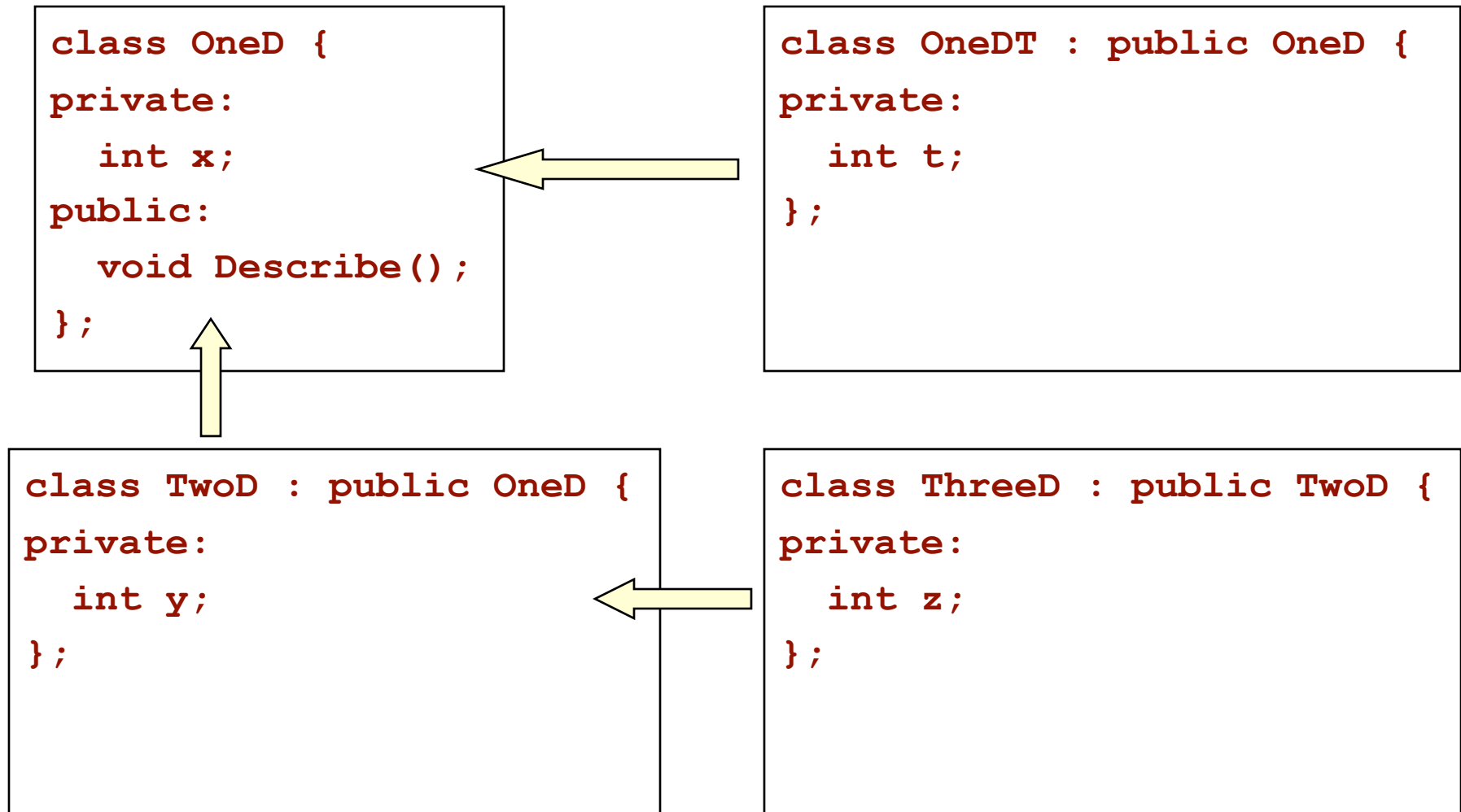


# Introduction to C, C++, and Unix/Linux

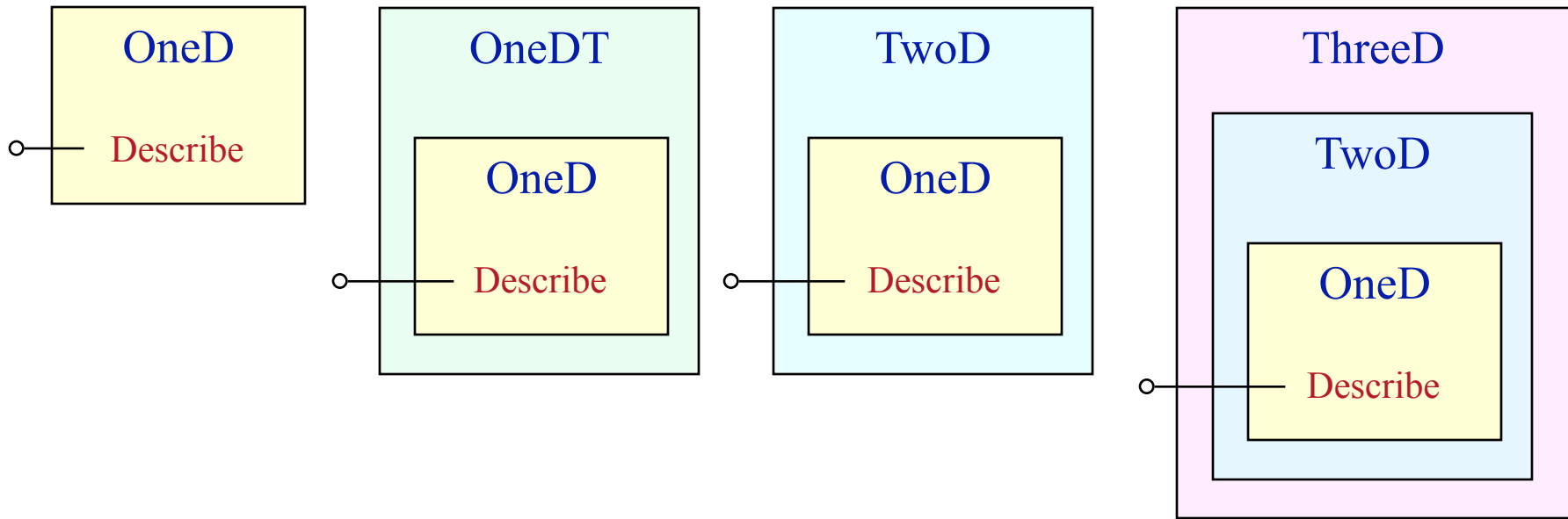
CS 60

Today

- Virtual class functions, abstract classes
- Reading [KR] Chapters 1-7
- Read [So] chapters 1, 3, 4 (Boolean), 9, 13, 14 & 18.



(Assume there are also functions to get and set the variable values)



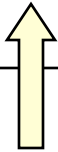
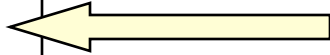
```
OneD pt1d;  
OneDT pt1dt;  
TwoD pt2d;  
ThreeD pt3d;
```

```
pt1d.Describe();    "I'm a OneD object"  
pt1dt.Describe();  "I'm a OneD object"  
pt2d.Describe();    "I'm a OneD object"  
pt3d.Describe();    "I'm a OneD object"
```

Calls the OneD function **Describe**

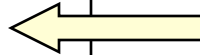
```
class OneD {  
private:  
    int x;  
public:  
    void Describe();  
};
```

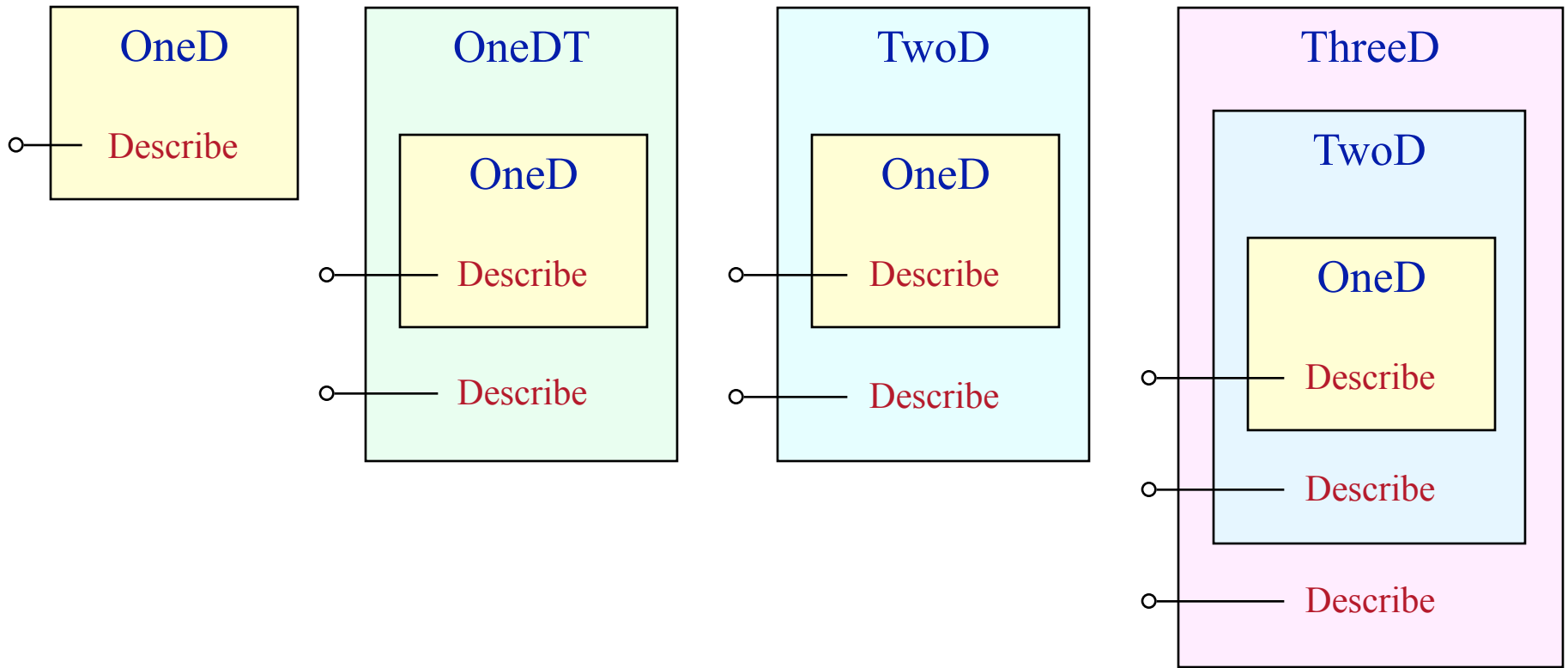
```
class OneDT : public OneD {  
private:  
    int t;  
public:  
    void Describe();  
};
```



```
class TwoD : public OneD {  
private:  
    int y;  
public:  
    void Describe();  
};
```

```
class ThreeD : public TwoD {  
private:  
    int z;  
public:  
    void Describe();  
};
```





```
OneD one_d;  
OneDT one_dt;  
TwoD two_d;  
ThreeD three_d;
```

```
one_d.Describe();    "I'm a OneD object"  
one_dt.Describe();  "I'm a OneDT object"  
two_d.Describe();   "I'm a TwoD object"  
three_d.Describe(); "I'm a ThreeD object"
```

Calls the derived class function **Describe**

If the derived class doesn't have a **Describe** function,  
it looks for one in the class that it was derived from

## Accessing via the base class #1

- Let's say we create many (thousands?) of points of all types (1D, 1DT, 2D, 3D)
- It's efficient to gather them in the array

```
OneD *point_array[5000];  
point_array[0] = new TwoD();  
... // create the rest  
for (i=0; i<5000; i++)  
    point_array[i]->Render();
```

← Render each point  
type correctly!

Avoids big, ungainly  
“switch” statements

## Accessing via the base class #2

- We'd like to define one function that works on the base class and all derived classes
  - Even those we haven't created yet
  - Rather than creating one function for every possible type

```
void Transform(OneD& pt, double scale,  
              double angle)  
{  
    pt.Resize(scale);  
    pt.Rotate(angle);  
}
```

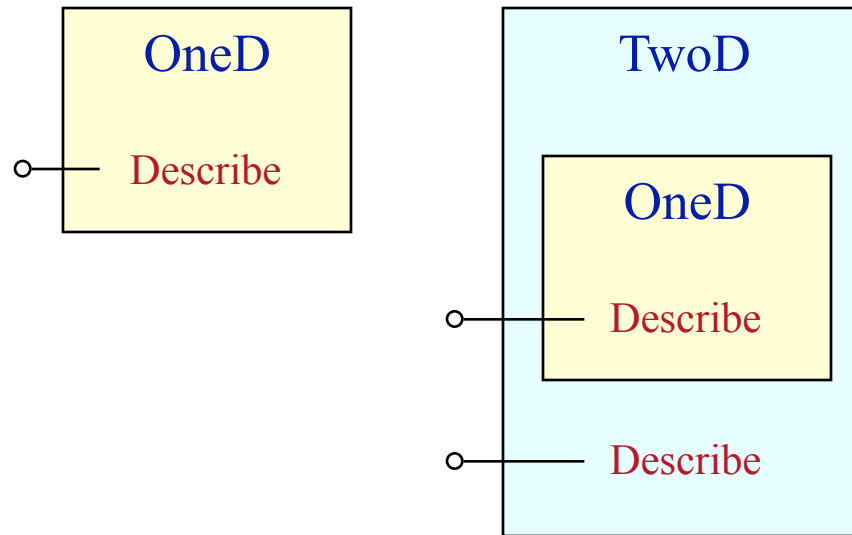
Global function  
(not class function)

```
OneD one_d;  
TwoD two_d;
```

```
one_d.Describe(); "OneD"  
two_d.Describe(); "TwoD"
```

```
OneD *p = &two_d;  
OneD& rp = two_d;
```

```
p->Describe(); "OneD"  
rp.Describe(); "OneD"
```



```
void Report(OneD& obj)
{
    // Misc. code

    obj.Describe();
}
```

This function works for all objects derived from OneD

```
Report(one_d);           "I'm a OneD object"
Report(one_dt);          "I'm a OneD object"
Report(two_d);           "I'm a OneD object"
Report(three_d);         "I'm a OneD object"
```

But this isn't the behavior we'd like!

How can we call the class-specific **Describe** functions inside of **Report**?  
And through **OneD** pointers?

# Virtual class functions

- When a base class function is defined as **virtual**, C++ will look for the derived class function first – *even if the object is a base class object*

```
class OneD {  
private:  
    int x;  
public:  
    virtual void Describe();  
};
```

```
OneD one_d;  
TwoD two_d;
```

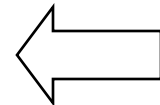
```
one_d.Describe(); "OneD"  
two_d.Describe(); "TwoD"
```

```
void Report(OneD& obj)  
{  
    obj.Describe();  
}
```

```
OneD *p = &two_d;  
OneD& rp = two_d;
```

```
p->Describe(); "TwoD"  
rp.Describe(); "TwoD"
```

```
Report(one_d); "OneD"  
Report(two_d); "TwoD"  
Report(*p); "TwoD"  
Report(rp); "TwoD"
```



```

class Base {
public:
    void func1() { cout <<
        "func1: Base\n"; }
    void func2() { cout <<
        "func2: Base\n"; }
};

class Derived : public Base {
};

void Test(Base& b)
{
    b.func1();
    b.func2();
}

```

```

Base b;
Derived d;

b.func1(); "func1: Base"
b.func2(); "func2: Base"
d.func1(); "func1: Base"
d.func2(); "func2: Base"

Test(b); "func1: Base"
"func2: Base"

Test(d); "func1: Base"
"func2: Base"

```

```

class Base {
public:
    void func1() { cout <<
        "func1: Base\n"; }
    void func2() { cout <<
        "func2: Base\n"; }
};
class Derived : public Base {
public:
    void func1() { cout <<
        "func1: Derived\n"; }
    void func2() { cout <<
        "func2: Derived\n"; }
};
void Test(Base& b)
{
    b.func1();
    b.func2();
}

```

```

Base b;
Derived d;

b.func1();    "func1: Base"
b.func2();    "func2: Base"
d.func1();    "func1: Derived"
d.func2();    "func2: Derived"

Test(b);      "func1: Base"
              "func2: Base"

Test(d);      "func1: Base"
              "func2: Base"

```

```

class Base {
public:
    void func1() { cout <<
        "func1: Base\n"; }
    virtual void func2() { cout
        << "func2: Base\n"; }
};
class Derived : public Base {
public:
    void func1() { cout <<
        "func1: Derived\n"; }
    void func2() { cout <<
        "func2: Derived\n"; }
};
void Test(Base& b)
{
    b.func1();
    b.func2();
}

Base b;
Derived d;

b.func1(); "func1: Base"
b.func2(); "func2: Base"
d.func1(); "func1: Derived"
d.func2(); "func2: Derived"

Test(b); "func1: Base"
"func2: Base"

Test(d); "func1: Base"
"func2: Derived"

```

```

class Base {
public:
    void func1() { cout <<
        "func1: Base\n"; }
    virtual void func2() { cout
        << "func2: Base\n"; }
};
class Derived : public Base {
public:
    void func1() { cout <<
        "func1: Derived\n"; }
    void func2() { cout <<
        "func2: Derived\n"; }
};
void Test(Base b)
{
    b.func1();
    b.func2();
}

```

```

Base b;
Derived d;

b.func1(); "func1: Base"
b.func2(); "func2: Base"
d.func1(); "func1: Derived"
d.func2(); "func2: Derived"

Test(b); "func1: Base"
"func2: Base"

Test(d); "func1: Base"
"func2: Base"

```

Why?

# Virtual constructors and destructor

- Class constructors cannot be **virtual**
  - But class destructors can be
- What if the base class allocates memory from the heap and the derived class allocates memory from the heap, and then:

```
Base *pb = new Derived;
```

```
...
```

```
delete pb;
```

```
pb = NULL;
```

← This calls the Base destructor, but not the Derived destructor

Solution: Make the base class destructor **virtual**

First **~Derived()** is called, then **~Base()**

# Abstract classes

- You can specify a virtual class that *must* be overridden in a derived class

```
virtual int GetPixelWidth() = 0;
```

- This is called a *pure virtual function*, and makes the class an *abstract class*
- Abstract classes cannot be instantiated
  - They are essentially templates for other classes to inherit

```
Base b; // ERROR! Abstract class
```

```
class Image {
protected:
    int rows;
    int cols;
    PixelType type;
    string comment;
    void *data;
public:
    virtual int GetPixelWidth() = 0;
    virtual int GetNRows() = 0;
    virtual int GetNCols() = 0;
};
```

```
class Gray16bitImage : public Image {
    ...
};
class Color24bitImage : public Image {
    ...
};
```

```
Image im1; // ERROR! Abstract class
Gray16bitImage im2; // Okay
Color24bitImage im3; // Okay
```

# Virtual

- Why have a virtual function?
- Why have a pure virtual function?
- Let's say we wrote an abstract Image class, then derived from it several classes:
  - Gray8bitImage, Float32bitImage, Color24bitImage, ...
- How would we declare – and write – ReadImage()?

```
Gray8bitImage im;  
im = ReadImage();
```

```
Gray8bitImage im;  
im.ReadImage();
```

```
class Image {
public:
    virtual void Print() = 0;
    virtual void ClearImage() = 0;
    static Image* ReadImage(); ←
};

class Gray8Image : public Image {
public:
    void Print();
    void ClearImage();
};

Image *im = Image::ReadImage();
im->SetPixel(100, 100, 0);
```

```
class Image {
public:
    virtual void Print() = 0;
    virtual void ClearImage() = 0;
    static Image& ReadImage();
};

class Gray8Image : public Image {
public:
    void Print();
    void ClearImage();
};

Image& im = Image::ReadImage();
im.SetPixel(100, 100, 0);
```