Inheritance: A simple example

```
class sphere {

public:
   sphere();
   sphere(double r);
   double getVolume();
   void setRadius(double r);
   void display();

private:
   double theRadius;


};
```

```
class ball : public sphere {

public:
   ball();
   ball(double r, string n);
   string getName();
   void setName(string n);
   void display();

private:
   string name;

};
```

Inheritance rules:

-
-
-

Class substitution (via examples)

```
void printVolume(sphere t) {
  cout << t.getVolume() << endl;
}

int main() {
   sphere s(3.0);
   ball b(2.2, "handful");

   double a = b.getVolume();

   printVolume(s);
   printVolume(b);
```

```
Base b;
Derived d;

b = d;

d = b;
```

```
Base * b;
Derived * d;

b = d;

d = b;
```

More …

```
class sphere {

public:
   sphere();
   sphere(double r);
   double getVolume();
   void setRadius(double r);
   void sphere::display(){
      cout << "sphere" << endl;
   }
p
   double theRadius;


};
```

```
class ball : public sphere {

public:
   ball();
   ball(double r, string n);
   string getName();
   void setName(string n);
void ball::display(){
   cout << "ball" << endl;
}
   string name;

};
```

```
sphere s;
ball b;
s.display();
b.display();
```

```
sphere * sptr;
sptr = &s;
sptr->display();
```

```
sphere * sptr;
sptr = &b;
sptr->display();
```

"virtual" functions

```
class sphere {

public:
  sphere();
  void sphere::display(){
     cout << "sphere" << endl;
  }
        void display();

private:
  double theRadius;


};
```

```
class ball : public sphere {

public:
  ball();
  void ball::display(){
     cout << "ball" << endl;
  }
        void display();

private:
  string name;

};
```

```
cin << x;
if (x == 0)
     sptr = &s;
else sptr = &b;
sptr->display();
```

“virtual” functions – the rules

A virtual method is one a _____ can override.

A class’s virtual methods _____ be implemented. If not, then the class is an “abstract base class” and no objects of that type can be declared.

A derived class is not *required* to override an existing implementation of an _____ virtual method.

Constructors _____ be virtual

Destructors can and _____ virtual

Virtual method return type _____ be overwritten

Constructors for derived class:

(suppose base class has 0 and 1 arg. ctors.)

```
ball::ball(): sphere() {
    name = "unknown";
}
```

```
ball b;
```

```
ball::ball(double r, string n):
sphere(r) {
    name = n;
}
```

```
ball b(2.1, "watermelon");
```

“virtual” destructors:

```
class Base {
public:
  Base() {cout << “Ctor: B” << endl; }
  ~Base() {cout << “Dtor: B” << endl; }
};

class Derived: public Base {
public:
  Derived() {cout << “Ctor: D” << endl; }
  ~Derived() {cout << “Dtor: D” << endl; }
};
```

```
void main() {
   Base *v = new Derived();
   delete v;
}
```

Abstract Base classes:

```
class flower {
public:
  flower();
  virtual void drawBlossom() = 0;
  virtual void drawStem() = 0;
  …
};
```

```
void rose::drawBlossom() {
  // whatever
}

void rose::drawStem() {
  // whatever
}
```

```
class rose : public flower {
public:
  virtual void drawBlossom();
  virtual void drawStem();
  …
private:
  int blossom; // 꽃잎수
  int stem; // 길이
};
```

```
flower f;

rose r;

flower * fptr;
```

Concluding remarks on inheritance

Polymorphism: objects of different types can employ methods of the same name and parameterization.

```
animal ** farm;

farm = new animal *[5];
farm[0] = new dog;
farm[1] = new pig;
farm[2] = new horse;
farm[3] = new cow;
farm[4] = new duck;

for (int i = 0; i<5; i++)
   farm[i]->speak();
```

Inheritance provides DYNAMIC polymorphism—type dependent functions can be selected at run-time. Wikipedia: Polymorphism in OOP.

Next topic: “templates” are C++ implementation of static polymorphism, where type dependent functions are chosen at compile-time.

What do you notice about this code?

```
void swapInt(int x, int y) {
   int temp;
   temp = x;
   x = y;
   y = temp;
```

```
void swapChar(char x, char y) {
   char temp;
   temp = x;
   x = y;
   y = temp;
```

```
void main() {
   int a = 1; int b = 2;
   char c = 'r'; char d = 'y';
   swapInt(a, b);
   swapChar(c, d);
   cout << a << " " << b << endl;
   cout << c << " " << d << endl;
}
```

Function templates:

```
template <class T>
void swap(T & x, T & y) {
   T temp;
   temp = x;
   x = y;
   y = temp;
}
```

```
void main() {
   int a = 1; int b = 2;
   char c = 'r'; char d = 'y';
   swap   (a, b);
   swap   (c, d);
   cout << a << " " << b << endl;
   cout << c << " " << d << endl;
}
```

Class templates:

```
template <class T>

class objpair {

private:
   T a, b;

public:
   objpair(T first, T second);
   T getmax();

};
```

```
template <class T>
T objpair<T>::getmax(){
  T retmax;
  retmax = (a > b ? a : b);
  return retmax;
}

template <class T>
objpair<T>::objpair(T first, T second){
  a = first;
  b = second;
}
```

```
int main() {
   objpair<int> twoNums(100, 75);
   cout << twoNums.getmax() << endl;
   return 0;
}
```

Class templates:

```
template <class T>

class objpair {

private:
   T a, b;

public:
   objpair(T first, T second);
   T getmax();

};
```

```
template <class T>
T objpair<T>::getmax(){
  T retmax;
  retmax = (a > b ? a : b);
  return retmax;
}

template <class T>
objpair<T>::objpair(T first, T second){
  a = first;
  b = second;
};
```

Challenge1: write the function declaration for the copy constructor (if we needed one) for this class.

_____ _____:: _____ (_____)

Challenge2: How do you declare and allocate a dynamic array of `objpair`s of integers? (We want that array to have 8 elements.)

A note on templates:

```
template <class T, class U>

T addEm(T a, U b) {
   return a + b;
}

int main() {
  addEm<int, int>(3, 4);
  addEm<double, int>(3.2, 4);
  addEm<int, double>(3, 4.2);
  addEm<string, int>("hello", 4);
  addEm<int, string>(3,"hello");
}
```