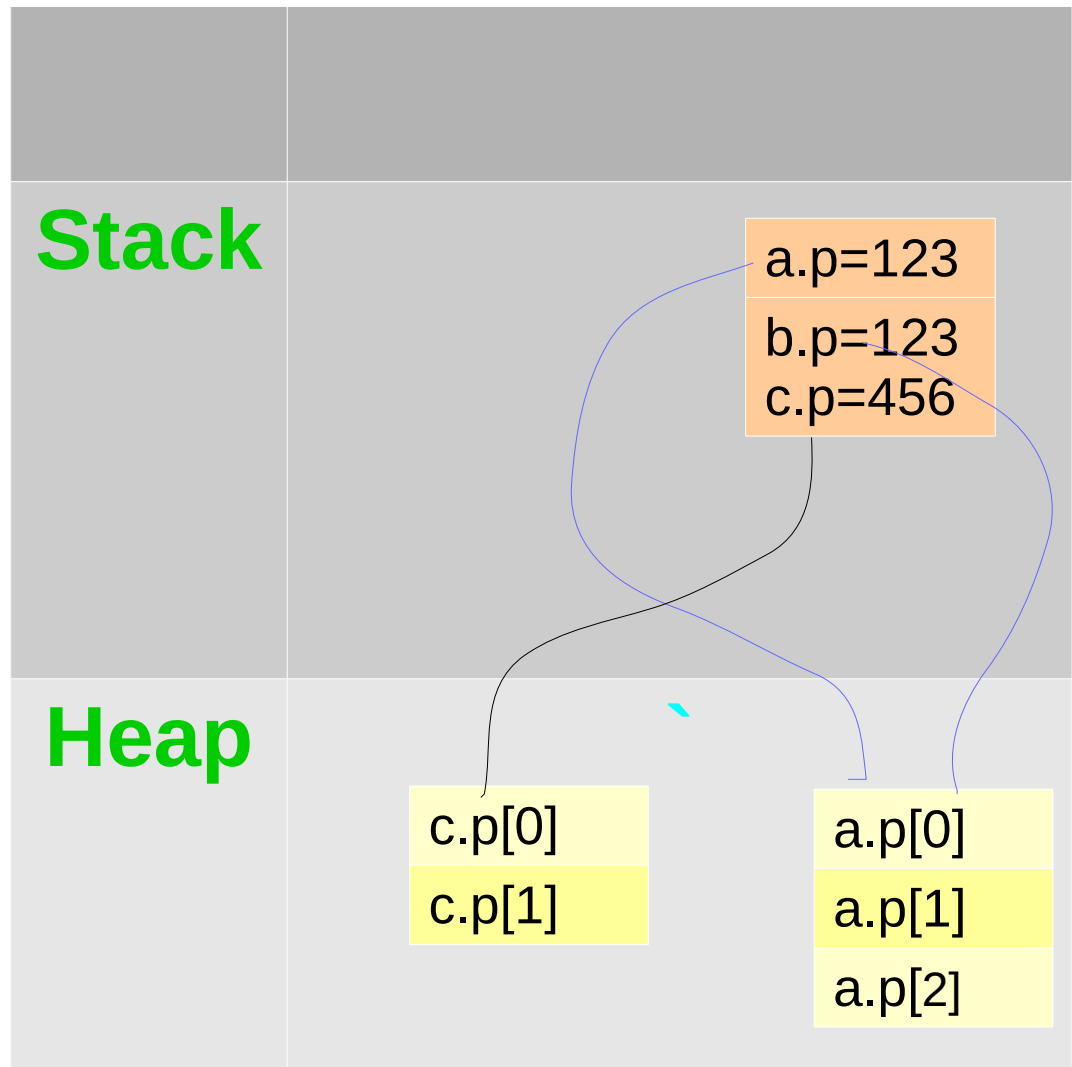


C++: copy, assign, convert

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

Shallow vs. Deep Copy (*folder 1*)

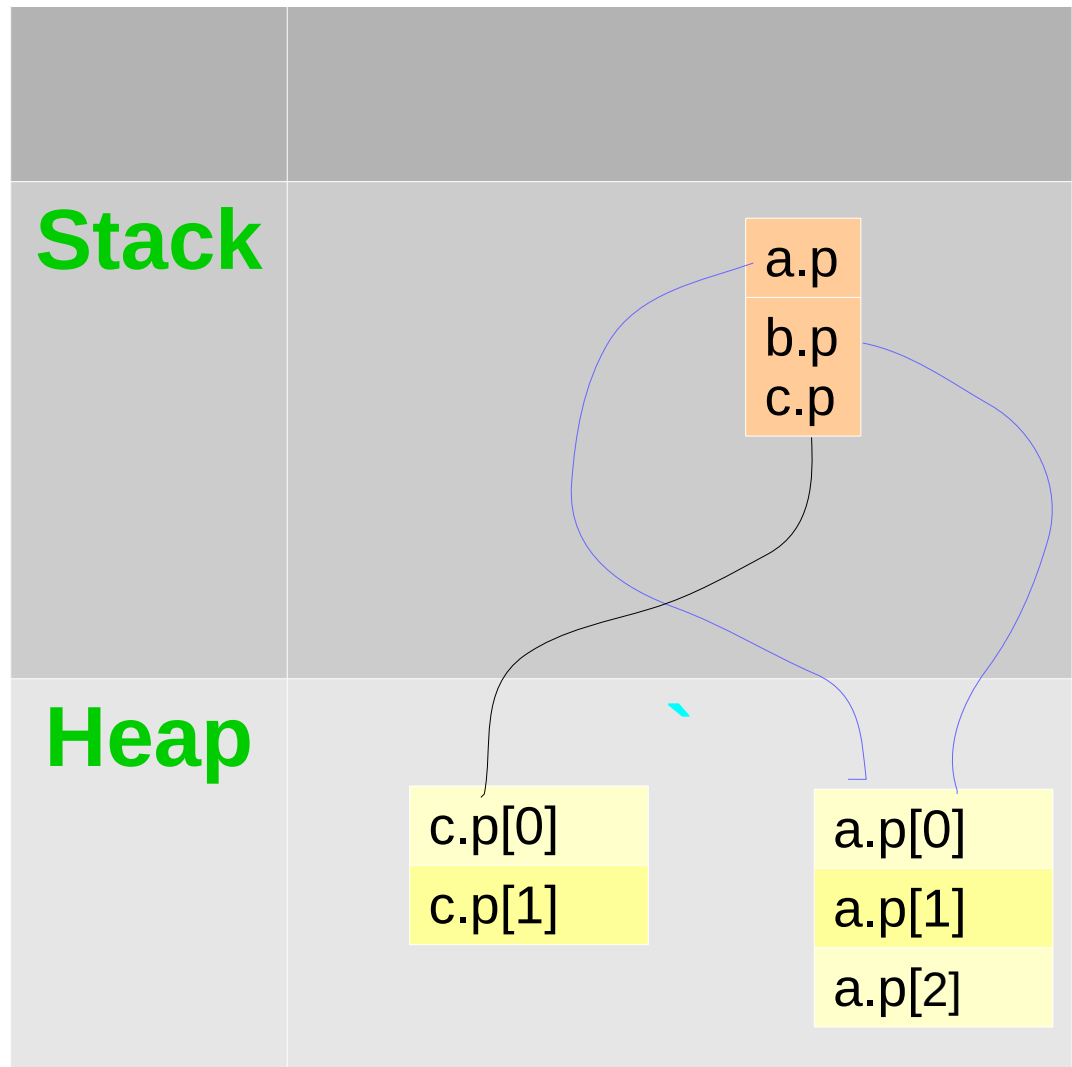
```
class IntList {  
    int* p;  
public:  
    IntList(uint n):  
        p(new int[n]) { }  
    ~IntList() {  
        if (!p) delete[] p;  
        p = nullptr; }  
};  
int main() {  
    IntList a(3);  
    IntList b=a;  
    IntList c(2);  
}
```



Shallow vs. Deep Copy (*folder 1*)

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

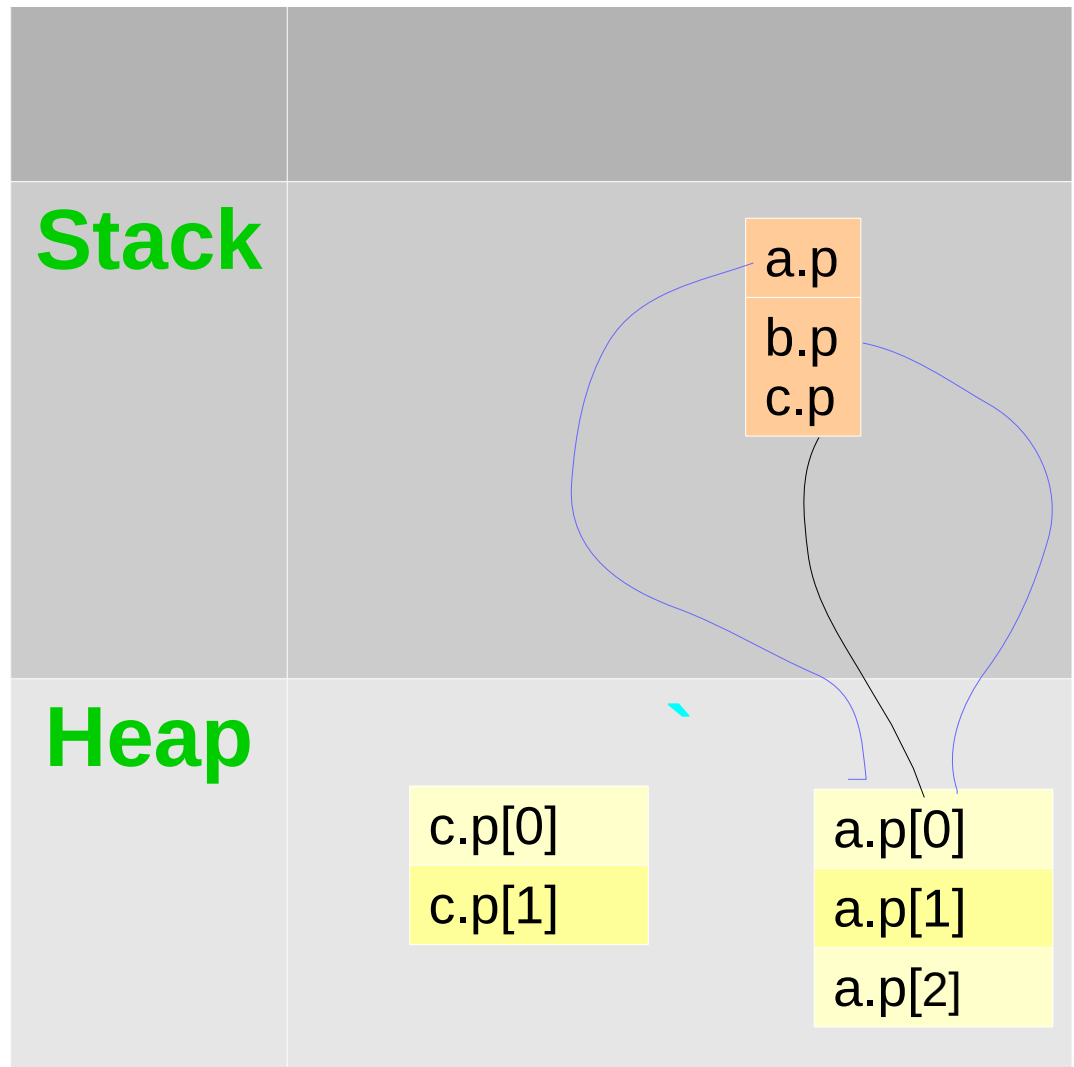
```
int main() {  
    int* p = new int[5];  
    int* q = p;  
    delete[] q; // OK  
    delete[] p; // Error  
}
```



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;  
    // c.p = a.p  
}
```



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 (folder 1)

// 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= ...
```

Conversions of types

done in two cases:

1. Explicit casting;
2. Implicit casting - 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
```


User defined conversion (folders 5,6)

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

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!

The `explicit` keyword (folder 4)

```
explicit Vector(size_t length) // ctor
```

```
...
```

```
int sum(const Vector& v) // function
```

```
...
```

```
int i=3;
```

```
sum(i); // Won't compile
```

```
sum(Vector{i}); // compiles
```