



# 6.096 Lecture 8: Memory Management

Clean up after your pet program

Geza Kovacs

# Review: Constructors

- ✗ Method that is called when an instance is created

```
class Integer {  
public:  
    int val;  
    Integer() {  
        val = 0; cout << "default constructor" << endl;  
    }  
};
```

```
int main() {  
    Integer i;  
}
```

**Output:**  
default constructor

- When making an array of objects, default constructor is invoked on each

```
class Integer {  
public:  
    int val;  
    Integer() {  
        val = 0; cout << "default constructor" << endl;  
    }  
};
```

```
int main() {  
    Integer arr[3];  
}
```

**Output:**

default constructor  
default constructor  
default constructor

- When making a class instance, the default constructor of its fields are invoked

```
class Integer {
public:
    int val;
    Integer() {
        val = 0; cout << "Integer default constructor" << endl;
    }
};

class IntegerWrapper {
public:
    Integer val;
    IntegerWrapper() {
        cout << "IntegerWrapper default constructor" << endl;
    }
};

int main() {
    IntegerWrapper q;
}
```

**Output:**

Integer default constructor

IntegerWrapper default constructor

- Constructors can accept parameters

```
class Integer {  
public:  
    int val;  
    Integer(int v) {  
        val = v; cout << "constructor with arg " << v << endl;  
    }  
};
```

```
int main() {  
    Integer i(3);  
}
```

**Output:**  
constructor with arg 3

- Constructors can accept parameters
  - Can invoke single-parameter constructor via assignment to the appropriate type

```
class Integer {  
public:  
    int val;  
    Integer(int v) {  
        val = v; cout << "constructor with arg " << v << endl;  
    }  
};
```

```
int main() {  
    Integer i(3);  
    Integer j = 5;  
}
```

two ways of doing  
it

### Output:

constructor with arg 3  
constructor with arg 5

- If a constructor with parameters is defined, the default constructor is no longer available

```
class Integer {  
public:  
    int val;  
    Integer(int v) {  
        val = v;  
    }  
};
```

```
int main() {  
    Integer i(3); // ok  
    Integer j;  
}
```



Error: No default constructor available for Integer

- If a constructor with parameters is defined, the default constructor is no longer available
- X (– Without a default constructor, can't declare arrays without initializing)

```
class Integer {  
public:  
    int val;  
    Integer(int v) {  
        val = v;  
    }  
};
```

```
int main() {  
    Integer a[] = { Integer(2), Integer(5) }; // ok  
    Integer b[2];  
}
```

Error: No default constructor available for Integer



- If a constructor with parameters is defined, the default constructor is no longer available
  - Can create a separate 0-argument constructor

```
class Integer {
public:
    int val;
    Integer() {
        val = 0;
    }
    Integer(int v) {
        val = v;
    }
};

int main() {
    Integer i; // ok
    Integer j(3); // ok
}
```

- If a constructor with parameters is defined, the default constructor is no longer available
  - Can create a separate 0-argument constructor
  - Or, use default arguments

```
class Integer {  
public:  
    int val;  
    Integer(int v = 0) {  
        val = v;  
    }  
};
```

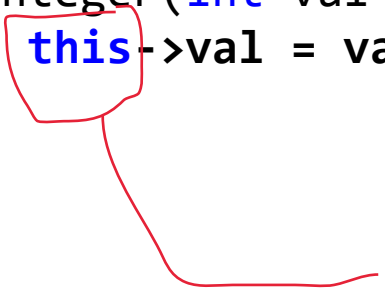
```
int main() {  
    Integer i; // ok  
    Integer j(3); // ok  
}
```

when nothing is  
passed v is  
initialized a 0.

when a value is  
passed the zero  
initialisation is  
bypassed.

- How do I refer to a field when a method argument has the same name?
- **this**: a pointer to the current instance

```
class Integer {           // defining a class
public:
    int val;
    Integer(int val = 0) { // element of class    initializing element
        this->val = val;
    }
};
```

 **this** is a shorthand for `(*this).val`

an inbuilt pointer for current instance

- How do I refer to a field when a method argument has the same name?
- this: a pointer to the current instance

```
class Integer {  
public:  
    int val;  
    Integer(int val = 0) {  
        this->val = val;  
    }  
    void setVal(int val) {  
        this->val = val;  
    }  
};
```

inside a function if i have to use a field variable of class  
may be later if i do manipulation inside the function , it wont  
effect the field variable of class

# Scoping and Memory

- Whenever we declare a new variable (int x), memory is allocated
- When can this memory be freed up (so it can be used to store other variables)?
  - When the variable goes out of scope

# Scoping and Memory

- When a variable goes out of scope, that memory is no longer guaranteed to store the variable's value

```
int main() {  
    if (true) {  
        int x = 5;  
    }  
    // x now out of scope, memory it used to occupy can be reused  
}
```

# Scoping and Memory

- When a variable goes out of scope, that memory is no longer guaranteed to store the variable's value

```
int main() {  
    int *p;  
    if (true) {  
        int x = 5;  
        p = &x;  
    }  
    cout << *p << endl; // ???  
}
```

# Scoping and Memory

- When a variable goes out of scope, that memory is no longer guaranteed to store the variable's value

```
int main() {  
    int *p;  
    if (true) {  
        int x = 5;  
        p = &x;  
    }  
    cout << *p << endl; // ???  
}
```




int \*p



# Scoping and Memory

- When a variable goes out of scope, that memory is no longer guaranteed to store the variable's value

```
int main() {  
    int *p;  
    if (true) {  
        int x = 5;   
        p = &x;  
    }  
    cout << *p << endl; // ???  
}
```

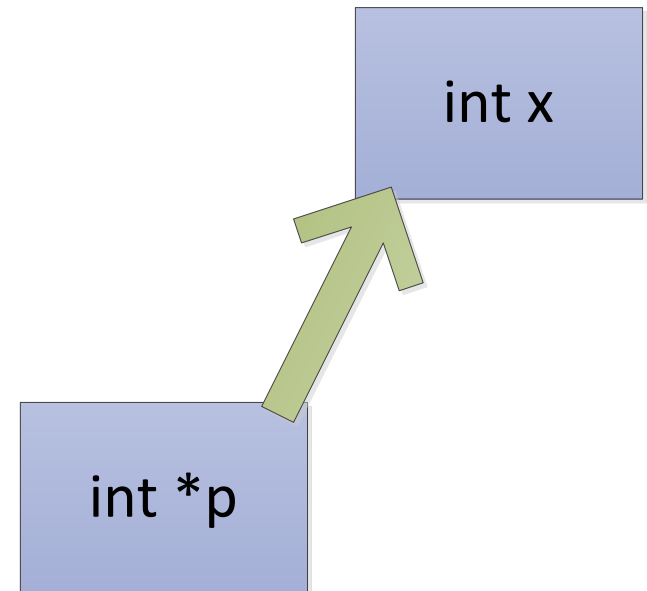
int x

int \*p

# Scoping and Memory

- When a variable goes out of scope, that memory is no longer guaranteed to store the variable's value

```
int main() {  
    int *p;  
    if (true) {  
        int x = 5;  
        p = &x;  
    }  
    cout << *p << endl; // ???  
}
```



# Scoping and Memory

- When a variable goes out of scope, that memory is no longer guaranteed to store the variable's value
  - Here, p has become a **dangling pointer** (points to memory whose contents are undefined)

```
int main() {  
    int *p;  
    if (true) {  
        int x = 5;  
        p = &x;  
    }  
    cout << *p << endl; // ???  
}
```

need clarity

even if 'if (true)' is removed and only with curly brackets . definition goes out of scope.

???

here

int \*p

# A Problematic Task

- Implement a function which returns a pointer to some memory containing the integer 5
- Incorrect implementation:

```
int* getPtrToFive() {  
    int x = 5;  
    return &x;  
}
```

- Implement a function which returns a pointer to some memory containing the integer 5
- Incorrect implementation:
  - x is declared in the function scope

```
int* getPtrToFive() {  
    int x = 5;  
    return &x;  
}
```



int x

```
int main() {  
    int *p = getPtrToFive();  
    cout << *p << endl; // ???  
}
```

- Implement a function which returns a pointer to some memory containing the integer 5
- Incorrect implementation:
  - x is declared in the function scope
  - As `getPtrToFive()` returns, x goes out of scope. So a dangling pointer is returned

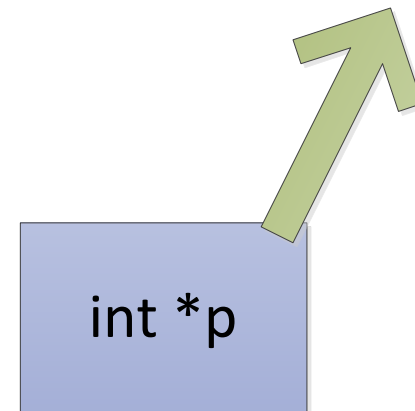
```
int* getPtrToFive() {  
    int x = 5;  
    return &x;  
}
```



```
int main() {  
    int *p = getPtrToFive();  
    cout << *p << endl; // ???  
}
```

this code works fine in the vs 19  
will have to clarify.

???



# The **new** operator

- Another way to allocate memory, where the memory will remain allocated until you manually de-allocate it
- Returns a pointer to the newly allocated memory

```
int *x = new int;
```

# The **new** operator

- Another way to allocate memory, where the memory will remain allocated until you manually de-allocate it
- Returns a pointer to the newly allocated memory

```
int *x = new int;
```

Type parameter needed to determine how much memory to allocate



# The **new** operator

- Another way to allocate memory, where the memory will remain allocated until you manually de-allocate it
- Returns a pointer to the newly allocated memory
- Terminology note:
  - If using **int x**; the allocation occurs on a region of memory called **the stack**
  - If using **new int**; the allocation occurs on a region of memory called **the heap**

# The **delete** operator

- De-allocates memory that was previously allocated using **new**
- Takes a pointer to the memory location

```
int *x = new int;  
// use memory allocated by new  
delete x;
```

- Implement a function which returns a pointer to some memory containing the integer 5
  - Allocate memory using **new** to ensure it remains allocated

```
int *getPtrToFive() {  
    int *x = new int;  
    *x = 5;  
    return x;  
}
```

- Implement a function which returns a pointer to some memory containing the integer 5
  - Allocate memory using **new** to ensure it remains allocated.
  - When done, de-allocate the memory using **delete**

```
int *getPtrToFive() {  
    int *x = new int;  
    *x = 5;  
    return x;  
}  
  
int main() {  
    int *p = getPtrToFive();  
    cout << *p << endl; // 5  
    delete p;  
}
```



this code is working fine without using new int.  
variables declared inside the curly brackets not accessible even after using new int



important to think about data transfer in pointers here

# Delete Memory When Done Using It

- If you don't use de-allocate memory using **delete**, your application will waste memory

```
int *getPtrToFive() {  
    int *x = new int;  
    *x = 5;  
    return x;  
}
```

```
int main() {  
    int *p;  
    for (int i = 0; i < 3; ++i) {  
        p = getPtrToFive();  
        cout << *p << endl;  
    }  
}
```



incorrect

- If you don't use de-allocate memory using **delete**, your application will waste memory

```
int *getPtrToFive() {  
    int *x = new int;  
    *x = 5;  
    return x;  
}
```

```
int main() {  
    int *p;  
    for (int i = 0; i < 3; ++i) {  
        p = getPtrToFive();  
        cout << *p << endl;  
    }  
}
```

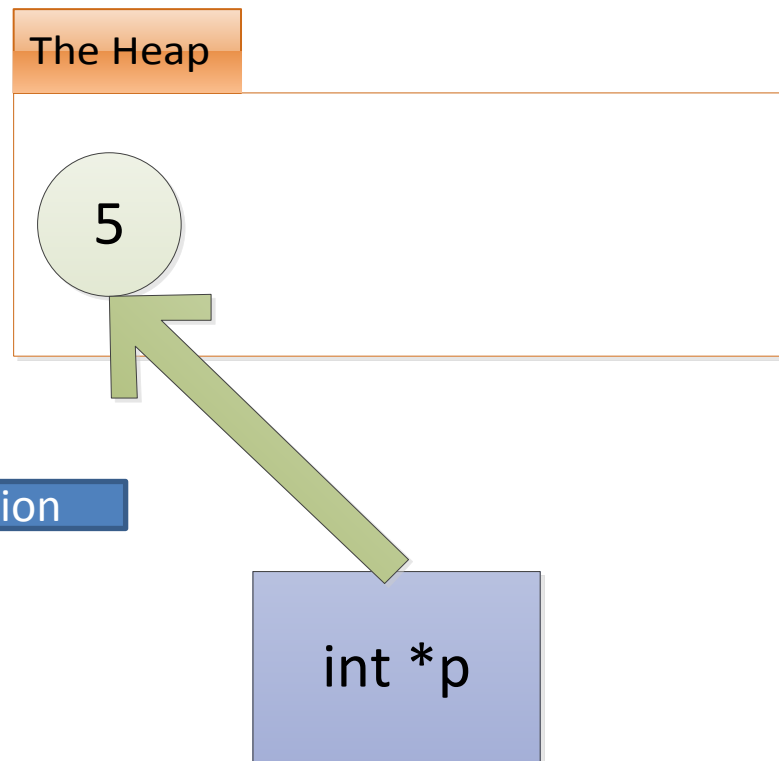


int \*p

- If you don't use de-allocate memory using **delete**, your application will waste memory

```
int *getPtrToFive() {  
    int *x = new int;  
    *x = 5;  
    return x;  
}
```

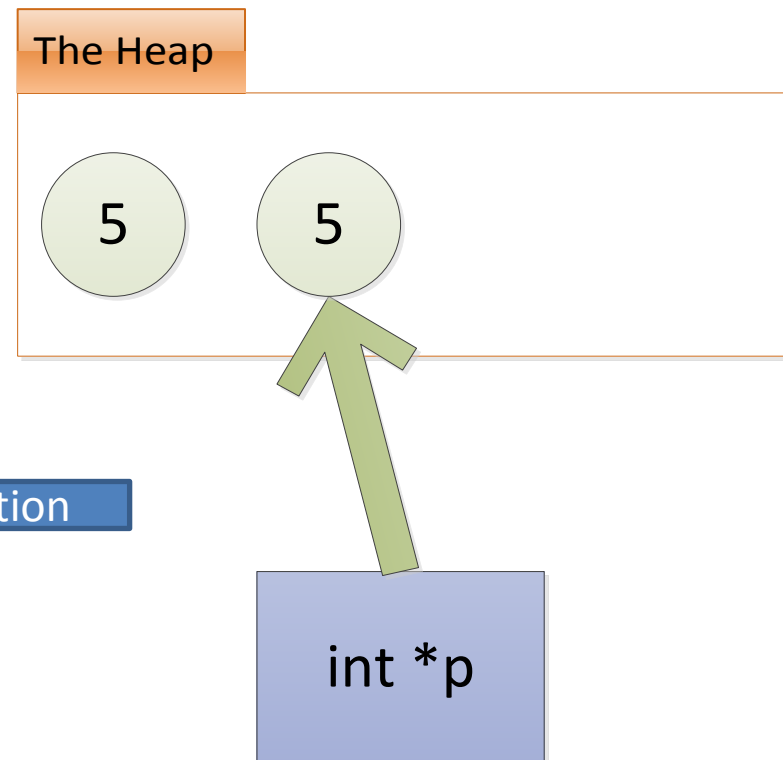
```
int main() {  
    int *p;  
    for (int i = 0; i < 3; ++i) {  
        p = getPtrToFive();  
        cout << *p << endl;  
    }  
}
```



- If you don't use de-allocate memory using **delete**, your application will waste memory

```
int *getPtrToFive() {  
    int *x = new int;  
    *x = 5;  
    return x;  
}
```

```
int main() {  
    int *p;  
    for (int i = 0; i < 3; ++i) {  
        p = getPtrToFive();  
        cout << *p << endl;  
    }  
}
```

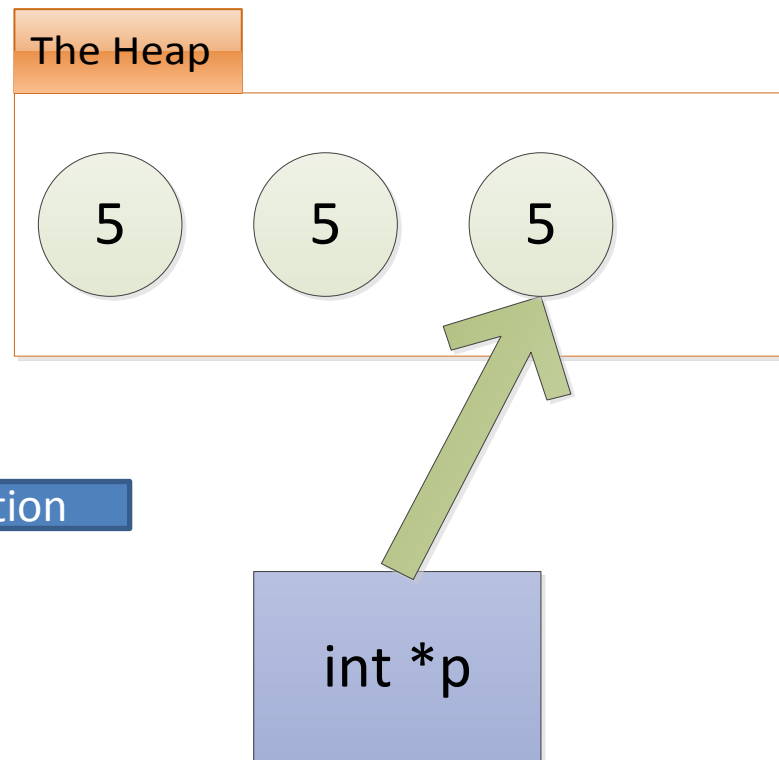




- If you don't use de-allocate memory using **delete**, your application will waste memory
- When your program allocates memory but is unable to de-allocate it, this is a **memory leak**

```
int *getPtrToFive() {  
    int *x = new int;  
    *x = 5;  
    return x;  
}
```

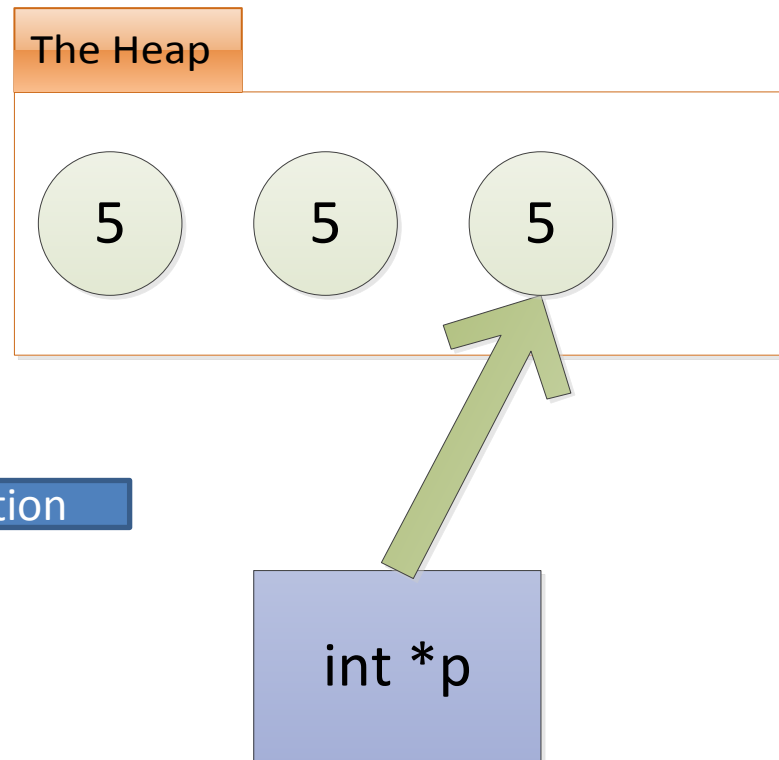
```
int main() {  
    int *p;  
    for (int i = 0; i < 3; ++i) {  
        p = getPtrToFive();  
        cout << *p << endl;  
    }  
}
```



- Does adding a delete after the loop fix this memory leak?

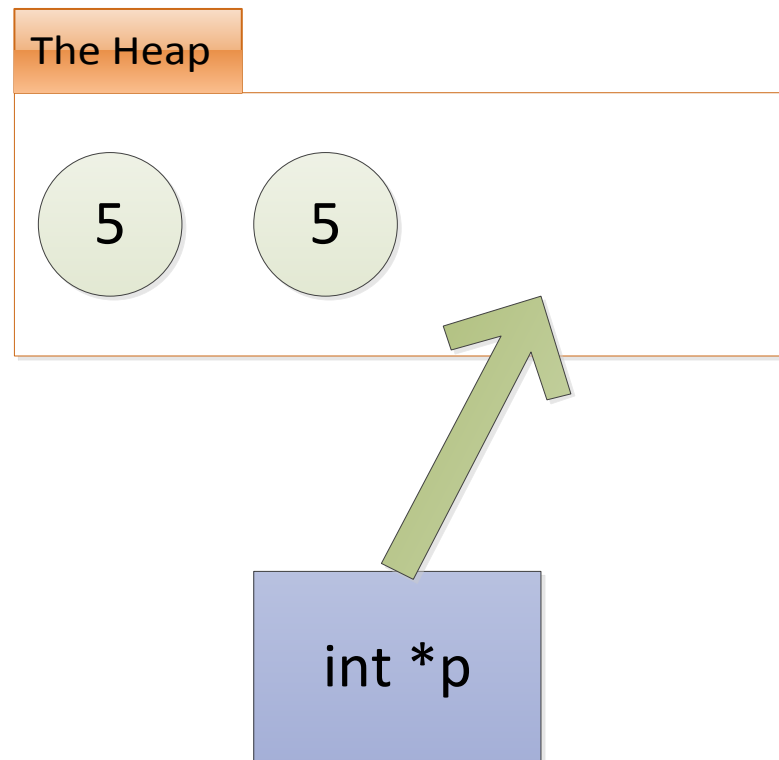
```
int *getPtrToFive() {  
    int *x = new int;  
    *x = 5;  
    return x;  
}
```

```
int main() {  
    int *p;  
    for (int i = 0; i < 3; ++i) {  
        p = getPtrToFive();  
        cout << *p << endl;  
    }  
    delete p;  
}
```



- Does adding a delete after the loop fix this memory leak?
  - No; only the memory that was allocated on the last iteration gets de-allocated

```
int *getPtrToFive() {  
    int *x = new int;  
    *x = 5;  
    return x;  
}  
  
int main() {  
    int *p;  
    for (int i = 0; i < 3; ++i) {  
        p = getPtrToFive();  
        cout << *p << endl;  
    }  
    delete p;  
}
```



- To fix the memory leak, de-allocate memory within the loop


```
int *getPtrToFive() {  
    int *x = new int;  
    *x = 5;  
    return x;  
}
```

```
int main() {  
    int *p;  
    for (int i = 0; i < 3; ++i) {  
        p = getPtrToFive();  
        cout << *p << endl;  
        delete p;  
    }  
}
```

- To fix the memory leak, de-allocate memory within the loop

```
int *getPtrToFive() {  
    int *x = new int;  
    *x = 5;  
    return x;  
}
```

```
int main() {  
    int *p;  
    for (int i = 0; i < 3; ++i) {  
        p = getPtrToFive();  
        cout << *p << endl;  
        delete p;  
    }  
}
```

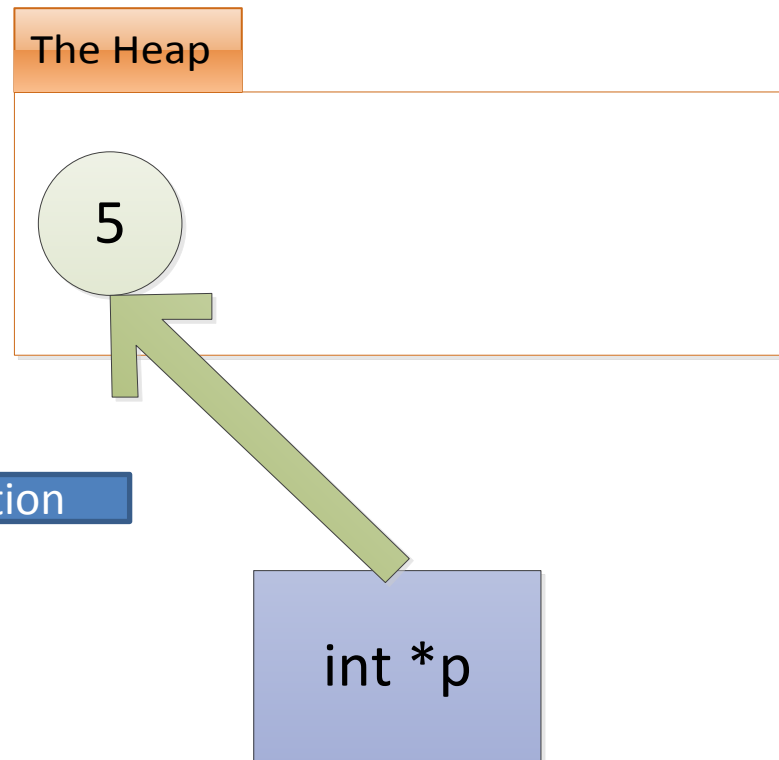


int \*p

- To fix the memory leak, de-allocate memory within the loop

```
int *getPtrToFive() {  
    int *x = new int;  
    *x = 5;  
    return x;  
}
```

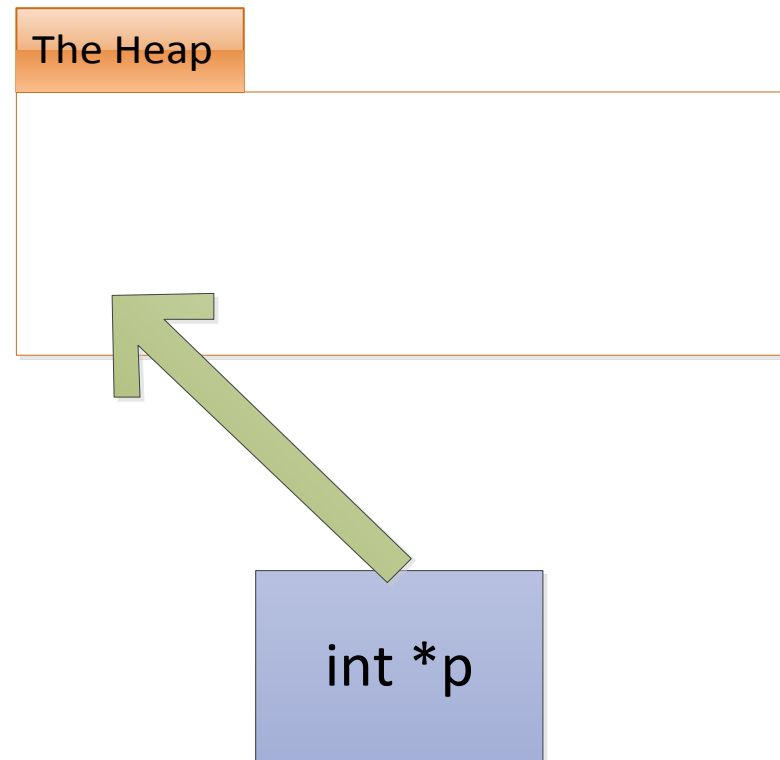
```
int main() {  
    int *p;  
    for (int i = 0; i < 3; ++i) {  
        p = getPtrToFive();  
        cout << *p << endl;  
        delete p;  
    }  
}
```



- To fix the memory leak, de-allocate memory within the loop

```
int *getPtrToFive() {  
    int *x = new int;  
    *x = 5;  
    return x;  
}  
  
int main() {  
    int *p;  
    for (int i = 0; i < 3; ++i) {  
        p = getPtrToFive();  
        cout << *p << endl;  
        delete p;   
    }  
}
```

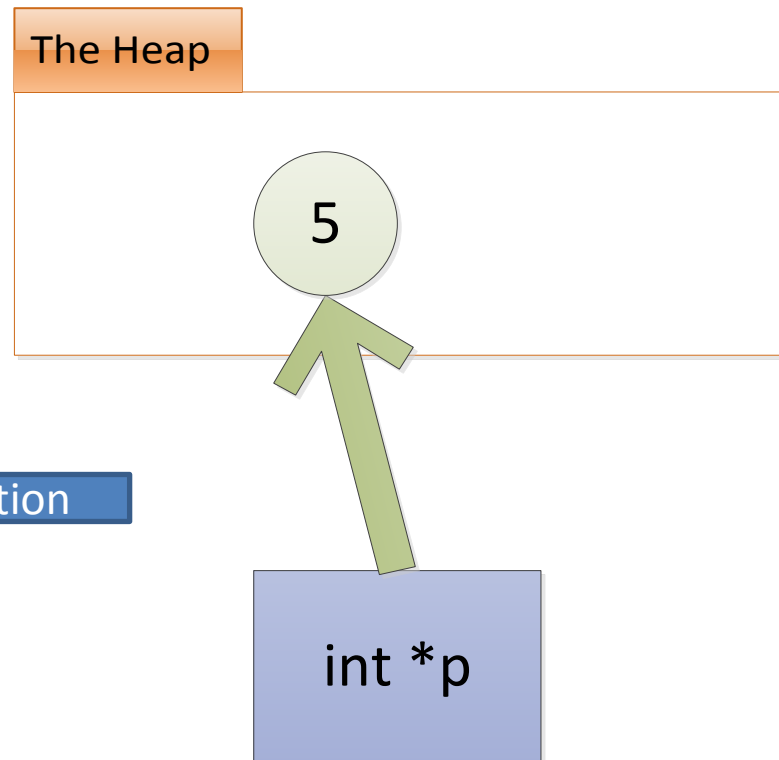
← 1<sup>st</sup> iteration



- To fix the memory leak, de-allocate memory within the loop

```
int *getPtrToFive() {  
    int *x = new int;  
    *x = 5;  
    return x;  
}
```

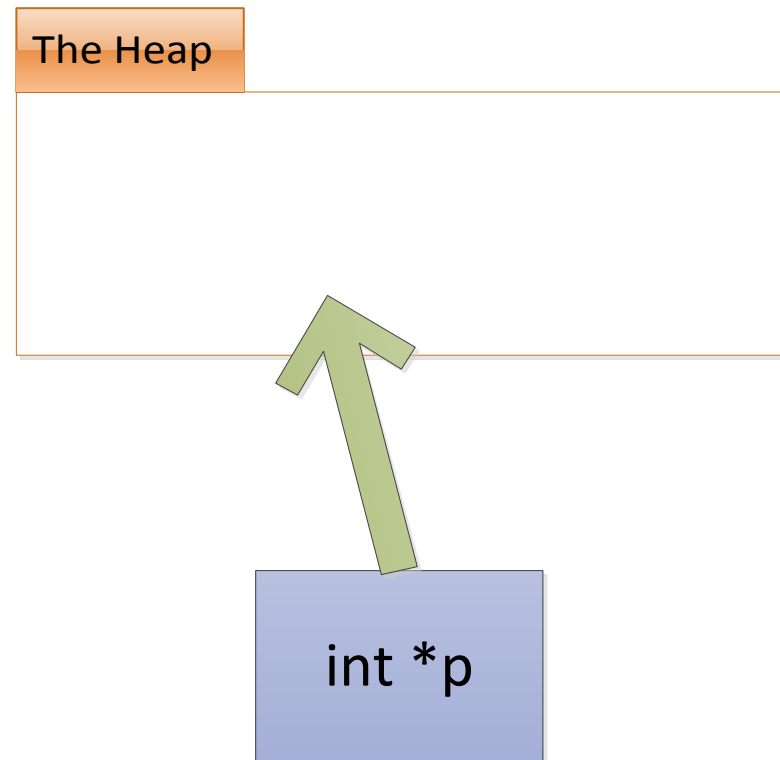
```
int main() {  
    int *p;  
    for (int i = 0; i < 3; ++i) {  
        p = getPtrToFive();  
        cout << *p << endl;  
        delete p;  
    }  
}
```





- To fix the memory leak, de-allocate memory within the loop

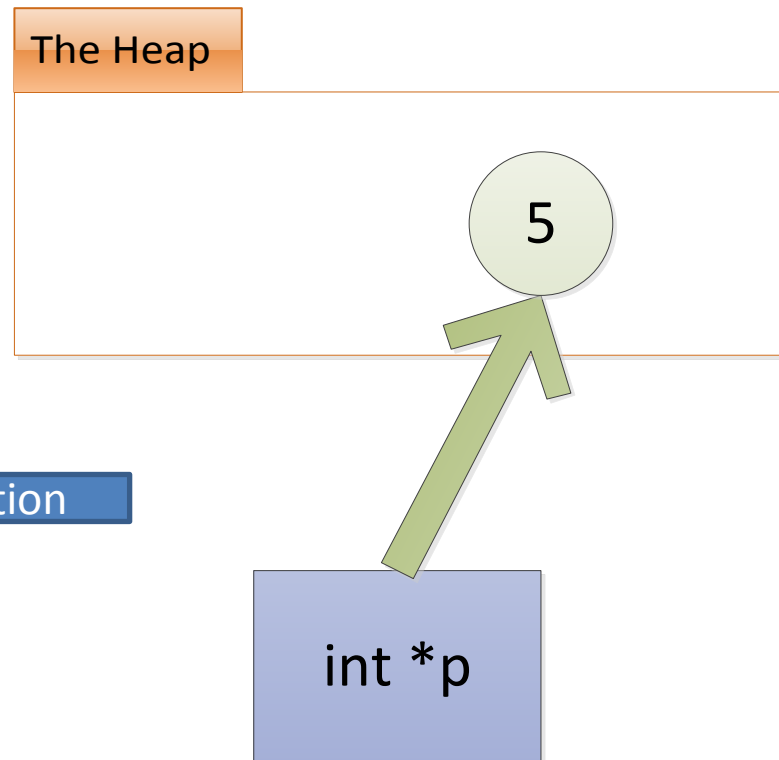
```
int *getPtrToFive() {  
    int *x = new int;  
    *x = 5;  
    return x;  
}  
  
int main() {  
    int *p;  
    for (int i = 0; i < 3; ++i) {  
        p = getPtrToFive();  
        cout << *p << endl;  
        delete p;  
    }  
}
```



- To fix the memory leak, de-allocate memory within the loop

```
int *getPtrToFive() {  
    int *x = new int;  
    *x = 5;  
    return x;  
}
```

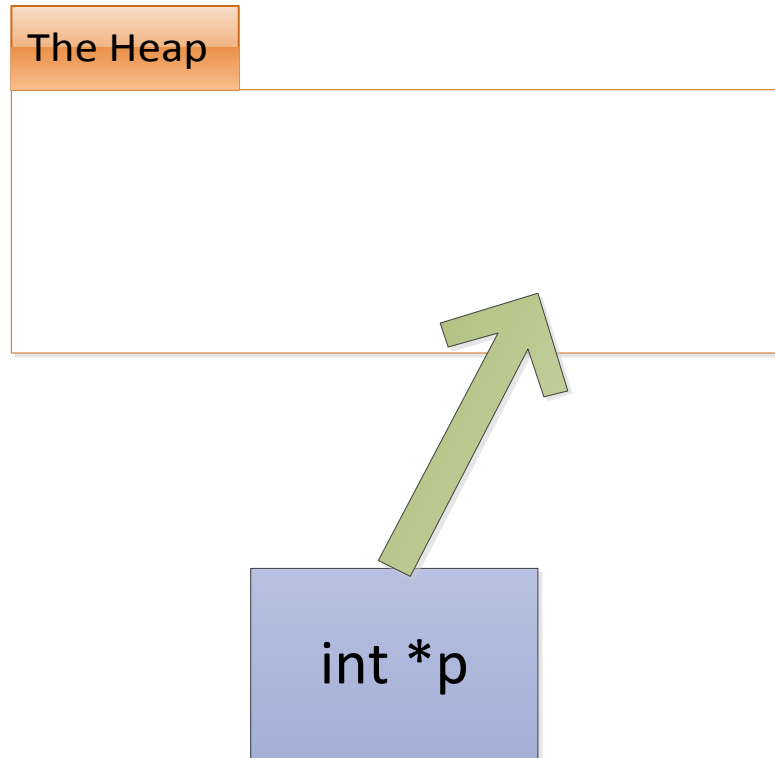
```
int main() {  
    int *p;  
    for (int i = 0; i < 3; ++i) {  
        p = getPtrToFive();  
        cout << *p << endl;  
        delete p;  
    }  
}
```



- To fix the memory leak, de-allocate memory within the loop

```
int *getPtrToFive() {  
    int *x = new int;  
    *x = 5;  
    return x;  
}  
  
int main() {  
    int *p;  
    for (int i = 0; i < 3; ++i) {  
        p = getPtrToFive();  
        cout << *p << endl;  
        delete p;   
    }  
}
```

3<sup>rd</sup> iteration

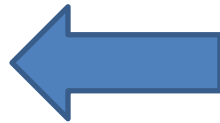


# Don't Use Memory After Deletion

incorrect

```
int *getPtrToFive() {  
    int *x = new int;  
    *x = 5;  
    return x;  
}
```

```
int main() {  
    int *x = getPtrToFive();  
    delete x;  
    cout << *x << endl; // ???  
}
```



# Don't Use Memory After Deletion

incorrect

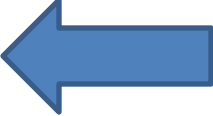
```
int *getPtrToFive() {  
    int *x = new int;  
    *x = 5;  
    return x;  
}  
  
int main() {  
    int *x = getPtrToFive();  
    delete x;  
    cout << *x << endl; // ???  
}
```

correct

```
int *getPtrToFive() {  
    int *x = new int;  
    *x = 5;  
    return x;  
}  
  
int main() {  
    int *x = getPtrToFive();  
    cout << *x << endl; // 5  
    delete x;  
}
```

# Don't delete memory twice

incorrect

```
int *getPtrToFive() {  
    int *x = new int;  
    *x = 5;  
    return x;  
}  
  
int main() {  
    int *x = getPtrToFive();  
    cout << *x << endl; // 5  
    delete x;  
    delete x;   
}
```

# Don't delete memory twice

incorrect

```
int *getPtrToFive() {  
    int *x = new int;  
    *x = 5;  
    return x;  
}  
  
int main() {  
    int *x = getPtrToFive();  
    cout << *x << endl; // 5  
    delete x;  
    delete x;  
}
```

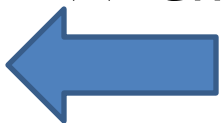
correct

```
int *getPtrToFive() {  
    int *x = new int;  
    *x = 5;  
    return x;  
}  
  
int main() {  
    int *x = getPtrToFive();  
    cout << *x << endl; // 5  
    delete x;  
}
```

# Only **delete** if memory was allocated by **new**

incorrect

```
int main() {  
    int x = 5;  
    int *xPtr = &x;  
    cout << *xPtr << endl;  
    delete xPtr;  
}
```





# Only **delete** if memory was allocated by **new**

incorrect

```
int main() {  
    int x = 5;  
    int *xPtr = &x;  
    cout << *xPtr << endl;  
    delete xPtr;  
}
```

correct

```
int main() {  
    int x = 5;  
    int *xPtr = &x;  
    cout << *xPtr << endl;  
}
```

# Allocating Arrays

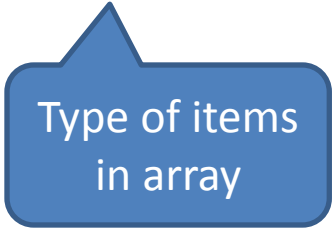
- When allocating arrays on the stack (using “int arr[SIZE]”), size must be a constant

```
int numItems;  
cout << "how many items?";  
cin >> numItems;  
int arr[numItems]; // not allowed
```

# Allocating Arrays

- If we use **new[]** to allocate arrays, they can have variable size

```
int numItems;  
cout << "how many items?";  
cin >> numItems;  
int *arr = new int[numItems];
```

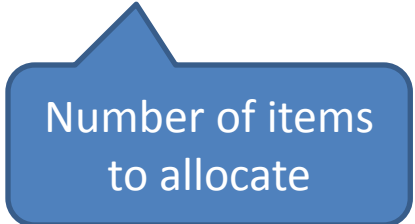


Type of items  
in array

# Allocating Arrays

- If we use **new[]** to allocate arrays, they can have variable size

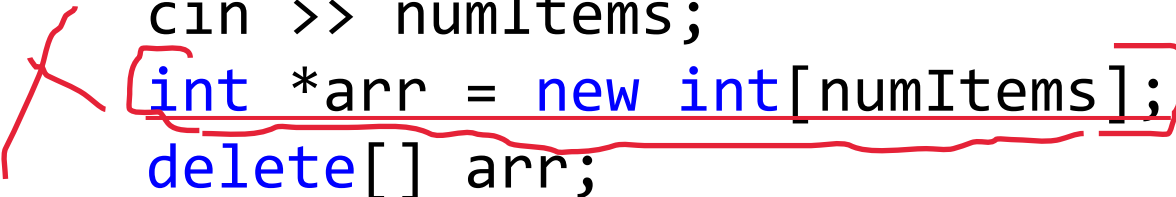
```
int numItems;  
cout << "how many items?";  
cin >> numItems;  
int *arr = new int[numItems];
```



Number of items  
to allocate

# Allocating Arrays

- If we use **new[]** to allocate arrays, they can have variable size
- De-allocate arrays with **delete[]**

```
int numItems;  
cout << "how many items?";  
cin >> numItems;  
  
int *arr = new int[numItems];  
delete[] arr;
```

# Ex: Storing values input by the user

```
int main() {  
    int numItems;  
    cout << "how many items? ";  
    cin >> numItems;  
    int *arr = new int[numItems];  
    for (int i = 0; i < numItems; ++i) {  
        cout << "enter item " << i << ": ";  
        cin >> arr[i];  
    }  
    for (int i = 0; i < numItems; ++i) {  
        cout << arr[i] << endl;  
    }  
    delete[] arr;  
}
```

how many items? **3**  
enter item 0: **7**  
enter item 1: **4**  
enter item 2: **9**  
**7**  
**4**  
**9**

# Allocating Class Instances using **new**

- **new** can also be used to allocate a class instance

```
class Point {  
public:  
    int x, y;  
};
```

```
int main() {  
    Point *p = new Point;  
    delete p;  
}
```

# Allocating Class Instances using **new**

- **new** can also be used to allocate a class instance
- The appropriate constructor will be invoked

```
class Point {  
public:  
    int x, y;  
    Point() {  
        x = 0; y = 0; cout << "default constructor" << endl;  
    }  
};
```

```
int main() {  
    Point *p = new Point;  
    delete p;  
}
```

**Output:**  
default constructor



# Allocating Class Instances using **new**

- **new** can also be used to allocate a class instance
- The appropriate constructor will be invoked

```
class Point {  
public:  
    int x, y;  
    Point(int nx, int ny) {  
        x = nx; y = ny; cout << "2-arg constructor" << endl;  
    }  
};
```

```
int main() {  
    Point *p = new Point(2, 4);  
    delete p;  
}
```

**Output:**

2-arg constructor

# Destructor

- Destructor is called when the class instance gets de-allocated

```
class Point {  
public:  
    int x, y;  
    Point() {  
        cout << "constructor invoked" << endl;  
    }  
    ~Point() {  
        cout << "destructor invoked" << endl;  
    }  
}
```

- Destructor is called when the class instance gets de-allocated
  - If allocated with **new**, when **delete** is called

```
class Point {  
public:  
    int x, y;  
    Point() {  
        cout << "constructor invoked" << endl;  
    }  
    ~Point() {  
        cout << "destructor invoked" << endl;  
    }  
};  
int main() {  
    Point *p = new Point;  
    delete p;  
}
```

**Output:**

constructor invoked  
destructor invoked

- Destructor is called when the class instance gets de-allocated

- If allocated with **new**, when **delete** is called

- If stack-allocated, when it goes out of scope

```
class Point {  
public:  
    int x, y;  
    Point() {  
        cout << "constructor invoked" << endl;  
    }  
    ~Point() {  
        cout << "destructor invoked" << endl;  
    }  
};  
int main() {  
    if (true) {  
        Point p;  
    }  
    cout << "p out of scope" << endl;  
}
```

not clear

### Output:

constructor invoked  
destructor invoked  
p out of scope

# Representing an Array of Integers

- When representing an array, often pass around both the pointer to the first element and the number of elements
  - Let's make them fields in a class


```
class IntegerArray {  
public:  
    int *data;  
    int size;  
};
```



Pointer to the first element

# Representing an Array of Integers

- When representing an array, often pass around both the pointer to the first element and the number of elements
  - Let's make them fields in a class

```
class IntegerArray {  
public:  
    int *data;  
    int size;   
};
```

```
class IntegerArray {  
public:  
    int *data;  
    int size;  
};
```

```
int main() {  
    IntegerArray arr;  
    arr.size = 2;  
    arr.data = new int[arr.size];  
    arr.data[0] = 4; arr.data[1] = 5;  
    delete[] a.data;  
}
```

similar to \*x =  
values



```
class IntegerArray {  
public:  
    int *data;  
    int size;  
};
```

```
int main() {  
    IntegerArray arr;  
    arr.size = 2;  
    arr.data = new int[arr.size];  
    arr.data[0] = 4; arr.data[1] = 5;  
    delete[] a.data;  
}
```



Can move this into a constructor



```
class IntegerArray {  
public:  
    int *data;  
    int size;  
    IntegerArray(int size) {  
        data = new int[size];  
        this->size = size;  
    }  
};
```

why to do it like  
this ?

```
int main() {  
    IntegerArray arr(2);  
    arr.data[0] = 4; arr.data[1] = 5;  
    delete[] arr.data;  
}
```

say arrays  
brr(4),krr(6),mrr(9)  
are defined having  
deferent sizes.

```
class IntegerArray {  
public:  
    int *data;  
    int size;  
    IntegerArray(int size) {  
        data = new int[size];  
        this->size = size;  
    }  
};
```

```
int main() {  
    IntegerArray arr(2);  
    arr.data[0] = 4; arr.data[1] = 5;  
    delete[] arr.data;  
}
```



Can move this into a destructor

```
class IntegerArray {  
public:  
    int *data;  
    int size;  
    IntegerArray(int size) {  
        data = new int[size];  
        this->size = size;  
    }  
    ~IntegerArray () {  
        delete[] data;  
    }  
};
```



De-allocate memory used by fields in destructor

```
int main() {  
    IntegerArray arr(2);  
    arr.data[0] = 4; arr.data[1] = 5;  
}
```

incorrect

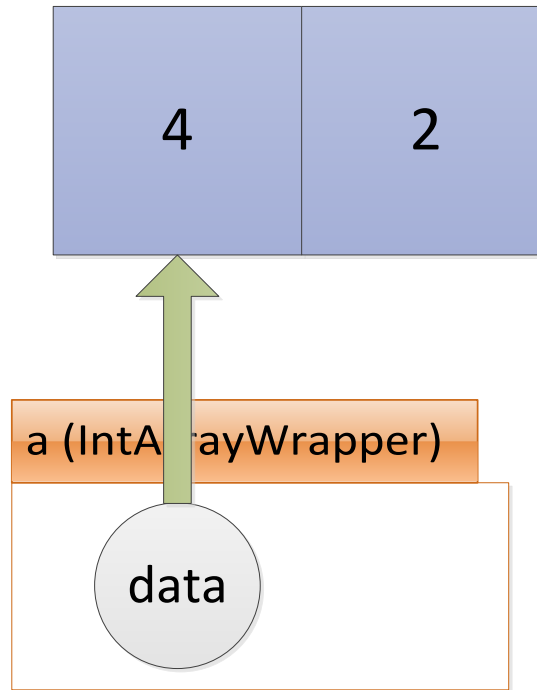
```
class IntegerArray {
public:
    int *data;
    int size;
    IntegerArray(int size) {
        data = new int[size];
        this->size = size;
    }
    ~IntegerArray() {
        delete[] data;
    }
};

int main() {
    IntegerArray a(2);
    a.data[0] = 4; a.data[1] = 2;
    if (true) {
        IntegerArray b = a;
    }
    cout << a.data[0] << endl; // not 4!
}
```

```

class IntegerArray {
public:
    int *data;
    int size;
    IntegerArray(int size) {
        data = new int[size];
        this->size = size;
    }
    ~IntegerArray() {
        delete[] data;
    }
};

```



```

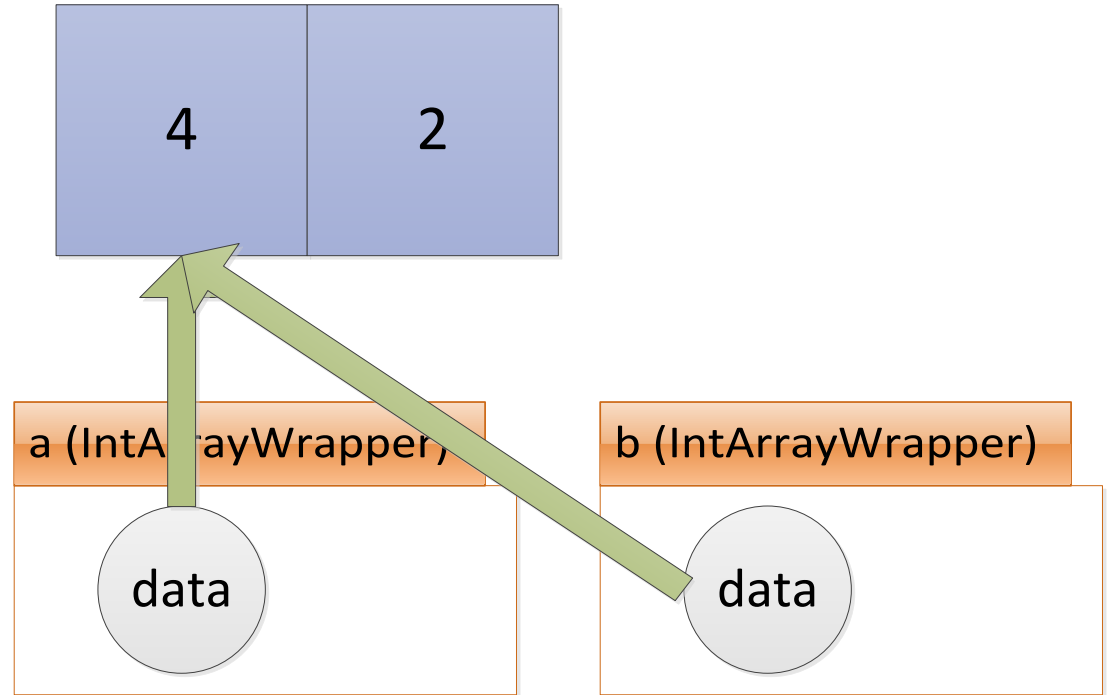
int main() {
    IntegerArray a(2);
    a.data[0] = 4; a.data[1] = 2;
    if (true) {
        IntegerArray b = a;
    }
    cout << a.data[0] << endl; // not 4!
}

```



- Default copy constructor copies fields

```
class IntegerArray {  
public:  
    int *data;  
    int size;  
    IntegerArray(int size) {  
        data = new int[size];  
        this->size = size;  
    }  
    ~IntegerArray() {  
        delete[] data;  
    }  
};
```

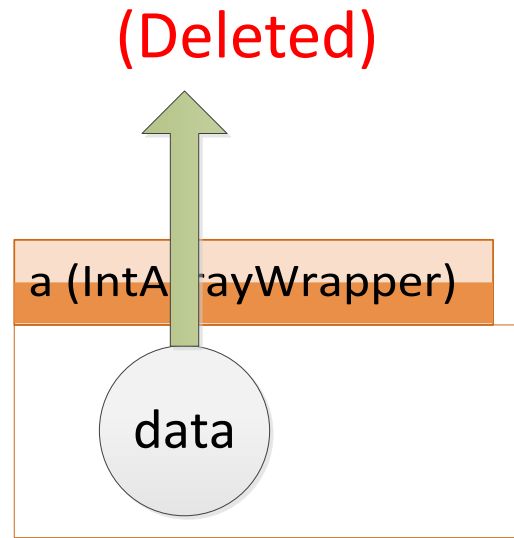


```
int main() {  
    IntegerArray a(2);  
    a.data[0] = 4; a.data[1] = 2;  
    if (true) {  
        IntegerArray b = a;  
    }  
    cout << a.data[0] << endl; // not 4!  
}
```



- When b goes out of scope, destructor is called (deallocates array), a.data now a dangling pointer

```
class IntegerArray {  
public:  
    int *data;  
    int size;  
    IntegerArray(int size) {  
        data = new int[size];  
        this->size = size;  
    }  
    ~IntegerArray() {  
        delete[] data;  
    }  
};
```

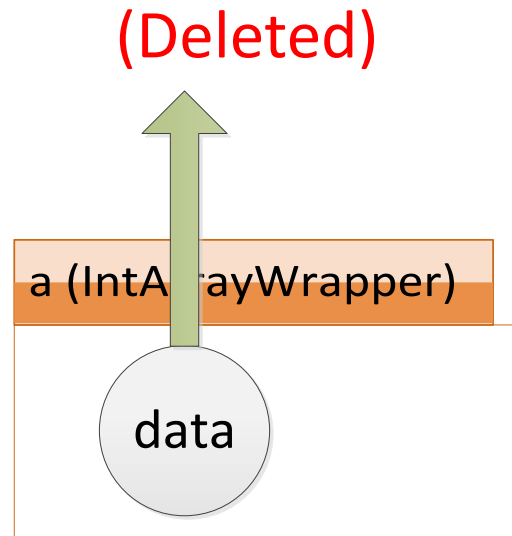


```
int main() {  
    IntegerArray a(2);  
    a.data[0] = 4; a.data[1] = 2;  
    if (true) {  
        IntegerArray b = a;  
    }  
    cout << a.data[0] << endl; // not 4!  
}
```



- 2<sup>nd</sup> bug: when a goes out of scope, its destructor tries to delete the (already-deleted) array

```
class IntegerArray {  
public:  
    int *data;  
    int size;  
    IntegerArray(int size) {  
        data = new int[size];  
        this->size = size;  
    }  
    ~IntegerArray() {  
        delete[] data;  
    }  
};
```



```
int main() {  
    IntegerArray a(2);  
    a.data[0] = 4; a.data[1] = 2;  
    if (true) {  
        IntegerArray b = a;  
    }  
    cout << a.data[0] << endl; // not 4!  
}
```

Program crashes as it terminates



- Write your own a copy constructor to fix these bugs

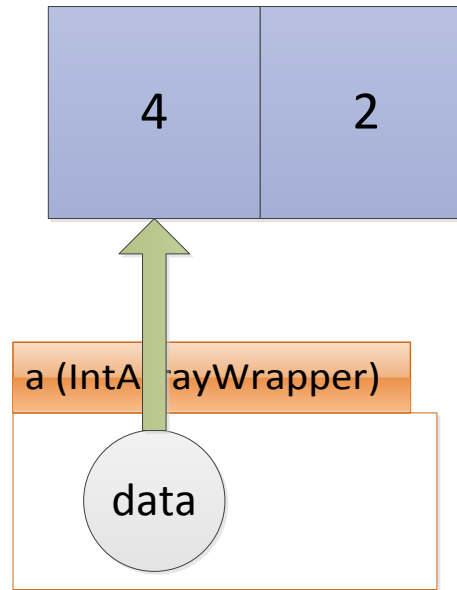
```
class IntegerArray {
public:
    int *data;
    int size;
    IntegerArray(int size) {
        data = new int[size];
        this->size = size;
    }
    IntegerArray(IntegerArray &o) {
        data = new int[o.size];
        size = o.size;
        for (int i = 0; i < size; ++i)
            data[i] = o.data[i];
    }
    ~IntegerArray() {
        delete[] data;
    }
};
```

```

class IntegerArray {
public:
    int *data; int size;
    IntegerArray(int size) {
        data = new int[size];
        this->size = size;
    }
    IntegerArray(IntegerArray &o) {
        data = new int[o.size];
        size = o.size;
        for (int i = 0; i < size; ++i)
            data[i] = o.data[i];
    }
    ~IntegerArray() {
        delete[] data;
    }
};

int main() {
    IntegerArray a(2);
    a.data[0] = 4; a.data[1] = 2;
    if (true) {
        IntegerArray b = a;
    }
    cout << a.data[0] << endl; // 4
}

```

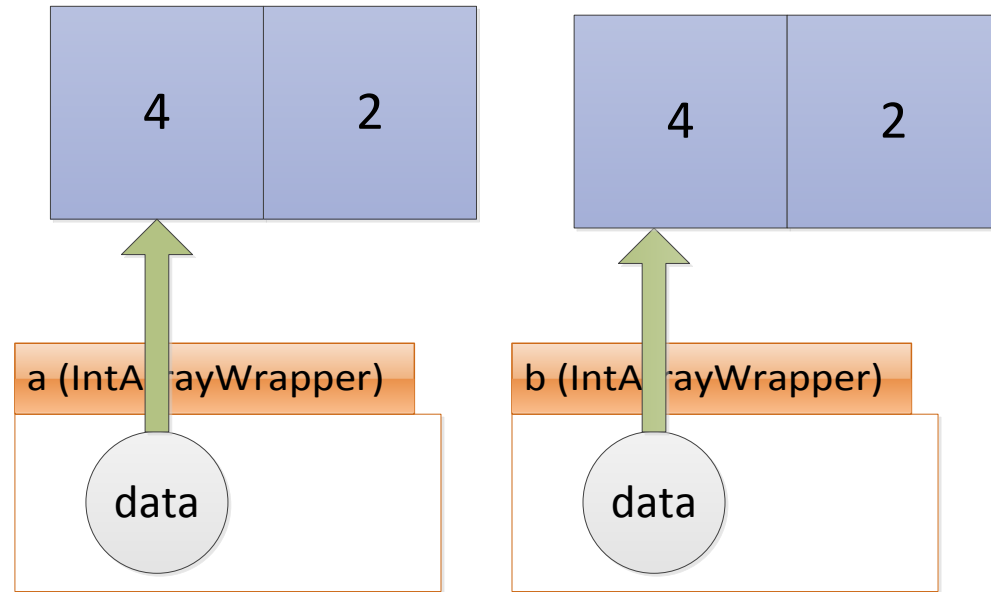


```

class IntegerArray {
public:
    int *data; int size;
    IntegerArray(int size) {
        data = new int[size];
        this->size = size;
    }
    IntegerArray(IntegerArray &o) {
        data = new int[o.size];
        size = o.size;
        for (int i = 0; i < size; ++i)
            data[i] = o.data[i];
    }
    ~IntegerArray() {
        delete[] data;
    }
};

int main() {
    IntegerArray a(2);
    a.data[0] = 4; a.data[1] = 2;
    if (true) {
        IntegerArray b = a;
    }
    cout << a.data[0] << endl; // 4
}

```



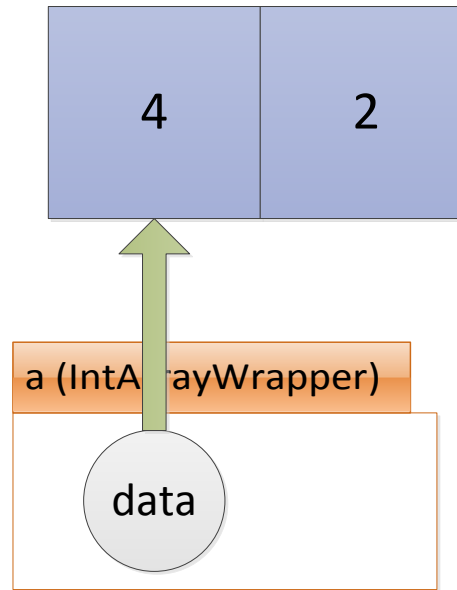
← Copy constructor invoked

```

class IntegerArray {
public:
    int *data; int size;
    IntegerArray(int size) {
        data = new int[size];
        this->size = size;
    }
    IntegerArray(IntegerArray &o) {
        data = new int[o.size];
        size = o.size;
        for (int i = 0; i < size; ++i)
            data[i] = o.data[i];
    }
    ~IntegerArray() {
        delete[] data;
    }
};

int main() {
    IntegerArray a(2);
    a.data[0] = 4; a.data[1] = 2;
    if (true) {
        IntegerArray b = a;
    }
    cout << a.data[0] << endl; // 4
}

```



here

MIT OpenCourseWare

<http://ocw.mit.edu>

6.096 Introduction to C++

January (IAP) 2011

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.