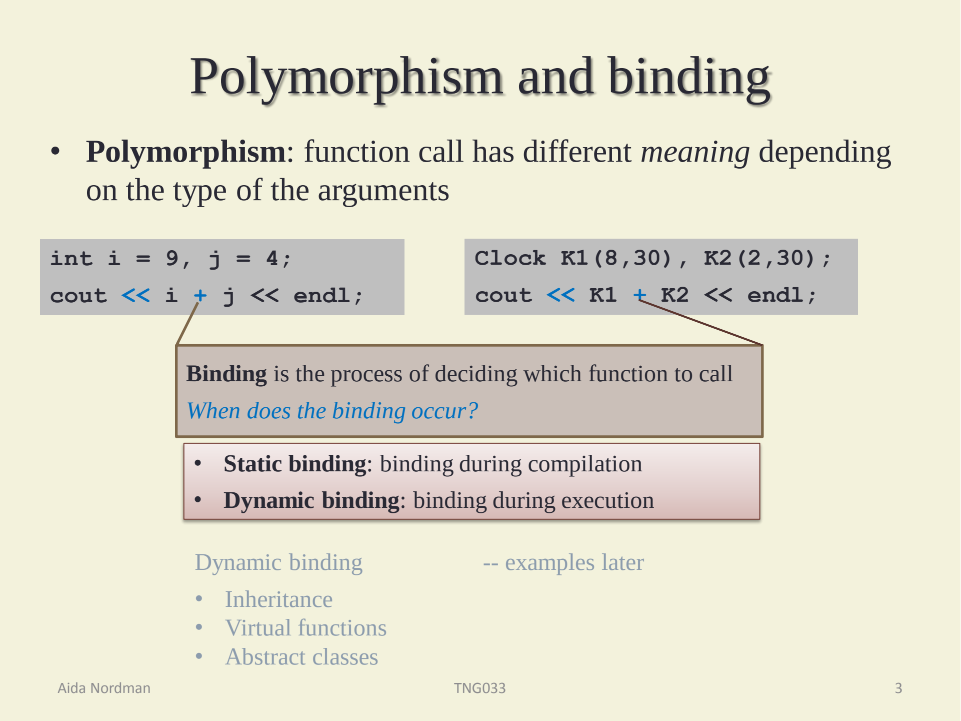
Lecture 11

- **Inheritance** *(arv)* [sec. 9]
  - Polymorphism and dynamic binding [sec. 9.6]
    - Virtual member functions
    - Polymorphic classes
  - Virtual destructors [sec. 9.9]
  - Abstract classes [sec. 9.10]
- Examples
  - Geometric figures hierarchy
  - `Employee`s hierarchy

Info

- **Labs**
  - Attend your lab lab session in the corresponding scheduled room
    - List of lab groups and labs schedule is available from course web site
  - Each session has at most 10 groups
    - MT2a.2 has few groups registered
    - MT2a.1, MT2b.1, and MT2b.2 are **full** classes
  - Week 51 there is an extra redovisning session
  - Require to work outside scheduled lab sessions
- **Lessons**
  - Important to read and attempt exercises in advance
  - More exercises than can be solved in a lesson -- extra exercises included
- **Duggor**
  - Ask to the course's staff if you don't understand why you got a certain remark
  - Passing test examples is not a guarantee of correct code
  - Three "!ok" implies that the dugga is underkänt
  - But, there may other reasons to not approve a dugga



Polymorphism and binding

- **Polymorphism**: function call has different *meaning* depending on the type of the arguments

```
int i = 9, j = 4;
cout << i + j << endl;
```

```
Clock K1(8,30), K2(2,30);
cout << K1 + K2 << endl;
```

Binding is the process of deciding which function to call

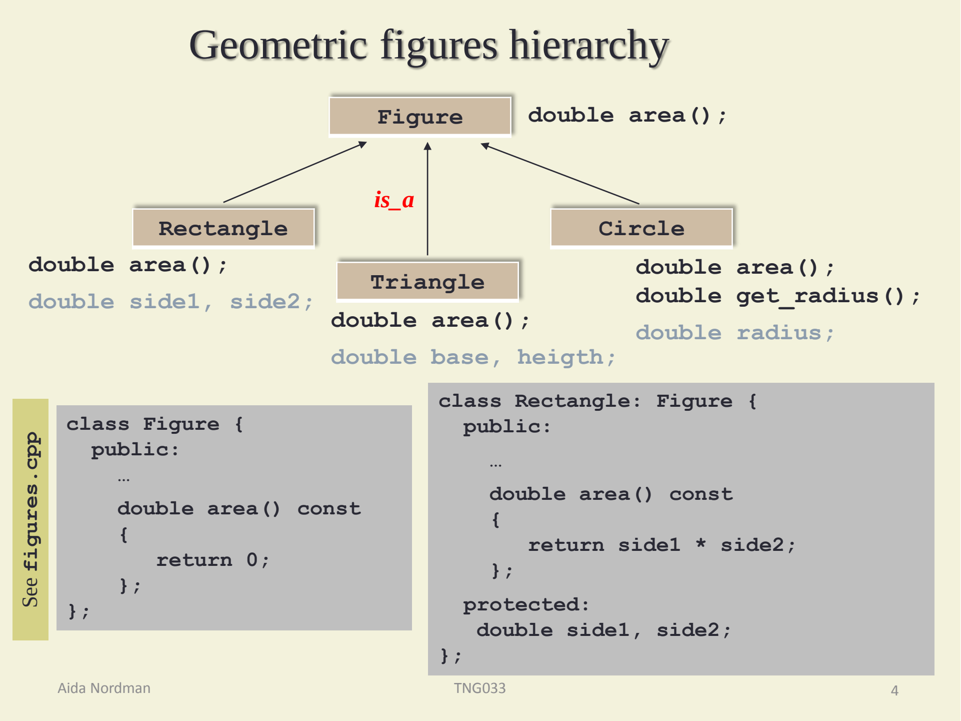
When does the binding occur?

- **Static binding**: binding during compilation
- **Dynamic binding**: binding during execution

Dynamic binding -- examples later

- Inheritance
- Virtual functions
- Abstract classes

Aida Nordman TNG033 3



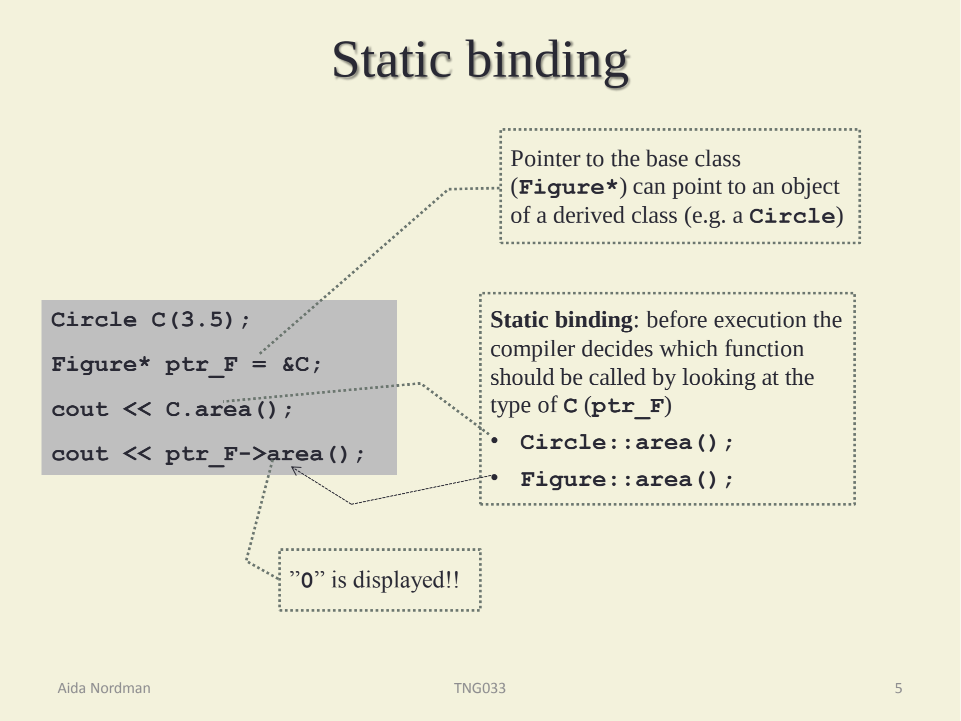
Geometric figures hierarchy

See **figures.cpp**

```
class Figure {
  public:
    …
    double area() const
    {
       return 0;
    };
};
```

```
class Rectangle: Figure {
  public:
    …
    double area() const
    {
       return side1 * side2;
    };
  protected:
   double side1, side2;
};
```

Aida Nordman TNG033 4



Static binding

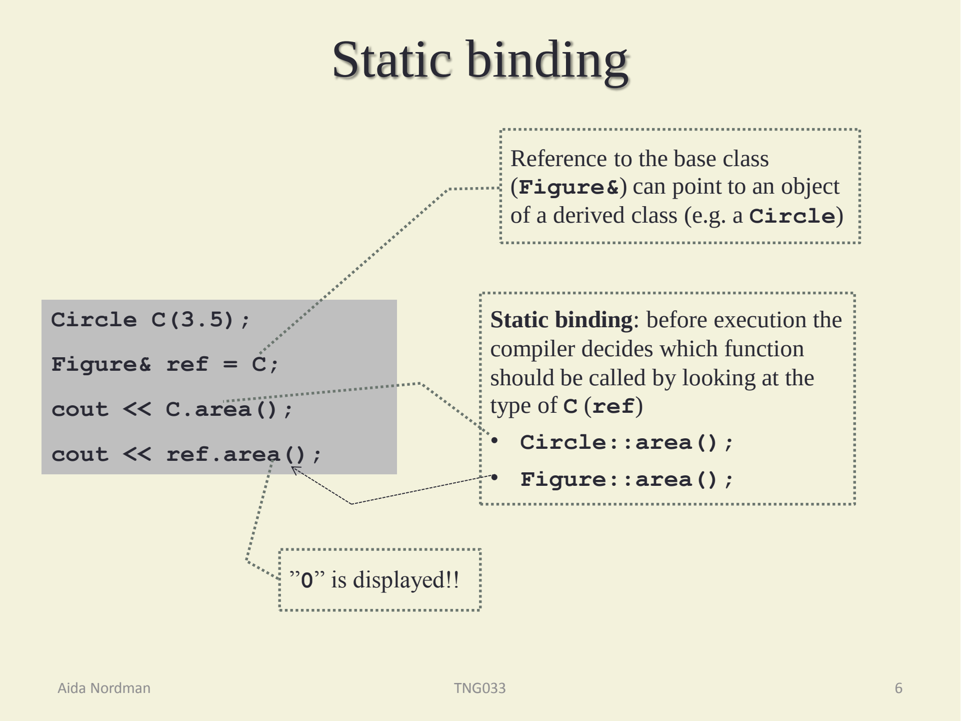
Pointer to the base class (**`Figure*`**) can point to an object of a derived class (e.g. a **`Circle`**)

```
Circle C(3.5);
Figure* ptr_F = &C;
cout << C.area();
cout << ptr_F->area();
```

Static binding: before execution the compiler decides which function should be called by looking at the type of **`C`** (**`ptr_F`**)

- **`Circle::area();`**
- **`Figure::area();`**

”`0`” is displayed!!



Static binding

Reference to the base class (**`Figure&`**) can point to an object of a derived class (e.g. a **`Circle`**)

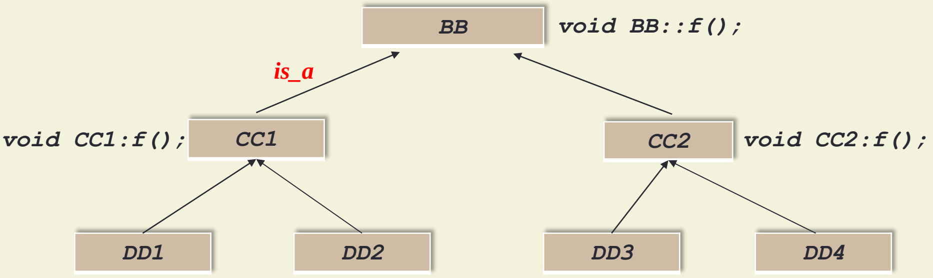
```
Circle C(3.5);
Figure& ref = C;
cout << C.area();
cout << ref.area();
```

Static binding: before execution the compiler decides which function should be called by looking at the type of **`C`** (**`ref`**)

- **`Circle::area();`**
- **`Figure::area();`**

”`0`” is displayed!!

Static binding



```
DD2 d2;
d2.f();
BB *ptr_B = &d2;
ptr_B->f();
```

Compiler searches for `f()` starting from the class of `d2` upwards (`CC1::f()` called)

Compiler searches for `f()` starting from the declared class of `*ptr_B` upwards (`BB::f()` called)

Is static binding enough?

```
const int MAX = 100;

//data base of Figures
Figure* DB[MAX];

int howMany = 4;

DB[0] = new Circle(3.5);
DB[1] = new Rectangle(4,2);
DB[2] = new Triangle(8,2.4);
DB[3] = new Circle(6.6);

for(int i = 0; i < howMany; i++)
        cout << DB[i]->area() << endl ;
```

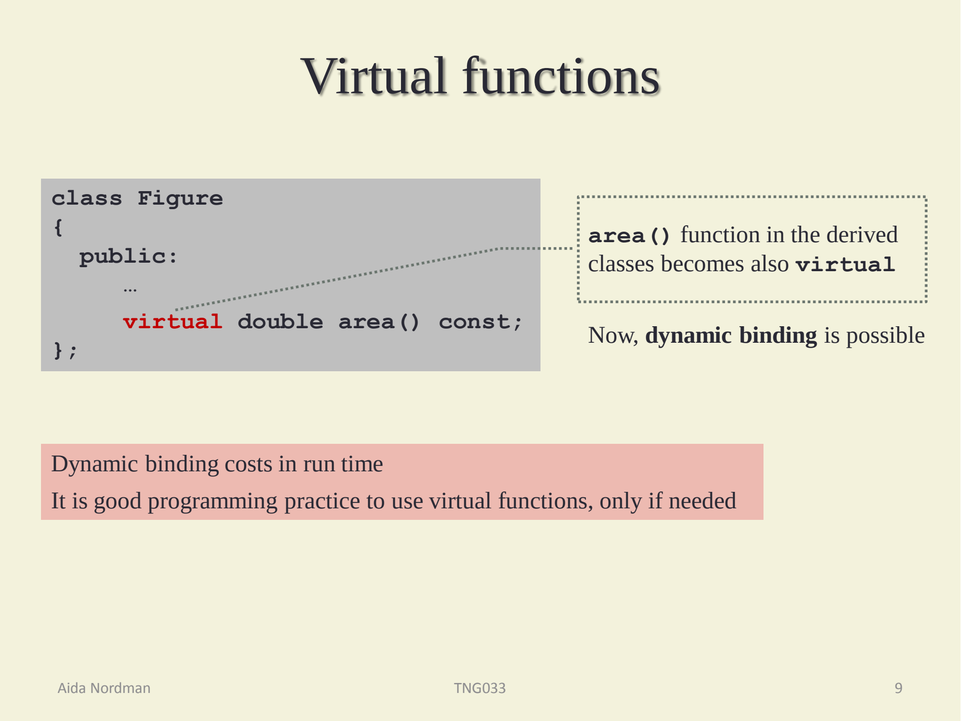
can be user given options

”0” is displayed ☹

What would we like?

It’s desirable that **`area()`** function called would depend on the object pointed by **`DB[i]`** not on the type of the pointer (**`Figure*`**)

Solution: virtual functions and dynamic binding



Virtual functions

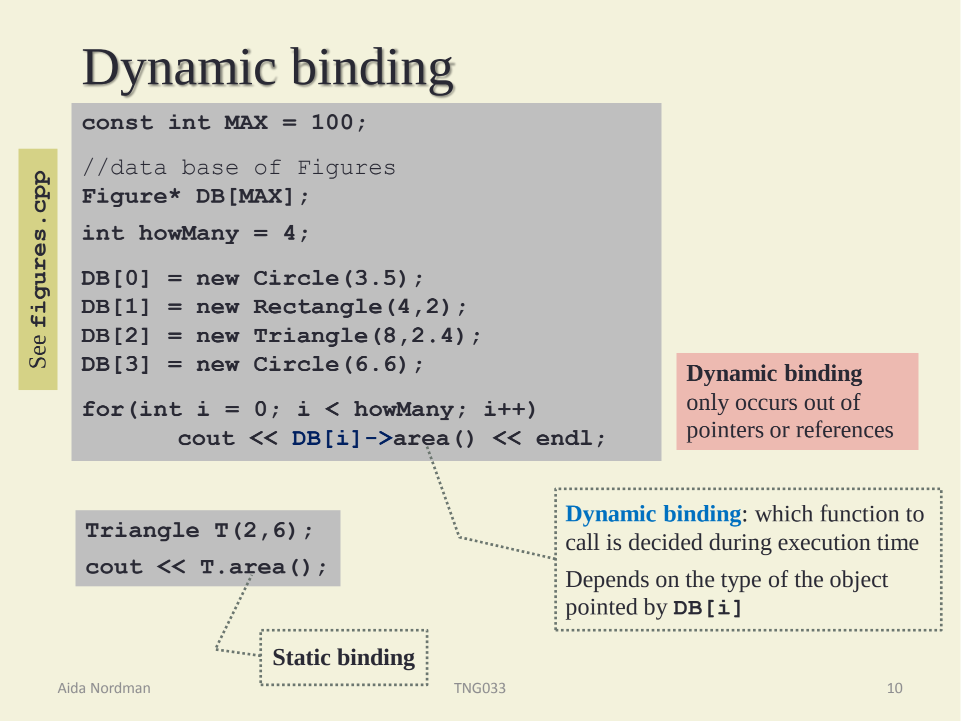
```
class Figure
{
  public:
     …
     virtual double area() const;
};
```

area() function in the derived classes becomes also **virtual**

Now, **dynamic binding** is possible

Dynamic binding costs in run time

It is good programming practice to use virtual functions, only if needed



Dynamic binding

See **figures.cpp**

```
const int MAX = 100;

//data base of Figures
Figure* DB[MAX];

int howMany = 4;

DB[0] = new Circle(3.5);
DB[1] = new Rectangle(4,2);
DB[2] = new Triangle(8,2.4);
DB[3] = new Circle(6.6);

for(int i = 0; i < howMany; i++)
       cout << DB[i]->area() << endl;
```

Dynamic binding only occurs out of pointers or references

```
Triangle T(2,6);
cout << T.area();
```

Dynamic binding: which function to call is decided during execution time

Depends on the type of the object pointed by **DB[i]**

Static binding

Dynamic Binding: summary

```
class B
{
  public:
    …
    virtual return_type f(parameter);
    …
};
```

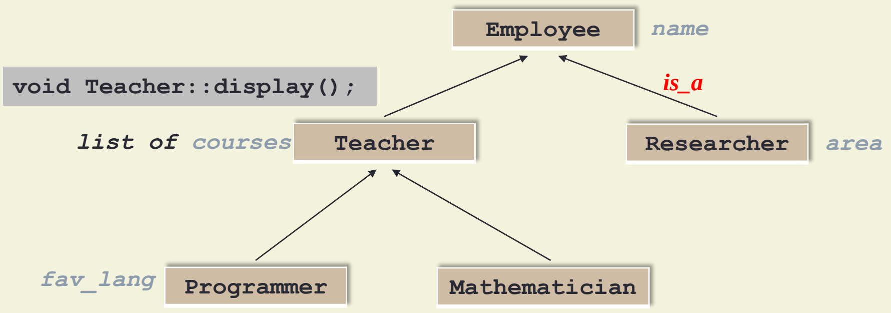
Every derived class D_i of **B** redefines function **f**, with same parameters

```
D1 d1;
B *ptr = &d1;
ptr->f(…);
D2 d2;
B &ref = d2;
ref.f(…);
```

Dynamic binding

Employee hierarchy (*Fö 10*)

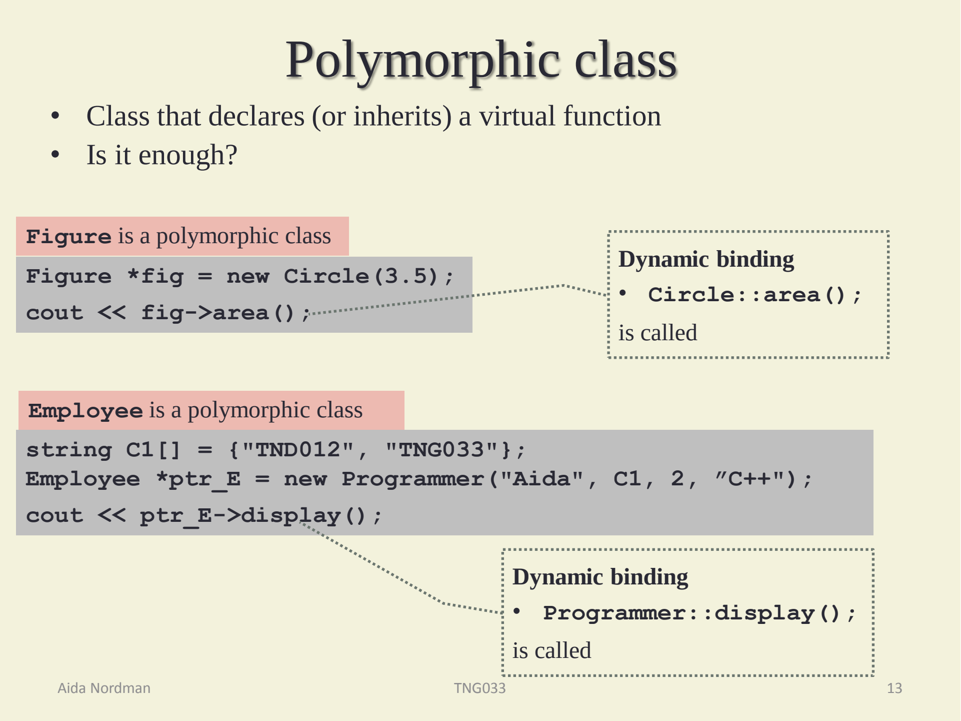
```
void Employee::set_name(string s);
virtual void Employee::display();
```



```
void Teacher::display();
```

```
void Programmer::display();
```

See **arv2.cpp**



Polymorphic class

- Class that declares (or inherits) a virtual function
- Is it enough?

`Figure` is a polymorphic class

```
Figure *fig = new Circle(3.5);
cout << fig->area();
```

Dynamic binding
- **`Circle::area();`**

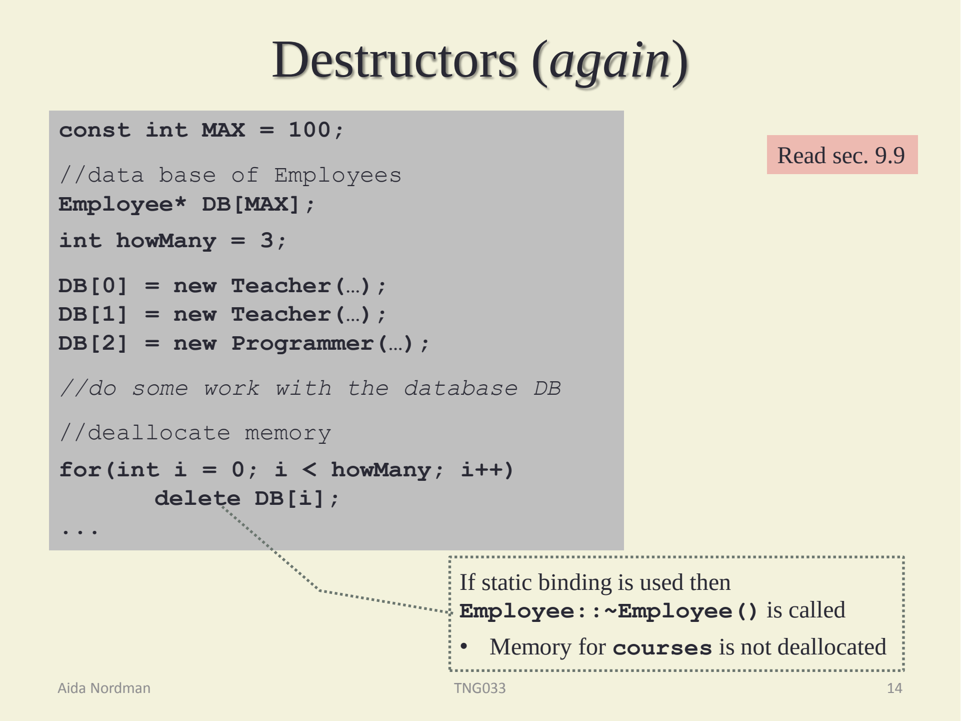
is called

`Employee` is a polymorphic class

```
string C1[] = {"TND012", "TNG033"};
Employee *ptr_E = new Programmer("Aida", C1, 2, ”C++");
cout << ptr_E->display();
```

Dynamic binding
- **`Programmer::display();`**

is called

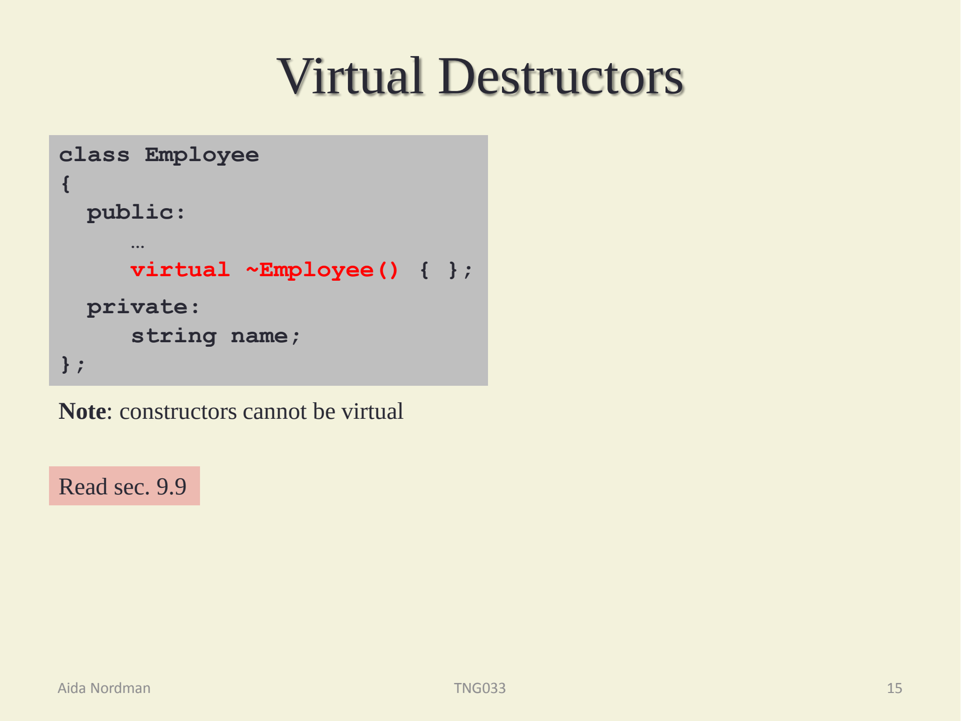


Destructors (*again*)

```
const int MAX = 100;
//data base of Employees
Employee* DB[MAX];
int howMany = 3;
DB[0] = new Teacher(…);
DB[1] = new Teacher(…);
DB[2] = new Programmer(…);
//do some work with the database DB
//deallocate memory
for(int i = 0; i < howMany; i++)
       delete DB[i];
...
```

Read sec. 9.9

If static binding is used then **`Employee::~Employee()`** is called
- Memory for **`courses`** is not deallocated

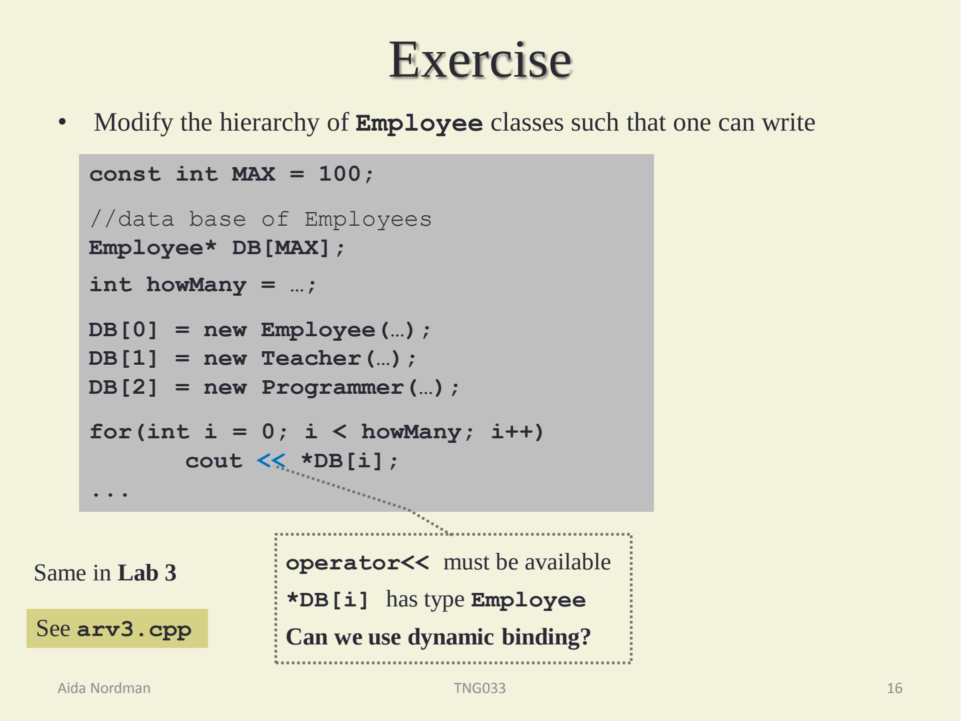


Virtual Destructors

```
class Employee
{
  public:
     …
     virtual ~Employee() { };
  private:
     string name;
};
```

Note: constructors cannot be virtual

Read sec. 9.9



Exercise

- Modify the hierarchy of **`Employee`** classes such that one can write

```
const int MAX = 100;

//data base of Employees
Employee* DB[MAX];

int howMany = …;

DB[0] = new Employee(…);
DB[1] = new Teacher(…);
DB[2] = new Programmer(…);

for(int i = 0; i < howMany; i++)
       cout << *DB[i];
...
```

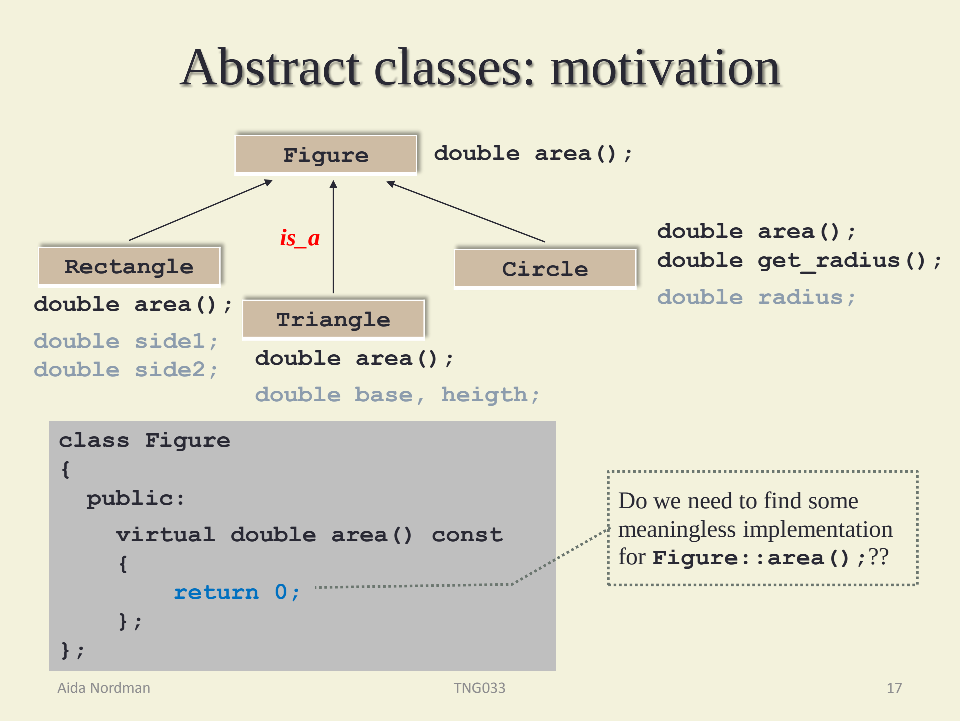
`operator<<` must be available

`*DB[i]` has type **`Employee`**

Can we use dynamic binding?

Same in **Lab 3**

See **`arv3.cpp`**



Abstract classes: motivation

Figure `double area();`

is_a

Rectangle

```
double area();
double side1;
double side2;
```

Triangle

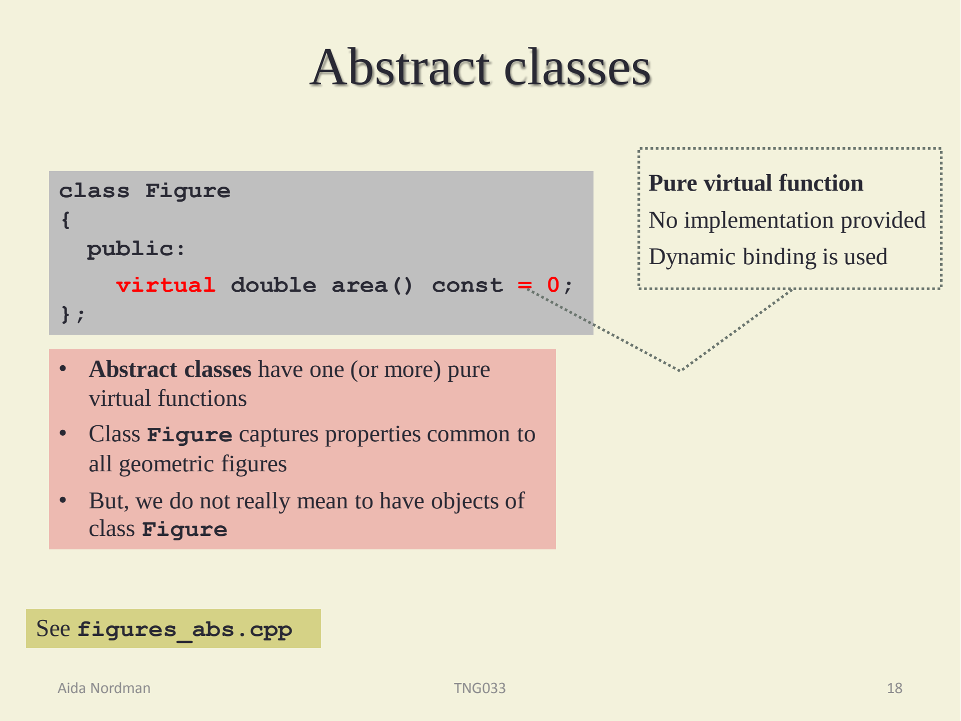
```
double area();
double base, heigth;
```

Circle

```
double area();
double get_radius();
double radius;
```

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

Do we need to find some meaningless implementation for **`Figure::area();`**??



Abstract classes

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

Pure virtual function

No implementation provided

Dynamic binding is used

- **Abstract classes** have one (or more) pure virtual functions
- Class **`Figure`** captures properties common to all geometric figures
- But, we do not really mean to have objects of class **`Figure`**

See **`figures_abs.cpp`**

Abstract classes

- Have one (or more) pure virtual functions
  - Derived classes must provide the implementation
- Forbidden to declare objects as instances of an abstract class
- Pattern for how the subclasses should look like

```
const int MAX = 100;

//data base of Figures
Figure* DB[MAX];

int howMany = 4;

~~DB[0] = new Figure~~;
DB[1] = new Rectangle(4,2);
DB[2] = new Triangle(8,2.4);
DB[3] = new Circle(6.6);

for(int i = 0; i < howMany; i++)
        cout << DB[i]->area() << endl ;
```

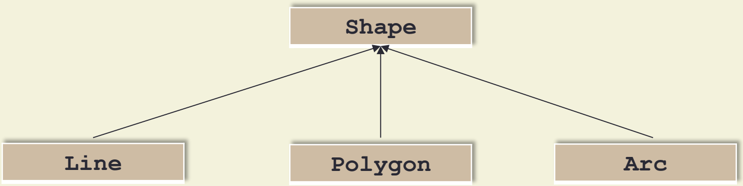
Dynamic binding

Read sec. 9.10

Abstract classes: example

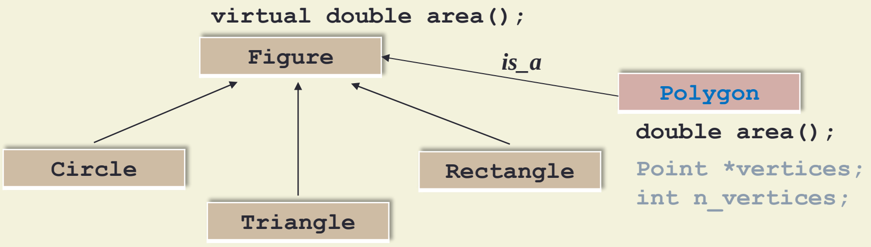
```
class Shape
{
  public:
    virtual ~shape() { };
    virtual void draw() = 0;
    virtual void rotate(double angle) = 0;
    …
};
```

Class to represent an arbitrary shape



Is it enough for a polymorphic class to have a virtual function?

Dynamic binding



`virtual double area();`

Figure

is_a

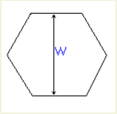
Polygon

```
double area();
Point *vertices;
int n_vertices;
```

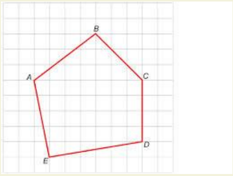
Circle

Triangle

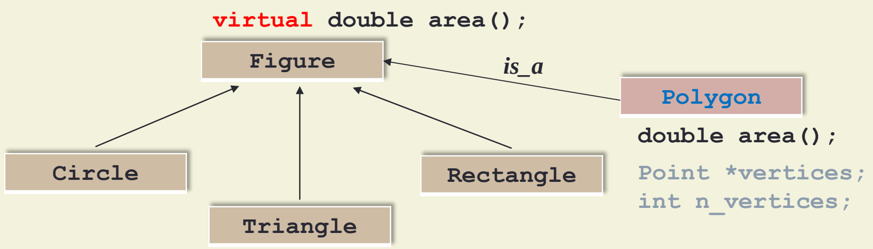
Rectangle



Figures hierarchy is well designed if it allows programmers to add new sub-classes (e.g. **Polygonon**) without having to change any of the existing classes of the hierarchy



Dynamic binding



virtual double area();

Figure

is_a

Polygon

```
double area();
Point *vertices;
int n_vertices;
```

Circle

Triangle

Rectangle

```
class Figure
{
  public:
    Figure() = default;
    //virtual ~Figure() { };
    virtual double area() const = 0;
};
```

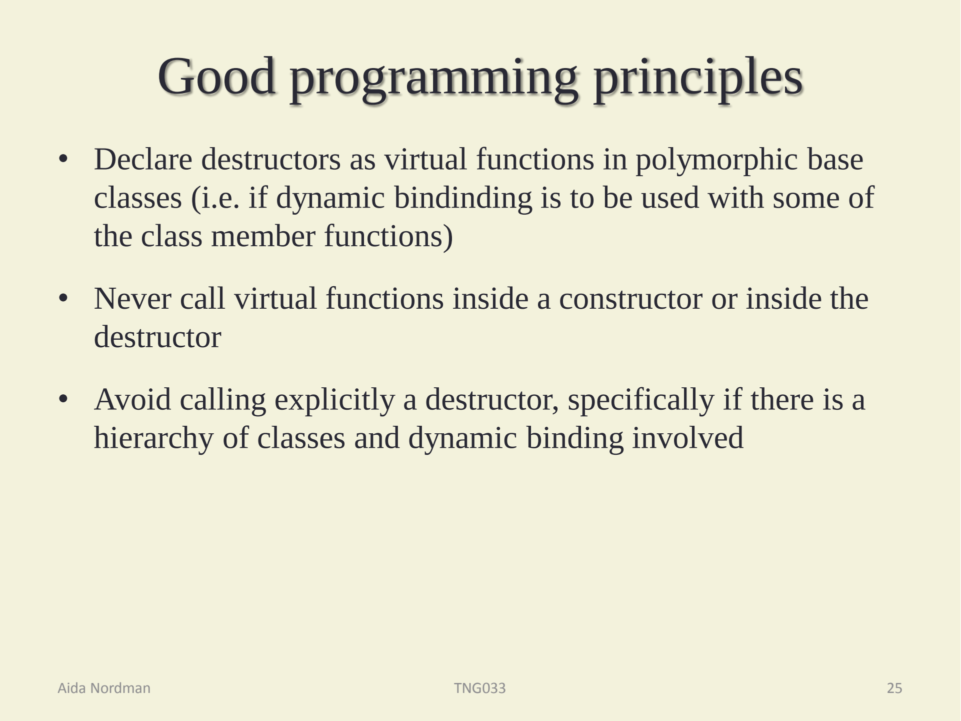
Figure cannot be used as base class of any class that uses dynamic memory allocation for its data members (e.g. **Polygonon**)

```
const int MAX = 100;
Figures* scene[MAX];
int howMany = 3;
scene[0] = new Rectangle(4,2);
scene[1] = new Triangle(8,2.4);
scene[2] = new Polygonon(…);
//do some work with the scene
//deallocate memory
for(int i = 0; i < howMany; i++)
        delete scene[i];
...
```

Add a virtual destructor to polymorphic bases classes like **Figure**

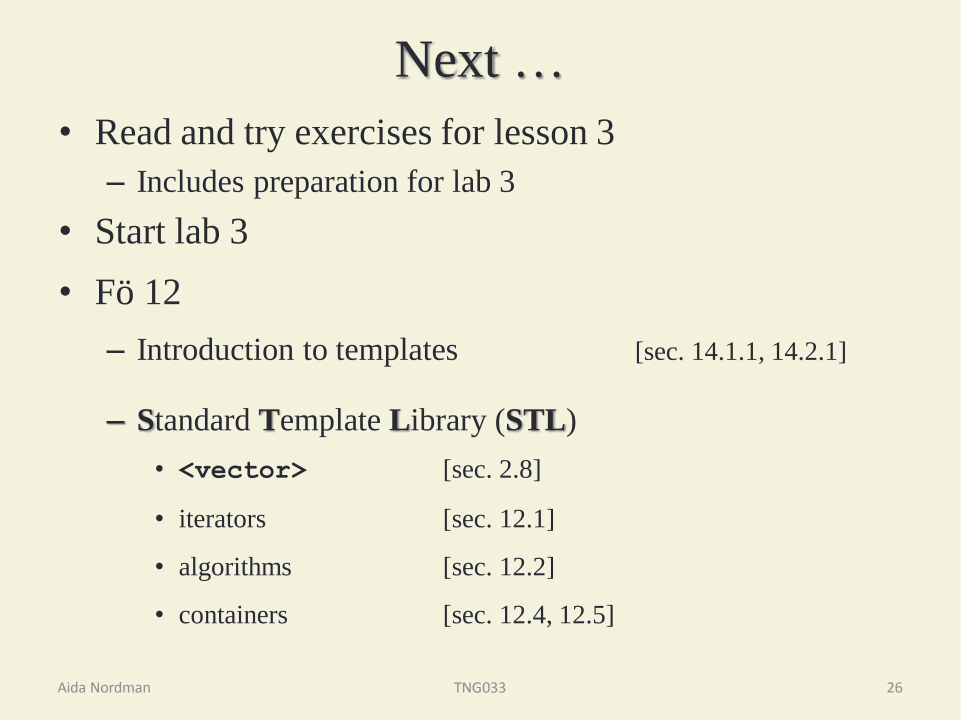
If static binding is used then there is a memory leak

Figure::~Figure() is called



Good programming principles

- Declare destructors as virtual functions in polymorphic base classes (i.e. if dynamic bindinding is to be used with some of the class member functions)
- Never call virtual functions inside a constructor or inside the destructor
- Avoid calling explicitly a destructor, specifically if there is a hierarchy of classes and dynamic binding involved



Next …

- Read and try exercises for lesson 3
  - Includes preparation for lab 3
- Start lab 3
- Fö 12
  - Introduction to templates [sec. 14.1.1, 14.2.1]
  - **S**tandard **T**emplate **L**ibrary (**STL**)

• **`<vector>`**	[sec. 2.8]
• iterators	[sec. 12.1]
• algorithms	[sec. 12.2]
• containers	[sec. 12.4, 12.5]