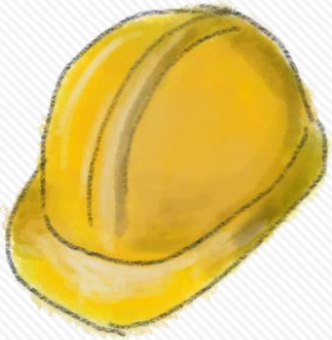
Inheritance

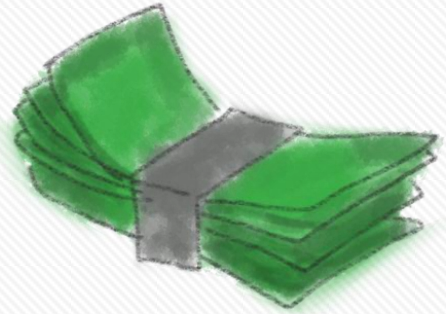
Inheritance of Classes and Virtual Functions

A Problem

- Assume we wish to create software to handle human resources in our company

- We identify that we have **two types** of employees in our company:
  - Engineers
  - Sales people





- What classes should we create?

One-Class Solution

- Creating one class for all types of employees will cause **incoherent code**:

```
class Employee {
public:
  // ...
  string getName() const;
  void giveRaise(int amount);
  enum Type { ENGINEER, SALESPERSON };

private:
  string name;
  Date birth;
  int salary;
  Type type;
  set<string> degrees;  // for an engineer
  double comissionRate; // for a sales person
};
```

signs of a class trying to fill too many roles

Two-Class Solution

- Creating one class for engineers, and another for sales people, will cause **code duplication**:

```
class Engineer {
public:
  // ...
  string getName() const;
  void giveRaise(int amount);

private:
  string name;
  Date birth;
  int salary;
  set<string> degrees;
};
```

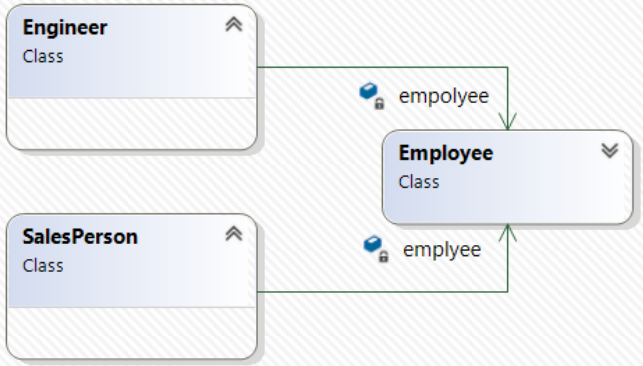
```
class SalesPerson {
public:
  // ...
  string getName() const;
  void giveRaise(int amount);

private:
  string name;
  Date birth;
  int salary;
  double comissionRate;
};
```

```
void printAddress(const Engineer& e);
void printAddress(const SalesPerson& s);
```

need to duplicate functions to handle both classes

Identifying the Common Part

- Engineers and SalesPerson have **common functionality**
  - We can extract this part into a third class:




```
class Employee {
public:
  // ...
  string getName() const;
  void giveRaise(int amount);
private:
  string name;
  Date birth;
  int salary;
};
```

common to all employees

just calls the function
employee.getName()

```
class Engineer {
public:
  // ...
  string getName() const;
  void giveRaise(int amount);
private:
  Employee employee;
  set<string> degrees;
};
```

```
class SalesPerson {
public:
  // ...
  string getName() const;
  void giveRaise(int amount);
private:
 Employee employee;
 double comissionRate;
};
```

Using the Common Part

- This solution is better, but still has problems:
  - We need to write **extra code** to expose the functionality of the common class in the specialized classes:

```
string Engineer::getName() const {
  return employee.getName();
}
```

  - What if we want code that can work on **any** employee?

```
void printEmployeeDetails(const Employee& emp) {

  cout << "Employee details: " << endl;
  cout << "Name: " << emp.getName() << endl;
  cout << "Salary: " << emp.getSalary() << endl;

}
```

this function will not work on an Engineer or SalesPerson object, even though both have getName() and getSalary() functions

Inheritance

- In object-oriented languages, we solve this using **inheritance**
- Inheritance allows us to take an existing class, and **derive new classes** from it
  - So Engineer and SalesPerson can **inherit the functionality** of an Employee class

```
class Employee {
public:
  // ...
  string getName() const;
  void giveRaise(int amount);
private:
  string name;
  Date birth;
  int salary;
};
```

```
class Engineer : public Employee {
public:
  // engineer-only functions ...

private:
  set<string> degrees;

};
```

Inheritance

- The inheriting class has all the **fields and methods** of the base class, plus the **new ones** it defines

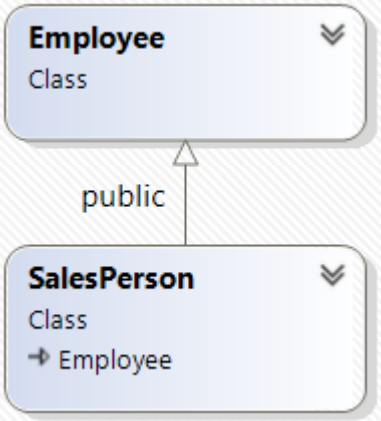
```
class Employee {
public:
  // ...
  string getName() const;
  void giveRaise(int amount);
private:
  string name;
  Date birth;
  int salary;
};
```

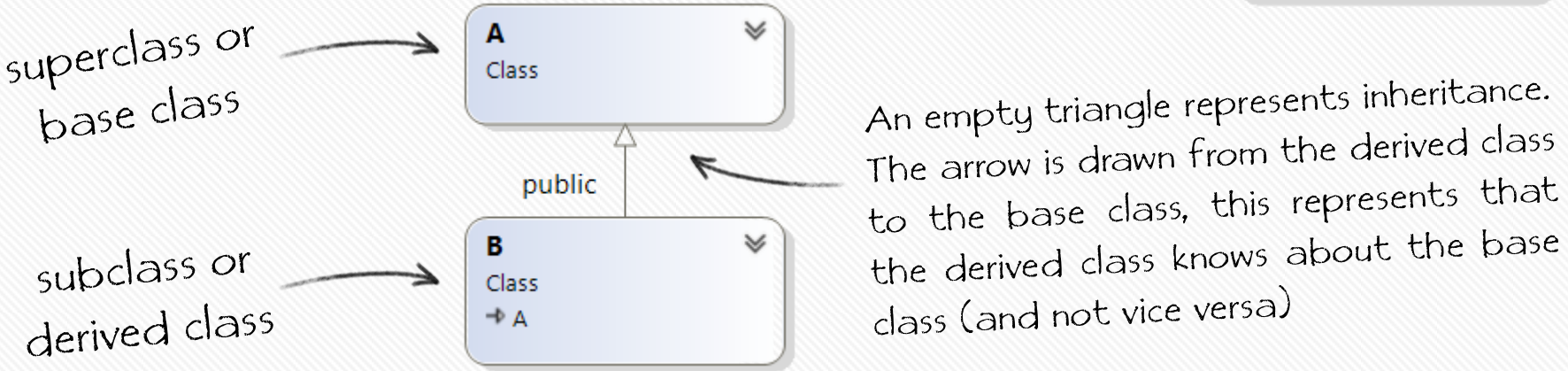
```
class Engineer : public Employee {
public:
  void addDegree(string degree);

private:
  set<string> degrees;
};
```

```
void f(Engineer& engineer) {
  engineer.addDegree("B.Sc. Computer Science");
  engineer.giveRaise(10);
}
```

Inheritance - Terminology

- There are several equivalent terms for describing inheritance relations:
  - Class B **inherits** class A
  - Class B **derives from** class A
  - A is the **base class** and B is the **derived class**
  - A is the **superclass** and B is the **subclass**





Access Control

- The derived class **cannot access** the **private section** of the base class (even though it contains it!)

```
class Employee {
public:
  string getName() const;
  void giveRaise(int amount);

private:
  string name;
  Date birth;
  int salary;
};
```

```
void Engineer::addReview(string text, int score) {
  reviews.push_front(text);
  if (score == 5) {
    salary += 1000;
  }
}
```

error: salary is private!

```
void f(Engineer& engineer) {
  engineer.salary = 1000;
}
```

```
class Engineer : public Employee {
public:
  void addReview(string text, int score);

private:
  list<string> reviews;
};
```

Access Control

- We can declare class members as **protected**
  - Protected members can be accessed from within **any class deriving from the base class**
  - Protected members **cannot be accessed** from outside the class

```
class Employee {
public:
  string getName() const;
  void giveRaise(int amount);
private:
  string name;
  Date birth;
protected:
  int salary;
};
```

```
void Engineer::addReview(string text, int score) {
  reviews.add(text);
  if (score == 5) {
    salary += 1000;
  }
}
```

o.k., salary is protected

```
class Engineer : public Employee {
public:
  void addReview(string text, int score);
private:
  list<string> reviews;
};
```

```
void f(Engineer& engineer) {
  engineer.salary = 1000;
}
```

error: salary is protected

Subtyping

- A derived class such as Engineer is a **subtype** of the base class Employee
  - This means that an Engineer **can be used** wherever an Employee is acceptable
- This is possible because all the public members of the base class are guaranteed to be available in the derived class as well

```
void giveRaiseAndLog(Employee& e) {
  e.giveRaise(1000);
  cout << "gave raise to ";
  cout << e.getName() << endl;
}
```

```
Engineer engineer("Jane Doe");
SalesPerson salesPerson("John Doe");
giveRaiseAndLog(engineer);
giveRaiseAndLog(salesPerson);
```

An engineer *is an* employee

Subtyping

- In C++, if B is a subtype of A, this means that **B* can be used as A***, and **B& can be used as A&**
  - But not vice versa!
  - Note that this requires use of either pointers or references

```
Engineer engineer("Jane Doe");
SalesPerson salesPerson("John Doe");
Employee employee("John Smith");

Employee& ref = engineer; // o.k.
Employee* ptr = &salesPerson; // o.k.

Engineer* ptr2 = &employee; // error

Employee copy = engineer; // works,
                // but probably a bug

Employee* array[] = {
  &engineer, &salesPerson, &employee
};
```

Engineer is an Employee

SalesPerson is an Employee

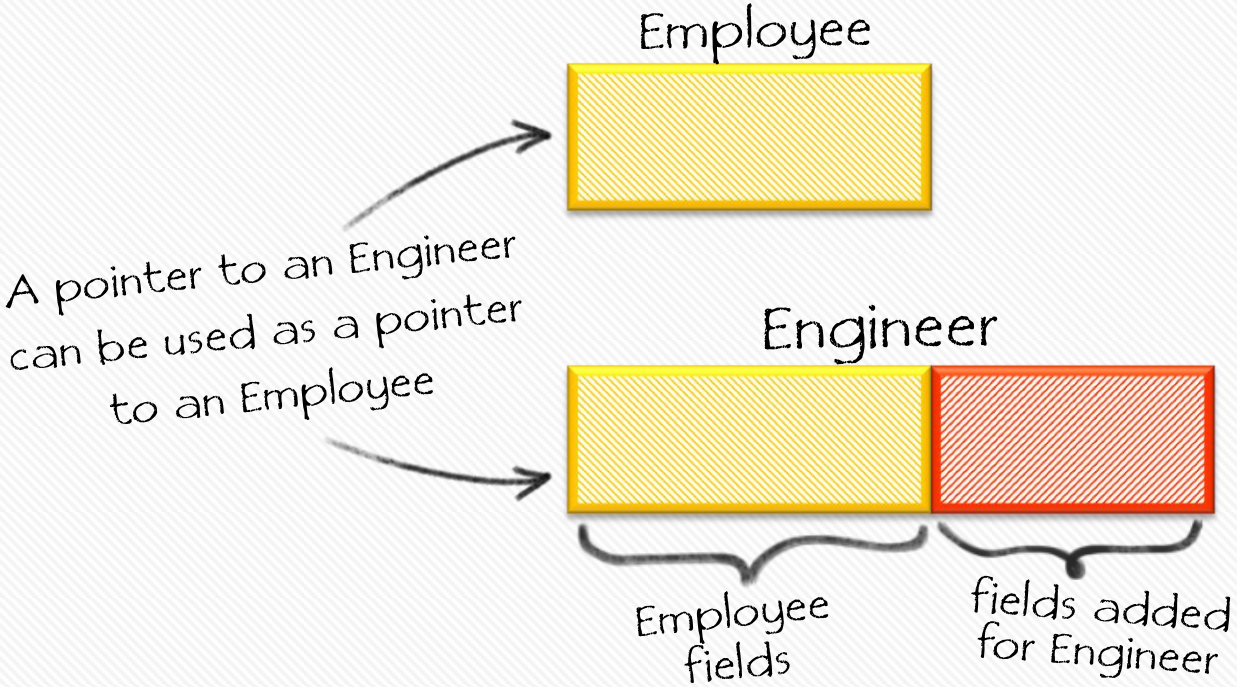
An Employee is not necessarily an Engineer

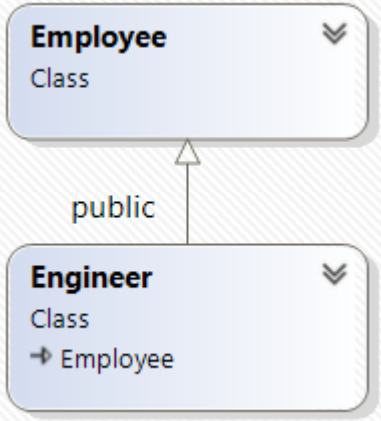
Copies only the "Employee part" of Engineer

Subtyping allows grouping
all types of employees
(in an array or any other container of pointers)

Subtyping

- The common implementation of subtyping is done by **adding the data** of the derived class **at the end** of a base class object





Constructors

- When constructing a derived class, its **base class part is constructed first**. The order of operations is:
  1. The **base class** members are constructed
  2. The **base class** constructor code is called
  3. The **derived class** members are constructed
  4. The **derived class** constructor code is called

- We can pass arguments to the base class constructor in the **initialization list**
  - Otherwise, the base class default constructor is used

```
class Employee {
public:
  Employee(string name);
private:
  string name;
  int salary;
};
```

```
class SalesPerson : public Employee {
public:
  SalesPerson(string name, double commission);
private:
  double commission;
};
```

Employee part is constructed first

```
Employee::Employee(string name)
  : name(name), salary(STARTING_SALARY)
{}
```

```
SalesPerson::SalesPerson(string name, double commission)
  : Employee(name), commission(commission)
{}
```

what would happen without this explicit initialization?

a compilation error, since the compiler would try to call Employee::Employee() which does not exist

Adding a Manager Class

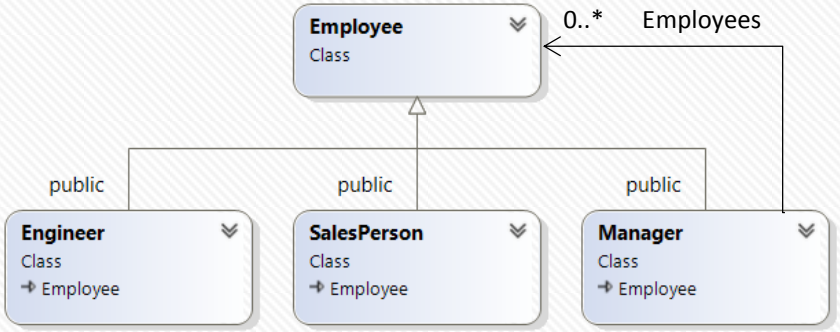
- Let's add a **Manager** class

```
class Employee {
public:
  Employee(string name);
private:
  string name;
  int salary;
};
```

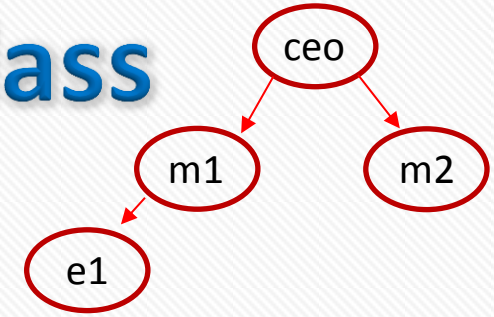
```
Employee::Employee(string name) :
  name(name), salary(STARTING_SALARY)
{}
```

```
Manager::Manager(string name) :
  Employee(name), employees()
{}
```

```
class Manager : public Employee {
public:
  Manager(string name);
private:
  set<Employee*> employees;
};
```



Adding a Manager Class



- Let's add a **Manager** class
  - Using subtyping, existing code for Employees will work with this new class!
  - **No extra code** is needed for this

```
class Manager : public Employee {
public:
  Manager(string name);
  bool isManagerOf(const Employee* emp) const;
  void addEmployee(Employee* emp);
private:
  set<Employee*> employees;
};
```

some of the employees can be Managers themselves

```
Manager ceo("Ron");

Manager m1("Liav");
ceo.addEmployee(&m1);

Manager m2("Aharon");
ceo.addEmployee(&m2);

Engineer e1("Yossi");
m1.addEmployee(&e1);

giveRaiseAndLog(ceo);
```

this function will work with a Manager object, even though the class didn't even exist when the function was written!

Compiler-Generated Functions

- A derived class automatically has a **default constructor**, a **copy constructor**, and an **assignment operator**, like any other class
  - The compiler-generated default constructor calls the base class default constructor
  - The compiler-generated copy constructor calls the base class copy constructor
  - The compiler-generated assignment operator calls the base class assignment operator
- A default constructor is only generated when no other constructors are defined for the derived class
- The base class constructors are **not inherited**
  - We must explicitly define any constructors the derived class needs

```
class Employee {
public:
  Employee(string name);
private:
  string name;
  int salary;
};
```

```
class Manager : public Employee {
public:
  Manager(string name);
private:
  set<Employee*> group;
};
```

the constructor from Employee that takes a string name is not inherited – we must define it explicitly

Destructors

- Destructors are also **not inherited**
- When destructing a derived class, its **base class part is destructed last**
  - First the derived class destructor is called, then the derived class members are destructed, then the base class destructor is called, and finally the base class members are destructed
- There is **no need to explicitly call** the base class destructor from the derived class destructor – it is called automatically

```
class Manager : public Employee {
public:
  ~Manager();
private:
  set<Employee*> group;
};
```

```
Manager::~Manager() {
  // ...
}
```

Employee part is destructed here

```
class Engineer : public Employee {
public:
  void addReview(string text, int score);
private:
  list<string> reviews;
};
```

the compiler-generated Engineer::~Engineer() simply calls Employee::~Employee()

Overriding Methods

- What if different employees have different computations for giveRaise()?
- We can achieve this by **overriding** the base class function
  - Member variables cannot be overridden

```
class Employee {
public:
  // ...
  void giveRaise(int amount);
};
```

```
void Employee::giveRaise(int amount) {
  salary += amount;
}
```

```
class SalesPerson : public Employee {
public:
  // ...
  void giveRaise(int amount);
};
```

```
void SalesPerson::giveRaise(int amount) {
  Employee::giveRaise(amount + comissionRate*sales);
}
```

the overridden function is declared again in the derived class

the overridden method of the base class can be called using the base class name

(it is common for the overriding function to call the overridden one)

Overriding Methods

- Using overrides, each class in the hierarchy can have a **different behavior** for the **same message**:

```
class Employee {
public:
  // ...
  void giveRaise(int amount);
};
```

```
class SalesPerson : public Employee {
public:
  // ...
  void giveRaise(int amount);
};
```

```
class Manager : public Employee {
public:
  // ...
  void giveRaise(int amount);
};
```

```
void SalesPerson::giveRaise(int amount) {
  Employee::giveRaise(amount + comissionRate * sales);
}
```

```
void timeForBonus(SalesPerson& salesPerson,
                  Manager& manager,
                  Engineer& engineer) {

  salesPerson.giveRaise(1000);
  manager.giveRaise(1000);
  engineer.giveRaise(1000);
}
```

SalesPerson::giveRaise(int)

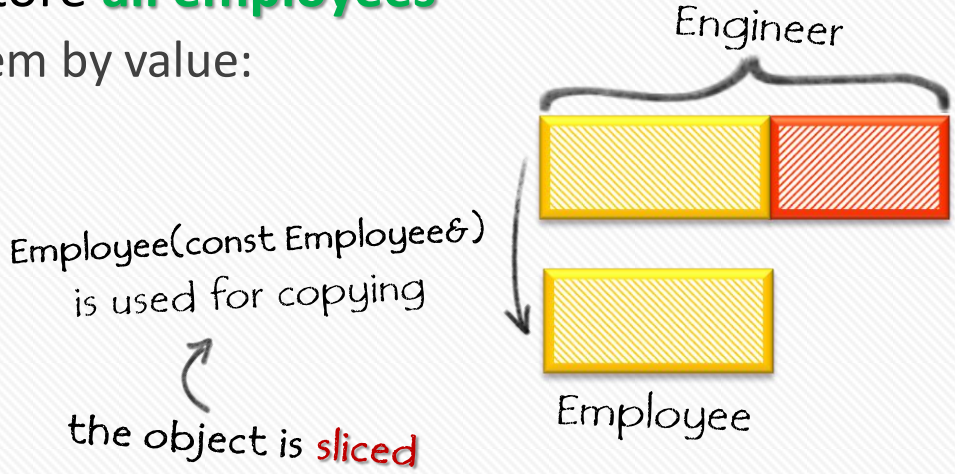
Manager::giveRaise(int)

Employee::giveRaise(int)
(not overridden)

Overriding Methods: Caveat

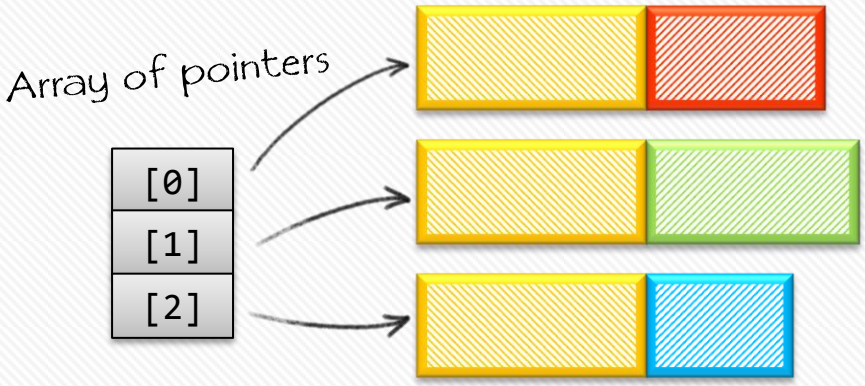
- We would like a container to store **all employees**
  - However, we cannot store them by value:

```
Engineer engineer("Jane Doe");
SalesPerson salesPerson("John Doe");
Manager manager("Mr. Bossman");

Employee employees[] = {
  engineer, salesPerson, manager
};
```



- When we wish to create an array (or any other container) of **mixed types**, we must **store pointers**

```
Employee* employees[] = {
  &engineer, &salesPerson, &manager
};

list<Employee*> employees2;
employees2.push_back(&engineer);
employees2.push_back(&salesPerson);
employees2.push_back(&manager);
```



Overriding Methods: Caveat

- Calling a function through a pointer (or reference) to a base class imposes **a problem**:
  - The compiler has to choose which function to call **during compilation** ← this is called "static binding"
  - But the function that **should** be invoked is only known **at run-time**

```
Engineer engineer("Jane Doe");
SalesPerson salesPerson("John Doe");
Manager manager("Mr. Bossman");

Employee* employees[] = {
  &engineer, &salesPerson, &manager
};

for (int i = 0; i < 3; ++i) {
  employees[i]->giveRaise(100);
}
```

Which function should be called here?

Engineer::giveRaise(int)
SalesPerson::giveRaise(int)
Manager::giveRaise(int)
Employee::giveRaise(int)

the compiler only knows that employees[i] is an Employee pointer, so Employee::giveRaise(int) is invoked!

Overriding Methods

- How can we solve this problem?
  - First attempt: **explicitly determine the object's type at run-time** using a field with the object's type

```
class Employee {
  string type;
public:
  // ...
  Employee(string type) : type(type) {}
  string getType() const;
};
```

```
class Manager : public Employee {
public:
  // ...
  Manager() : Employee("manager") {}
};
```

```
Employee* employees[] = { ... };

for (int i = 0; i < n; ++i) {
  if (employees[i].getType()=="manager") {
    ((Manager*)e)->giveRaise(100);
  }
  // repeat for every type ...
}
```

static_cast

- First improvement: replace C-style cast with C++ **static_cast**
  - A static cast is **checked by the compiler**
  - If there is **no reasonable connection** between the two types, the code **will not compile**
    - Unlike C-style casts, which will compile with **undefined behavior**

```
Employee* employees[] = { ... };

for (int i = 0; i < n; ++i) {
  if (employees[i].getType() == "manager") {
    static_cast<Manager*>(employees[i])->giveRaise(100);
  }
  // repeat for every type ...
}
```

no casting needed

```
Manager* m = ...;
Employee* emp = m;
Engineer* eng = static_cast<Engineer*>(m);
Manager* m1 = static_cast<Manager*>(emp);
```

compilation error: cannot cast from Manager* to Engineer*

o.k.

Note: in a static_cast from type A to type B, the compiler only checks that the conversion A->B makes sense. It does NOT check that the parameter is really of type B (and if it's not, the result is still undefined)

Virtual Functions

- Better solution: instead of **"asking" each employee for his type**, let's simply **"tell" him to invoke the right behavior**
- For this we declare a function in the base class as **virtual**
  - A virtual function has **dynamic binding** (as opposed to **static binding**), which means the identity of the function is **determined at run-time**

```
class Employee {
public:
  // ...
  virtual void giveRaise(int amount);
};
```

```
class SalesPerson : public Employee {
public:
  // ...
  void giveRaise(int amount);
};
```

when giveRaise() is invoked - choose the correct function at run-time

```
SalesPerson salesPerson("John Doe");
Employee* emp = &salesPerson;

emp->giveRaise(1000);
```

calls SalesPerson::giveRaise()

Virtual Functions

```
class Employee {
public:
  // ...
  virtual void giveRaise(int amount);
};
```

```
class SalesPerson : public Employee {
public:
  // ...
  void giveRaise(int amount);
};
```

```
class Engineer : public Employee {
public:
  // ...
  void giveRaise(int amount);
};
```

```
class Manager : public Employee {
public:
  // does not override giveRaise()
};
```

```
Engineer engineer("Jane Doe");
SalesPerson salesPerson("John Doe");
Manager manager("Mr. Bossman");

Employee* employees[] = {
  &engineer, &salesPerson, &manager
};

for (int i = 0; i < 3; ++i) {
  e->giveRaise(1000);
}
```

this will automatically call the correct function for each object!

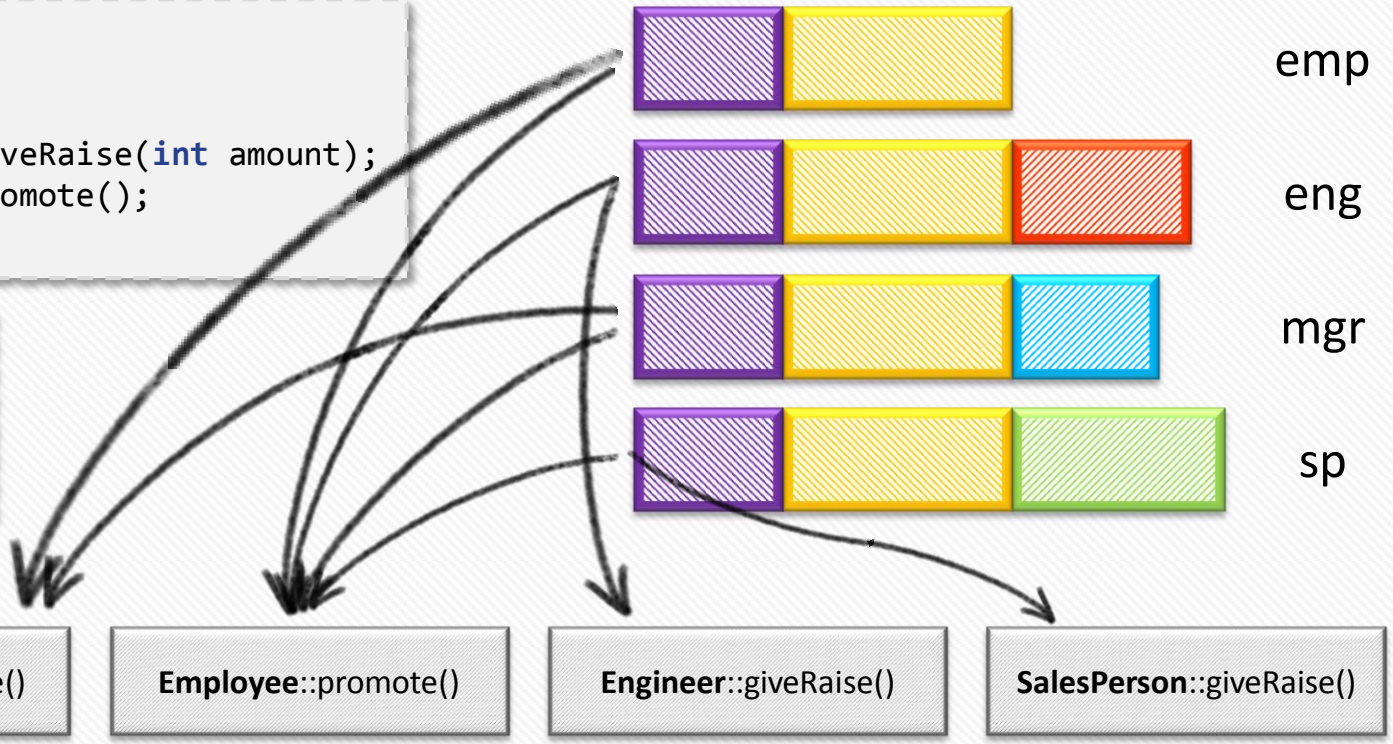
Engineer::giveRaise for employees[0]
SalesPerson::giveRaise for employees[1]
Employee::giveRaise for employees[2]

Implementing Virtual Functions

- To implement virtual functions, the compiler adds a list of **function pointers** to each object when it is created

```
class Employee {
public:
  // ...
  virtual void giveRaise(int amount);
  virtual void promote();
};
```

```
Employee emp;
Engineer eng;
Manager mgr;
SalesPerson sp;
```

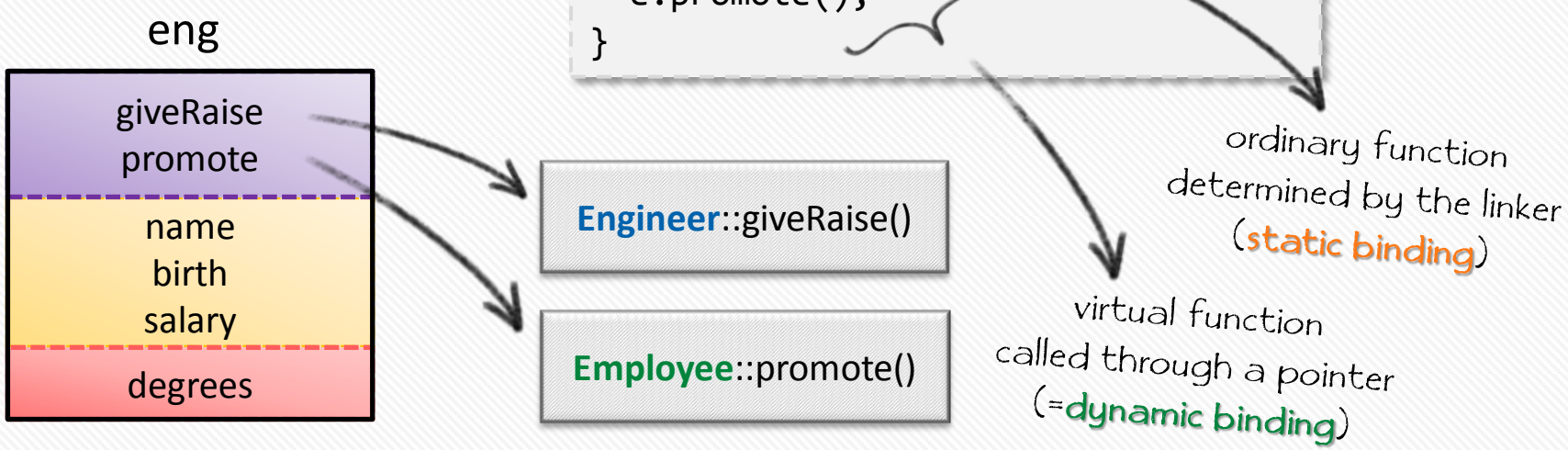


Implementing Virtual Functions

- Virtual function calls and ordinary function calls are thus treated **very differently** by the compiler

```
Engineer eng;
promoteAndRaise(eng);
```

```
void promoteAndRaise(Employee& e) {
  cout << e.getName() << endl;
  e.giveRaise(500);
  e.promote();
}
```



* In reality, C++ implements this in a slightly different way, using less memory per object (not in the material)

Abstract Classes

- In many cases, the base class represents an **abstract concept**
  - There is no reasonable implementation for some of the virtual functions
- In such cases, we can declare a **pure virtual function**
  - Pure virtual functions have **no implementation** in the base class

```
class Employee {
public:
  // ...
  virtual void giveRaise(int amount) = 0;
};
```

this function is not implemented for a general Employee

```
class SalesPerson : public Employee {
public:
  // ...
  void giveRaise(int amount);
};
```

SalesPerson provides a concrete implementation of giveRaise

Abstract and Concrete Classes

- A class with **one or more pure virtual functions** is called an **abstract class**
  - Objects of an abstract class **cannot be created**
  - **Pointers** and **references** to abstract classes **can still be created**
- A derived class that **implements all the pure-virtual functions** becomes a **concrete class,** and objects can be created from this class
  - Otherwise, it remains **abstract**

```
class Employee {
public:
  // ...
  virtual void giveRaise(int amount) = 0;
};
```

```
class Manager : public Employee {
public:
  // ...
  void giveRaise(int amount);
};
```

```
Employee e; // error (abstract)
Manager m;  // o.k.

Employee* ptr = &m;
Employee& ref = m;
```

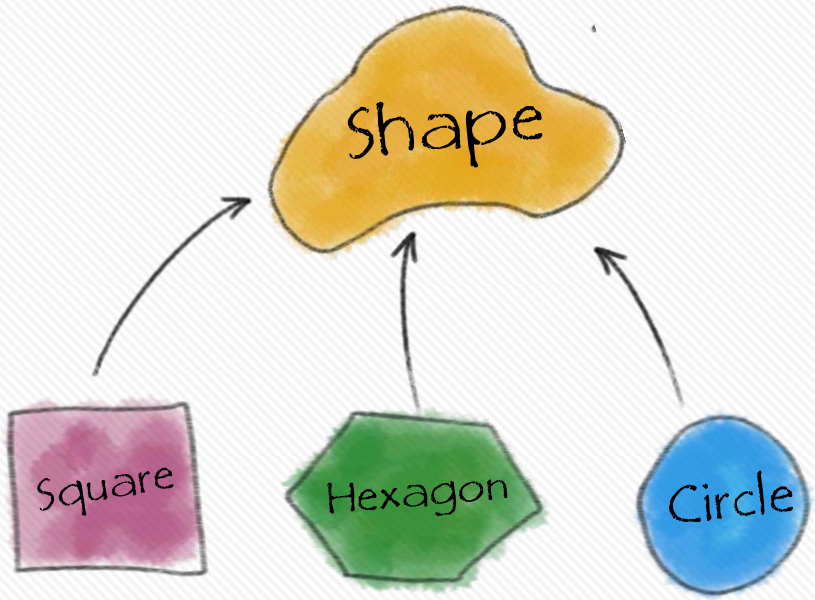
Interfaces

- An abstract class that has **only pure virtual functions** is called an **interface**

```
class Shape {
public:
  virtual void rotate(double angle) = 0;
  virtual void draw() const = 0;
  virtual double area() const = 0;
};
```

```
class Circle : public Shape {
  // ...
public:
  Circle (double radius);
  void rotate(double angle);
  void draw() const;
  double area() const;
};
```

```
double Circle::area() const {
  return radius * radius * PI;
}
// … implementation of all virtual functions
```



Constructors and Destructors

- **Constructors are never virtual** – we always use the constructor of the exact class we want to create
- **Destructors must be virtual** when using inheritance
  - This ensures that if we delete an object through a pointer to its base class, the **correct destructor** will be called

```
class Employee {
  // ...
public:
  virtual ~Employee() {}
  // ...
};
```

C++11 "=default" can also be used

```
Employee* emp = new Engineer("John");

// ...

delete emp;
```

the correct d'tor, Engineer::~Engineer(), will be called at run-time

Constructors and Destructors

- What if we wanted to create an **exact copy** of an object, but we only had a pointer to its base class?
  - This happens very often with inheritance

```
Shape* duplicateShape(const Shape* s) {
  Shape* newShape = ??? <create a copy of the shape in s>
  newShape->moveX(10);
  newShape->moveY(10);
  newShape->show();
  return newShape;
}
```

Somewhere in PowerPoint's code...

Constructors and Destructors

- Solution: define a virtual **clone()** function in the base class
  - Sometimes referred to as a "virtual copy constructor"

```
class Shape {
  // ...
public:
  virtual Shape* clone() const = 0;
  // ...
};
```

```
class Circle : public Shape {
  // ...
public:
  Circle(double radius);
  virtual Shape* clone() const;
  // ...
};
```

```
Shape* Circle::clone() const {
  return new Circle(*this);
}
```

```
Shape* ptr = new Circle(3.0);

Shape* copy = ptr->clone();
```

the true type of *copy is the same type as *ptr

Polymorphism

- The **main use** of inheritance and virtual functions is to enable **polymorphic behavior**

```
class Employee {
public:
  virtual void giveRaise(int amount);
  virtual double getYearlyCost() const;
  virtual ~Employee() {}
};
```

```
double yearlyCost(const Employee* employees[], int n)
{
  double totalCost = 0.0;
  for (int i = 0; i < n; ++i) {
    totalCost += employees[i]->getYearlyCost();
  }
  return totalCost;
}
```

this code will even work for future employee types that haven't been written yet!

we don't need to know the real type of each employee, we just ask it for its cost!

Polymorphism

- **Polymorphism** – from Greek, meaning **"having multiple forms"**
- Polymorphic code works on all types that share a common base class, **without having to know the real type of each object**
- Polymorphism can improve code:
  - **Simplifies** code by making it oblivious to the specific type of the object
  - Allows extending the code in the future **without changing old code**
- First hint to consider polymorphism:
  - The code contains `if` statements that separate to **different behaviors based on the "type" of the object** (e.g., if it's a square, do X, else if it's a circle…)

The override keyword

- The C++11 **override** keyword allows to explicitly indicate that a function is overriding a virtual function of a base class
  - Using it causes a **compilation error** if there is no virtual function with a matching signature in the base class

```
class Employee {
public:
  virtual void giveRaise(int amount);
};
```

```
class QASpecialist : public Employee {
  // ...
public:
  QASpecialist(int qualification);
  void giveRaise(int amount) override;
  // ...
};
```

C++11

```
void QASpecialist::giveRaise(int amount) {
  Employee::giveRaise(amount * qualification);
}
```

if class Employee has no function void giveRaise(int amount), or if this function is not virtual, this line will not compile

use the override keyword to:

- improve code readability
- avoid nasty bugs

you are required to use override in our course

Run Time Type Information

- Sometimes virtual functions are not enough and we must **query the type of an object**
  - For example, assume we want to print all engineers and their degrees

```
class Engineer : public Employee {
  // ...
public:
  void printDegrees() const;
};
```

a function that only engineers have

```
void printDegrees(const Employee* employees[], int n)
{
  for (int i = 0; i < n; ++i) {
    if (/** employees[i] is an Engineer **/) {
      cout << employees[i]->getName() << endl;
      employees[i]->printDegrees();
    }
  }
}
```

uh-oh! this will not compile since employees[i] is of type Employee*, which does not have a printDegrees() function

Run Time Type Information

- We wish to avoid the dirty solution of adding a "**type field**" or a "**type method**":

```
class Employee {
public:
  virtual string type() const = 0;
  // ...
};
```

```
class Engineer : public Employee {
  // ...
public:
  string type() const override {
    return "engineer";
  }
};
```

```
void printDegrees(const Employee* employees[], int n)
{
  for (int i = 0; i < n; ++i) {
    if (employees[i]->type() == "engineer") {
      const Engineer* eng =
        static_cast<const Engineer*>(employees[i]);
      cout << eng->getName() << endl;
      eng->printDegrees();
    }
  }
}
```

Run Time Type Information

the "real" type

- C++ allows querying the **dynamic type** of an object using the **dynamic_cast** and operator
- A **dynamic_cast** casts from a base class pointer to a derived class pointer in a **polymorphic hierarchy**
  - A class is polymorphic if it has **at least one virtual function**
  - A dynamic_cast is **checked for correctness at run-time**. If it fails, NULL is returned

```
void printDegrees(const Employee* employees[], int n) {
  for (int i = 0; i < n; ++i) {
    const Engineer* eng = dynamic_cast<const Engineer*>(employees[i]);
    if (eng != NULL) {
      eng->printDegrees();
    }
  }
}
```

typeid Operator

- The **typeid** operator is used to get the type of an object at **run time**
  - typeid returns an object of type **type_info**
  - A type_info object may be **compared** with other type_info objects
  - The **name()** method of type_info returns a string representing the type's name

```
Shape* ptr  = new Circle(3.0);
Shape* ptr2 = new Square(2.0);

if (typeid(*ptr) == typeid(*ptr2)) {
  cout << "same type" << endl;
}

if (typeid(*ptr) == typeid(Circle)) {
  cout << "*ptr is a Circle" << endl;
}

type_info ptrType = typeid(ptr);
type_info objType = typeid(*ptr);
cout << ptrType.name() << endl;
cout << objType.name() << endl;
```

```
bool lessThan(const Shape& s1,
              const Shape& s2) {

  type_info type1 = typeid(s1);
  type_info type2 = typeid(s2);
  if (type1 != type2) {
    return type1.before(type2);
  }
  return s1.area() < s2.area();
}
```

List and SortedList

- We want to create **two classes**:
  - A list of integers
  - A sorted list of integers

C++11

```
List list = { 3, 2, 4, 1, 5 };

for (List::Iterator it =
     list.begin();
     it != list.end(); ++it) {
  cout << *it << ", ";
}
```

prints: "3, 2, 4, 1, 5,"

```
SortedList list = { 3, 2, 4, 1, 5 };

for (List::Iterator it =
     list.begin();
     it != list.end(); ++it) {
  cout << *it << ",";
}
```

prints: "1, 2, 3, 4, 5,"

- **Should** SortedList be derived from List?
  - Is SortedList a List?

List and SortedList

- SortedList **is not** a List!
  - If SortedList were a type of List, then a SortedList could be used **anywhere a list can be used**

```
void list_test(List& list) {
  list.insertFirst(10);
  assert(*list.begin() == 10);
}
```

```
List list = { 3, 2, 4, 1, 5 };
list_test(list);
```

```
SortedList sorted = { 3, 2, 4, 1, 5 };
list_test(sorted);
```

surprise: assertion fails!

- The reason SortedList is not a List is because **some of List's functionality does not apply to SortedList**
  - For example, insertFirst() / insertLast() have no meaning in SortedList since it keeps its elements sorted

When Should We Use Inheritance?

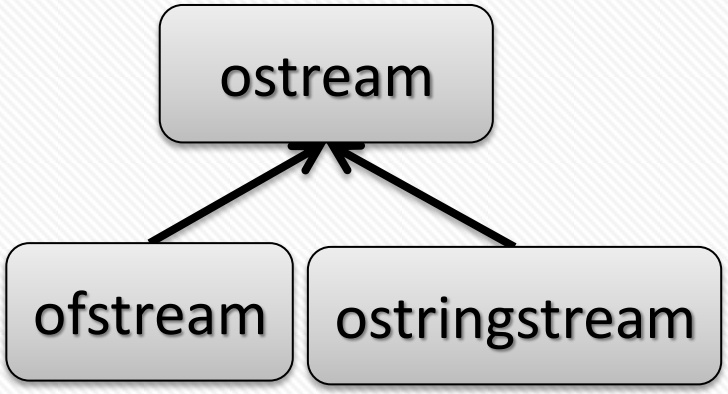
- Inheritance can be **easily misused** to create **confusing** and **complicated** code
- Use inheritance **when you need polymorphism**
  - This means that you should know **what code will benefit** from this design choice
- Otherwise, **prefer composition over inheritance**
  - A new class can be implemented using an existing class as a member variable

Input and Output Streams

- The C++ streams library is a **good example of polymorphism**
  - The output operator << is implemented for the base class std::ostream
  - Any class that derives from std::ostream will be able to use the overloaded operator
  - **New classes of streams** can be added by inheriting std::ostream

```
template<class T>
std::ostream& operator<<(std::ostream& os,
                         const Set<T>& set);
```

```
set<int> s;
// ...
ofstream outfile("output.txt");
outfile << s;
```



ofstream is an ostream, so this code will work

“There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.”

- C.A.R. Hoare, The 1980 ACM Turing Award Lecture