C++ Short Course Part 2

- Constructor (reprise)
- The Big 3 – memory management
  - Overloading operators
- Inheritance
- Templates
- Generic programming

C++ Short Course Part 2

- Constructor (reprise)
- The Big 3 – memory management
  - Overloading operators
- Inheritance
- Templates
- Generic programming

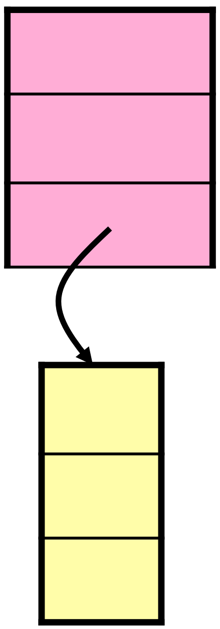
Constructors reprise:

```
…
//default constructor, alt syntax
sphere::sphere()
{


}
…
```

```
class sphere{
public:
sphere();
sphere(double r);
sphere(const sphere & orig);
void setRadius(double newRad);
double getDiameter() const;
…
private:
double theRadius;
int numAtts;
string * atts;
};
```

What do you want the object to look like when you declare it?

```
sphere a;
```



C++ Short Course Part 2

- Constructor (reprise)
- The Big 3 – memory management
  - Overloading operators
- Inheritance
- Templates
- Generic programming

Copy constructor - utility:

```
class sphere{
public:
sphere();
sphere(double r);
sphere(const sphere & orig);
void setRadius(double newRad);
double getDiameter() const;
…
private:
double theRadius;
int numAtts;
string * atts;
};
```

Use 1:

```
sphere myFun(sphere s){
   //play with s
   return s;
}

int main(){
  sphere a, b;
   // initialize a
   b = myFun(a);
   return 0;
}
```

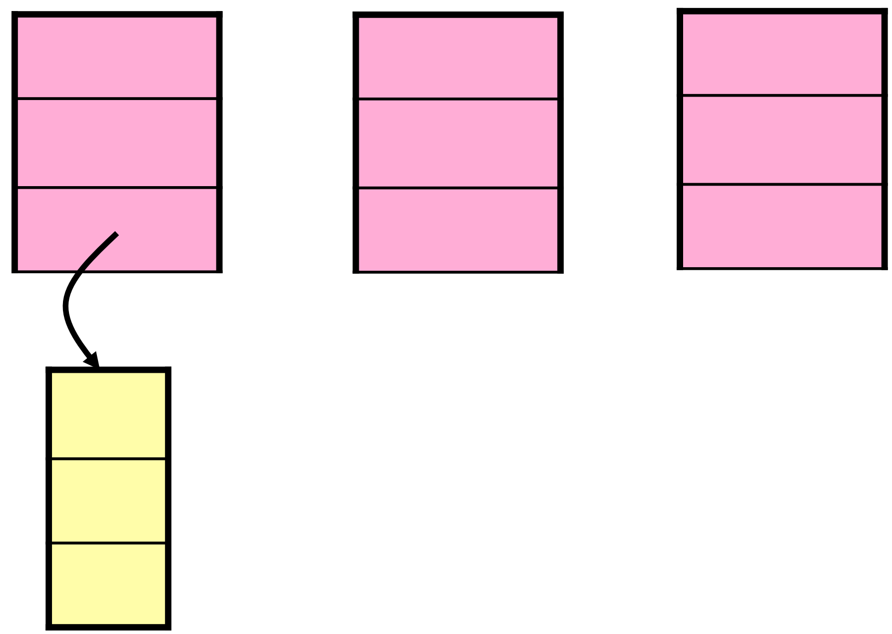
Use 2:

```
int main(){



};
```

Copy constructor:

```
class sphere{
public:
sphere();
sphere(double r);
sphere(const sphere & orig);
void setRadius(double newRad);
double getDiameter() const;
…
private:
double theRadius;
int numAtts;
string * atts;
};
```

```
…
//copy constructor
sphere::sphere(const sphere & orig)
{






}
…
```



Copy constructor discussion

```
class sphere{
public:
sphere();
sphere(const sphere & orig);
…
private:
    double theRadius;
    int numAtts;
    string * atts;
};
```

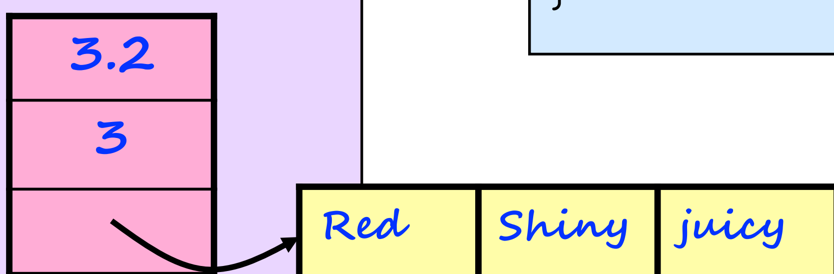
1. Why is the cctor's param pbr?

2. What does it mean that the cctor's param is const?

3. Why did we need to write a custom cctor?

Destructors:

```
class sphere{
public:
sphere();
sphere(double r);
sphere(const sphere & orig);
~sphere();
…
private:
double theRadius;
int numAtts;
string * atts;
};
```



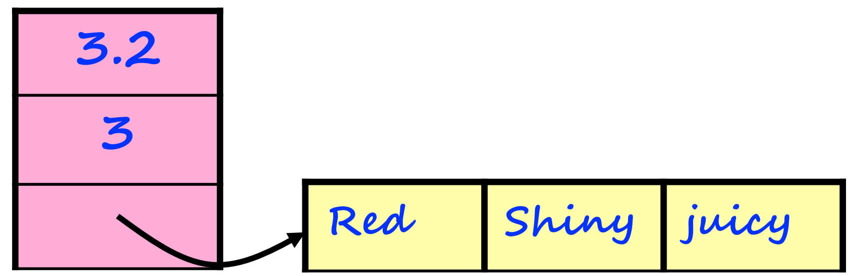

```
void myFun(sphere s){
   sphere t(s);
   …
   // play with s and t
   …
}

int main(){
   sphere a;
   myFun(a);
}
```

```
//destructor
sphere::~sphere(){


}
```

Destructors:

```
int main(){
   sphere * b = new sphere;
   delete b;
   return 0;
}
```

```
class sphere{
public:
sphere();
sphere(double r);
sphere(const sphere & orig);
~sphere();
…
private:
double theRadius;
int numAtts;
string * atts;
};
```




```
//destructor
sphere::~sphere(){


}
```

The destructor, a summary:

1. Destructor is never "called." Rather, we provide it for the system to use in two situations:

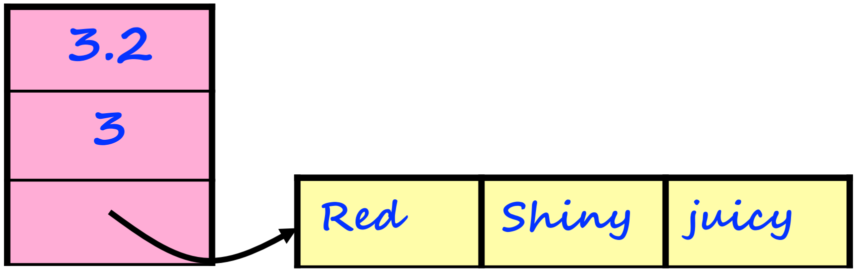
    a) ____________________

    b) ____________________

2. If your constructor, ____________________, allocates dynamic memory, then you need a destructor.

3. Destructor typically consists of a sequence of delete statements.

```
class sphere{
public:
   //tons of other stuff
   ~sphere();
private:
   double theRadius;
   int numAtts;
   string * atts;
};
```



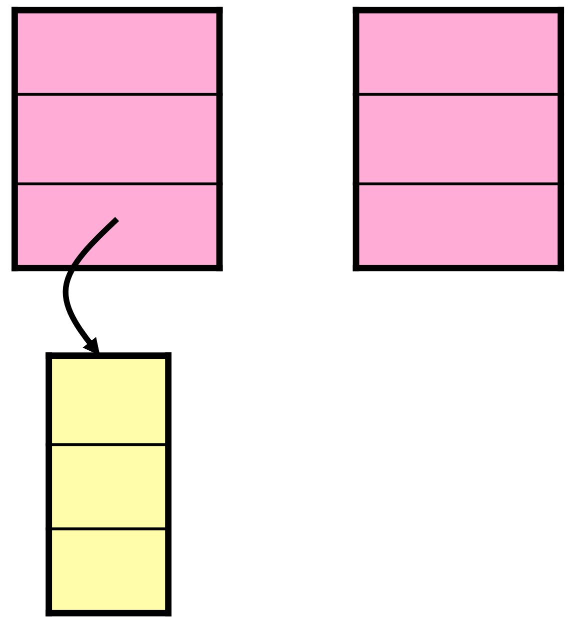
C++ Short Course Part 2

- Constructor (reprise)
- The Big 3 – memory management
  - Overloading operators
- Inheritance
- Templates
- Generic programming

One more problem:

```
class sphere{
public:
sphere();
sphere(double r);
sphere(const sphere & orig);
~sphere();
…
private:
double theRadius;
int numAtts;
string * atts;
};
```

```
int main(){
   sphere a, b;
   // initialize a
   b = a;
   return 0;
}
```



Overloaded operators:

```
int main(){
   // declare a,b,c









   // initialize a,b
   c = a + b;
   return 0;
}
```

```
// overloaded operator
sphere & sphere::operator+
     (const sphere & s){





}
```

Overloaded operators: what can be overloaded?

arithmetic operators, logical operators, I/O stream operators

```
    +    -    *    /    =    <    >    +=   -=   *=   /=
<<   >>   <<=   >>=   ==   !=   <=   >=   ++   --   %   &   ^
          !   |   ~   &=   ^=   |=   &&   ||   %=
               []   ()   ,   ->*   ->
          new   delete   new[]   delete[]
```

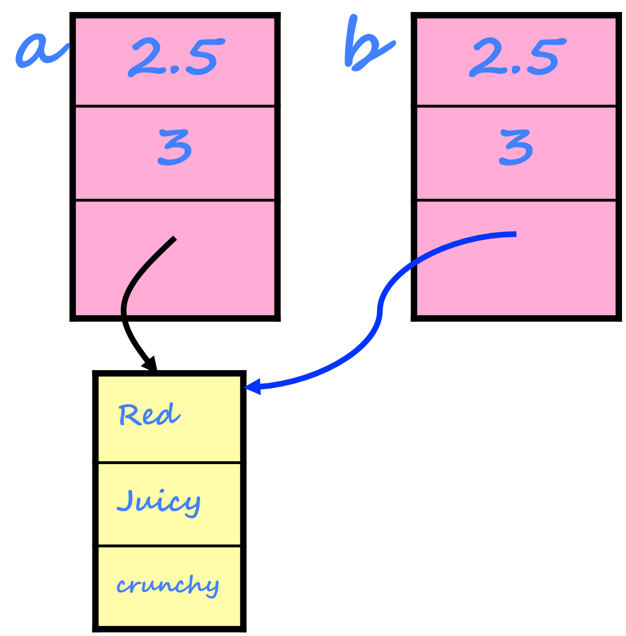
C++ Short Course Part 2

- Constructor (reprise)
- The Big 3 – memory management
  - Overloading operators
- Inheritance
- Templates
- Generic programming

One more problem: *default assignment is memberwise, so we redefine =.*

```
class sphere{
public:
sphere();
sphere(double r);
sphere(const sphere & orig);
~sphere();
_________ operator=(______________________);
…
private:
double theRadius;
int numAtts;
string * atts;
};
```

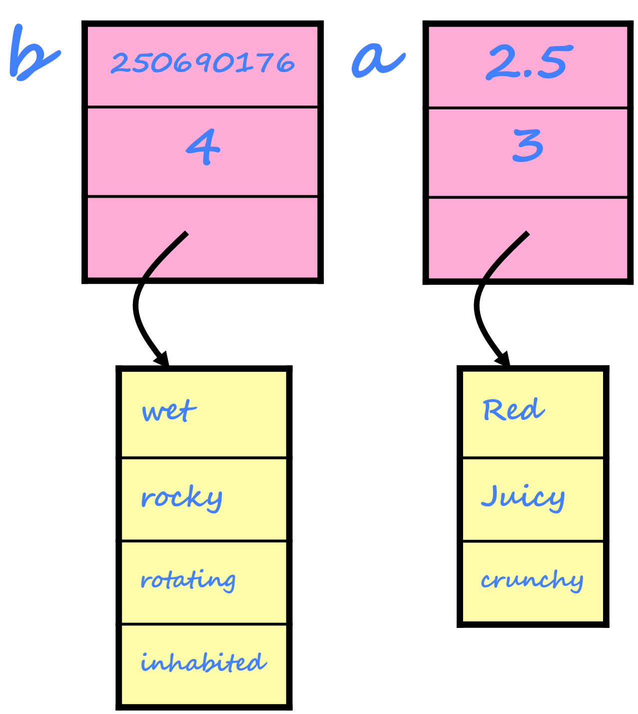
```
int main(){
   sphere a, b;
   // initialize a
   b = a;
   return 0;
}
```



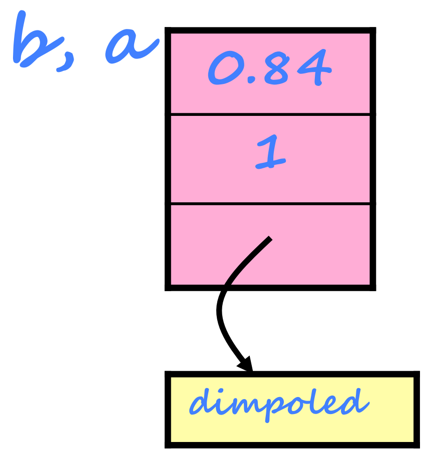
Overloading Operator=:

```
int main(){
  sphere a, b;
   // initialize a
   b = a;
   return 0;
}
```

b=a



b=a



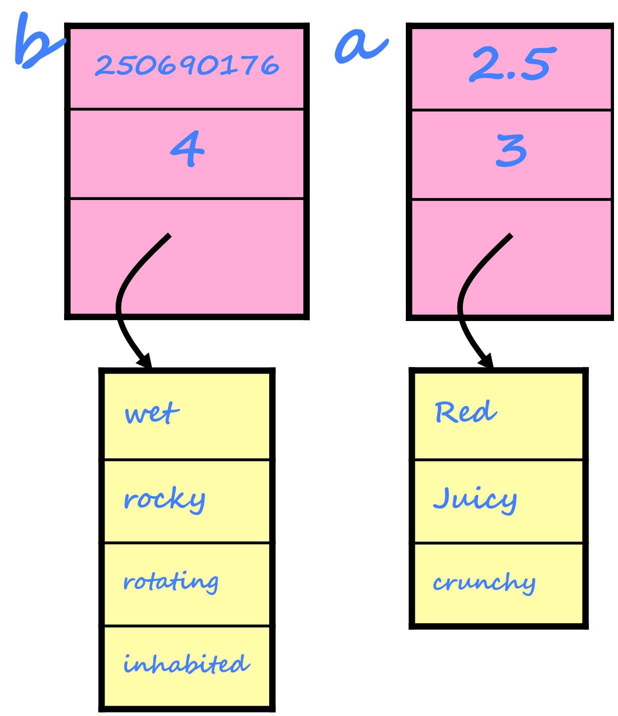
c=b=a

Operator=, the plan:

```
…
// overloaded =
sphere & sphere::operator=(const sphere & rhs){
   //protect against re-assignment

      //clear lhs

      //copy rhs

   //return a helpful value
}
```

```
c
pu
sp
sp
sp
~s

…
pr
do
int numAtts;
string * attributes;
};
```

```
int main(){
  sphere a, b;
   // initialize a
   b = a;
   return 0;
}
```



Operator=:

```
class sphere{
public:
sphere();
sphere(double r);
sphere(const sphere & o
~sphere();
sphere & operator=(cons
private:
double theRadius;
int numAtts;
string * attributes;
};
```

```
…

sphere & sphere::operator=(const sphere & rhs){

   if (this != &rhs) {

      clear();
      copy(rhs);

   }
   return *this;

}
```

Why not `(*this != rhs)` ?

-
-
-

The Rule of the Big Three:

If you have a reason to implement any one of

- ______________________
- ______________________
- ______________________

then you must implement all three.

Object Oriented Programming

Three fundamental characteristics:

encapsulation - separating an object's data and implementation from its interface.

inheritance -

polymorphism - a function can behave differently, depending on the type of the calling object.

C++ Short Course Part 2

- Constructor (reprise)
- The Big 3 – memory management
  - Overloading operators
- Inheritance
- Templates
- Generic programming

Inheritance: a simple first 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;
};
```

```
int main() {
sphere a;


}
```

inheritance rules:

-
-
-

Inheritance: write down 3 observations about prev slide

Subclass substitution (via examples):

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

int main() {
   sphere s(8.0);
   ball b(3.2, "pompom");

   double a = b.getVolume();

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

```
Base b;
Derived d;

b=d;

d=b;
```

```
Base * b;
Derived * d;

b=d;

d=b;
```

something to consider:

```
class sphere {
public:
    sphere();
    sphere(double r);
    …
    void display();
private:
    double theRadius;
};
```

```
class ball:public sphere {
public:
    ball();
    ball(double r string n);
    …
    trin
    oid
    void display();
private:
    string name;
};
```

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

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

ex1

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

ex2

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

ex3

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

“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();
```

```
oid
```

```
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;
     …
};
```

```
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
};
```

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

```
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.

C++ Short Course Part 2

- Constructor (reprise)
- The Big 3 – memory management
  - Overloading operators
- Inheritance
- Templates
- Generic programming

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;
}
```

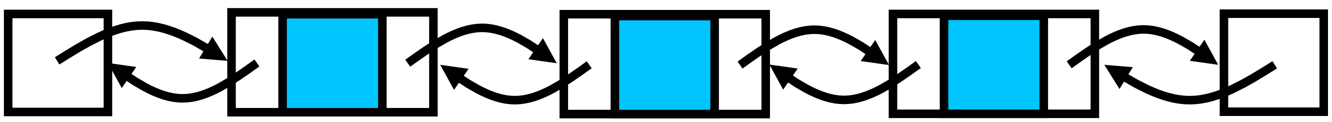
Function templates:

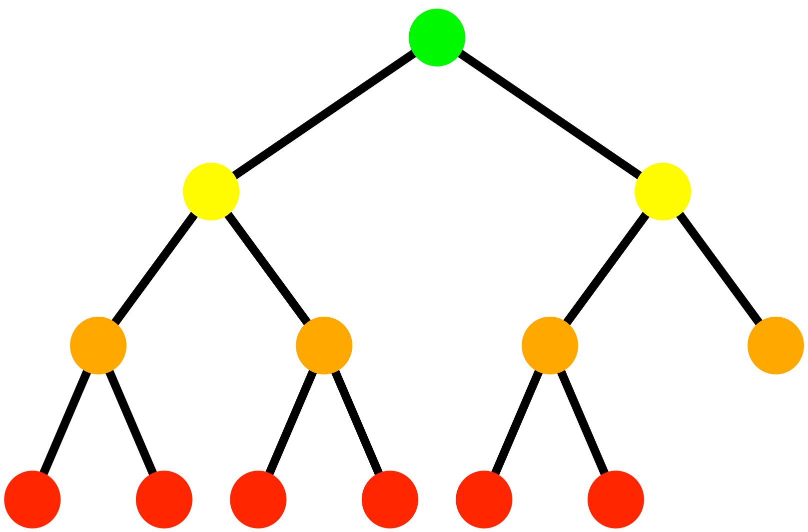
```
template <class T>
void swapUs(T & x, T & y){
   T temp;
   temp = x;
   x = y;
   y = temp;
}
```

```
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 `mypair`s of integers?

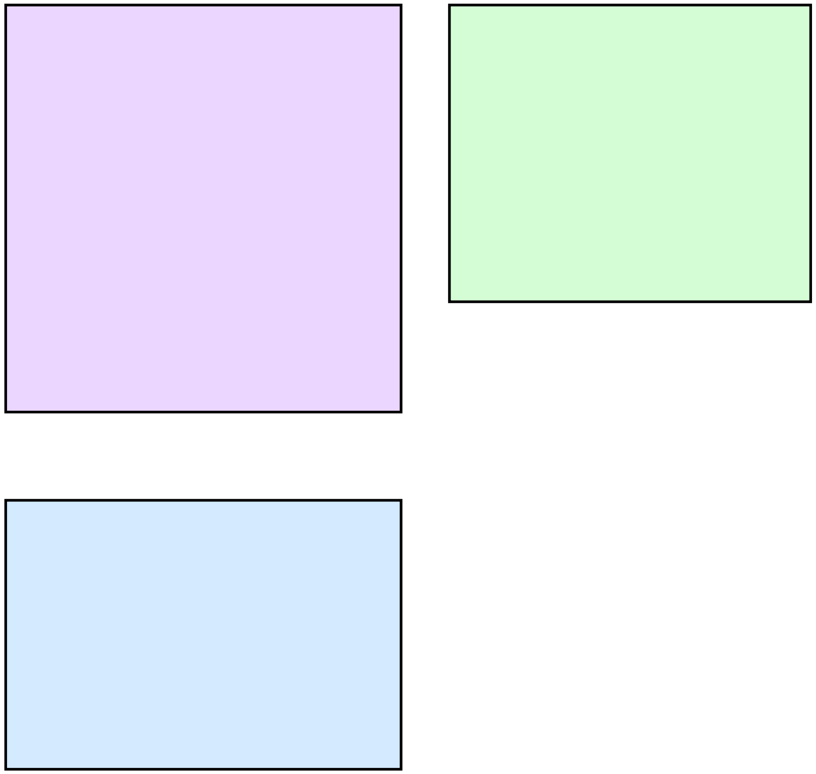
Challenge3: How do you allocate memory if you want that array to have 8 elements?

A note on templates:

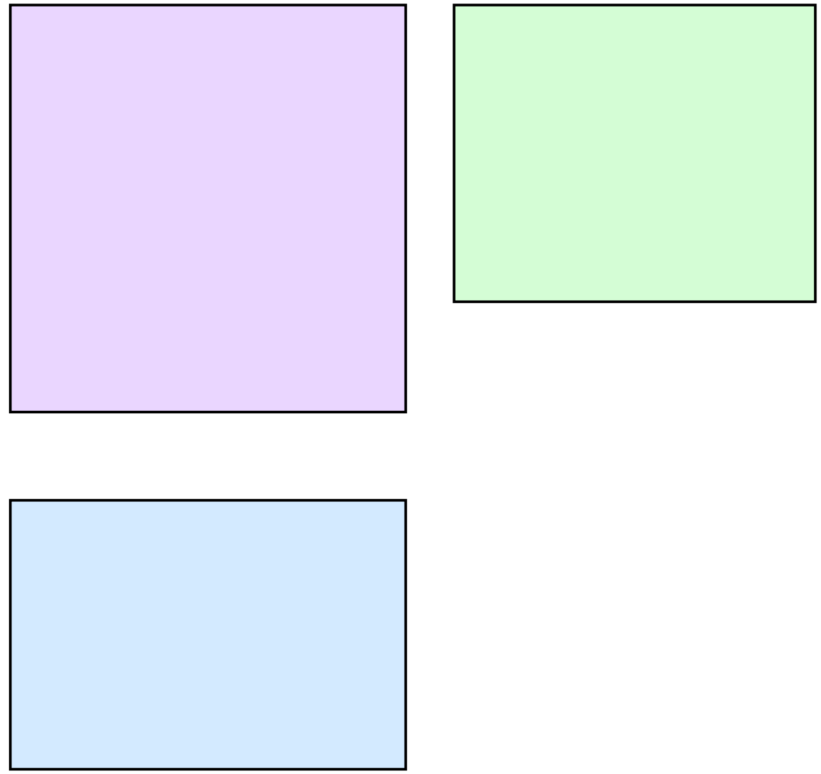
```
template <class T, class U>
T addEm(T a, U b) {
   return a + b;
}

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");
}
```

Template compilation:

Old:



New:



C++ Short Course Part 2

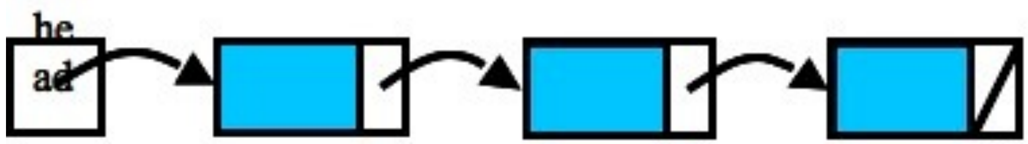
- Constructor (reprise)
- The Big 3 – memory management
  - Overloading operators
- Inheritance
- Templates
- Generic programming

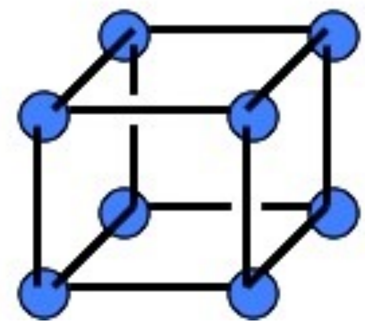
C++: A bit of Magic…

```
#include <list>
#include <iostream>
#include <string>
using namespace std;

struct animal {
   string name;
   string food;
   bool big;
   animal(string n="blob", string f="you”, bool b=true):name(n),food(f),big(b) {}
};

int main() {

   animal g("giraffe","leaves"), p("penguin","fish”,false), b("bear");
   list<animal> zoo;

   zoo.push_back(g); zoo.push_back(p); zoo.push_back(b); //STL list insertAtEnd

   for(list<animal>::iterator it = zoo.begin(); it != zoo.end(); it++)
      cout << (*it).name << "  " << (*it).food << endl;

   return 0;
}
```

Suppose these familiar structures were encapsulated.

Iterators give client the access it needs to traverse them anyway!





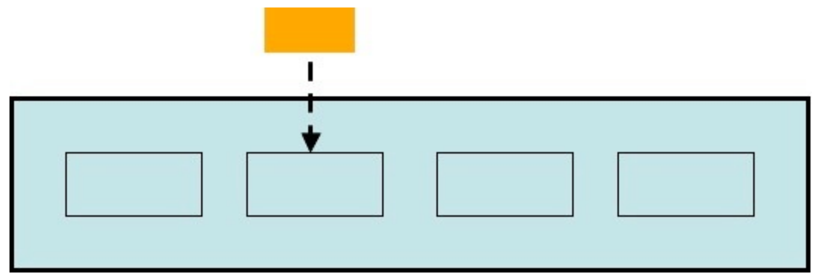
Objects of type “iterator” promise to have at least the following defined:

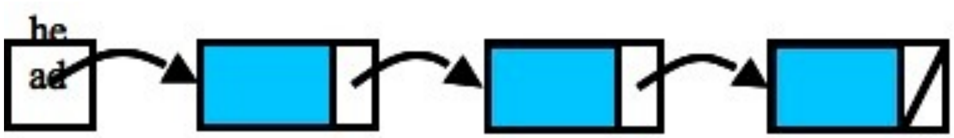
operator++
*operator**
operator!=
operator==
operator=

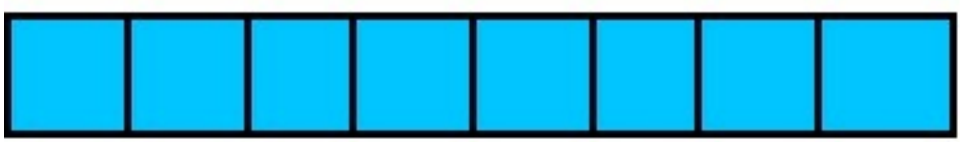
“Container classes” typically have a variety of iterators defined within:

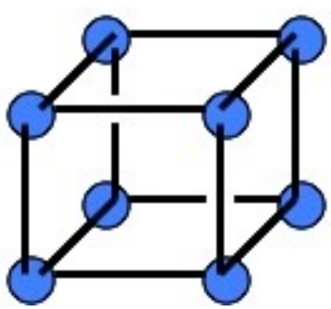
Forward
Reverse
Bidirectional

Iterator class:









	pm	++	*
linked list			
array			
hypercube			

```
class apartmentBldg {
public:






private:




};
```

Where do these constructs live?

iterator class

begin()/end()

op++/op*

iter representation

std library documentation: http://www.sgi.com/tech/stl/

Generic programming: (more magic)

```
#include <list>
#include <iostream>
#include <string>
using namespace std;

struct animal {
   string name;
   string foo
   bool big;
   animal(st
};

int main() {

   animal g("
   list<anima

   zoo.push_back(g); zoo.push_back(p); zoo.push_back(b); //STL list insertAtEnd

   for(list<animal>::iterator it = zoo.begin(); it != zoo.end(); it++)
      cout << (*it).name << " " << (*it).food << endl;

return 0;
}
```

```
class printIfBig {
public:
   void operator()(animal a) {
      if (a.big) cout << a.name << endl;
   }
};
```

1. Declare an object of type `animal`:
2. Declare an object of type `printIfBig`:
3. Using your answers for 1 and 2, invoke a member function of the `printIfBig` class:

Generic programming: (more magic)

```
template<class Iter, class Formatter>
void print(Iter first, Iter second, Formatter printer) {
   while (!(first==second)) {
      printer(*first);
      first++;
   }
}
```

Write a short description of this function:

This is a function called ____________, whose inputs are two ____________ and a ____________. The function appears to ____.

What is printer?

Generic programming: (more magic)

```
#include <list>
#include <ios
#include <str
using namespa

struct anima
   string nam
   string foo
   bool big;
   animal(str
};

int main() {

   animal g("
   list<anima

   zoo.push_b

   for(list<a
      cout <<
```

```
template<class Iter, class Formatter>
void print(Iter first, Iter second, Formatter printer) {
   while (!(first==second)) {
      printer(*first);
      first++;
   }
}
```

```
class printIfBig {
public:
   void operator()(animal a) {
      if (a.big) cout << a.name << endl;
   }
};
```

```
printIfBig myFun;
print<list<animal>::iterator,printIfBig>(zoo.begin(),zoo.end(),myFun);
```