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

```cpp
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

```cpp
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

```cpp
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

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

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

**Output:**
constructor with arg 3
constructor with arg 5
• If a constructor with parameters is defined, the default constructor is no longer available.

```cpp
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
  – Without a default constructor, can’t declare arrays without initializing

```cpp
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

```cpp
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

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

int main() {
    Integer i; // ok
    Integer j(3); // ok
}
```
• How do I refer to a field when a method argument has the same name?

• **this**: a pointer to the current instance

```cpp
class Integer {
public:
    int val;
    Integer(int val = 0) {
        this->val = val;
    }
};
```
• How do I refer to a field when a method argument has the same name?
• **this**: a pointer to the current instance

```cpp
class Integer {
public:
    int val;
    Integer(int val = 0) {
        this->val = val;
    }
    void setVal(int val) {
        this->val = val;
    }
};
```
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.
Scoping and Memory

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

```cpp
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
Scoping and Memory

• When a variable goes out of scope, that memory is no longer guaranteed to store the variable’s value.

```cpp
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.

```c++
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)

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

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

```c
int* getPtrToFive() {
    int x = 5;
    return &v;
}
```
• Implement a function which returns a pointer to some memory containing the integer 5
• Incorrect implementation:
  – x is declared in the function scope

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

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

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

```c
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

```cpp
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

```cpp
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

```cpp
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`

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

int main() {
    int *p = getPtrToFive();
    cout << *p << endl; // 5
    delete p;
}
```
Delete Memory When Done Using It

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

```cpp
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

```cpp
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.
If you don’t use de-allocate memory using `delete`, your application will waste memory.
• 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**

```cpp
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?
• 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

```cpp
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

```cpp
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.

```cpp
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

```cpp
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 *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.

```cpp
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.

```c++
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;  // 2nd iteration
    }
}
```
- To fix the memory leak, de-allocate memory within the loop

```cpp
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
Don’t Use Memory After Deletion

```cpp
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

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

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

correct

```cpp
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

```cpp
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

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

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

```
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

```cpp
int main() {
    int x = 5;
    int *xPtr = &x;
    cout << *xPtr << endl;
    delete xPtr;
}
```
Only **delete** if memory was allocated by **new**

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

```cpp
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

```cpp
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

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

• If we use `new[]` to allocate arrays, they can have variable size

• De-allocate arrays with `delete[]`

```c++
int numItems;
cout << "how many items?";
cin >> numItems;
int *arr = new int[numItems];
delete[] arr;
```
Ex: Storing values input by the user

```cpp
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

```cpp
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

```cpp
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

```cpp
class Point {
public:
    int x, y;
    Point(int nx, int ny) {
        x = ny; x = 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

```cpp
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

```cpp
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

```cpp
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;
}
```

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

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

```cpp
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;
}
```cpp
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;
}
```
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;
}

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;
    }
};

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

```cpp
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

```cpp
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!
}
```
2nd bug: when a goes out of scope, its destructor tries to delete the (already-deleted) array

```cpp
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

```cpp
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;  // Pointer to the array data
    int size;  // Size of the array

    IntegerArray(int size) {
        data = new int[size];  // Allocate memory for the array
        this->size = size;  // Set the size
    }

    IntegerArray(IntegerArray &o) {
        data = new int[o.size];  // Allocate memory for the array
        size = o.size;  // Set the size
        for (int i = 0; i < size; ++i)
            data[i] = o.data[i];  // Copy the data
    }

    ~IntegerArray() {
        delete[] data;  // Free the allocated memory
    }
};

int main() {
    IntegerArray a(2);  // Create an IntegerArray object
    a.data[0] = 4;  a.data[1] = 2;
    if (true) {
        IntegerArray b = a;  // Copy constructor invoked
    }
    cout << a.data[0] << endl;  // Output 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