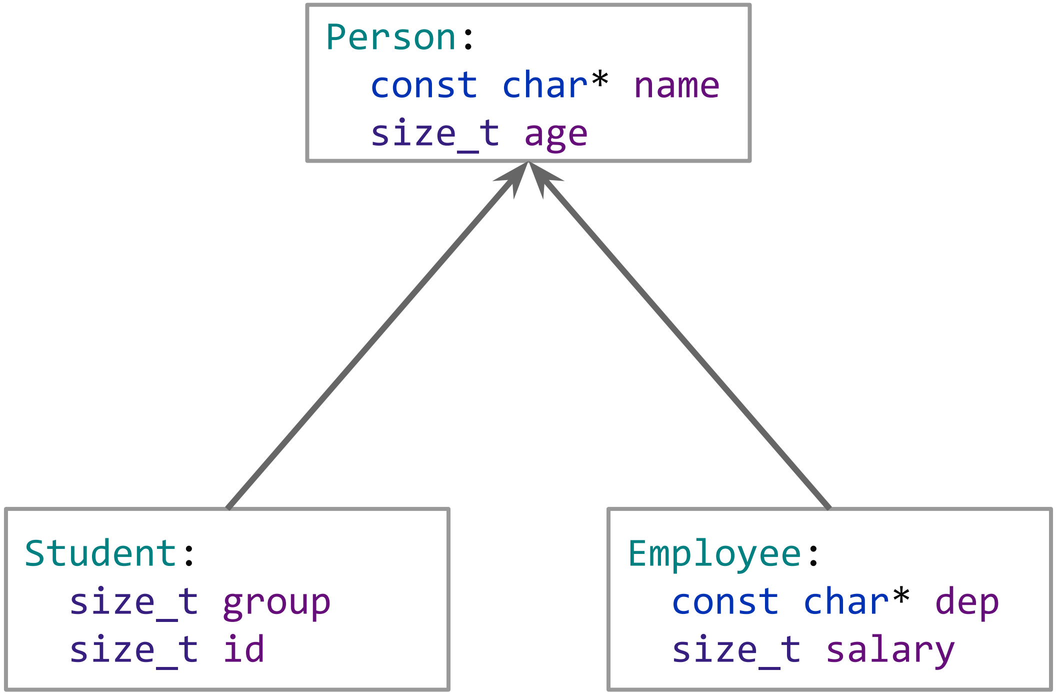


Derived class extends base: «Student — is a Person, who also has some group and id».
It specifies the particular case of base class, narrows set of objects.

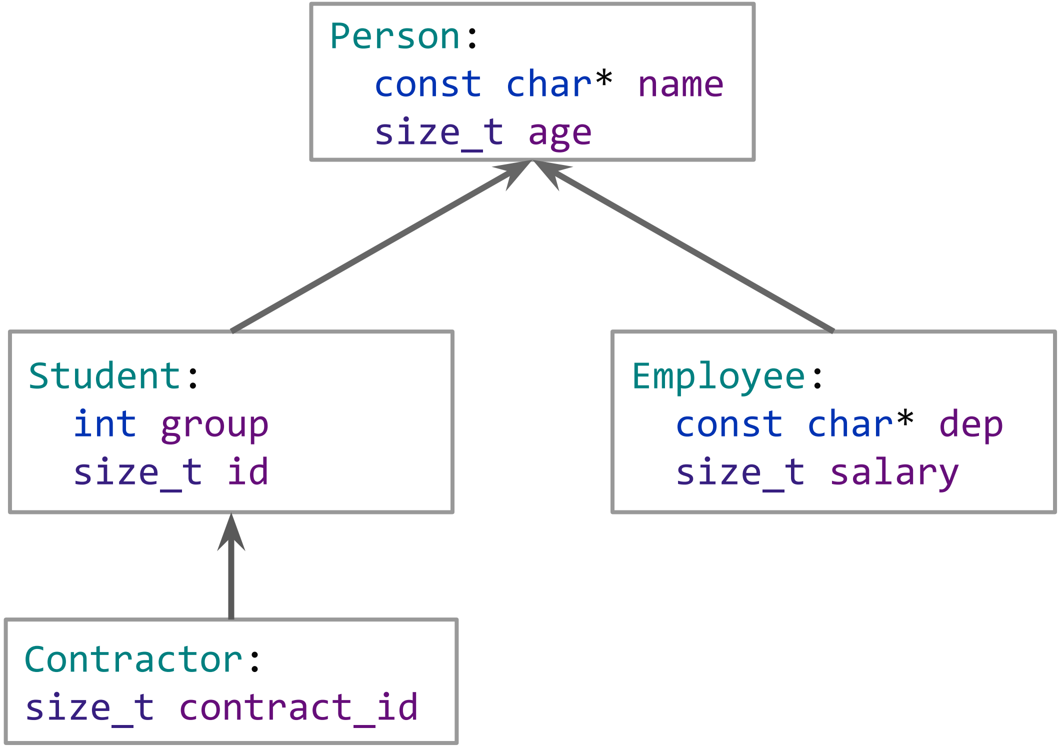
Inheritance



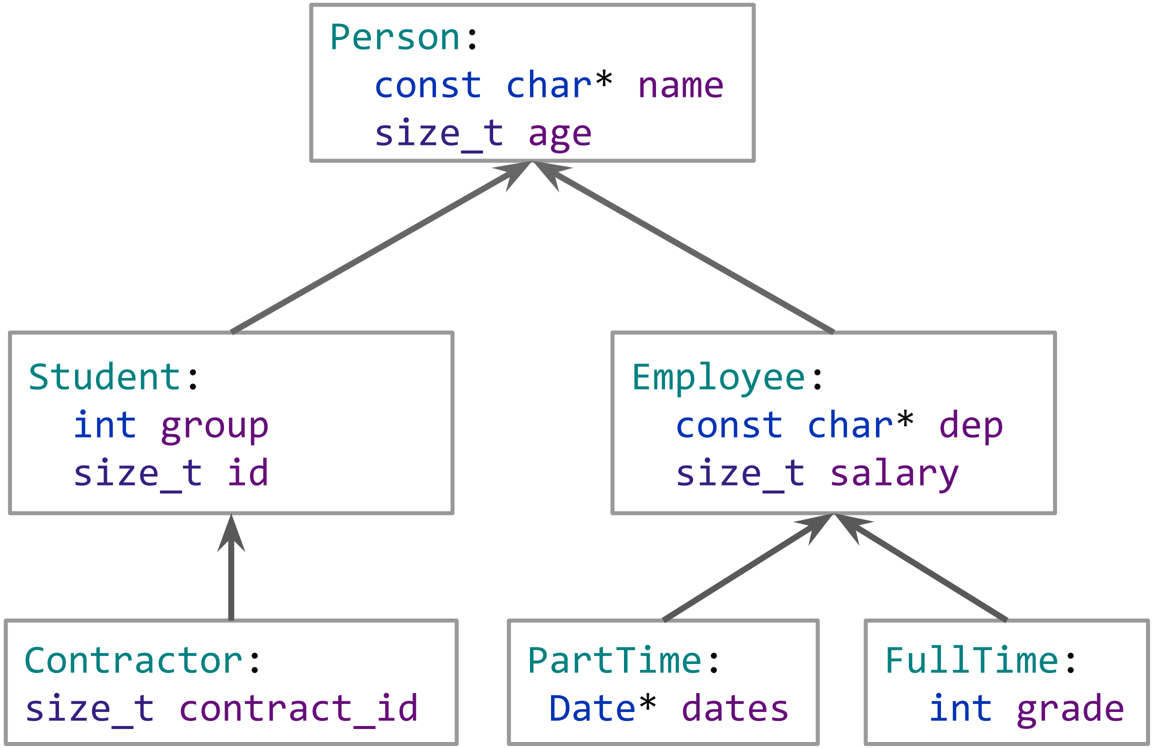
Inheritance



Inheritance

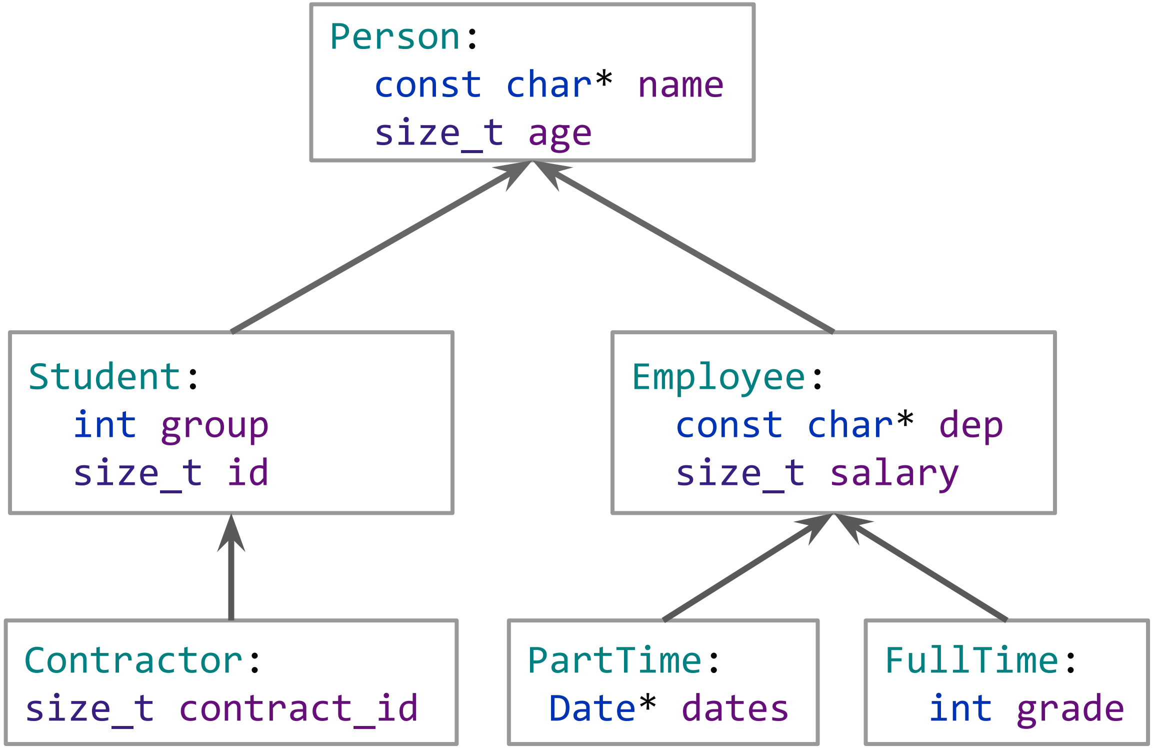


Inheritance



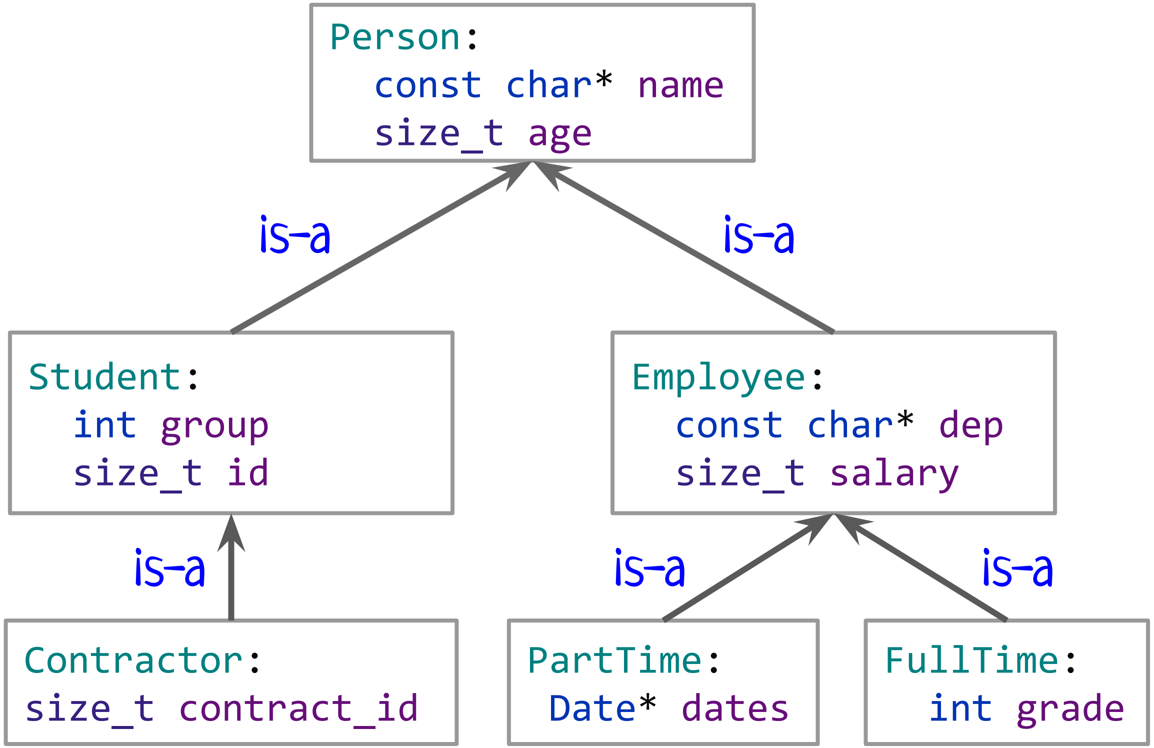
Inheritance

inheritance defines relationship: "is-a"



Inheritance

inheritance defines relationship: "is-a"



Talk is cheap.

Show me the code! (c)

Inheritance (structs)

```
struct Person {
    const char* name;
    size_t age;
};
```

Inheritance (structs)

```
struct Person {
    const char* name;
    size_t age;
};
```

```
struct Student: Person {
    size_t group;
    size_t id;
};
```

Inheritance (structs)

base class

```
struct Person {
    const char* name;
    size_t age;
};
```

```
struct Student: Person {
    size_t group;
    size_t id;
};
```

Inheritance (structs)

base class

```
struct Person {
    const char* name;
    size_t age;
};
```

```
struct Student: Person {
    size_t group;
    size_t id;
};
```

```
Person p;
p.name = "Bob"; p.age = 30;

Student s;
s.name = "Alice"; s.age = 18;
s.group = 22126;  s.id = 1234;
```

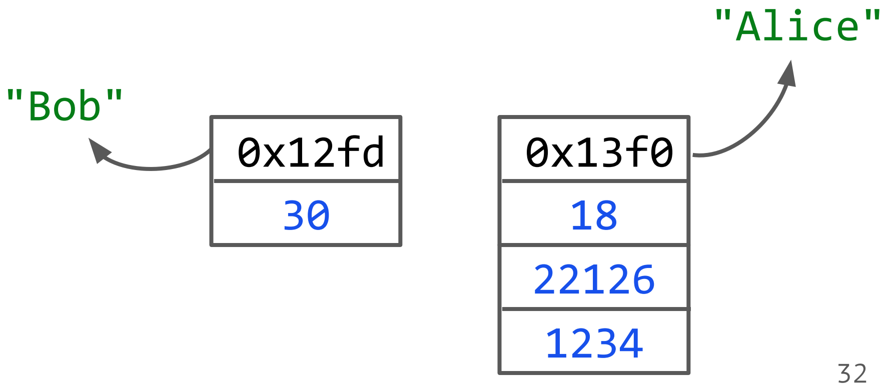
Inheritance (structs)

base class

```
struct Person {
    const char* name;
    size_t age;
};
```

```
struct Student: Person {
    size_t group;
    size_t id;
};
```

```
Person p;
p.name = "Bob"; p.age = 30;

Student s;
s.name = "Alice"; s.age = 18;
s.group = 22126;  s.id = 1234;
```



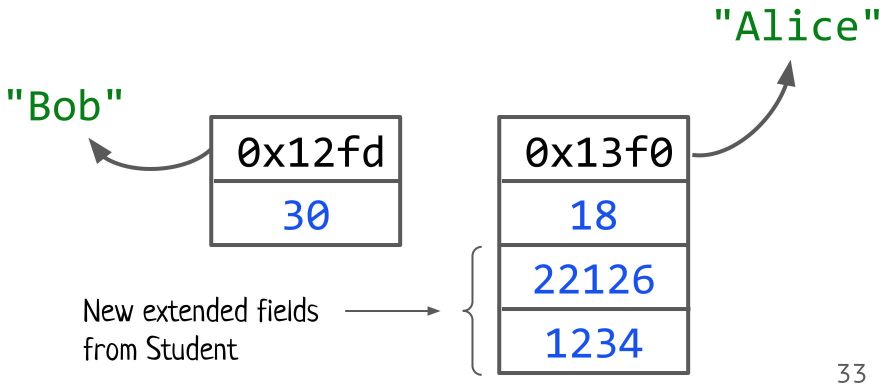
Inheritance (structs)

base class

```
struct Person {
    const char* name;
    size_t age;
};
```

```
struct Student: Person {
    size_t group;
    size_t id;
};
```

```
Person p;
p.name = "Bob"; p.age = 30;

Student s;
s.name = "Alice"; s.age = 18;
s.group = 22126;  s.id = 1234;
```



Inheritance (structs)

```
struct Person {
    const char* name;
    size_t age;

    Person(const char* name, int age): name(name), age(age) {}
};
```

```
Person p("Bob", 30);
```

Inheritance (structs)

```
struct Student: Person {
    size_t group;
    size_t id;

    Student(const char* name, size_t age, size_t group, size_t id):
                                Person(name, age), group(group), id(id) {}
};
```

Initialize base part with help of corresponding constructor.

Inheritance (structs)

```
struct Student: Person {
    size_t group;
    size_t id;

    Student(const char* name, size_t age, size_t group, size_t id):
                                Person(name, age), group(group), id(id) {}
};
```

Initialize base part with help of corresponding constructor.

Don't confuse with delegating constructor, this one works fine with the rest of member initialization list.

Inheritance (structs)

```
struct Student: Person {
    size_t group;
    size_t id;

    Student(const char* name, size_t age, size_t group, size_t id):
                             Person(name, age), group(group), id(id) {}
};
```

Initialize base part with help of corresponding constructor.

```
Student s("Alice", 18, 22126, 1234);
```

What about methods?

```cpp
struct Person {
    const char* name;
    size_t age;

    Person(const char* name, size_t age): name(name), age(age){}

    void print() const {
        std::cout << "Person " << name
                  << "; age = " << age << std::endl;
    }

};

Person p("Bob", 30);
p.print(); // Person Bob; age = 30
```

```
struct Person {
    const char* name;
    size_t age;

    Person(const char* name, size_t age): name(name), age(age){}

    void print() const {
        std::cout << "Person " << name
                  << "; age = " << age << std::endl;
    }

};

Person p("Bob", 30);
p.print(); // Person Bob; age = 30
Student s("Alice", 18, 22126, 1234);
s.print(); // Person Alice; age = 18
```

Student also inherits method print from its base.

```cpp
struct Person {
    const char* name;
    size_t age;

    Person(const char* name, size_t age):
                     name(name), age(age) {}

    void print() const {
        std::cout << "Person " << name
                  << "; age = " << age
                  << std::endl;
    }
};
```

```cpp
struct Student: Person {
    size_t group;
    size_t id;

    Student(const char* name, size_t age,
            size_t group, size_t id):
                Person(name, age),
                group(group), id(id) {}
};
```

```cpp
Person p("Bob", 30);
p.print(); // Person Bob; age = 30
Student s("Alice", 18, 22126, 1234);
s.print(); // Person Alice; age = 18
```

```cpp
struct Person {
    const char* name;
    size_t age;

    Person(const char* name, size_t age):
                     name(name), age(age) {}

    void print() const {
        std::cout << "Person " << name
                  << "; age = " << age
                  << std::endl;
    }
};
```

```cpp
struct Student: Person {
    size_t group;
    size_t id;

    Student(const char* name, size_t age,
            size_t group, size_t id):
                Person(name, age),
                group(group), id(id) {}

    void print() const {
        std::cout << "Student " << name
                  << " from group " << group
                  << std::endl;
    }
};
```

```cpp
Person p("Bob", 30);
p.print(); // Person Bob; age = 30
Student s("Alice", 18, 22126, 1234);
s.print(); // Student Alice from group 22126
```

```cpp
struct Person {
    const char* name;
    size_t age;

    Person(const char* name, size_t age):
                 name(name), age(age) {}

    void print() const {
        std::cout << "Person " << name
                  << "; age = " << age
                  << std::endl;
    }
};
```

```cpp
struct Student: Person {
    size_t group;
    size_t id;

    Student(const char* name, size_t age,
            size_t group, size_t id):
                Person(name, age),
                group(group), id(id) {}

    void print() const {
        std::cout << "Student " << name
                  << " from group " << group
                  << std::endl;
    }
};
```

Method print was overridden in Student with some custom implementation.

```cpp
Person p("Bob", 30);
p.print(); // Person Bob; age = 30
Student s("Alice", 18, 22126, 1234);
s.print(); // Student Alice from group 22126
```

```cpp
struct Person {
    const char* name;
    size_t age;

    Person(const char* name, size_t age):
                      name(name), age(age) {}

    void print() const {
        std::cout << "Person " << name
                  << "; age = " << age
                  << std::endl;
    }
};
```

```cpp
struct Student: Person {
    size_t group;
    size_t id;

    Student(const char* name, size_t age,
            size_t group, size_t id):
                Person(name, age),
                group(group), id(id) {}

    void print() const {
        std::cout << "Student " << name
                  << " from group " << group
                  << std::endl;
    }
};
```

```cpp
Person p("Bob", 30);
p.print(); // Person Bob; age = 30
Student s("Alice", 18, 22126, 1234);
s.print(); // Student Alice from group 22126
s.Person::print(); // Person Alice; age = 18
```

Method print was overridden in Student with some custom implementation.

What about encapsulation?

```cpp
struct Person {
    const char* name;
    size_t age;

    Person(const char* name, size_t age):
                     name(name), age(age) {}

    void print() const {
        std::cout << "Person " << name
                  << "; age = " << age
                  << std::endl;
    }
};
```

```cpp
struct Student: Person {
    size_t group;
    size_t id;

    Student(const char* name, size_t age,
            size_t group, size_t id):
                Person(name, age),
                group(group), id(id) {}

    void print() const {
        std::cout << "Student " << name
                  << " from group " << group
                  << std::endl;
    }
};
```

```cpp
struct Person {
private:
    const char* name;
    size_t age;
public:
    Person(const char* name, size_t age):
                    name(name), age(age) {}

    void print() const {
        std::cout << "Person " << name
                  << "; age = " << age
                  << std::endl;
    }
};
```

```cpp
struct Student: Person {
private:
    size_t group;
    size_t id;
public:
    Student(const char* name, size_t age,
            size_t group, size_t id):
                Person(name, age),
                group(group), id(id) {}

    void print() const {
       std::cout << "Student " << name
                 << " from group " << group
                 << std::endl;
    }
};
```

Is everything is still ok here?

```cpp
struct Person {
private:
    const char* name;
    size_t age;
public:
    Person(const char* name, size_t age):
                    name(name), age(age) {}

    void print() const {
        std::cout << "Person " << name
                  << "; age = " << age
                  << std::endl;
    }
};
```

```cpp
struct Student: Person {
private:
    size_t group;
    size_t id;
public:
    Student(const char* name, size_t age,
            size_t group, size_t id):
                Person(name, age),
                group(group), id(id) {}

    void print() const {
        std::cout << "Student " << name
                  << " from group " << group
                  << std::endl;
    }
};
```

Compilation error: private field name is inaccessible out of the struct

Inheritance (structs)

It was quite straightforward previously: code was split into "internal" and "external"

Inheritance (structs)

It was quite straightforward previously: code was split into "internal" and "external"

But the question is: are fields and methods of the base class are internal or external for the derived class?

Inheritance (structs)

It was quite straightforward previously: code was split into "internal" and "external"

But the question is: are fields and methods of the base class are internal or external for the derived class?

Solution: new access modifier protected. Such fields and methods are accessible* in code of the class itself and derived one.

```cpp
struct Person {
private:
    const char* name;
    size_t age;
public:
    Person(const char* name, size_t age):
                    name(name), age(age) {}

    void print() const {
        std::cout << "Person " << name
                  << "; age = " << age
                  << std::endl;
    }
};
```

```cpp
struct Student: Person {
private:
    size_t group;
    size_t id;
public:
    Student(const char* name, size_t age,
            size_t group, size_t id):
                Person(name, age),
                group(group), id(id) {}

    void print() const {
        std::cout << "Student " << name
                  << " from group " << group
                  << std::endl;
    }
};
```

Compilation error: private field name is inaccessible out of the class

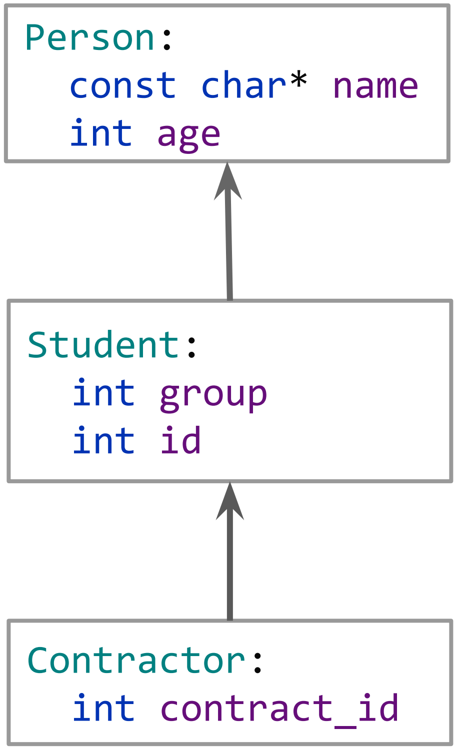
```cpp
struct Person {
protected:
    const char* name;
    size_t age;
public:
    Person(const char* name, size_t age):
                    name(name), age(age) {}

    void print() const {
        std::cout << "Person " << name
                  << "; age = " << age
                  << std::endl;
    }
};
```

```cpp
struct Student: Person {
private:
    size_t group;
    size_t id;
public:
    Student(const char* name, size_t age,
            size_t group, size_t id):
                Person(name, age),
                group(group), id(id) {}

    void print() const {
        std::cout << "Student " << name
                  << " from group " << group
                  << std::endl;
    }
};
```

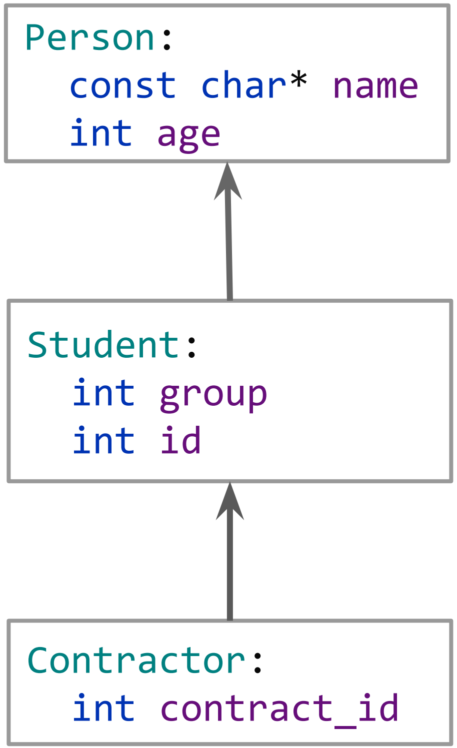
```
Person:
  const char* name
  int age
```

```
Student:
  int group
  int id
```

```
Contractor:
  int contract_id
```

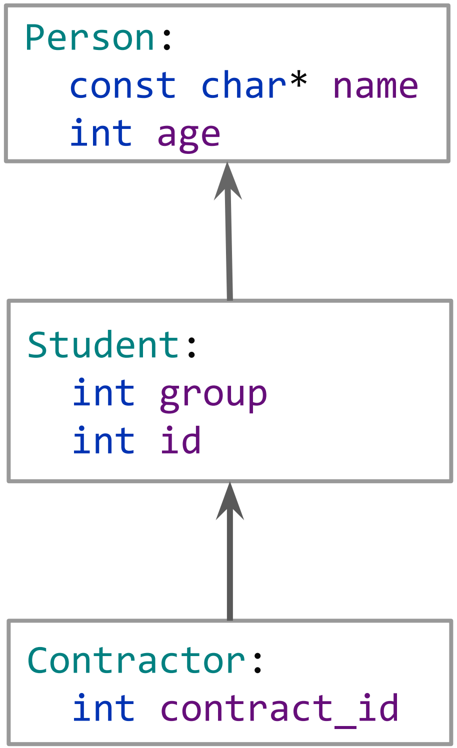


Should protected fields/methods of Person be accessible in Contractor?



Should protected fields/methods of Person be accessible in Contractor?

Should public fields/methods of Person be accessible through instances of Student or Contractor?



Should protected fields/methods of Person be accessible in Contractor?

Should public fields/methods of Person be accessible through instances of Student or Contractor?

In C++ you can control it!

```cpp
struct Foo {

private:
    int a;

protected:
    int b;

public:
    int c;
};
```

```
struct Foo {

private:
   int a;

protected:
   int b;

public:
   int c;
};
```

```
struct Bar:
    public Foo {

no access

accessed as
       protected

accessed as
          public
};
```

```
Bar bar;

std::cout << bar.c;   ✓
```

```
struct Foo {

private:
    int a;

protected:
    int b;

public:
    int c;
};
```

```
struct Bar:
    public Foo {

no access

accessed as
       protected

accessed as
          public
};
```

```
struct Foo {

private:
    int a;

protected:
    int b;

public:
    int c;
};
```

```
struct Bar:
     public Foo {

no access

accessed as
         protected

accessed as
             public
};
```

```
struct Baz:
     protected Foo {

no access

accessed as
         protected

accessed as
         protected
};
```

```
Baz baz;

std::cout << baz.c;
```

✗

```
struct Foo {

private:
    int a;

protected:
    int b;

public:
    int c;
};
```

```
struct Bar:
    public Foo {

no access

accessed as
        protected

accessed as
           public
};
```

```
struct Baz:
    protected Foo {

no access

accessed as
        protected

accessed as
        protected
};
```

```
struct Foo {

private:
    int a;

protected:
    int b;

public:
    int c;
};
```

```
struct Bar:
    public Foo {

no access

accessed as
        protected

accessed as
           public
};
```

```
struct Baz:
    protected Foo {

no access

accessed as
        protected

accessed as
        protected
};
```

```
struct Qux:
    private Foo {

no access

accessed as
          private

accessed as
          private
};
```

By default

```
struct Foo {

private:
    int a;

protected:
    int b;

public:
    int c;
};
```

```
struct Bar:
    public Foo {

no access

accessed as
        protected

accessed as
           public
};
```

```
struct Baz:
    protected Foo {

no access

accessed as
        protected

accessed as
        protected
};
```

```
struct Qux:
    private Foo {

no access

accessed as
          private

accessed as
          private
};
```

```
class Foo {

private:
    int a;

protected:
    int b;

public:
    int c;
};
```

```
class Bar:
     public Foo {

no access

accessed as
            protected

accessed as
               public
};
```

```
class Baz:
    protected Foo {

no access

accessed as
            protected

accessed as
            protected
};
```

```
class Qux:
    private Foo {

no access

accessed as
           private

accessed as
           private
};
```

By default

```
class Foo {

private:
    int a;

protected:
    int b;

public:
    int c;
};
```

```
class Bar:
    public Foo {

no access

accessed as
    protected

accessed as
    public
};
```

```
class Baz:
    protected Foo {

no access

accessed as
    protected

accessed as
    protected
};
```

```
class Qux:
    private Foo {

no access

accessed as
    private

accessed as
    private
};
```

```cpp
struct Person {
protected:
    const char* name;
    size_t age;
public:
    Person(const char* name, size_t age):
                    name(name), age(age) {}

    void print() const {
        std::cout << "Person " << name
                  << "; age = " << age
                  << std::endl;
    }
};
```

```cpp
struct Student: Person {
private:
    size_t group;
    size_t id;
public:
    Student(const char* name, size_t age,
            size_t group, size_t id):
                Person(name, age),
                group(group), id(id) {}

    void print() const {
        std::cout << "Student " << name ✓
                  << " from group " << group
                  << std::endl;
    }
};
```

```cpp
class Person {
protected:
    const char* name;
    size_t age;
public:
    Person(const char* name, size_t age):
                     name(name), age(age) {}

    void print() const {
        std::cout << "Person " << name
                  << "; age = " << age
                  << std::endl;
    }
};
```

```cpp
class Student: public Person {
private:
    size_t group;
    size_t id;
public:
    Student(const char* name, size_t age,
              size_t group, size_t id):
                  Person(name, age),
                  group(group), id(id) {}

    void print() const {
        std::cout << "Student " << name ✓
                  << " from group " << group
                  << std::endl;
    }
};
```

```cpp
class Person {
protected:
    const char* name;
    size_t age;
public:
    Person(const char* name, size_t age):
                    name(name), age(age) {}

    void print() const {
        std::cout << "Person " << name
                  << "; age = " << age
                  << std::endl;
    }
};
```

```cpp
class Student: public Person {
private:
    size_t group;
    size_t id;
public:
    Student(const char* name, size_t age,
            size_t group, size_t id):
                Person(name, age),
                group(group), id(id) {}

    void print() const {
        std::cout << "Student " << name ✓
                  << " from group " << group
                  << std::endl;
    }
};
```

Usually you need public inheritance in C++.

```cpp
class Person {
protected:
    const char* name;
    size_t age;
public:
    Person(const char* name, size_t age):
                        name(name), age(age) {}

    void print() const {
        std::cout << "Person " << name
                  << "; age = " << age
                  << std::endl;
    }
};
```

```cpp
class Student: public Person {
private:
    size_t group;
    size_t id;
public:
    Student(const char* name, size_t age,
            size_t group, size_t id):
                   Person(name, age),
                   group(group), id(id) {}

    void print() const {
        std::cout << "Student " << name
                  << " from group " << group
                  << std::endl;
    }
};
```

Usually you need public inheritance in C++.

Scenarios for non-public inheritance: when you want to remove something from public API that you've gotten from the base class.

```cpp
class Person {
protected:
    const char* name;
    size_t age;
public:
    Person(const char* name, size_t age):
                    name(name), age(age) {}

    void print() const {
        std::cout << "Person " << name
                  << "; age = " << age
                  << std::endl;
    }
};
```

```cpp
class Student: public Person {
private:
    size_t group;
    size_t id;
public:
    Student(const char* name, size_t age,
            size_t group, size_t id):
                Person(name, age),
                group(group), id(id) {}

    void print() const {
        std::cout << "Student " << name
                  << " from group " << group
                  << std::endl;
    }
};
```

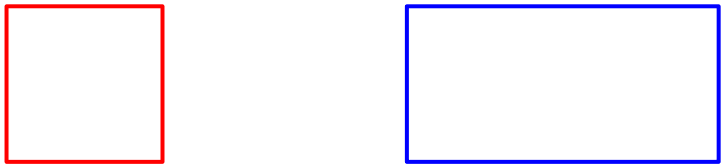
Usually you need public inheritance in C++.

Scenarios for non-public inheritance: when you want to remove something from public API that you’ve gotten from the base class. Details later.

About building hierarchies

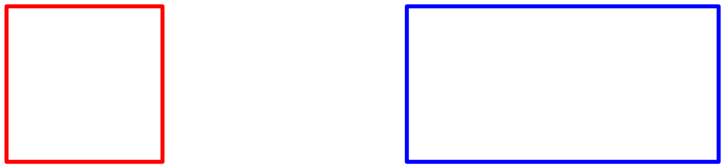
About building hierarchies

Task: define a hierarchy (?) of classes to work with both squares and rectangles.

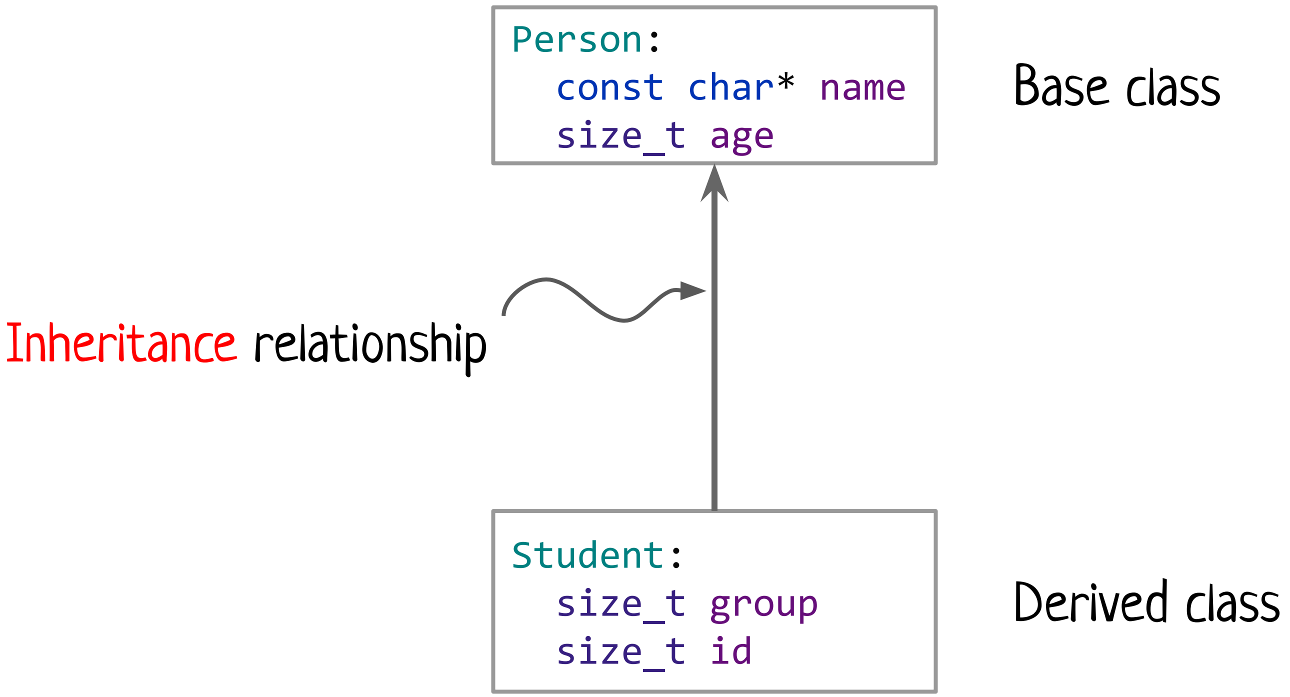


About building hierarchies

Task: define a hierarchy (?) of classes to work with both squares and rectangles.



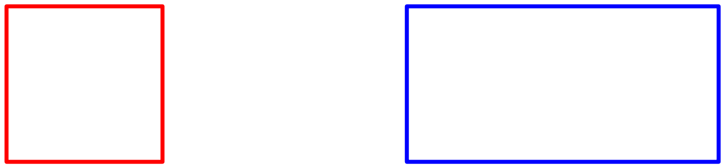
What should we do? Make Square a base class and Rectangle a derived? Or vice versa?



Derived class extends base: «Student — is a Person, who also has some group and id».
It specifies the particular case of base class, narrows set of objects.

About building hierarchies

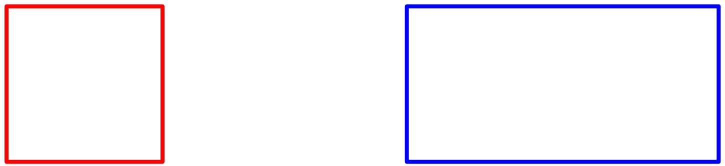
Task: define a hierarchy (?) of classes to work with both squares and rectangles.



Approach #1: Every square is a rectangle.

About building hierarchies

Task: define a hierarchy (?) of classes to work with both squares and rectangles.



Approach #1: Every square is a rectangle. So, square - is a particular case of rectangle. So, Rectangle should be base class, but square - a derived class.

```cpp
struct Rectangle {
protected:
   double w, h;
public:
   Rectangle(double w, double h): w(w), h(h) {}
};
```

```cpp
struct Square: Rectangle {
   Square(double length): Rectangle(length, length) {}
};
```

```cpp
struct Rectangle {
protected:
   double w, h;
public:
   Rectangle(double w, double h): w(w), h(h) {}
};
```

```cpp
// invariant of the class: w == h
struct Square: Rectangle {
   Square(double length): Rectangle(length, length) {}
};
```

```cpp
struct Rectangle {
protected:
   double w, h;
public:
   Rectangle(double w, double h): w(w), h(h) {}

   // w -> w*c
   // h is unchanged
   void stretchOnlyWidth(double c) {
       w *= c;
   }
};



// invariant of the class: w == h
struct Square: Rectangle {
   Square(double length): Rectangle(length, length) {}
};
```

```cpp
struct Rectangle {
protected:
   double w, h;
public:
   Rectangle(double w, double h): w(w), h(h) {}

   // w -> w*c
   // h is unchanged
   void stretchOnlyWidth(double c) {
       w *= c;
   }
};




// invariant of the class: w == h
struct Square: Rectangle {
   Square(double length): Rectangle(length, length) {}
};


Square s{10};
s.stretchOnlyWidth(); // ---> breaks invariants of Square!
```

```cpp
struct Rectangle {
protected:
   double w, h;
public:
   Rectangle(double w, double h): w(w), h(h) {}

   // w -> w*c
   // h is unchanged
   void stretchOnlyWidth(double c) {
       w *= c;
   }
};


// invariant of the class: w == h
struct Square: Rectangle {
   Square(double length): Rectangle(length, length) {}

   void stretchOnlyWidth(double c) {
      w *= c;
      h *= c;
   }
};
```

```cpp
Square s{10};
s.stretchOnlyWidth(); // ok, but...
```

```cpp
struct Rectangle {
protected:
   double w, h;
public:
   Rectangle(double w, double h): w(w), h(h) {}

   // w -> w*c
   // h is unchanged
   void stretchOnlyWidth(double c) {
       w *= c;
   }
};


// invariant of the class: w == h
struct Square: Rectangle {
   Square(double length): Rectangle(length, length) {}

   void stretchOnlyWidth(double c) {
      w *= c;
      h *= c;
   }
};
```

```cpp
Square s{10};
s.stretchOnlyWidth(); // ok, but...
```

ok, but obviously violates the contract of stretchOnlyWidth!

```cpp
struct Rectangle {
protected:
   double w, h;
public:
   Rectangle(double w, double h): w(w), h(h) {}

   // w -> w*c
   // h is unchanged
   void stretchOnlyWidth(double c) {
       w *= c;
   }
};


// invariant of the class: w == h
struct Square: Rectangle {
   Square(double length): Rectangle(length, length) {}

   void stretchOnlyWidth(double c) {
      w *= c;
      h *= c;
   }
};
```

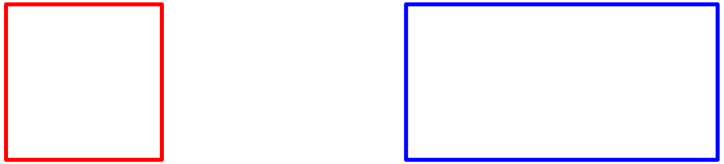
```cpp
Square s{10};
s.stretchOnlyWidth(); // ok, but...
```

ok, but obviously violates the contract of stretchOnlyWidth!

Another argument against such approach: we have two fields where we could have only one.

About building hierarchies

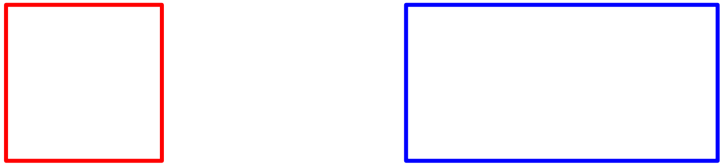
Task: define a hierarchy (?) of classes to work with both squares and rectangles.



~~Approach #1: Every square is a rectangle. So, square - is a particular case of rectangle. So, Rectangle should be base class, but square - a derived class.~~

About building hierarchies

Task: define a hierarchy (?) of classes to work with both squares and rectangles.



Approach #2: Rectangle is an extension of Square (it could have some additional fields). So, why not to make Square a base class?

```cpp
struct Square {
private:
   double length;
public:
   Square(double length): length(length) {}
};

struct Rectangle: Square {
   double height;
public:
   Rectangle(double width, double height): Square(width), height(height) {}
};
```

```cpp
struct Square {
private:
   double length;
public:
   Square(double length): length(length) {}
};

struct Rectangle: Square {
   double height;
public:
   Rectangle(double width, double height): Square(width), height(height) {}
};
```

Something is already so wrong here, it is obvious that not every Rectangle is a Square...

```
struct Square {
private:
   double length;
public:
   Square(double length): length(length) {}
};

struct Rectangle: Square {
   double height;
public:
   Rectangle(double width, double height): Square(width), height(height) {}
};
```

Something is already so wrong here, it is obvious that not every Rectangle is a Square... and that field length in Rectangle, so awkward

```cpp
struct Square {
private:
   double length;
public:
   Square(double length): length(length) {}
};

struct Rectangle: Square {
   double height;
public:
   Rectangle(double width, double height): Square(width), height(height) {}
};
```

Something is already so wrong here, it is obvious that not every Rectangle is a Square... and that field length in Rectangle, so awkward.

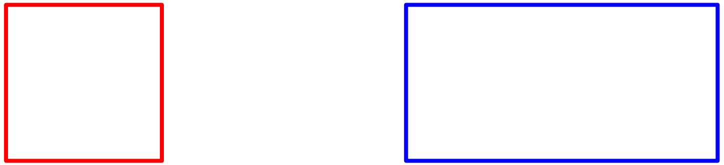
How to break it even more?

```cpp
struct Square {
private:
   double length;
public:
   Square(double length): length(length) {}

   double getInscribedCircleSquare() {
      return M_PI * (length / 2) * (length / 2);
   }
};

struct Rectangle: Square {
   double height;
public:
   Rectangle(double width, double height): Square(width), height(height) {}
};
```

```
struct Square {
private:
   double length;
public:
   Square(double length): length(length) {}

   double getInscribedCircleSquare() {
      return M_PI * (length / 2) * (length / 2);
   }
};

struct Rectangle: Square {
   double height;
public:
   Rectangle(double width, double height): Square(width), height(height) {}
};

Rectangle rect(42);
double d = rect.getInscribedCircleSquare();

// what it can even mean?
```

About building hierarchies

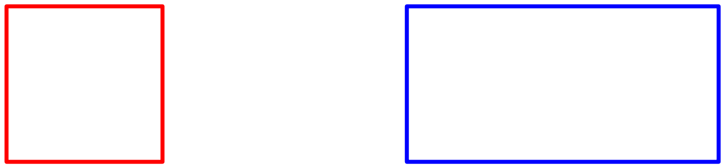
Task: define a hierarchy (?) of classes to work with both squares and rectangles.



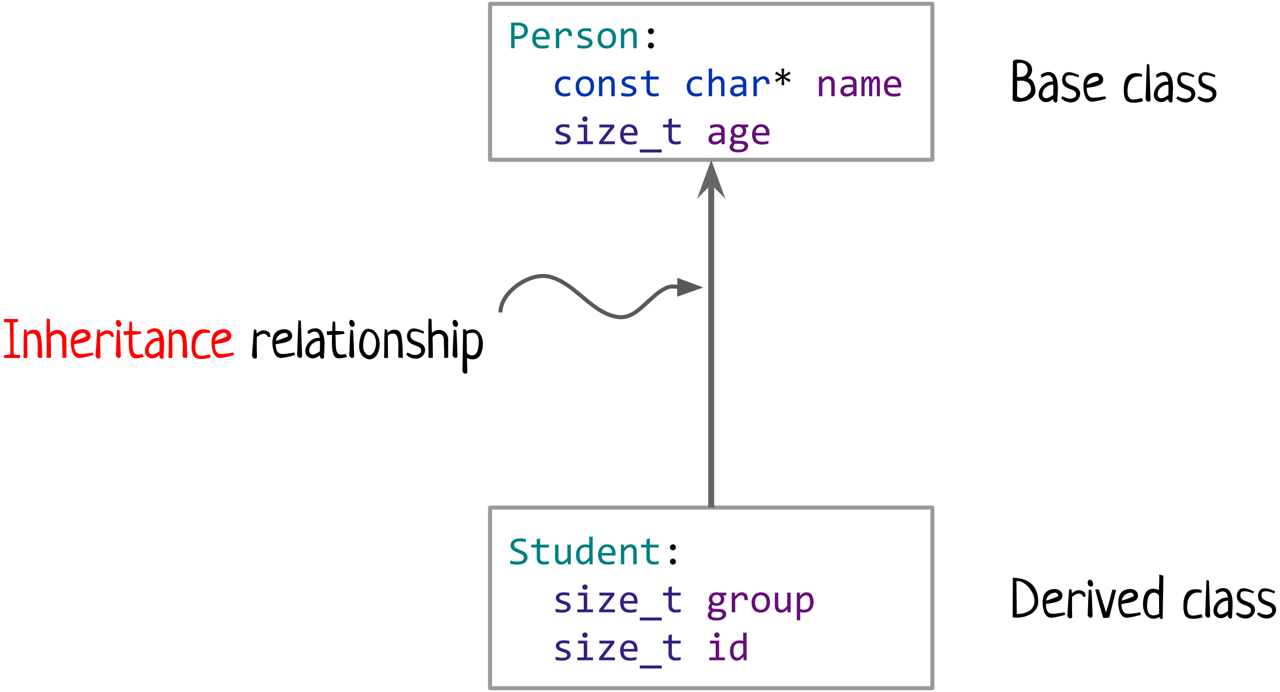
~~Approach #2: Rectangle is an extension of Square (it could have some additional fields). So, why not to make Square a base class?~~

About building hierarchies

Task: define a hierarchy (?) of classes to work with both squares and rectangles.



So, both approaches failed us. Why?



Derived class extends base: «Student — is a Person, who also has some group and id».
It specifies the particular case of base class, narrows set of objects.

```
Person:
  const char* name
  size_t age
```

Base class

Inheritance relationship

```
Student:
  size_t group
  size_t id
```

Derived class

Both should be true!

Derived class extends base: «Student — is a Person, who also has some group and id».
It specifies the particular case of base class, narrows set of objects.

About building hierarchies

`So, both approaches failed us. Why?`

About building hierarchies

So, both approaches failed us. Why?

Because when you build hierarchies with "is-a" relationship, you should check that:

About building hierarchies

So, both approaches failed us. Why?

Because when you build hierarchies with "is-a" relationship, you should check that:

If some predicate is true about all Base class instances, it should also be true for all Derived class instances.

About building hierarchies

If some predicate is true about all Base class instances, it should also be true for all Derived class instances.

This is called Liskov substitution principle (LSP).

```cpp
struct Rectangle {
protected:
   double w, h;
public:
   Rectangle(double w, double h): w(w), h(h) {}

   // w -> w*c
   // h is unchanged
   void stretchOnlyWidth(double c) {
       w *= c;
   }
};


// invariant of the class: w == h
struct Square: Rectangle {
   Square(double length): Rectangle(length, length) {}

   void stretchOnlyWidth(double c) {
      w *= c;
      h *= c;
   }
};
```

Predicate: "after calling stretchOnlyWidth(c), w is multiplied by c and h is unchanged" is true for any instance of Rectangle.

```cpp
struct Rectangle {
protected:
   double w, h;
public:
   Rectangle(double w, double h): w(w), h(h) {}

   // w -> w*c
   // h is unchanged
   void stretchOnlyWidth(double c) {
       w *= c;
   }
};
```

Predicate: "after calling stretchOnlyWidth(c), w is multiplied by c and h is unchanged" is true for any instance of Rectangle.

```cpp
// invariant of the class: w == h
struct Square: Rectangle {
   Square(double length): Rectangle(length, length) {}

   void stretchOnlyWidth(double c) {
      w *= c;
      h *= c;
   }
};
```

But it is not true for instances of Square.

```cpp
struct Rectangle {
protected:
   double w, h;
public:
   Rectangle(double w, double h): w(w), h(h) {}

   // w -> w*c
   // h is unchanged
   void stretchOnlyWidth(double c) {
       w *= c;
   }
};
```

Predicate: "after calling stretchOnlyWidth(c), w is multiplied by c and h is unchanged" is true for any instance of Rectangle.

```cpp
// invariant of the class: w == h
struct Square: Rectangle {
   Square(double length): Rectangle(length, length) {}

   void stretchOnlyWidth(double c) {
      w *= c;
      h *= c;
   }
};
```

But it is not true for instances of Square.

So, LSP is violated, "is-a" relationship is wrong here.

```
struct Square {
private:
   double length;
public:
   Square(double length): length(length) {}

   double getInscribedCircleSquare() {
      return M_PI * (length / 2) * (length / 2);
   }
};

struct Rectangle: Square {
   double height;
public:
   Rectangle(double width, double height): Square(width), height(height) {}
};
```

```cpp
struct Square {
private:
   double length;
public:
   Square(double length): length(length) {}

   double getInscribedCircleSquare() {
      return M_PI * (length / 2) * (length / 2);
   }
};

struct Rectangle: Square {
   double height;
public:
   Rectangle(double width, double height): Square(width), height(height) {}
};
```

Predicate: "calling getInscribedCircleSquare, will give you a square of inscribed circle" is true for any instance of Square.

```
struct Square {
private:
  double length;
public:
  Square(double length): length(length) {}

  double getInscribedCircleSquare() {
    return M_PI * (length / 2) * (length / 2);
  }
};

struct Rectangle: Square {
  double height;
public:
  Rectangle(double width, double height): Square(width), height(height) {}
};
```

Predicate: "calling getInscribedCircleSquare, will give you a square of inscribed circle" is true for any instance of Square.

And of course that's not true for instances of Rectangle.
So, LSP is violated again.

About building hierarchies

If some predicate is true about all Base class instances, it should also be true for all Derived class instances.

This is called Liskov substitution principle (LSP).

About building hierarchies

If some predicate is true about all Base class instances, it should also be true for all Derived class instances.

This is called Liskov substitution principle (LSP).

Practical consequence: you can write your code in terms of basic classes and be sure that it will work with derived classes as well.

About building hierarchies

If some predicate is true about all Base class instances, it should also be true for all Derived class instances.

This is called Liskov substitution principle (LSP).

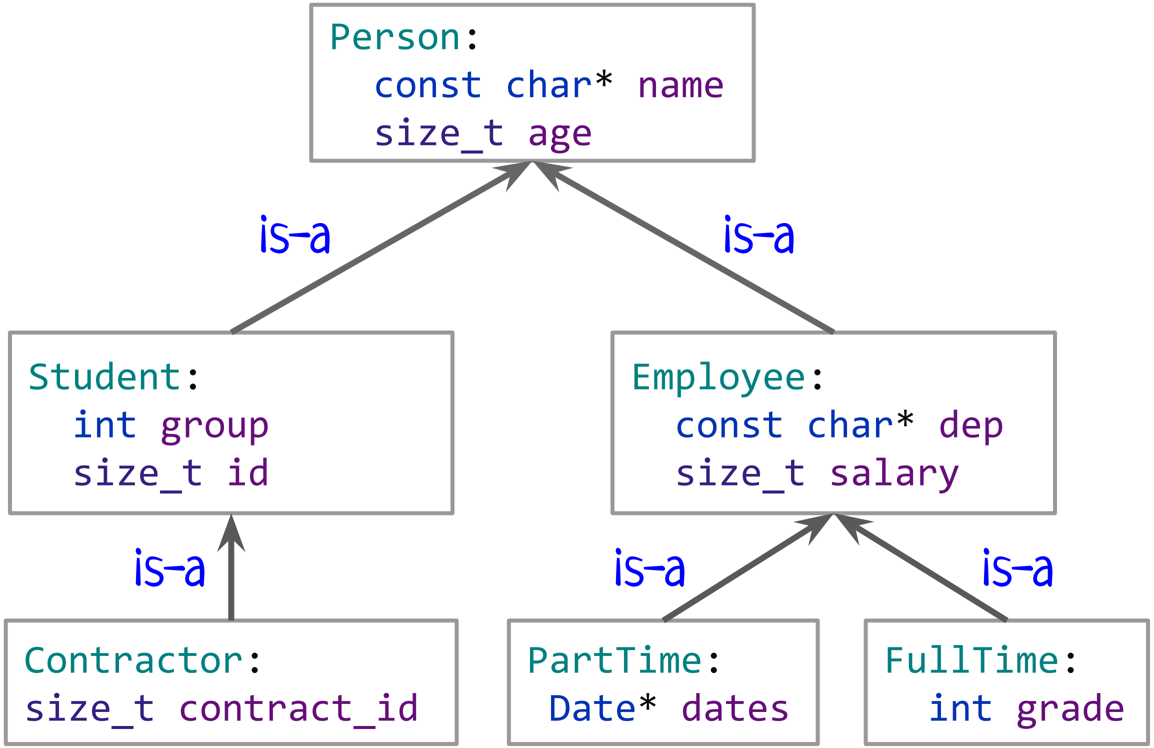
Practical consequence: you can write your code in terms of basic classes and instances of derived classes can be substituted there.

Talk is cheap.

Show me the code! (c)

Subtyping polymorphism

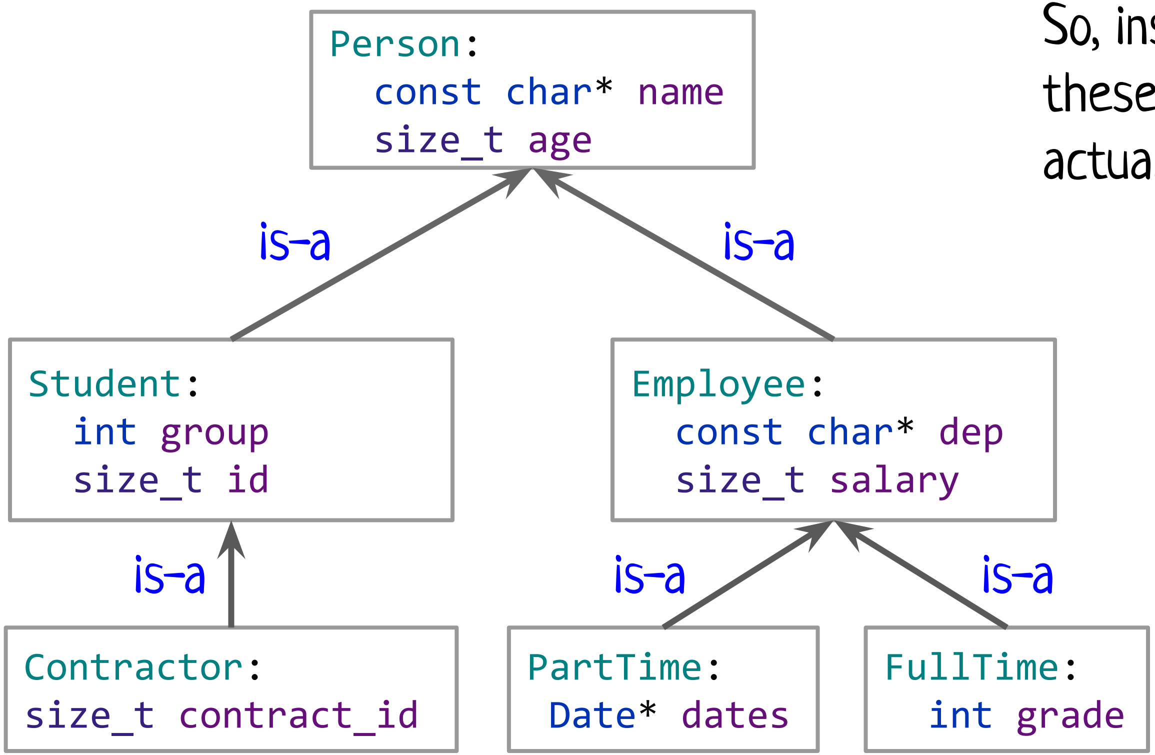
inheritance defines relationship: "is-a"



Subtyping polymorphism

inheritance defines relationship: "is-a"

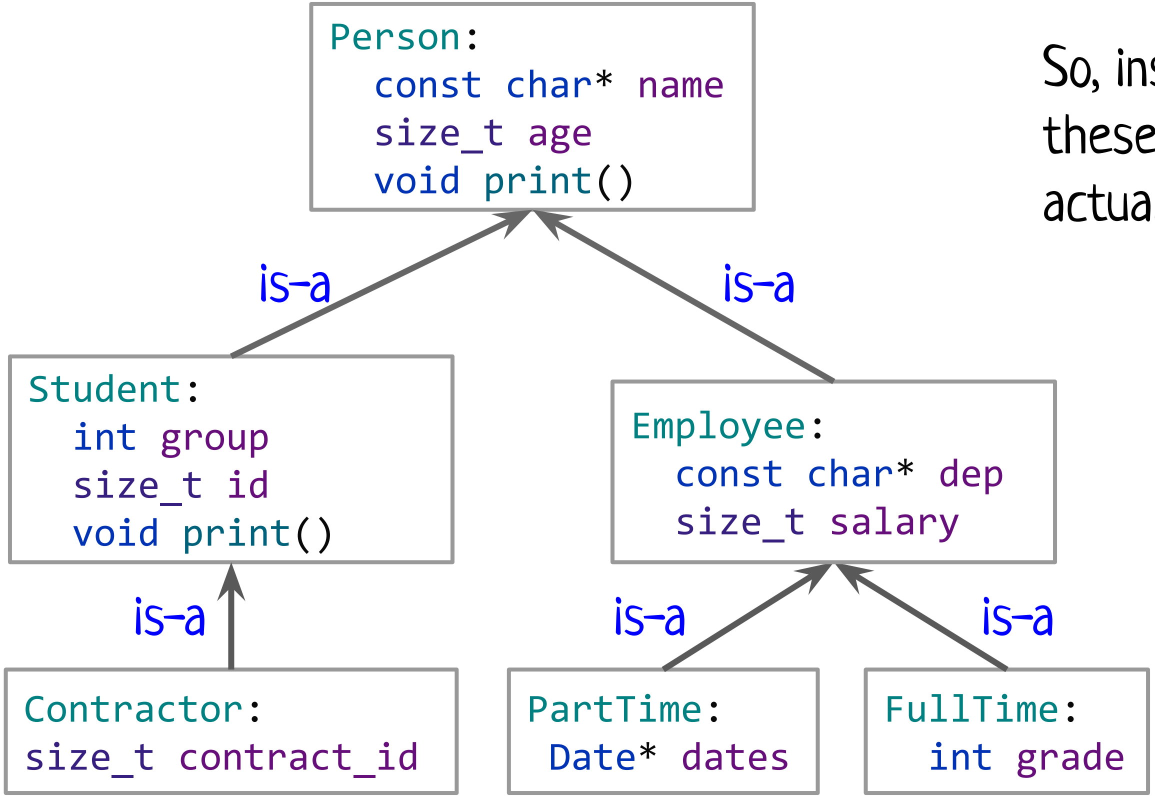
So, instances of all these classes are actually Persons.



Subtyping polymorphism

inheritance defines relationship: "is-a"

So, instances of all these classes are actually Persons.



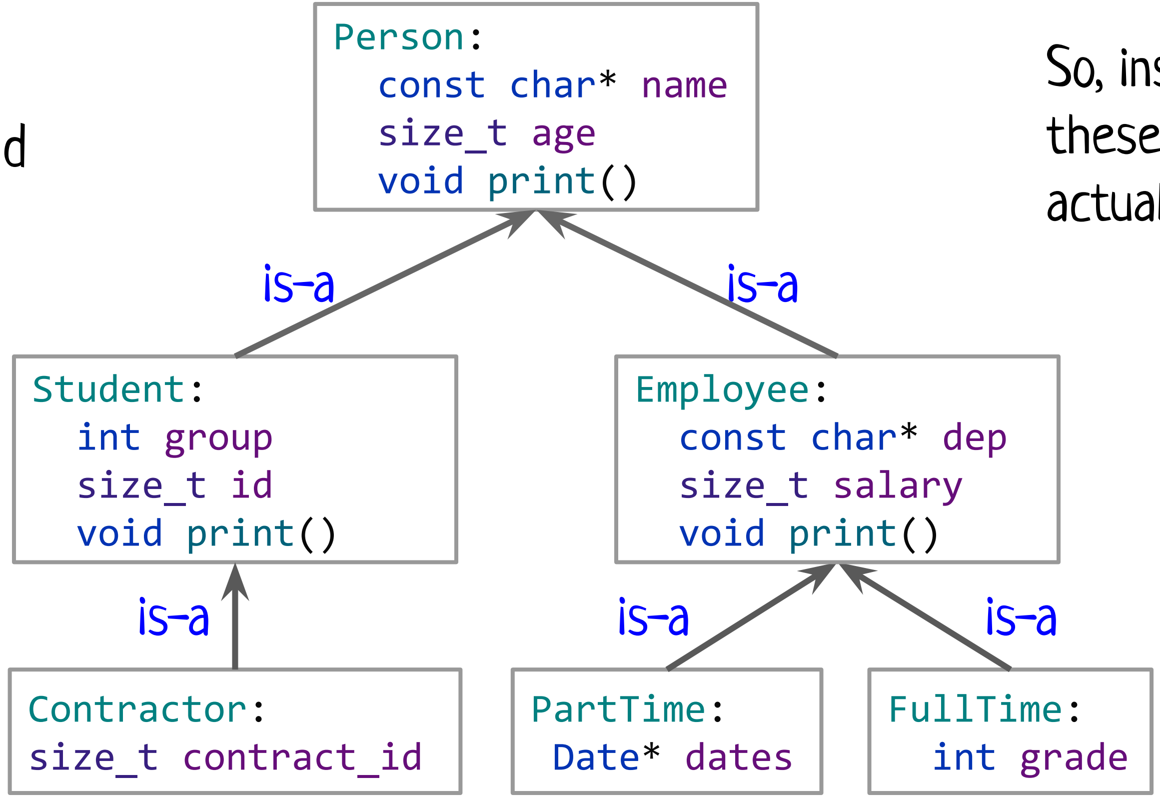
Subtyping polymorphism

```
class Employee: public Person {
private:
   const char* dep;
   size_t salary;
public:
   Employee(const char* name, size_t age,
            const char* dep, size_t salary):
            Person(name, age), dep(dep), salary(salary) {}

   void print() const {
       std::cout << "Employee " << name
                 << " from dep " << dep
                 << std::endl;
   }
};
```

Subtyping polymorphism

inheritance defines relationship: "is-a"

And all of them have print method (at least one)



So, instances of all these classes are actually Persons.

Subtyping polymorphism

Task: handle the whole hierarchy of these classes with a single generic algorithm

Subtyping polymorphism

Task: handle the whole hierarchy of these classes with a single generic algorithm

For example: iterate over collection of objects and just print them, using print method.

Subtyping polymorphism

LSP consequence: you can write your code in terms of basic classes and instances of derived classes can be substituted there.

Subtyping polymorphism

LSP consequence: you can write your code in terms of basic classes and instances of derived classes can be substituted there.

Implementation of this principle in C++:

A pointer (a reference) to the derived class can be substituted instead of a pointer (a reference) to the base class.

Subtyping polymorphism

LSP consequence: you can write your code in terms of basic classes and instances of derived classes can be substituted there.

Implementation of this principle in C++:

A pointer (a reference) to the derived class can be substituted instead of a pointer (a reference) to the base class.

The opposite it not true!

Subtyping polymorphism

Semantics guarantees!

LSP consequence: you can write your code in terms of basic classes and instances of derived classes can be substituted there.

Implementation of this principle in C++:

A pointer (a reference) to the derived class can be substituted instead of a pointer (a reference) to the base class.

The opposite it not true!

Subtyping polymorphism

Semantics guarantees!

LSP consequence: you can write your code in terms of basic classes and instances of derived classes can be substituted there.

Implementation of this principle in C++:

A pointer (a reference) to the derived class can be substituted instead of a pointer (a reference) to the base class.

Syntax guarantees by C++!

The opposite it not true!

Subtyping polymorphism

```
Person* p = new Person("Bob", 30);

Person* s = new Student("Alice", 18, 22126, 1234);
```

Subtyping polymorphism

```
Person* p = new Person("Bob", 30);
✓ Person* s = new Student("Alice", 18, 22126, 1234);
```

Quite logical, because every student is still a person.

Subtyping polymorphism

```
  Person* p = new Person("Bob", 30);

✓ Person* s = new Student("Alice", 18, 22126, 1234);

✓ Person* e = new Employee("John", 25, "MMF", 5000);

✗ Student* k = new Person("Tom", 42);
```

But not every person is a student.

Subtyping polymorphism

```
void print_info(Person& k) {
    k.print();
}
```

Subtyping polymorphism

```cpp
void print_info(Person& k) {
    k.print();
}
```

k can be a reference to some Person instance, but can be also a reference to instance of any Derived class

```cpp
Person p("Bob", 30);
Student s("Alice", 18, 22126, 1);
Employee e("John", 25, "MMF", 5000);
```

✓ `print_info(p);`
✓ `print_info(s);`
✓ `print_info(e);`

Subtyping polymorphism

```
void print_info(Person& k) {
    k.print();
}

Person p("Bob", 30);
Student s("Alice", 18, 22126, 1);
Employee e("John", 25, "MMF", 5000);

✓ print_info(p);
✓ print_info(s);
✓ print_info(e);
```

k can be a reference to some Person instance, but can be also a reference to instance of any Derived class

LSP guarantees us that no semantic invariants will be ruined here.

Subtyping polymorphism

```
void print_info(Person& k) {
    k.print();
}

Person p("Bob", 30);
Student s("Alice", 18, 22126, 1);
Employee e("John", 25, "MMF", 5000);

✓ print_info(p);
✓ print_info(s);
✓ print_info(e);
```

k can be a reference to some Person instance, but can be also a reference to instance of any Derived class

LSP guarantees us that no semantic invariants will be ruined here.

But what will be printed?

Subtyping polymorphism

```cpp
class Person {
protected:
    const char* name;
    size_t age;
public:
    Person(const char* name, size_t age):
                        name(name), age(age) {}

    void print() const {
        std::cout << "Person " << name
                  << "; age = " << age
                  << std::endl;
    }
};
```

```cpp
class Student: public Person {
    ...
public:
    ...

    void print() const {
        std::cout << "Student " << name
                  << " from group " << group
                  << std::endl;
    }
};
```

```cpp
class Employee: public Person {
    ...
public:
    ...

    void print() const {
        std::cout << "Employee " << name
                  << " from dep " << dep
                  << std::endl;
    }
};
```

Subtyping polymorphism

```
void print_info(Person& k) {
    k.print();
}
```

k can be a reference to some Person instance, but can be also a reference to instance of any Derived class

```
Person p("Bob", 30);
Student s("Alice", 18, 22126, 1);
Employee e("John", 25, "MMF", 5000);
```

But what will be printed?

```
✓ print_info(p);
✓ print_info(s);
✓ print_info(e);
```

Subtyping polymorphism

```cpp
void print_info(Person& k) {
    k.print();
}

Person p("Bob", 30);
Student s("Alice", 18, 22126, 1);
Employee e("John", 25, "MMF", 5000);

print_info(p); // Person Bob; age = 30
print_info(s); // Person Alice; age = 18
print_info(e); // Person John; age = 25
```

k can be a reference to some Person instance, but can be also a reference to instance of any Derived class

But what will be printed?

Subtyping polymorphism

```
void print_info(Person& k) {
    k.print();
}
```

k can be a reference to some Person instance, but can be also a reference to instance of any Derived class

```
Person p("Bob", 30);
Student s("Alice", 18, 22126, 1);
Employee e("John", 25, "MMF", 5000);
```

But what will be printed?

```
✓ print_info(p); // Person Bob; age = 30
✓ print_info(s); // Person Alice; age = 18
✓ print_info(e); // Person John; age = 25
```

is it what you expected?

Subtyping polymorphism

```cpp
void print_info(Person& k) {
    k.print();
}
```

k can be a reference to some Person instance, but can be also a reference to instance of any Derived class

```cpp
Person p("Bob", 30);
Student s("Alice", 18, 22126, 1);
Employee e("John", 25, "MMF", 5000);
```

```cpp
print_info(p); // Person Bob; age = 30
print_info(s); // Person Alice; age = 18
print_info(e); // Person John; age = 25
```

By default we will call the method print from type that is actually (statically) specified in the code.

Subtyping polymorphism

```cpp
class Person {
protected:
    const char* name;
    size_t age;
public:
    Person(const char* name, size_t age):
                          name(name), age(age) {}

    void print() const {
        std::cout << "Person " << name
                  << "; age = " << age
                  << std::endl;
    }
};
```

```cpp
class Student: public Person {
    ...
public:
    ...

    void print() const {
        std::cout << "Student " << name
                  << " from group " << group
                  << std::endl;
    }
};
```

```cpp
class Employee: public Person {
    ...
public:
    ...

    void print() const {
        std::cout << "Employee " << name
                  << " from dep " << dep
                  << std::endl;
    }
};
```

Subtyping polymorphism

```
class Person {
protected:
    const char* name;
    size_t age;
public:
    Person(const char* name, size_t age):
                        name(name), age(age) {}

    virtual void print() const {
        std::cout << "Person " << name
                  << "; age = " << age
                  << std::endl;
    }
};
```

Virtual modifier changes this behavior.

```
class Student: public Person {
    ...
public:
    ...

    void print() const {
        std::cout << "Student " << name
                  << " from group " << group
                  << std::endl;
    }
};
```

```
class Employee: public Person {
    ...
public:
    ...

    void print() const {
        std::cout << "Employee " << name
                  << " from dep " << dep
                  << std::endl;
    }
};
```

Subtyping polymorphism

```
class Person {
protected:
    const char* name;
    size_t age;
public:
    Person(const char* name, size_t age):
                     name(name), age(age) {}

    virtual void print() const {
        std::cout << "Person " << name
                  << "; age = " << age
                  << std::endl;
    }
};
```

Virtual modifier changes this behavior: the closest method to the real type of the instance will be called.

```
class Student: public Person {
    ...
public:
    ...

    void print() const {
      std::cout << "Student " << name
                << " from group " << group
                << std::endl;
    }
};
```

```
class Employee: public Person {
    ...
public:
    ...

    void print() const {
      std::cout << "Employee " << name
                << " from dep " << dep
                << std::endl;
    }
};
```

Subtyping polymorphism

In C++ values can have static and dynamic type.

```
void print_info(Person& k) {
    k.print();
}


Person p("Bob", 30);
Student s("Alice", 18, 22126, 1);
Employee e("John", 25, "MMF", 5000);

✓ print_info(p); // Person Bob; age = 30
✓ print_info(s); // Person Alice; age = 18
✓ print_info(e); // Person John; age = 25
```

By default we will call the method print from type that is actually (statically) specified in the code.

Subtyping polymorphism

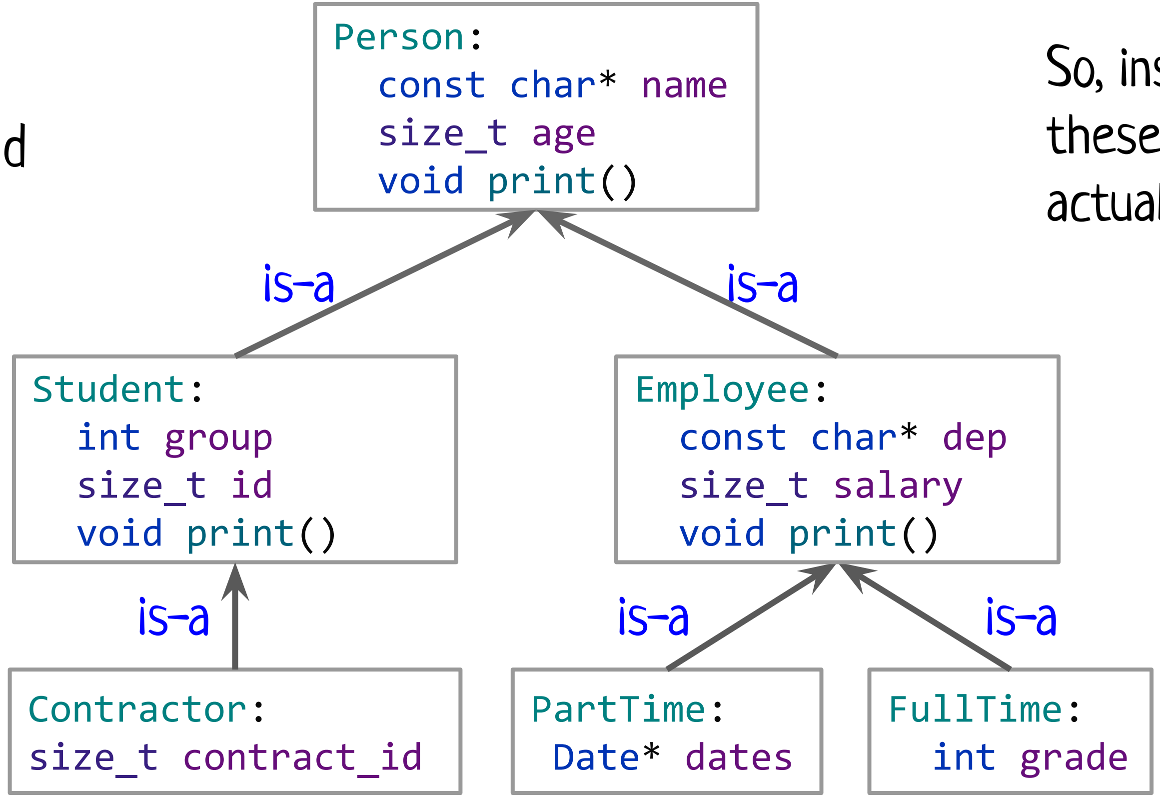
In C++ values can have static and dynamic type.

```
void print_info(Person& k) {
    k.print();
}


Person p("Bob", 30);
Student s("Alice", 18, 22126, 1);
Employee e("John", 25, "MMF", 5000);

✓ print_info(p); // Person Bob; age = 30
✓ print_info(s); // Student Alice from group 22126
✓ print_info(e); // Employee John from dep MMF
```

But if print is virtual method: the closest print to real derived class (that was passed here) will be called.

Subtyping polymorphism

And all of them have print method (at least one)



inheritance defines relationship: "is-a"

So, instances of all these classes are actually Persons.

Subtyping polymorphism

In C++ values can have static and dynamic type.

```
void print_info(Person& k) {
    k.print();
}
```

```
Person p("Bob", 30);
Student s("Alice", 18, 22126, 1);
PartTime pe("John", 25, "MMF", 5000, {Date{10, 12,}});
```

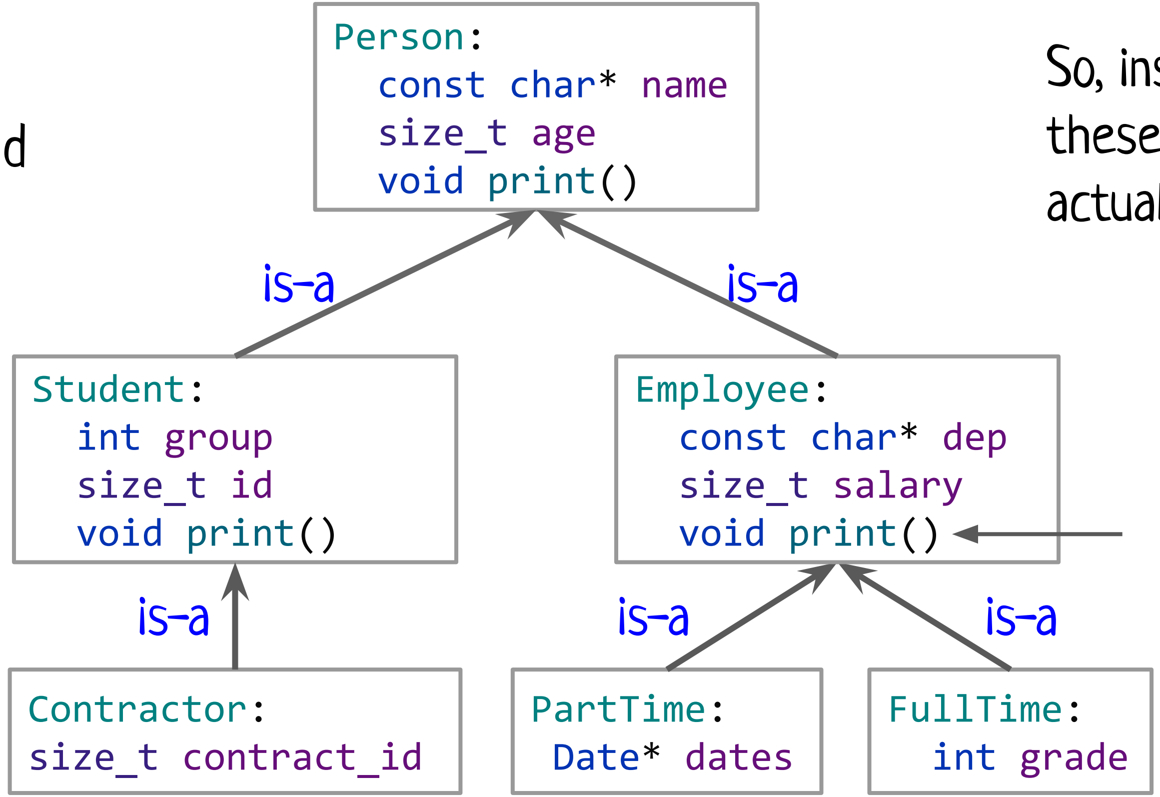
```
✓ print_info(p); // Person Bob; age = 30
✓ print_info(s); // Student Alice from group 22126
✓ print_info(pe); // Employee John from dep MMF
```

But if print is virtual method: the closest print to real derived class (that was passed here) will be called.

Subtyping polymorphism

inheritance defines relationship: "is-a"

And all of them have print method (at least one)



closest print to PartTime

So, instances of all these classes are actually Persons.

Subtyping polymorphism

- Based on Liskov substitution principle

Subtyping polymorphism

- Based on Liskov substitution principle
- Allows you to write generic algorithms that will work with any class from hierarchy

Subtyping polymorphism

- Based on Liskov substitution principle
- Allows you to write generic algorithms that will work with any class from hierarchy
- I can know nothing about the implementation of derived classes
- Derived classes can know nothing about how they will be used

Subtyping polymorphism

- Based on Liskov substitution principle
- Allows you to write generic algorithms that will work with any class from hierarchy
- I can know nothing about the implementation of derived classes
- Derived classes can know nothing about how they will be used
- But generic code will still work!

Subtyping polymorphism

```
void print_info(Person& k) {
    k.print();
}
```

k can be a reference to some Person instance, but can be also a reference to instance of any Derived class

```
Person p("Bob", 30);
Student s("Alice", 18, 22126, 1);
```

But what will be printed?

```
✓ print_info(p);
✓ print_info(s);
```

Subtyping polymorphism

```
void print_info(Person k) {
    k.print();
}
```

```
Person p("Bob", 30);
Student s("Alice", 18, 22126, 1);
```

But what will be printed?

```
✓ print_info(p);
✓ print_info(s);
```

Subtyping polymorphism

It will indeed compile

```
void print_info(Person k) {
    k.print();
}
```

```
Person p("Bob", 30);
Student s("Alice", 18, 22126, 1);
```

But what will be printed?

```
✓ print_info(p);
✓ print_info(s);
```

Subtyping polymorphism

It will indeed compile

```
void print_info(Person k) {
    k.print();
}
```

```
Person p("Bob", 30);
Student s("Alice", 18, 22126, 1);
```

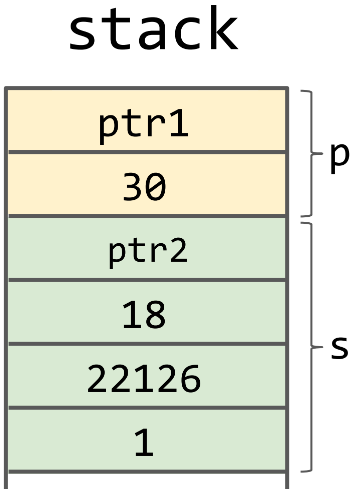
But what will be printed?

```
✓ print_info(p); // Person Bob; age = 30
✓ print_info(s); // Person Alice; age = 18
```

Why? What happened?

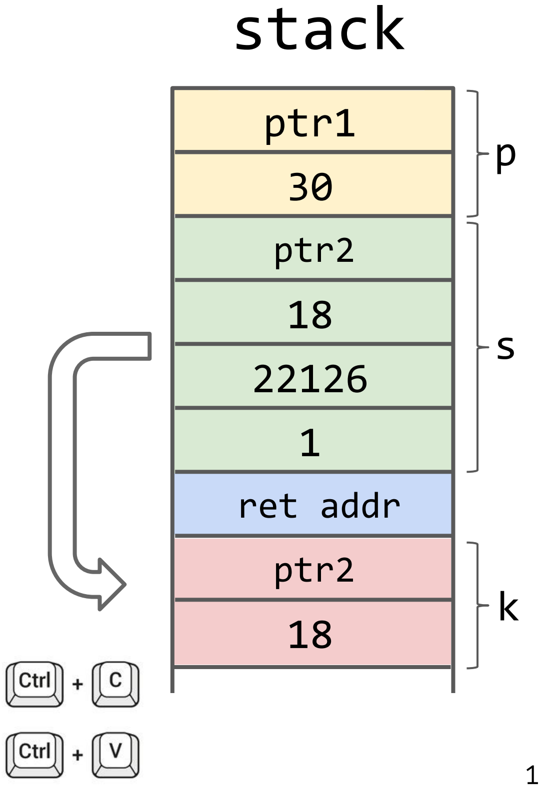
Subtyping polymorphism

```
void print_info(Person k) {
    k.print();
}

Person p("Bob", 30);
Student s("Alice", 18, 22126, 1);

print_info(p); // Person Bob; age = 30
print_info(s); // Person Alice; age = 18
```



Subtyping polymorphism

```
void print_info(Person k) {
    k.print();
}

Person p("Bob", 30);
Student s("Alice", 18, 22126, 1);

print_info(p); // Person Bob; age = 30
print_info(s); // Person Alice; age = 18
```



*layout is actually a bit different, we will discuss it soon

Subtyping polymorphism

Another example of how subtyping polymorphism doesn’t work with values (only pointers and refs):

Subtyping polymorphism

Another example of how subtyping polymorphism doesn’t work with values (only pointers and refs).

Imagine you want to have an array of different instances from the hierarchy (Persons, Students, Employees, etc). What type will it have?

Subtyping polymorphism

Another example of how subtyping polymorphism doesn't work with values (only pointers and refs).

Imagine you want to have an array of different instances from the hierarchy (Persons, Students, Employees, etc). What type will it have?

```
Person p[10];
```

Subtyping polymorphism

Another example of how subtyping polymorphism doesn't work with values (only pointers and refs).

Imagine you want to have an array of different instances from the hierarchy (Persons, Students, Employees, etc). What type will it have?

```
Person p[10]; // ???
```

In such case it contains only persons, no students (if you will try to assign a student it will be just copied to person).

Subtyping polymorphism

Another example of how subtyping polymorphism doesn’t work with values (only pointers and refs).

Imagine you want to have an array of different instances from the hierarchy (Persons, Students, Employees, etc). What type will it have?

```
Person* p[10];
```

Subtyping polymorphism

Another example of how subtyping polymorphism doesn’t work with values (only pointers and refs).

Imagine you want to have an array of different instances from the hierarchy (Persons, Students, Employees, etc). What type will it have?

```
Person* p[10];
p[0] = new Student(...);
p[1] = new Employee(...);
```

Subtyping polymorphism

Another example of how subtyping polymorphism doesn't work with values (only pointers and refs).

Imagine you want to have an array of different instances from the hierarchy (Persons, Students, Employees, etc). What type will it have?

```
Person* p[10];
p[0] = new Student(...);
p[1] = new Employee(...);
```

This is possible, pointer to derived is used instead of pointer to the base.

Subtyping polymorphism

Another example of how subtyping polymorphism doesn't work with values (only pointers and refs).

Imagine you want to have an array of different instances from the hierarchy (Persons, Students, Employees, etc). What type will it have?

```
Person* p[10];

for (size_t i = 0; i < 10; i++) {
    p->print();
}
```

Virtual methods mechanism will work as well!

```cpp
class Person {
protected:
    const char* name;
    size_t age;
public:
    Person(const char* name, size_t age):
                    name(name), age(age) {}

    void print() const {
        std::cout << "Person " << name
                  << "; age = " << age
                  << std::endl;
    }
};
```

```cpp
class Student: public Person {
private:
    size_t group;
    size_t id;
public:
    Student(const char* name, size_t age,
            size_t group, size_t id):
                Person(name, age),
                group(group), id(id) {}

    void print() const {
        std::cout << "Student " << name
                  << " from group " << group
                  << std::endl;
    }
};
```

Usually you need public inheritance in C++.

Scenarios for non-public inheritance: when you want to remove something from public API that you've gotten from the base class. Details later.

About building hierarchies

LSP: if some predicate is true about all Base class instances, it should also be true for all Derived class instances.

Practical consequence: you can write your code in terms of basic classes and instances of derived classes can be substituted there.

About building hierarchies

LSP: if some predicate is true about all Base class instances, it should also be true for all Derived class instances.

Practical consequence: you can write your code in terms of basic classes and instances of derived classes can be substituted there.

How is LSP connected with different types of inheritance in C++?

```
class LinkedList {
   Node* head;
public:

   Node* getHead() {
       return head;
   }

   bool isEmpty() {
       return head == nullptr;
   }

   void addToHead(int value) {
      ...
   }
};
```

```cpp
class LinkedList {
   Node* head;
public:

   Node* getHead() {
       return head;
   }

   bool isEmpty() {
       return head == nullptr;
   }

   void addToHead(int value) {
      ...
   }
};
```

```cpp
class Stack: public LinkedList {
public:
   void push(int value) {
       addToHead(value);
   }
   ...
};
```

```cpp
class LinkedList {
   Node* head;
public:

   Node* getHead() {
       return head;
   }

   bool isEmpty() {
       return head == nullptr;
   }

   void addToHead(int value) {
      ...
   }
};
```

```cpp
class Stack: public LinkedList {
public:
   void push(int value) {
       addToHead(value);
   }
   ...
};
```

What do you think about such hierarchy?

```cpp
class LinkedList {
   Node* head;
public:

   Node* getHead() {
       return head;
   }

   bool isEmpty() {
       return head == nullptr;
   }

   void addToHead(int value) {
      ...
   }
};
```

```cpp
class Stack: public LinkedList {
public:
   void push(int value) {
        addToHead(value);
   }
   ...
};
```

What do you think about such hierarchy? Is awful:

1. Stack has some unneeded public methods getHead and etc,

```cpp
class LinkedList {
   Node* head;
public:

   Node* getHead() {
       return head;
   }

   bool isEmpty() {
       return head == nullptr;
   }

   void addToHead(int value) {
      ...
   }
};
```

```cpp
class Stack: public LinkedList {
public:
   void push(int value) {
       addToHead(value);
   }
   ...
};
```

What do you think about such hierarchy? Is awful:

1. Stack has some unneeded public methods getHead and etc,
2. Logic is ruined here: Stack is NOT necessary a LinkedList!

```cpp
class LinkedList {
   Node* head;
public:

   Node* getHead() {
       return head;
   }

   bool isEmpty() {
       return head == nullptr;
   }

   void addToHead(int value) {
      ...
   }
};
```

```cpp
class Stack: public LinkedList {
public:
   void push(int value) {
       addToHead(value);
   }
   ...
};
```

What do you think about such hierarchy? Is awful:

1. Stack has some unneeded public methods getHead and etc,
2. Logic is ruined here: Stack is NOT necessary a LinkedList!
3. LSP?

```cpp
class LinkedList {
   Node* head;
public:

   Node* getHead() {
       return head;
   }

   bool isEmpty() {
       return head == nullptr;
   }

   void addToHead(int value) {
      ...
   }
};
```

```cpp
class Stack: public LinkedList {
public:
   void push(int value) {
       addToHead(value);
   }
   ...
};
```

What do you think about such hierarchy? Is awful:

1. Stack has some unneeded public methods getHead and etc,
2. Logic is ruined here: Stack is NOT necessary a LinkedList!
3. LSP? Violated as hell (LinkedList invariants can be broken)

```cpp
class LinkedList {
   Node* head;
public:

   Node* getHead() {
       return head;
   }

   bool isEmpty() {
       return head == nullptr;
   }

   void addToHead(int value) {
      ...
   }
};
```

```cpp
class Stack: /* private */ LinkedList {
public:
   void push(int value) {
       addToHead(value);
   }
   ...
};

Stack s;
s.getHead(); // compilation error
```

But what if inheritance would be private?

1. No more unneeded public methods getHead and etc,

```cpp
class LinkedList {
   Node* head;
public:

   Node* getHead() {
       return head;
   }

   bool isEmpty() {
       return head == nullptr;
   }

   void addToHead(int value) {
      ...
   }
};
```

```cpp
class Stack: /* private */ LinkedList {
public:
   void push(int value) {
       addToHead(value);
   }
   ...
};

LinkedList* ll = new Stack();
// compilation error!!!
```

But what if inheritance would be private?

1. No more unneeded public methods getHead and etc,
2. Logic is different here Stack is not a LinkedList indeed,
3. LSP is violated, but it is ok, it is just not "is-a" relationship

```cpp
class LinkedList {
   Node* head;
public:

   Node* getHead() {
       return head;
   }

   bool isEmpty() {
       return head == nullptr;
   }

   void addToHead(int value) {
      ...
   }
};
```

```cpp
class Stack: /* private */ LinkedList {
public:
   void push(int value) {
       addToHead(value);
   }
   ...
};

LinkedList* ll = new Stack();
// compilation error!!!
```

What relationship do we have here?

But what if inheritance would be private?

1. No more unneeded public methods getHead and etc,
2. Logic is different here Stack is not a LinkedList indeed,
3. LSP is violated, but it is ok, it is just not "is-a" relationship

```cpp
class LinkedList {
   Node* head;
public:

   Node* getHead() {
       return head;
   }

   bool isEmpty() {
       return head == nullptr;
   }

   void addToHead(int value) {
     ...
   }
};
```

```cpp
class Stack: /* private */ LinkedList {
public:
   void push(int value) {
       addToHead(value);
   }
   ...
};

LinkedList* ll = new Stack();
// compilation error!!!
```

What relationship do we have here?
LinkedList is part of Stack.

But what if inheritance would be private?

1. No more unneeded public methods getHead and etc,
2. Logic is different here Stack is not a LinkedList indeed,
3. LSP is violated, but it is ok, it is just not "is-a" relationship

Imagine we have one more class

```
Student:
  const char* name
  size_t age
  - - - - - - - - - - - -
  size_t group
  size_t id
```

```
Student:
  Person base
  - - - - - - - - - - - -
  size_t group
  size_t id
```

This is called composition.

There are some benefits of such approach, but it just doesn't look logical here (why Person should be a part of Student?)

Also: how can I get a name from Student? Some forwarding method to base => boilerplate code! Something we want to get rid of.

```cpp
class LinkedList {
   Node* head;
public:

   Node* getHead() {
       return head;
   }

   bool isEmpty() {
       return head == nullptr;
   }

   void addToHead(int value) {
      ...
   }
};
```

```cpp
class Stack: /* private */ LinkedList {
public:
   void push(int value) {
       addToHead(value);
   }
   ...
};

LinkedList* ll = new Stack();
// compilation error!!!
```

What relationship do we have here? Linked List is part of Stack. This is composition!!!

But what if inheritance would be private?

1. No more unneeded public methods getHead and etc,
2. Logic is different here Stack is not a LinkedList indeed,
3. LSP is violated, but it is ok, it is just not "is-a" relationship

About building hierarchies

1) Use `public` inheritance where LSP works.

About building hierarchies

1) Use public inheritance where LSP works.

2) Use private inheritance where you want to have composition, but still want to have direct access to private parts of Base without getters.

About building hierarchies

1) Use `public` inheritance where LSP works.

2) Use `private` inheritance where you want to have composition, but still want to have direct access to private parts of Base without getters.

3) Use protected inheritance when you want to `seal` the hierarchy in some reason.

Abstract classes

Abstract classes

```
class Figure {
public:
   virtual double area() {
       throw "error!";
   }
};
```

```
class Square: public Figure {
    double length;
public:
    Square(double l): length(l) {}
    double area() {
        return length * length;
    }
};
```

Abstract classes

```cpp
class Figure {
public:
   virtual double area() {
       throw "error!";
   }
};
```

⇑

Why do we need instances of such classes?

```cpp
class Square: public Figure {
    double length;
public:
    Square(double l): length(l) {}
    double area() {
        return length * length;
    }
};
```

Abstract classes

```
class Figure {
public:
    virtual double area() = 0;
};
```

```
class Square: public Figure {
    double length;
public:
    Square(double l): length(l) {}
    double area() {
        return length * length;
    }
};
```

Abstract classes

```cpp
class Figure {
public:
    virtual double area() = 0;
};
```

Pure virtual function

Abstract class

```cpp
class Square: public Figure {
    double length;
public:
    Square(double l): length(l) {}
    double area() {
        return length * length;
    }
};
```

Abstract classes

```
class Figure {
public:
    virtual double area() = 0;
};
```

Pure virtual function

Abstract class

```
class Square: public Figure {
    double length;
public:
    Square(double l): length(l) {}
    double area() {
        return length * length;
    }
};
```

If class either defines or inherits pure virtual function it becomes abstract

Abstract classes

```
class Figure {
public:
    virtual double area() = 0;
};
```

Pure virtual function

Abstract class

```
class Square: public Figure {
    double length;
public:
    Square(double l): length(l) {}
    double area() {
        return length * length;
    }
};
```

You can't instantiate such classes.
But you can have pointers or references to them.

```
✗ Figure f();
✗ Figure arr[10];
✓ Figure* p = new Square(3.14);
```

Abstract classes

- We need abstract classes to form the correct hierarchies (they are usually roots or their direct successors)

- You can’t create instances of such classes

- If pure virtual function is not overridden in the derived class it also becomes abstract.

Virtual destructors

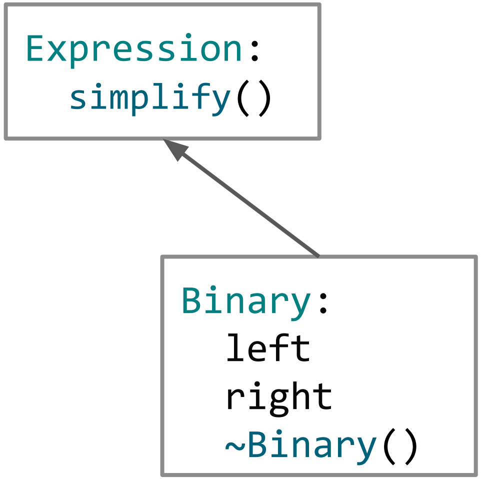
Virtual destructors

```
class Expression {
public:
   virtual Expression* simplify() = 0;
};
```

```
Expression:
  simplify()
```

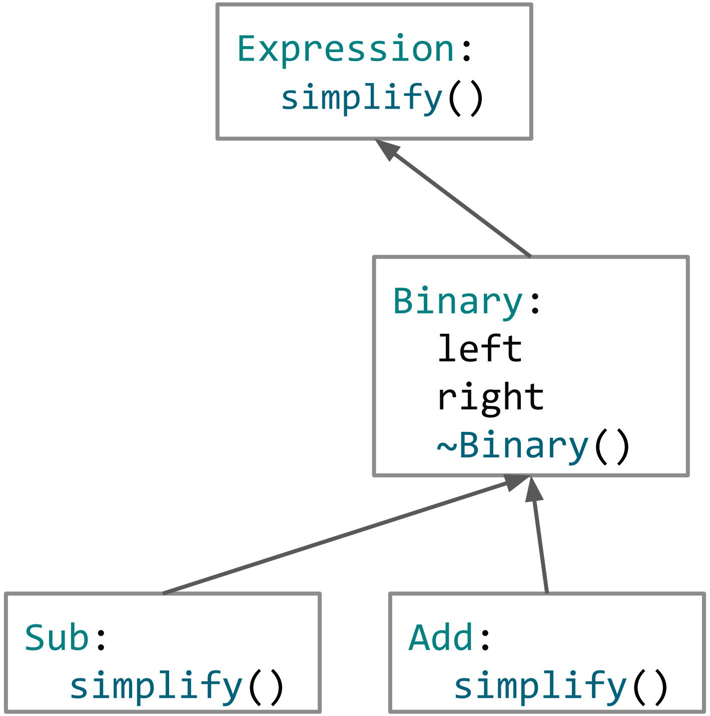
Virtual destructors

```
class Expression {
public:
   virtual Expression* simplify() = 0;
};

class Binary: public Expression {
   Expression* left;
   Expression* right;
public:
   ~Binary() {
       delete left;
       delete right;
   }
};
```



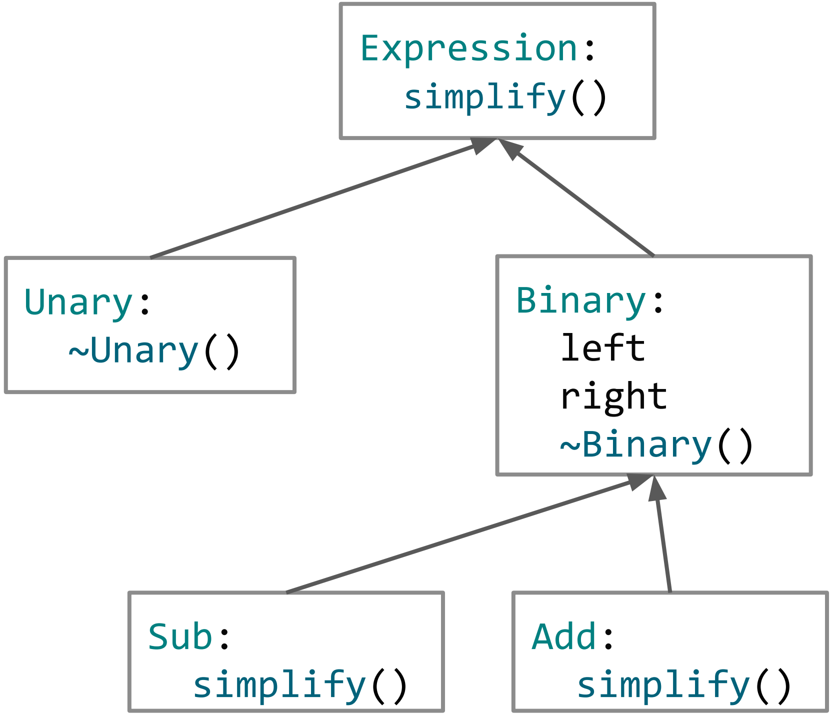
Virtual destructors

```
class Expression {
public:
    virtual Expression* simplify() = 0;
};

class Binary: public Expression {
    Expression* left;
    Expression* right;
public:
    ~Binary() {
        delete left;
        delete right;
    }
};

class Add: public Binary { ... };
class Sub: public Binary { ... };
```



Virtual destructors

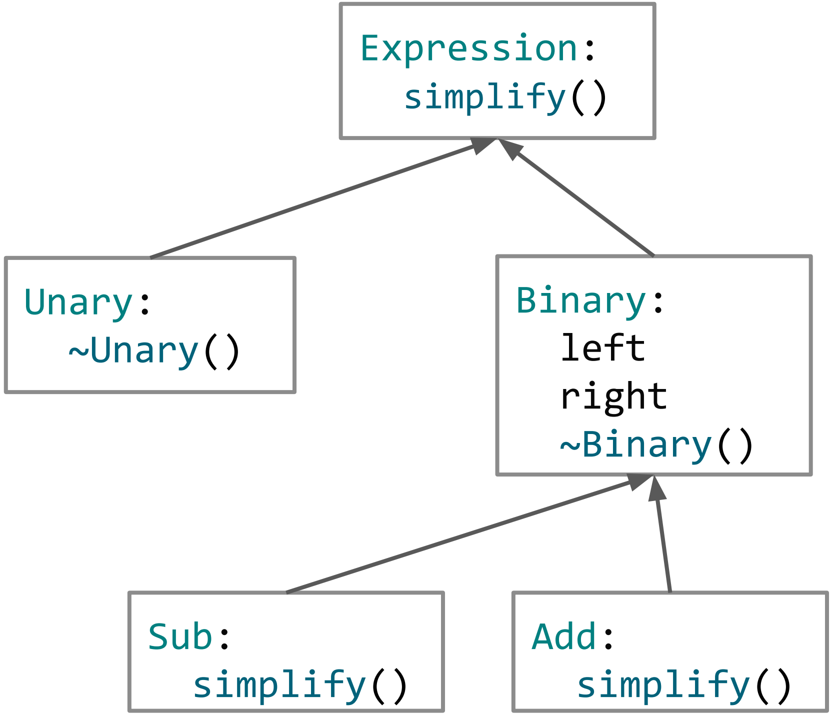
```
class Expression {
public:
   virtual Expression* simplify() = 0;
};

class Binary: public Expression {
   Expression* left;
   Expression* right;
public:
   ~Binary() {
       delete left;
       delete right;
   }
};

class Add: public Binary { ... };
class Sub: public Binary { ... };
```



Do you see any problems here?

Virtual destructors

```
class Expression {
public:
   virtual Expression* simplify() = 0;
};

class Binary: public Expression {
   Expression* left;
   Expression* right;
public:
   ~Binary() {
       delete left;
       delete right;
   }
};

class Add: public Binary { ... };
class Sub: public Binary { ... };
```



Virtual destructors

```
class Expression {
public:
   virtual Expression* simplify() = 0;
};

class Binary: public Expression {
   Expression* left;
   Expression* right;
public:
   ~Binary() {
       delete left;
       delete right;
   }
};

class Add: public Binary { ... };
class Sub: public Binary { ... };
```

```
Expression* exp = new Add(...);
```

Virtual destructors

```
class Expression {
public:
   virtual Expression* simplify() = 0;
};

class Binary: public Expression {
   Expression* left;
   Expression* right;
public:
   ~Binary() {
       delete left;
       delete right;
   }
};

class Add: public Binary { ... };
class Sub: public Binary { ... };
```

```
Expression* exp = new Add(...);
exp->simplify();
// what will be called?
```

Virtual destructors

```
class Expression {
public:
   virtual Expression* simplify() = 0;
};

class Binary: public Expression {
   Expression* left;
   Expression* right;
public:
   ~Binary() {
       delete left;
       delete right;
   }
};

class Add: public Binary { ... };
class Sub: public Binary { ... };
```

```
Expression* exp = new Add(...);
exp->simplify();
// Add:simplify will be called
```

Virtual destructors

```
class Expression {
public:
   virtual Expression* simplify() = 0;
};

class Binary: public Expression {
   Expression* left;
   Expression* right;
public:
   ~Binary() {
       delete left;
       delete right;
   }
};

class Add: public Binary { ... };
class Sub: public Binary { ... };
```

```
Expression* exp = new Add(...);
exp->simplify();
// Add:simplify will be called

delete exp;
// what will be called?
```

Virtual destructors

```
class Expression {
public:
   virtual Expression* simplify() = 0;
};

class Binary: public Expression {
   Expression* left;
   Expression* right;
public:
   ~Binary() {
       delete left;
       delete right;
   }
};

class Add: public Binary { ... };
class Sub: public Binary { ... };
```

```
Expression* exp = new Add(...);
exp->simplify();
// Add:simplify will be called

delete exp;
// Expression::~Expression
// Memory leak!
```

How to fix?

Virtual destructors

```
class Expression {
public:
   virtual Expression* simplify() = 0;
};

class Binary: public Expression {
   Expression* left;
   Expression* right;
public:
   ~Binary() {
       delete left;
       delete right;
   }
};

class Add: public Binary { ... };
class Sub: public Binary { ... };
```

```
Expression* exp = new Add(...);
exp->simplify();
// Add:simplify will be called

delete exp;
// Expression::~Expression
// Memory leak!
```

How to fix? We need to have a virtual constructor in Expression.

Virtual destructors

```
class Expression {
public:
   virtual Expression* simplify() = 0;
   virtual ~Expression() {};
};

class Binary: public Expression {
   Expression* left;
   Expression* right;
public:
   ~Binary() {
       delete left;
       delete right;
   }
};
class Add: public Binary { ... };
class Sub: public Binary { ... };
```

```
Expression* exp = new Add(...);
exp->simplify();
// Add:simplify will be called

delete exp;
// Binary::~Binary() will be called
// No more memory leak 😊
```

Not So Tiny Task №7 (2 points)

Implement a hierarchy of classes for symbolic differentiation of expressions.

- Base class: Expression;
- Derived classes: Binary, Unary, Add, Sub, Mult, Div, Exponent, Var, Val;
- Base class expression should contain pure virtual function Expression* diff(std::string var); (its implementations should return differentiation result for the expression by the given variable);
- Tests should be prepared as usual.

Not So Tiny Task №7 (2 points)

```
Example:

Expression* e = new Add(new Var("x"),
                        new Mult(new Val(10), new Var("y")));

// e = x + 10*y

Expression* res1 = e->diff("x");
// res1 = 1 + (0*y + 10*0) (it is ok to have non-simplified exprs)

Expression* res2 = e->diff("y");
// res2 = 0 + (0*y + 10*1) (it is ok to have non-simplified exprs)
```

Takeaways

- Inheritance in subtyping polymorphism in C++
- LSP as its theoretical base (please don't be too serious about it)
- Virtual functions in C++ and virtual destructors
- Pure virtual functions, abstract classes

How virtual functions actually work?