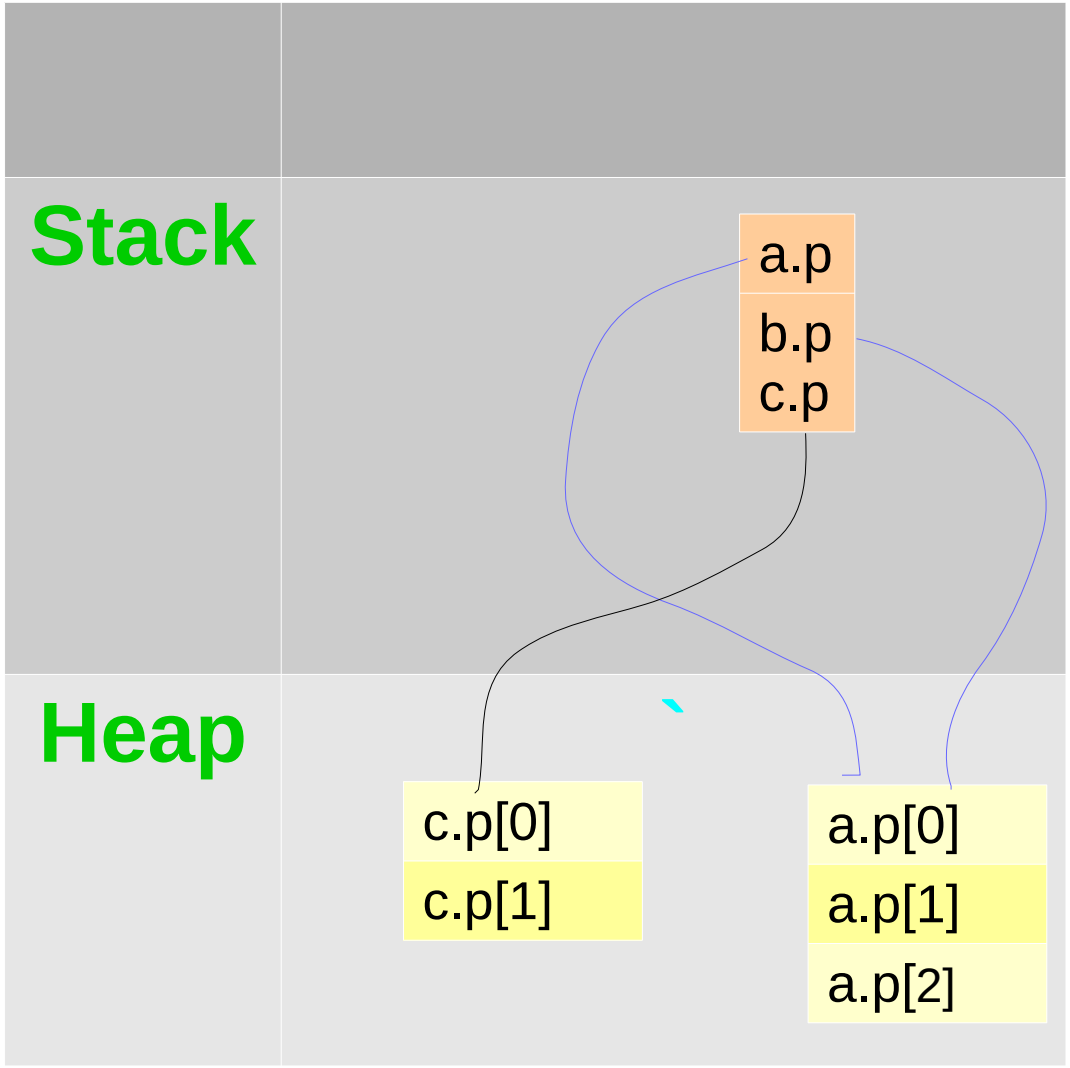
Copying
Conversions
Friends

- Version 1: Dr. Ofir Pele
- Version 2: Dr. Erel Segal-Halevi

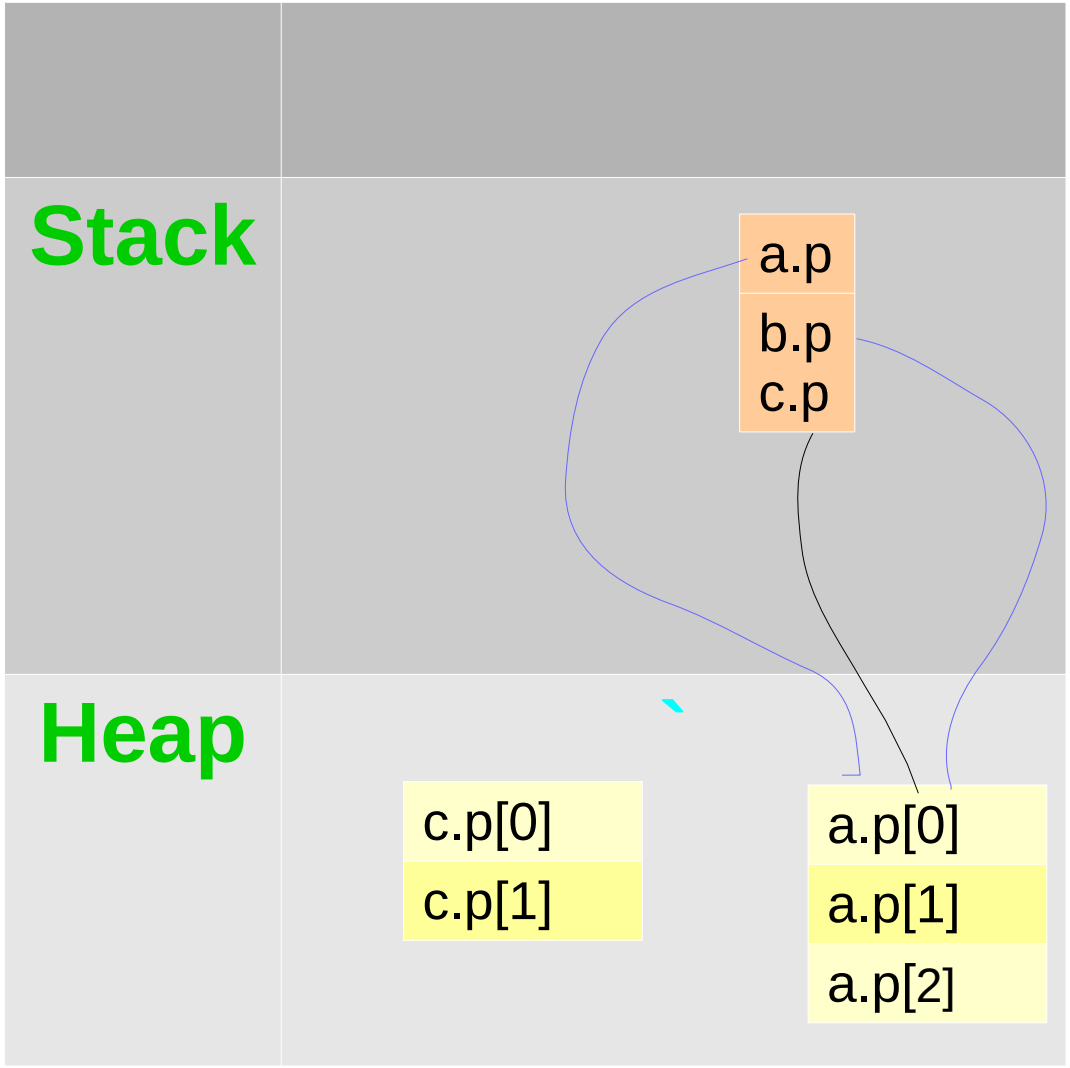
Shallow vs. Deep Copy *(folder 1)*

```
class IntList {
   int* p;
public:
   IntList(uint n):
      p(new int[n]) {  }
   ~IntList() {
      delete[] p;  }
};

int main() {
   IntList a(3);
   IntList b=a;
   IntList c(2);
}
```



Shallow vs. Deep Copy

```
class IntList {
   int* p;
public:
   IntList(uint n):
      p(new int[n]) {  }
   ~IntList() {
      delete[] p;  }
};

int main() {
   IntList a(3);
   IntList b=a;
   IntList c(2); c=a;
}
```



Copying

An object is copied when:

1. Constructing new object from existing
2. Passing parameter by value.
3. Returning by value.
4. Assigning existing to existing.

Cases 1-3 are handled by

copy constructor.

Case 4 is handled by

assignment operator.

By default, both do **shallow copy**.

Rule of Three

- A rule of thumb:
  - When you need to make a deep copy of an object, you need to define all of these:
    1. Copy constructor
    2. Destructor
    3. Operator =
  - Or in other words:

    when you need one, you need all.

A skeleton for deep copy

```
// Copy constructor
A (const A& other) : init {
    copy_other(other);
}
```

```
// Destructor
~A() {
    clear();
}
```

```
// Operator =
A& operator=(const A& other) {
    if (this!=&other) { // preventing problems in a=a
      clear(); init // or recycle
      copy_other(other);
    } return *this; } // allows a= b= c= …
```

IntBuffer example *(folder 2)*

Conversions of types is done in two cases:

1. Explicit casting (we'll learn more about it in next lessons)
2. When a function gets **X** type while it was expecting to get **Y** type, and there is a casting from **X** to **Y**:

```
void foo(Y y)
...
X x;
foo(x); // a conversion from X to Y is done
```

Conversions danger: unexpected behavior

```
Vector(size_t length) // ctor
…
int sum(const Vector& v) // function
...
int i=3;
sum(i); // Equivalent to: sum(Vector(i))
// Did the user really wanted this?
```

The Vector and the size_t objects are not logically the same objects!

Conversions danger: unexpected behavior

```
explicit Vector(size_t length) // ctor
…
int sum(const Vector& v) // function
...
int i=3;
sum(i); // Won’t compile
```

Conversion example (folder 4)

User defined conversion (folders 5,6)

```
class Fraction {
    ...
    // double --> Fraction conversion
    Fraction (const double& d) {
        ...
    }
    ...
    // Fraction --> double conversion
    operator double() const {
        ...
    }
```

Operator Suffix (folder 8)

Operator Comma (folder 9)

friend

friend functions

Friend function in a class:

- Not a method of the class
- Have access to the class's private and protected data members
- Defined inside the class scope

Used properly does not break encapsulation

friend functions example: Complex revisited

friend classes

- A class can allow other classes to access its private data members
- *QUESTION: Is the friendship link one-sided or two-sided? I.e:*
  - *Suppose class A is a friend of class B.*
  - *Does it mean that class B is a friend of A?*

friend classes - example

```
class IntTree {
    ...
    friend class IntTreeIterator;
};


// TreeIterator can access Tree's data members
IntTreeIterator& IntTreeIterator::operator++() {
    ...
    return *this;
}
```