

Announcements

MP2 available, due 2/10, 11:59p. Exam2:

“virtual” functions:

```
class sphere {  
public:  
    sphere();  
    sphere(double r);  
    ...  
};
```

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

void display();

```
private:  
    double theRadius;  
};
```

```
class ball:public sphere {  
public:  
    ball();  
    ball(double r string n);  
    string getName();  
};
```

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

void display();

```
private:  
    string name;  
};
```

ex4

```
if (a==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:

```
ball::ball() : sphere()  
{  
    name = "not known";  
}
```

```
ball b;
```

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

```
ball b(0.5, "grape");
```

“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;
    virtual void drawFoliage() = 0;
    ...
};
```

```
void daisy::drawBlossom() {
// whatever
}
void daisy::drawStem() {
// whatever
}
void daisy::drawFoliage() {
// whatever
}
```

```
class daisy:public flower {
public:
    virtual void drawBlossom();
    virtual void drawStem();
    virtual void drawFoliage();
    ...
private:
    int blossom; // number of petals
    int stem; // length of stem
    int foliage // leaves per inch
};
```

```
flower f;
daisy d;
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;  
}
```

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

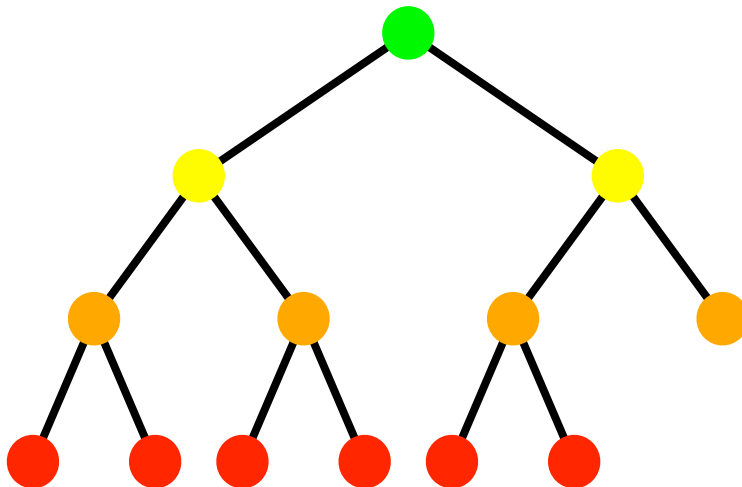
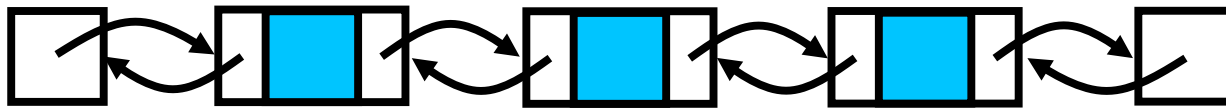
Function templates:

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

```
int main() {
    int a = 1; int b = 2;
    char c = 'n'; char d = 'm';
    swapUs      (a,b);
    swapUs      (c,d);
    cout << a << " " << b << endl;
    cout << c << " " << d << endl;
}
```


Classes can be given templates too:

0	1	2	3	4	5	6	7



Class templates:

```
template <class T>
class ezpair {
private:
    T a, b;
public:
    ezpair (T first, T second);
    T getmax ();
};
```

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

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

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

Class templates:

```
template <class T>
class ezpair {
private:
    T a, b;
public:
    ezpair (T first, T second);
    T getmax ();
};
```

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

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

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

_____ :: _____ (_____)

Challenge2: How do you declare a dynamic array of `mypairs` of integers?

Challenge3: How do you allocate memory if you 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>(4,3.2);
    addEm<string,int>("hi",4);
    addEm<int,string>(4,"hi");
}
```