# Constructors, destructors and call orders

# What we will learn about in these slides

- The order that constructors and destructors are called in, and why they are called in that order
- Constructor initializer lists
- Why destructors should always be virtual
- Special constructors
  - *copy* constructors
  - *converting* constructors
  - See https://en.cppreference.com/w/cpp/language/default_constructor for other types of constructors

# Constructor call order is from least to most derived

(See Example1 code)

```
class Employee {
public:
  int yearsWorked;
  int daysWorked
  Employee(int days);
};

Employee:: Employee(int days) {
  yearsWorked = days/365;
  daysWorked = days % 365;
  std::cout << “Employee” << std:: stdout;
}
```

The Employee constructor must execute first so that the more derived Staff constructor can access initialized Employee state

Output

```
Employee
Staff
```

```
class Staff : public Employee {
public;
  bool eligibleToRetire;
  Staff(int days);
};

Staff::Staff(int days) : Employee(days) {
  eligibleToRetire = yearsWorked > 20;
  std::cout << “Staff” << std::endl;
}

int main(int argc, char* argv[ ]) {
  Staff staff = new Staff(37522);
}
```

# Initializer lists Foo::Foo(Bar v) : Base(v), x(v) { . . . }

- Actions in the initializer list are performed before the body of the constructor, i.e. before what is in the { . . . }
- Four reasons this is a good thing.
  1. It allows a non-default base constructor to be called early.
  2. When initializing an object attribute (i.e. field) that is an object it is more efficient than an assignment
     a. x = *v;* may execute code that copies *v* into a temporary, and copies the temporary into *x* (compilers try to clean this up, but are not always successful)
     b. "*: x(v)"* directly copies *v* value into *x*'s space
  3. Easy to see where object attributes are initialized, and easy to type
  4. Initializer lists are the only way to initialize **const** refs and reference members (we'll talk about both of these later.)

# The negative/downside of initializer lists

- Initializations are done in *declaration order, not the order they appear in the initializer list*. This can lead to subtle, hard to find bugs when it makes a difference (it often doesn't)

# What order are initializations done?

(See Example2 code)

```
Base::Base(int u, int v, int w) : d(u), c(v), b(w), a(-1) { }

int main(int argc, char* argv[ ]) {
  Base base(10, 20, 30); // a: -1, b: 30, c: 20, d: 10
  base.print( );
}
```

- Initializations in the list are not done in the order listed
- The order they are done in is:
  - Execute base class constructor first
  - Perform the initializations in the order *declared*, where variables in a less derived class are considered to be declared earlier than variables in more derived classes
- As shown here, this often doesn't make any difference

# Initialization order can, however, make a difference

(see Example3 code)

## Base::Base(int u, int v, int w) : d(8), c(b), b(4), a(d) { }

Assume the variables are declared in the order a, b, c, d, as in Base.h in the previous example (Example2 code)

Assume the call Base *b = new Base(90, 100, 110);

1. Storage is allocated for the object and zeroed out, i.e., a=0; b=0; c=0; d=0
2. a(d) is performed, initializing a to 0 (the value of d)
3. b(4) is performed, initializing b to 4
4. c(b) is performed, initializing c to 4
5. d(8) is performed, initializing d to 8

```
a: 0, b: 4, c: 4, d: 8
```

# Constructor call order is from least to most derived

(See Example4 code)

```
class Base {
public:
  Base( );
};

Base::Base( ) {
  std::cout << "Base" << std::endl;
}
```

```
class Derived : public Base {
public:
  Derived( );
};

Derived::Derived( ) {
  std::cout << "Derived" << std::endl;
}

int main(int argc, char* argv[ ]) {
  Base* base = new Derived( );
  delete base;
}
```

Output

```
Base
Derived
```

# Destructors

- Constructors initialize objects when they are created
- Destructors are called when objects are destroyed
  - Stack allocated object variables are deleted when the stack frame of the function they are declared in is popped off the stack
  - Heap allocated objects are deleted when they are freed from the stack, that is, when *delete* is called on some pointer that points to them.
- Destructors are typically used to free storage owned by the object, to close files, etc.
- Destructors are called from most derived to least derived – the opposite of constructors
  - This allows a derived object destructor to have access to the state of less derived objects while executing its destructor
  - After a destructor executes, you should assume none of the object's data is valid

# Destructors

- If you do not supply a destructor, C++ supplies a zero arg default destructor.
  - This has been the case with the earlier examples in this course
  - A derived default constructor will call either a default or a user defined destructor in the more base class
  - We only need to provide destructors where there are actions for it
- Destructors should always be virtual
  - Failing to do this will cause problems, as we'll see soon

# Destructor call order is from least to most derived

(See Example5 code)

```
class Base {
public:
  Base( );
  virtual ~Base( );
};

Base::Base( ) {
  std::cout << "Base" << std::endl;
}

Base::~Base( ) {
  std::cout << "~Base" << std::endl;
}
```

```
class Derived : public Base {
public:
  Derived( );
  virtual ~Derived( );
};

Derived::Derived( ) {
  std::cout << "Derived" << std::endl;
}

Derived::~Derived( ) {
  std::cout << "~Derived" << std::endl;
}
```

```
int main(int argc, char* argv[ ]) {
  Base* base = new Derived( );
  delete base;
}
```

Output

```
Base
Derived
~Derived
~Base
```

# Non-virtual destructors can yield surprising results – and memory leaks (See Example6 code)

```
class Base {
public:
   Base( );
  ~Base( ); // note no virtual
};

Base::Base( ) {
   std::cout << "Base" << std::endl;
}

Base::~Base( ) {
   std::cout << "~Base" << std::endl;
}
```

```
class Derived : public Base {
public:
   Derived( );
   virtual ~Derived( );
};

Derived::Derived( ) {
   std::cout << "Derived" << std::endl;
}

Derived::~Derived( ) {
   std::cout << "~Derived" << std::endl;
}
```

```
int main(int argc, char* argv[ ]) {
   Base* base = new Derived( );
   delete base;
}
```

Output

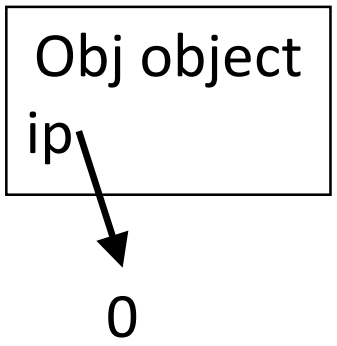
```
Base
Derived
~Derived
~Base
```

# The usefulness of Destructors (See Example7 code)

```
class Obj {
public:
  Obj(int);
  virtual ~Obj( );
private:
  int* ip;
};

Obj::Obj(int i) {
  ip = new int;
  *ip = i;
}

Obj::~Obj( ) {
  std::cout << "~Obj " << *ip;
  delete ip;
}
```



# The usefulness of Destructors

(See Example7 code)

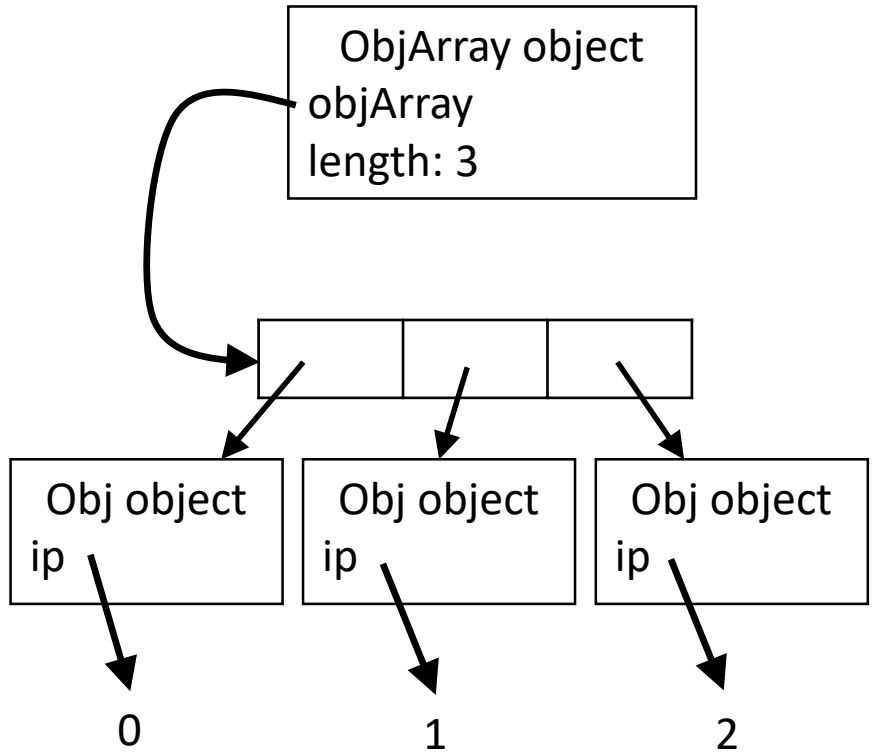
```
class Obj {
public:
  Obj(int);
  virtual ~Obj( );
private:
  int* ip;
};

Obj::Obj(int i) {
  ip = new int;
  *ip = i;
}

Obj::~Obj( ) {
  std::cout << "~Obj " << *ip;
  delete ip;
}
```

```
class ObjArray {
public:
  ObjArray(int);
  virtual ~ObjArray( );
private:
  Obj** objArray;
  int length;
};
ObjArray::ObjArray(int len) {
  length = len;
  objArray = new Obj*[len];
  for (int i = 0; i < len; i++) {
    objArray[i] = new Obj(i);
  }
}
```

```
ObjArray::~ObjArray( ) {
  for (int i = 0; i < length; i++) {
    std::cout << "~ObjArray " << i ;
    delete objArray[i];
  }
  std::cout << "~ObjArray objArray" ;
  delete objArray;
}
```



# The usefulness of Destructors (See Example7 code)

```
class Obj {
public:
  Obj(int);
  virtual ~Obj( );
private:
  int* ip;
};

Obj::Obj(int i) {
  ip = new int;
  *ip = i;
}

Obj::~Obj( ) {
  std::cout << "~Obj " << *ip;
  delete ip;
}
```

```
class ObjArray {
public:
  ObjArray(int);
  virtual ~ObjArray( );
private:
  Obj** objArray;
  int length;
};
ObjArray::ObjArray(int len) {
  length = len;
  objArray = new Obj*[len];
  for (int i = 0; i < len; i++) {
    objArray[i] = new Obj(i);
  }
}
ObjArray::~ObjArray( ) {
  for (int i = 0; i < length; i++) {
    std::cout << "~ObjArray " << i ;
    delete objArray[i];
  }
  std::cout << "~ObjArray objArray" ;
  delete objArray;
}
```

```
class ThreeArray : public ObjArray {
public:
  ThreeArray( );
  virtual ~ThreeArray( );
};

ThreeArray::ThreeArray( ) : ObjArray(3) { }

ThreeArray::~ThreeArray( ) {
  std::cout << "~ThreeAway" << std::endl;
}
```

# The usefulness of Destructors (See Example3b code)

```
class Obj {
public:
  Obj(int);
  virtual ~Obj( );
private:
  int* ip;
};

Obj::Obj(int i) {
  ip = new int;
  *ip = i;
}

Obj::~Obj( ) {
  std::cout << "~Obj " << *ip;
  delete ip;
}
```

```
class ObjArray {
public:
  ObjArray(int);
  virtual ~ObjArray( );
private:
  Obj** objArray;
  int length;
};
ObjArray::ObjArray(int len) {
  length = len;
  objArray = new Obj*[len];
  for (int i = 0; i < len; i++) {
    objArray[i] = new Obj(i);
  }
}
ObjArray::~ObjArray( ) {
  for (int i = 0; i < length; i++) {
    std::cout << "~ObjArray " << i ;
    delete objArray[i];
  }
  std::cout << "~ObjArray objArray" ;
  delete objArray;
}
```

```
class ThreeArray : public ObjArray {
public:
  ThreeArray( );
  virtual ~ThreeArray( );
};

ThreeArray::ThreeArray( ) : ObjArray(3) { }

ThreeArray::~ThreeArray( ) {
  std::cout << "~ThreeAway" << std::endl;
}
```

```
int main(int argc, char* argv[ ]) {
  ObjArray* threeArray = new ThreeArray( );
  std::cout << "deleting ThreeArray";
  delete threeArray;
}
```

# The usefulness of Destructors (See Example7 code)

```
Obj::Obj(int i) {
  ip = new int;
  *ip = i;
}

Obj::~Obj( ) {
  std::cout << "~Obj " << *ip;
  delete ip;
}

ObjArray::ObjArray(int len) {
  length = len;
  objArray = new Obj*[len];
  for (int i = 0; i < len; i++) {
    objArray[i] = new Obj(i);
  }
}
```

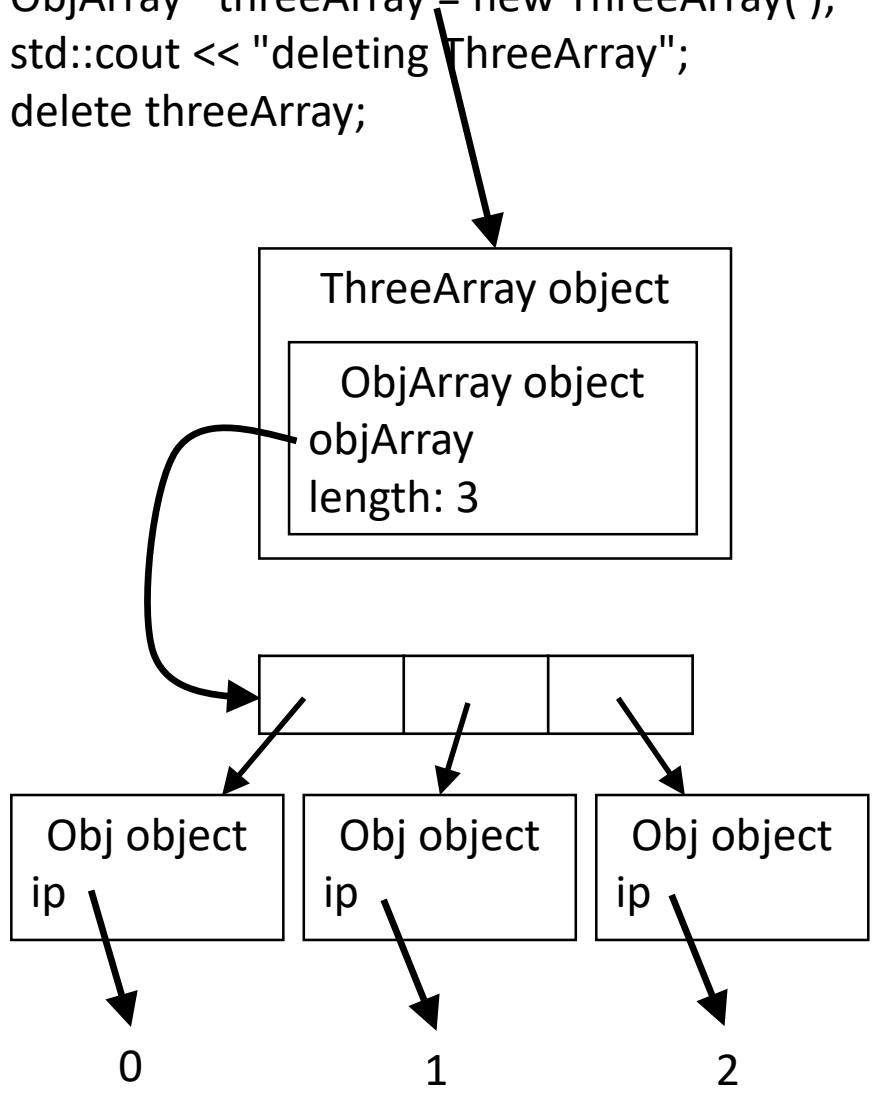
```
ObjArray::~ObjArray( ) {
  for (int i = 0; i < length; i++) {
    std::cout << "~ObjArray " << i ;
    delete objArray[i];
  }
  std::cout << "~ObjArray objArray" ;
  delete objArray;
}

class ThreeArray : public ObjArray {
public:
  ThreeArray( );
  virtual ~ThreeArray( );
};

ThreeArray::ThreeArray( ) : ObjArray(3) { }

ThreeArray::~ThreeArray( ) {
  std::cout << "~ThreeAway" << std::endl;
}
```

```
int main(int argc, char* argv[ ]) {
  ObjArray* threeArray = new ThreeArray( );
  std::cout << "deleting ThreeArray";
  delete threeArray;
}
```



# The usefulness of Destructors (See Example7 code)

```
Obj::Obj(int i) {
  ip = new int;
  *ip = i;
}

Obj::~Obj( ) {
  std::cout << "~Obj " << *ip;
  delete ip;
}

ObjArray::ObjArray(int len) {
  length = len;
  objArray = new Obj*[len];
  for (int i = 0; i < len; i++) {
    objArray[i] = new Obj(i);
  }
}
```

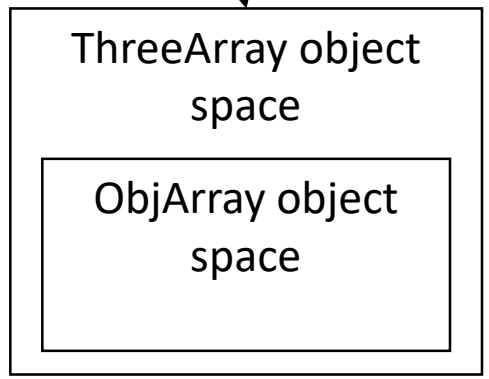
```
ObjArray::~ObjArray( ) {
  for (int i = 0; i < length; i++) {
    std::cout << "~ObjArray " << i ;
    delete objArray[i];
  }
  std::cout << "~ObjArray objArray" ;
  delete objArray;
}

class ThreeArray : public ObjArray {
public:
  ThreeArray( );
  virtual ~ThreeArray( );
};

ThreeArray::ThreeArray( ) : ObjArray(3) { }

ThreeArray::~ThreeArray( ) {
  std::cout << "~ThreeAway" << std::endl;
}
```

```
int main(int argc, char* argv[ ]) {
  ObjArray* threeArray = new ThreeArray( );
  std::cout << "deleting ThreeArray";
  delete threeArray;
}
```



# The usefulness of Destructors (See Example7 code)

```
Obj::Obj(int i) {
  ip = new int;
  *ip = i;
}

Obj::~Obj( ) {
  std::cout << "~Obj " << *ip;
  delete ip;
}

ObjArray::ObjArray(int len) {
  length = len;
  objArray = new Obj*[len];
  for (int i = 0; i < len; i++) {
    objArray[i] = new Obj(i);
  }
}
```

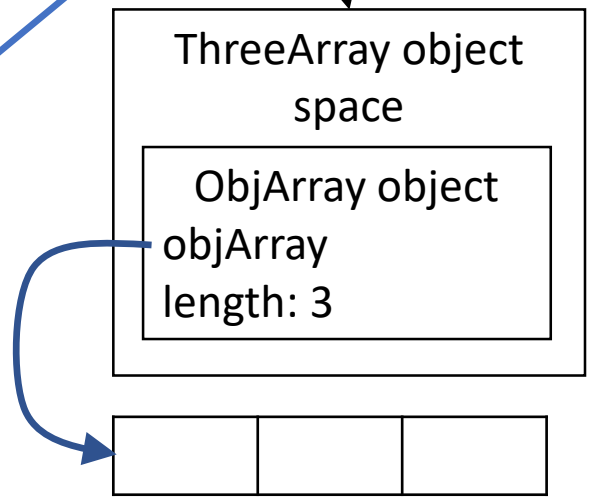
```
ObjArray::~ObjArray( ) {
  for (int i = 0; i < length; i++) {
    std::cout << "~ObjArray " << i ;
    delete objArray[i];
  }
  std::cout << "~ObjArray objArray" ;
  delete objArray;
}

class ThreeArray : public ObjArray {
public:
  ThreeArray( );
  virtual ~ThreeArray( );
};

ThreeArray::ThreeArray( ) : ObjArray(3) { }

ThreeArray::~ThreeArray( ) {
  std::cout << "~ThreeAway" << std::endl;
}
```

```
int main(int argc, char* argv[ ]) {
  ObjArray* threeArray = new ThreeArray( );
  std::cout << "deleting ThreeArray";
  delete threeArray;
}
```



# The usefulness of Destructors (See Example7 code)

```
Obj::Obj(int i) {
  ip = new int;
  *ip = i;
}

Obj::~Obj( ) {
  std::cout << "~Obj " << *ip;
  delete ip;
}

ObjArray::ObjArray(int len) {
  length = len;
  objArray = new Obj*[len];
  for (int i = 0; i < len; i++) {
    objArray[i] = new Obj(i);
  }
}
```

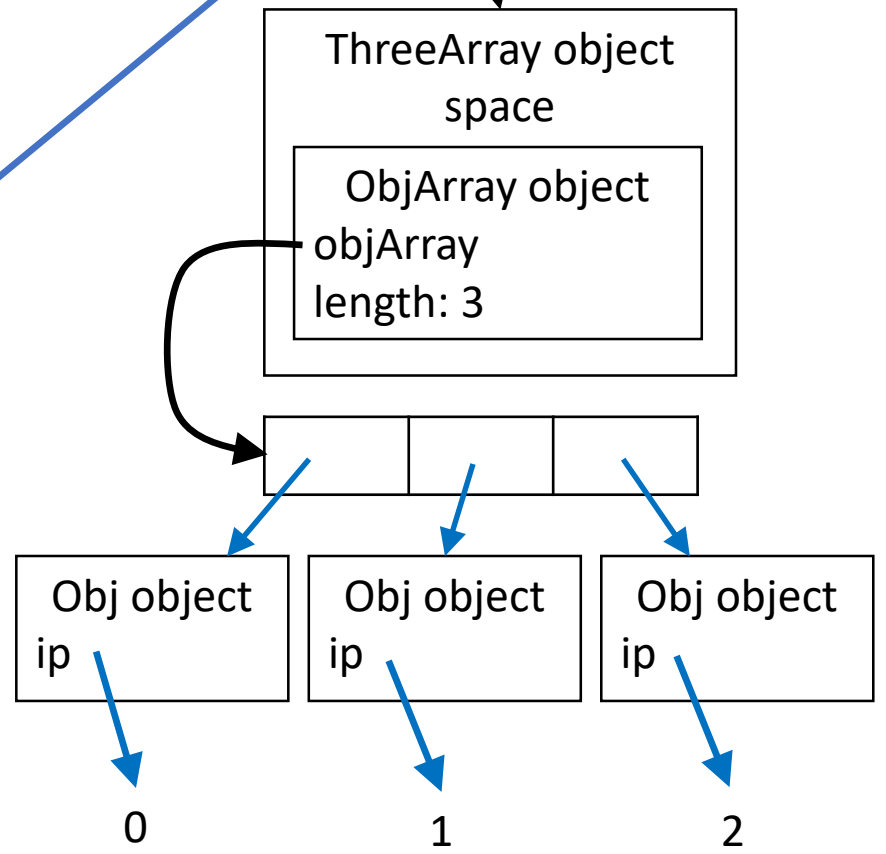
```
ObjArray::~ObjArray( ) {
  for (int i = 0; i < length; i++) {
    std::cout << "~ObjArray " << i ;
    delete objArray[i];
  }
  std::cout << "~ObjArray objArray" ;
  delete objArray;
}

class ThreeArray : public ObjArray {
public:
  ThreeArray( );
  virtual ~ThreeArray( );
};

ThreeArray::ThreeArray( ) : ObjArray(3) { }

ThreeArray::~ThreeArray( ) {
  std::cout << "~ThreeAway" << std::endl;
}
```

```
int main(int argc, char* argv[ ]) {
  ObjArray* threeArray = new ThreeArray( );
  std::cout << "deleting ThreeArray";
  delete threeArray;
}
```



# The usefulness of Destructors (See Example7 code)

```
Obj::Obj(int i) {
  ip = new int;
  *ip = i;
}

Obj::~Obj( ) {
  std::cout << "~Obj " << *ip;
  delete ip;
}

ObjArray::ObjArray(int len) {
  length = len;
  objArray = new Obj*[len];
  for (int i = 0; i < len; i++) {
    objArray[i] = new Obj(i);
  }
}
```

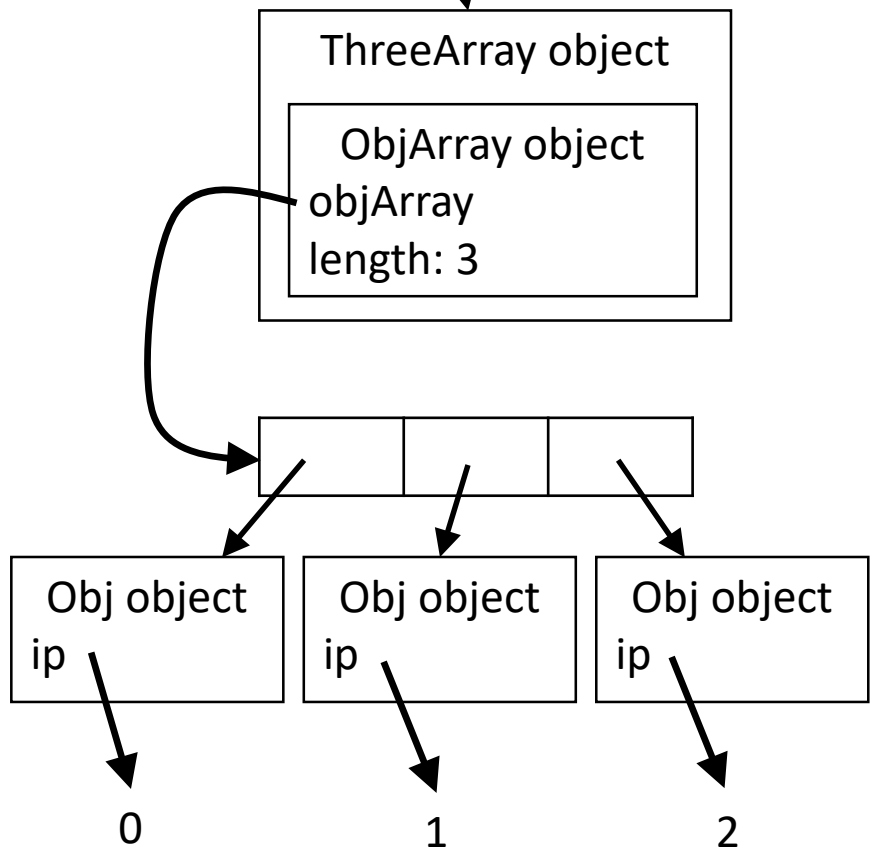
```
ObjArray::~ObjArray( ) {
  for (int i = 0; i < length; i++) {
    std::cout << "~ObjArray " << i ;
    delete objArray[i];
  }
  std::cout << "~ObjArray objArray" ;
  delete objArray;
}

class ThreeArray : public ObjArray {
public:
  ThreeArray( );
  virtual ~ThreeArray( );
};

ThreeArray::ThreeArray( ) : ObjArray(3) { }

ThreeArray::~ThreeArray( ) {
  std::cout << "~ThreeAway" << std::endl;
}
```

```
int main(int argc, char* argv[ ]) {
  ObjArray* threeArray = new ThreeArray( );
  std::cout << "deleting ThreeArray";
  delete threeArray;
}
```




deleting ThreeArray
~ThreeAway

# The usefulness of Destructors (See Example7 code)

```
Obj::Obj(int i) {
  ip = new int;
  *ip = i;
}

Obj::~Obj( ) {
  std::cout << "~Obj " << *ip;
  delete ip;
}

ObjArray::ObjArray(int len) {
  length = len;
  objArray = new Obj*[len];
  for (int i = 0; i < len; i++) {
    objArray[i] = new Obj(i);
  }
}
```

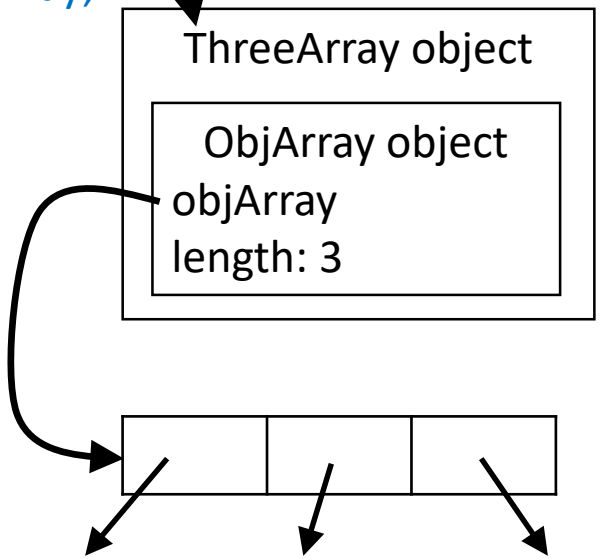
```
ObjArray::~ObjArray( ) {
  for (int i = 0; i < length; i++) {
    std::cout << "~ObjArray " << i ;
    delete objArray[i];
  }
  std::cout << "~ObjArray objArray" ;
  delete objArray;
}

class ThreeArray : public ObjArray {
public:
  ThreeArray( );
  virtual ~ThreeArray( );
};

ThreeArray::ThreeArray( ) : ObjArray(3) { }

ThreeArray::~ThreeArray( ) {
  std::cout << "~ThreeAway" << std::endl;
}
```

```
int main(int argc, char* argv[ ]) {
  ObjArray* threeArray = new ThreeArray( );
  std::cout << "deleting ThreeArray";
  delete threeArray;
}
```




```
deleting ThreeArray
~ThreeAway
~ObjArray 0
~Obj 0
~ObjArray 1
~Obj 1
~ObjArray 2
~Obj 2
```

# The usefulness of Destructors (See Example7 code)

```
Obj::Obj(int i) {
  ip = new int;
  *ip = i;
}

Obj::~Obj( ) {
  std::cout << "~Obj " << *ip;
  delete ip;
}

ObjArray::ObjArray(int len) {
  length = len;
  objArray = new Obj*[len];
  for (int i = 0; i < len; i++) {
    objArray[i] = new Obj(i);
  }
}
```

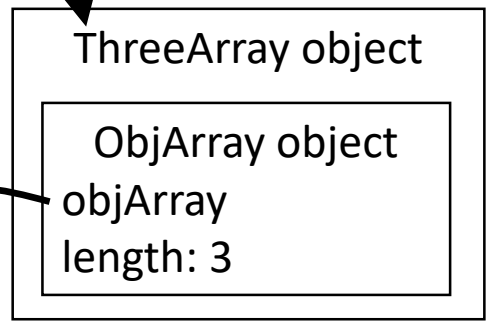
```
ObjArray::~ObjArray( ) {
  for (int i = 0; i < length; i++) {
    std::cout << "~ObjArray " << i ;
    delete objArray[i];
  }
  std::cout << "~ObjArray objArray" ;
  delete objArray;
}

class ThreeArray : public ObjArray {
public:
  ThreeArray( );
  virtual ~ThreeArray( );
};

ThreeArray::ThreeArray( ) : ObjArray(3) { }

ThreeArray::~ThreeArray( ) {
  std::cout << "~ThreeAway" << std::endl;
}
```

```
int main(int argc, char* argv[ ]) {
  ObjArray* threeArray = new ThreeArray( );
  std::cout << "deleting ThreeArray";
  delete threeArray;
}
```




```
deleting ThreeArray
~ThreeAway
~ObjArray 0
~Obj 0
~ObjArray 1
~Obj 1
~ObjArray 2
~Obj 2
~ObjArray objArray
```

# The usefulness of Destructors (See Example7 code)

```
Obj::Obj(int i) {
  ip = new int;
  *ip = i;
}

Obj::~Obj( ) {
  std::cout << "~Obj " << *ip;
  delete ip;
}

ObjArray::ObjArray(int len) {
  length = len;
  objArray = new Obj*[len];
  for (int i = 0; i < len; i++) {
    objArray[i] = new Obj(i);
  }
}
```

```
ObjArray::~ObjArray( ) {
  for (int i = 0; i < length; i++) {
    std::cout << "~ObjArray " << i ;
    delete objArray[i];
  }
  std::cout << "~ObjArray objArray" ;
  delete objArray;
}

class ThreeArray : public ObjArray {
public:
  ThreeArray( );
  virtual ~ThreeArray( );
};

ThreeArray::ThreeArray( ) : ObjArray(3) { }

ThreeArray::~ThreeArray( ) {
  std::cout << "~ThreeAway" << std::endl;
}
```

```
int main(int argc, char* argv[ ]) {
  ObjArray* threeArray = new ThreeArray( );
  std::cout << "deleting ThreeArray";
  delete threeArray;
}
```

nullptr

# What we have learned (1)

- Base class constructors are call first, then the constructor for the class that inherits from Base, and so forth to the constructor for the most derived class.
  - This insures that base class objects are all initialized when the most derived object is initialized
  - This allows the more derived constructors to use attributes and methods of the base classes without fear
- Initializer lists
  - provide place to call the immediate base class constructor
  - provide convenient place to initialize object attributes
  - Initializations are executed in the order the initialized attributes are declared
- Destructors are called in the opposite order, i.e., the most derived constructor is called first, down to the base constructor
  - This insures that base class objects' attributes and methods can be accessed while executing the derived classes' destructors

# What we have learned (2)

- Destructors should always be virtual
  - If this is not the case, then deleting an object pointed to by a base class pointer will only call the base class destructor
  - If the more derived class holds pointers to allocated memory, open files, etc., this can lead to resource leaks
- Copy constructors are called whenever C++ needs to make a copy of an object
  - The copy constructor for a class T has the signature `T(T&);`
  - If we do not provide a copy constructor, C++ will provide a default copy constructor that does a bit-for-bit copy
- Converting constructors are called whenever C++ needs to convert between an object of some type $T_{targ}$ and an object of type $T_{orig}$ or a primitive
  - Implicit copy constructors are called automatically without needing a cast
  - Explicit constructors require a cast or the compiler will give an error
  - Always use explicit constructors for safety unless you have a good reason not to (and I cannot think of a good reason not to)