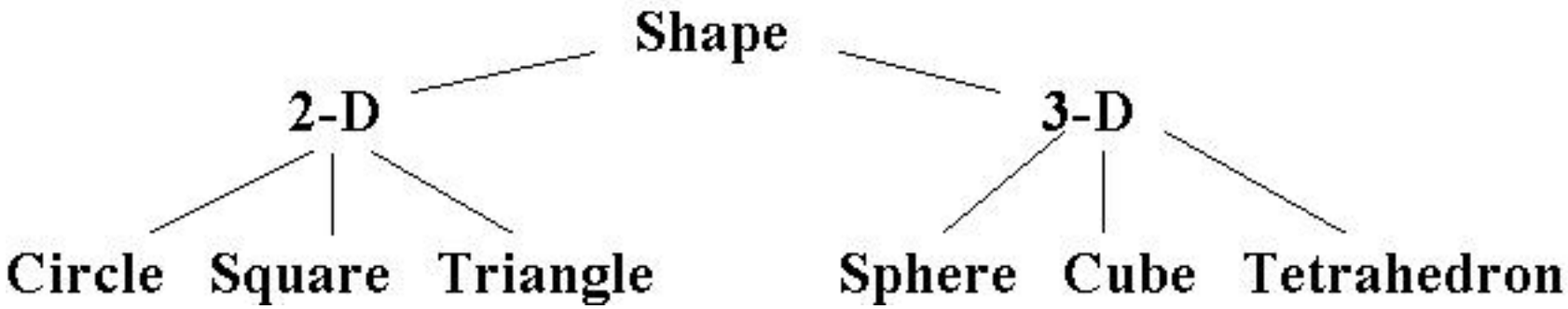
Inheritance and Overloading

Week 11

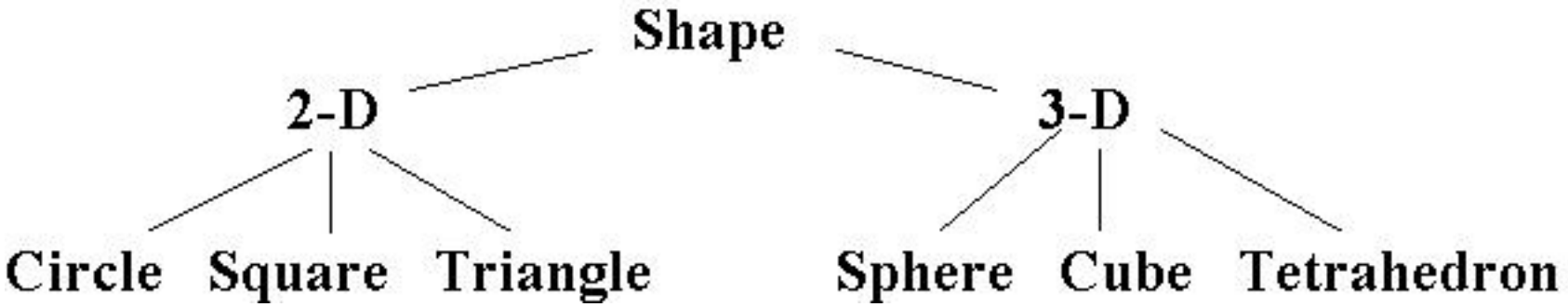
Inheritance

- Objects are often defined in terms of hierarchical classes with a base class and one or more levels of classes that inherit from the classes that are above it in the hierarchy.

- For instance, graphics objects might be defined as follows:



Inheritance (continued)



- This hierarchy could, of course, be continued for more levels.
- Each level inherits the attributes of the above level. Shape is the base class. 2-D and 3-D are derived from Shape and Circle, Square, and Triangle are derived from 2-D. Similarly, Sphere, Cube, and Tetrahedron are derived from 3-D.

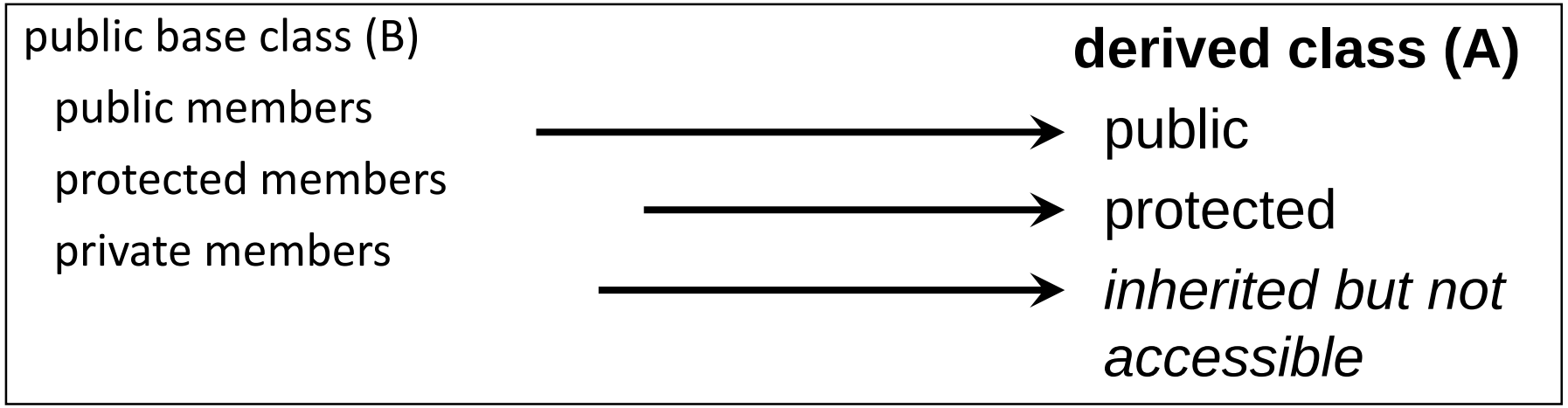
Inheritance (continued)

```
class A : base class access specifier B
{
  member access specifier(s):
  ...
        member data and member function(s);
  ...
}
```

Valid access specifiers include public, private, and protected

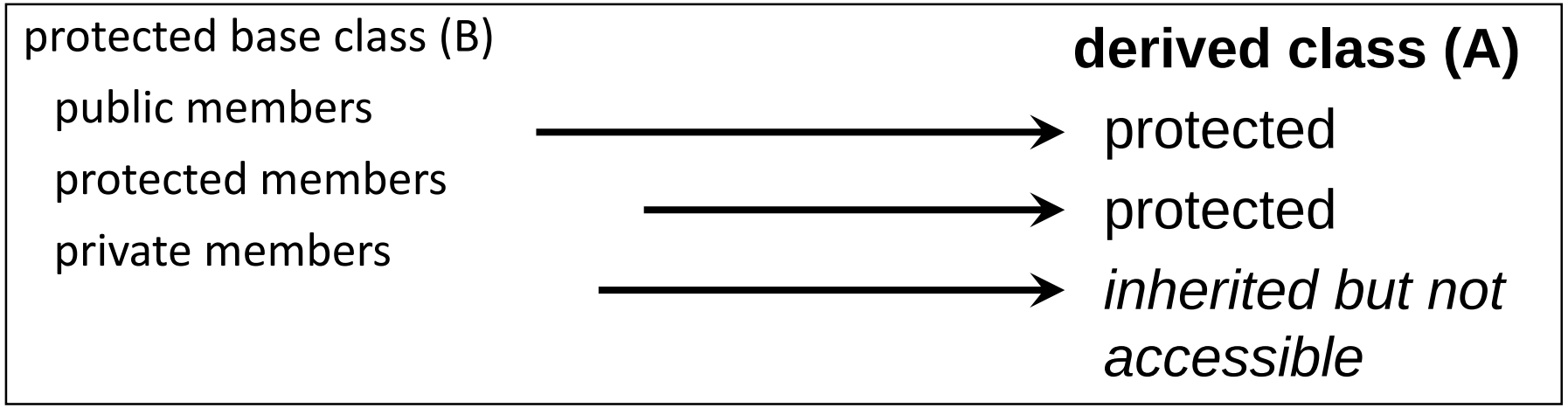
Public Inheritance

```
class A : public B
{       // Class A now inherits the members of Class B
        // with no change in the "access specifier" for
}       // the inherited members
```



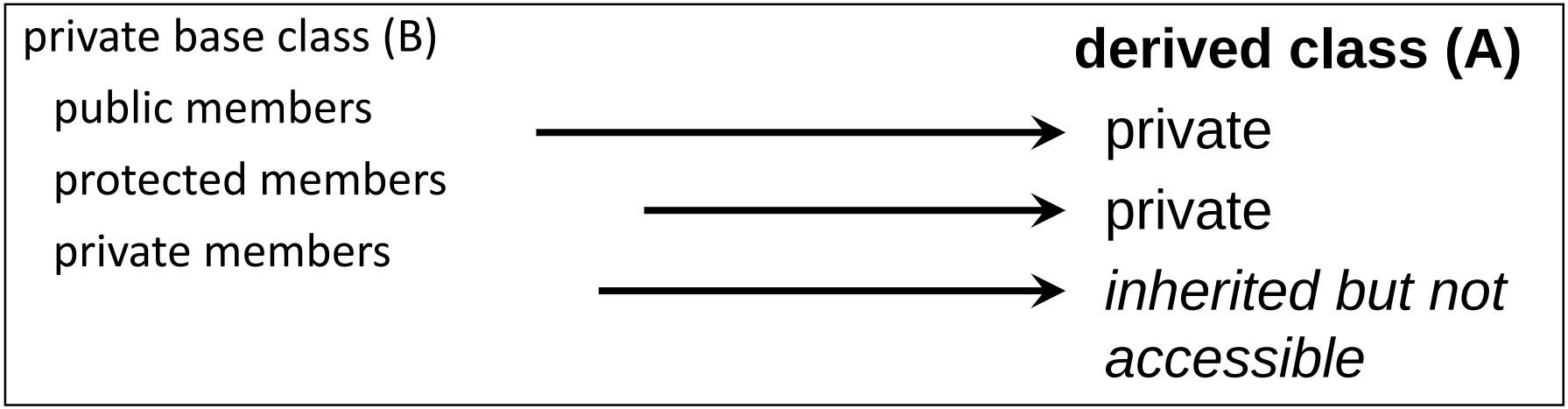
Protected Inheritance

```
class A : protected B
{       // Class A now inherits the members of Class B
        // with public members "promoted" to protected
}       // but no other changes to the inherited members
```



Private Inheritance

```
class A : private B
{       // Class A now inherits the members of Class B
        // with public and protected members
}       // "promoted" to private
```



Inheritance (continued)

```
class Shape
 {
   public:
          int GetColor ( ) ;
   protected:              // so derived classes can access it
          int color;
};
class Two_D : public Shape
{
   // put members specific to 2D shapes here
};
class Three_D : public Shape
{
   // put members specific to 3D shapes here
};
```

Inheritance (continued)

```
class Square : public Two_D
{
  public:
        float getArea ( ) ;
  protected:
        float edge_length;
} ;
class Cube : public Three_D
{
  public:
        float getVolume ( ) ;
  protected:
        float edge_length;
} ;
```

Inheritance (continued)

```
int main ( )
{
  Square mySquare;
  Cube myCube;

  mySquare.getColor ( );   // Square inherits getColor()
  mySquare.getArea ( );
  myCube.getColor ( );     // Cube inherits getColor()
  myCube.getVolume ( );
}
```

Function Overloading

- C++ supports writing more than one function with the same name but different argument lists. This could include:
  - different data types
  - different number of arguments

- The advantage is that the same apparent function can be called to perform similar but different tasks. The following will show an example of this.

Function Overloading

```
void swap (int *a, int *b) ;
void swap (float *c, float *d) ;
void swap (char *p, char *q) ;
int main ( )
{
  int a = 4, b = 6 ;
  float c = 16.7, d = -7.89 ;
  char p = 'M' , q = 'n' ;
  swap (&a, &b) ;
  swap (&c, &d) ;
  swap (&p, &q) ;
 }
```

Function Overloading

```
void swap (int *a, int *b)
{ int temp;  temp = *a;  *a = *b;  *b = temp; }

void swap (float *c, float *d)
{ float temp;  temp = *c;  *c = *d;  *d = temp; }

void swap (char *p, char *q)
{ char temp;  temp = *p;  *p = *q;  *q = temp; }
```

Function Templates

- We have discussed overloaded functions as a way to perform similar operations on data of different types. The swap functions were an example.

- We wrote three functions with the same name but different data types to perform the swap operations. Then we could call swap (&a, &b), for example, and C++ would select which function to use by matching the data type of a and b to one of the functions.

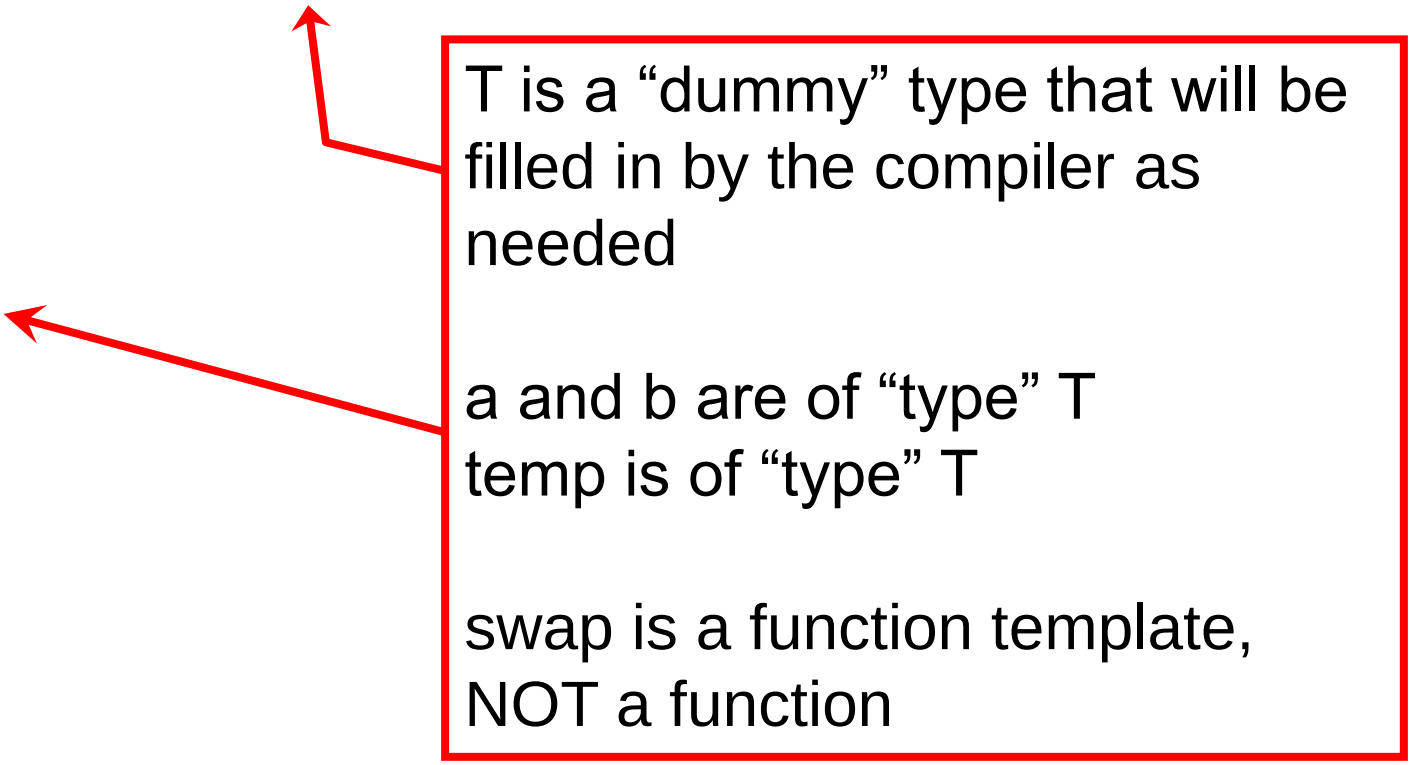
Function Templates

- Another way to perform this task would be to create a *function template* definition.

- With a *function template* defined, when we call swap (&a, &b), C++ will generate the object code functions for us. The program on the following slides is an example.

Function Templates

```
template <typename T> void swap (T *a, T *b)
{
  T temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```



Function Templates

```
int main ( )
{
   int a = 5, b = 6;
   float c = 7.6, d = 9.8;
   char e = 'M', f = 'Z';
   swap (&a, &b);            // compiler puts int in for T
   swap (&c, &d);            // compiler puts float in for T
   swap (&e, &f);            // compiler puts char in for T
   cout << "a=" << a << " and b=" << b << endl;
   cout << "c=" << c << " and d=" << d << endl;
   cout << "e=" << e << " and f=”  << f  << endl;
}
```