Constructors

- Class constructors are called when objects of that class is created.
- A constructor creates an object in 4 steps:
  1. Allocate memory
  2. If it has base class, call the base class's constructor.
  3. Initialize member values using the constructor initialization list.
  4. Execute the body of the constructor

Constructors - Continue

- The first step (memory allocation) can be carried out by compiler:
  - `Book b("C++", 1990);` //In this case, compiler also call destructor ~Book() to de-allocate the memory when b is out-of-scope
- Or by programmers:
  - `Book *b= new Book("C++", 1990);` //In this case, programmers are also responsible to call delete b; which calls destructor ~Book() to de-allocate the memory used by b.
Destructors

- Destructors are called either by the compiler automatically (when object goes out of scope) or by programmer (delete object).
- Destructors are responsible to re-claim the memory of the destroyed object.

```cpp
Book::~Book()
{
    // Would the compiler automatically free the allocated memory or we need to manually re-claim the allocated memory? 
    // Answer: depends on the data member of the class
}
```

Class Members are Pointers

```cpp
class Book{
    private:
        vector<string>* authors;
        string title;
        int year;
    }
Book::~Book()
{
    delete authors;
}
```

The Rule of Three

- Compiler automatically generates default constructor, copy constructor and assignment operator=, if they are not explicitly defined.
- These default functions perform shallow copy of data members.
- When any data member is a pointer (*), you need to explicitly define the following 3:
  - Copy constructor (memory allocation)
  - Assignment operator (memory de-allocation/ allocation)
  - Destructor (memory de-allocation)

Memory Allocation for authors

```cpp
authors=new vector<string>();
int i=0;
vector<string>* authorsList=other.authors;
while(i < (*authorsList).size())
{ (*authors).push_back((*authorsList).at(i++));
}
```

This is the other Book object To be copied over.

- Constructors have to use new to allocate memory for authors.
  authors=new vector<string>();
- Destructors have to use delete to reclaim the memory of authors.
  delete authors;

- Copy Constructor: check other.authors is not NULL.
- Values provided Constructor: check provided authors is not NULL.
- Default Constructor: C++ does not initialize pointer as null, or int to 0. They are garbage, which causes the delete authors in assignment- operator and update function to produce segv. The default constructor has to initialize “authors” to null and “year” to 0.
Assignment= and Update

• Assignment operator: lhs=rhs;
  — delete lhs.authors and new lhs.authors to add rhs.authors information.
• Update function: a.update(b);
  — delete a.authors and new a.authors to add b information.
• Do not use authors.clear() because vector.clear() calls the destructor of the vector element (which is ~string()) to de-allocate memory. When the Book object is created using the default constructor (Book a), authors=null. authors.clear() causes segv.
• Also check rhs.authors and b.authors are not NULL before loop the vector.

Assignment 3: Operator==

bool Book::operator== (const Book& other)
{
  //return 1 if book and other are the same.
  return (year==other.year && (name.compare(other.name) == 0));
}

class Book{
public:
  Book();
  Book(string, int);
  bool operator== (const Book& other);
private:
  string name;
  int year;
};

Constructors Operation

//constructor
AudioBook::AudioBook(string title_in, int year_in, double length_in): Book(title_in, year_in, length_in){}
//class AudioBook: public Book
//Book b(.); //?
//Book::f(int);
//Book::f(string);
//AudioBook a(.);
//AudioBook::f(Book);
//AudioBook::f(string);
//assign length_in to length – step 3
b.f(2); //?
AudioBook c("JavaScript",2008.2.5);
//assign length_in to length – step 3
//assign length_in to length – step 3
//assign length_in to length – step 3
//assign length_in to length – step 3
//assign length_in to length – step 3
//assign length_in to length – step 3
//assign length_in to length – step 3
//assign length_in to length – step 3
//assign length_in to length – step 3
//assign length_in to length – step 3
//assign length_in to length – step 3
//assign length_in to length – step 3
//assign length_in to length – step 3
//assign length_in to length – step 3
//assign length_in to length – step 3

Static Binding

• The type of implicit parameter determines which of the overriding functions to call.
• The type of explicit parameter determines which of the overloading functions to call.
10/19/12

Static vs. Dynamic Binding

- C++ supports both static and dynamic binding for overriding functions calls.
- Dynamic binding only applies to virtual functions that are called by object pointers.

<table>
<thead>
<tr>
<th>Function calls</th>
<th>binding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ObjectPointer -&gt; virtualFunction</td>
<td>dynamic</td>
</tr>
<tr>
<td>Object.virtualFunction</td>
<td>static</td>
</tr>
<tr>
<td>ObjectPointer -&gt; nonVirtualFunction</td>
<td>static</td>
</tr>
</tbody>
</table>

Examples

class Book{
    public:
        virtual double shipping();
        void update(int);
    ...
}
class AudioBook : public Book{
    public:
        double shipping();
        void update(double);  
    ...
}

AudioBook a(...);
Book b=a;  //ok
b.shipping();  //call ?, dynamic/static
a.shipping();  //call ?, dynamic/static
Book* ap=&a;
ap->shipping();  //call ?, dynamic/static
ap->update(1995);  //call ?, dynamic/static

Dynamic Binding: Assignment 4 Q3

double shippingCost(vector<Book*> order) {
    double cost=0.0;
    for(int i=0; i < order.size(); i++)
        cost+=order.at(i)->shipping();
    return cost;
}
virtual double Book::shipping(){return 5.0;}
double AudioBook::shipping(){return 3.0;}
double eBook::shipping(){return 0.0;}

c++ Objects

int main()
{
    Point p1(10,10);
    Point p2=p1;
    std::cout << p2.my_x() << " , " <<  
                p2.my_y();  //?
    p2.set_x(100);
p2.set_y(100);
    std::cout << p2.my_x() << " , " <<  
                p2.my_y();  //?
    std::cout << p1.my_x() << " , " <<  
                p1.my_y();  //?
}

Point p1 X=10 Y=10
Point p2 X=10 Y=10

C++ compiler manipulates object itself for
computation.
- This is NOT efficient.
  - Assignment operator (=): copy the
    entire object value;
  - function passed the object argument by
    copying the object value;
  - When an object is returned from a
    function, a copy of the object is returned.
C++ Parameter Variable is Pass by Value – Object Variables

Point swapObject(Point a)
{
  // is an object value, the
  // changes made are NOT visible
  // outside the function.
  int temp;
  temp=a.my_x();
  a.set_x(a.my_y());
  Point localV(100,101);
  //return local object is ok as a copy of
  //the object reference is returned
  return localV;
}

The same applies to template class objects, such as vector<T> objects

C++ Pass by reference using &

Point swapObject(Point & a)
{
  // is an object pointer, the
  // changes made are visible
  // outside the function.
  int temp;
  temp=a->my_x();
  a->set_x(a->my_y());
  Point localV(100,101);
  return localV;
}

int main()
{
  Point p(10,20);
  std::cout << p.my_x() << " , " << p.my_y() << std::endl; //?
  //object "p" value is copied to the "a"
  //parameter variable
  Point returnPoint=swapObject(p);
  std::cout << p.my_x() << " , " << p.my_y() << std::endl; //?
  std::cout << returnPoint.my_x() << " , " << returnPoint.my_y() << std::endl; //?
}

Don't return &localV or localV*,
as the results are unreliable.

int main()
{
  Point returnPoint=swapObject(&p);
}

Point returnPoint=swapObject(&p);