EECS 280 – Lecture 13

Memory Models and Dynamic Memory

2/21/2022

Recall: Compound Object Lifetimes

- Constructors are called whenever a class object is created for the first time.

```
Bird(const string &name_in)
  : name(name_in) {
  cout << "Bird ctor: " << name << endl;
}
```

- Destructors are called whenever a class object's lifetime ends (depends on storage duration).
  - e.g. For local variables, when they go out of scope.

```
~Bird() {
  cout << "Bird dtor: " << name << endl;
}
```

Example: Local Object Lifetimes


```
Bird(int id_in)
  : ID(id_in) {
  cout << "Bird ctor: " << ID << endl;
}
```

```
~Bird() {
  cout << "Bird dtor: " << ID << endl;
}
```

```
Bird b_global(0);

int main() {
  Bird b1(1);
  for (int i = 0; i < 3; ++i) {
    Bird b2(2);
    b2.talk();
  }
  b1.talk();
  if (100 < 2) {
    Bird b3(3);
    b3.talk();
  }
  else {
    Bird *ptrToB1 = &b1;
    ptrToB1->talk();
  }
}
```

Output

```
Bird ctor 0

Bird ctor 1

Bird ctor 2
tweet
Bird dtor 2
Bird ctor 2
tweet
Bird dtor 2
Bird ctor 2
tweet
Bird dtor 2

tweet

tweet

Bird dtor 1

Bird dtor 0
```

Recall: C++ Memory Model

- An **object** is a piece of data in memory.
- An object lives at an **address** in memory.
- You can use an object during its **lifetime**.
- Lifetimes are managed according to **storage duration**. Three options in C++:

Managed by the compiler.

  - ***Static***
    Lives for the whole program.
  - ***Automatic (Local)***
    Lives during the execution of its local block.

Managed by you!

  - ***Dynamic***
    You control the lifetime!

Static Storage

Managed by the compiler.

- Includes:
  - Global variables
  - Static member variables
  - Static variables in functions

- They are created at the very start of the program and live until the very end.
  - You can use them whenever.

Automatic Storage (Local Variables)

Managed by the compiler.

- A local variable lives inside a **block**.
  - A block is a set of curly braces.
  - Usually a function or loop body, but you can also make just a plain block.
  - Initialized when the declaration is run.
  - Dies when the block is finished.
  - Scope *usually* keeps us from using a dead local.

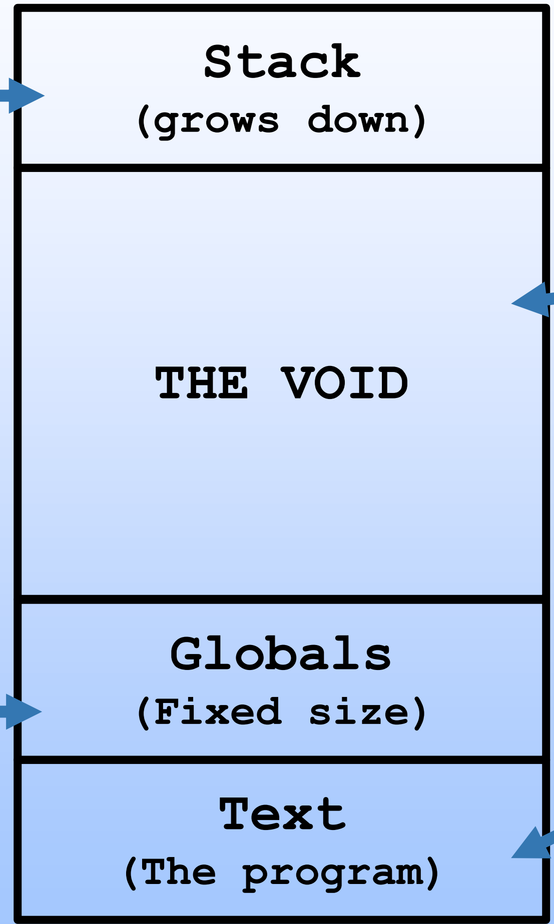
Memory Address Space

- The operating system allows each running program to use a particular range of addresses.
- This is called the program's **address space**.
- It is divided into several segments…

Memory Address Space

Automatic Storage
Stack frames are added and removed as functions are called.

Address MAX



Address 0

Free space.

Machine code for your program.

Static Storage
Objects that can be used throughout the whole program.

The Factory Pattern*

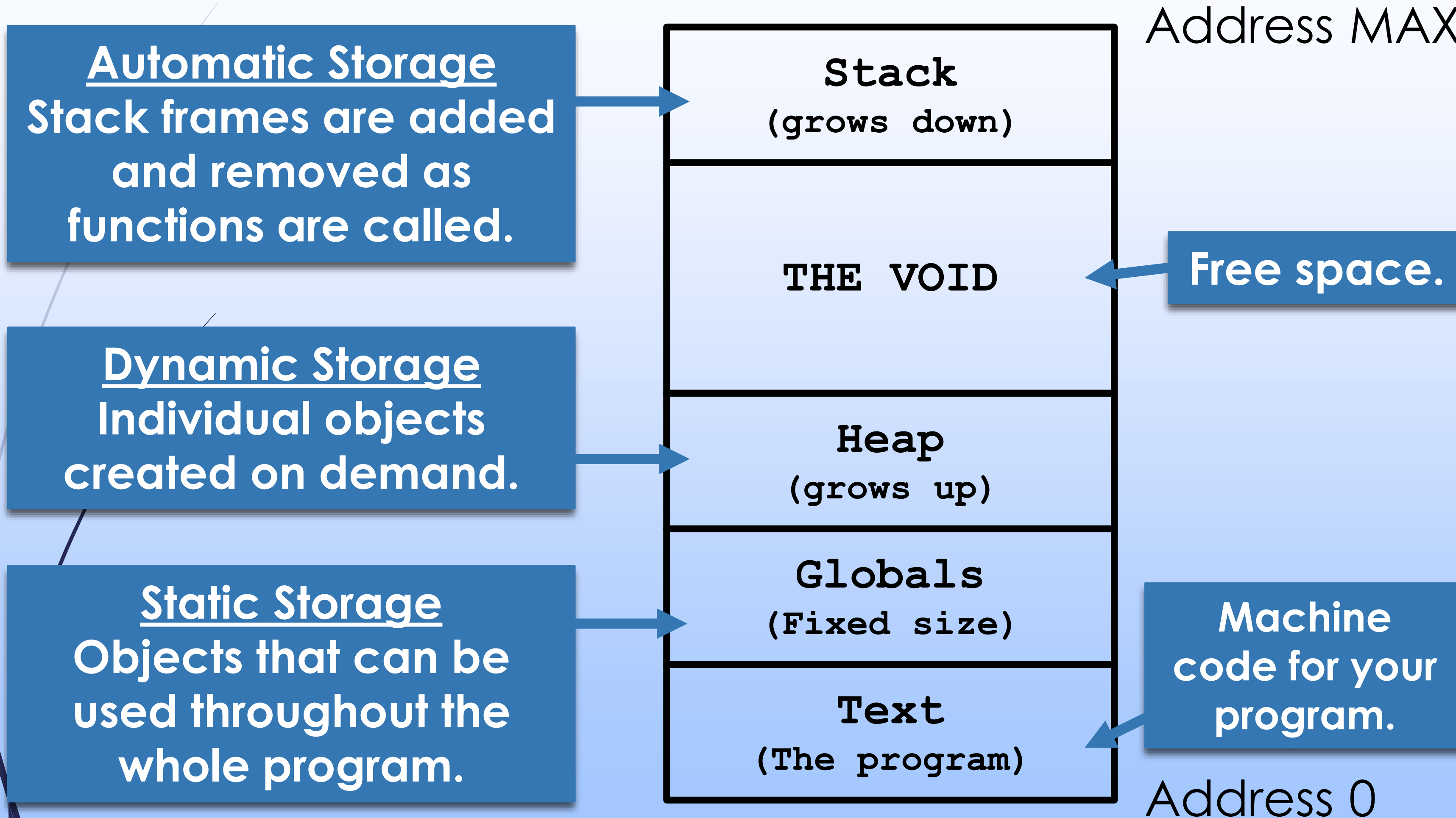
- A factory function creates and returns objects to be used elsewhere.
- However, automatic storage (i.e. using local variables) does not allow this. Why?

```
// REQUIRES: Color must be blue or black
// EFFECTS:  Creates a bird with the given color
//           and returns a pointer to it.
Bird * Bird_factory(const string &color, const string &name) {
  if (color == "blue") {
    BlueBird b(name);
    return &b;
  }
  else if (color == "black") {
    Raven b(name);
    return &b;
  }
  else {
    assert(false);
  }
}
```

```
int main() {
  string color;
  cin >> color; // User enters "black"
  Bird *bird = Bird_factory(color, "Poe");
  bird->talk(); // Prints "nevermore!"
}
```

*This factory function is broken! Stay tuned for a fixed version…

Memory Address Space



What is the Heap?

- The heap is a separate region of memory used for **dynamic storage**.
- Each object on the heap has its own lifetime. (No “frames” like the stack.)

Dynamic Storage

Managed by the programmer!

- We just saw one example of when we might not want the compiler to automatically destroy objects in memory.
- **Dynamic memory** lets *us* manage memory!
  - Objects are created on the heap by a `new` expression.
  - Those objects are destroyed by a `delete` expression.
  - You put these in the code yourself! You decide!

* The `malloc` and `free` functions also work with the heap, but they aren't generally used in C++ style programming.

The `new` Operator

- Use `new` to create a dynamic object.

```
int *ptr = new int(3);
```

- Here's what happens:
  - Space for an `int` is created on the **heap**.
  - The `int` is initialized with value 3.
  - The `new` expression *evaluates* to the address of the object.

The Factory Pattern

- A factory function creates and returns objects to be used elsewhere.
- Dynamic memory allows the object to live beyond the scope of the factory function.

```
// REQUIRES: Color must be blue or black
// EFFECTS:  Creates a bird with the given color
//           and returns a pointer to it.
Bird * Bird_factory(const string &color, const string &name) {
  if (color == "blue") {
    return new BlueBird(name);
  }
  else if (color == "black") {
    return new Raven(name);
  }
  else {
    assert(false);
  }
}
```

```
int main() {
  string color;
  cin >> color; // User enters "black"
  Bird *bird = Bird_factory(color, "Poe");
  bird->talk(); // Prints "nevermore!"
  // One more line missing here...
}
```

Keeping Track of Dynamic Objects

- For a **local object** (automatic storage)…
  - The lifetime of the object is tied to the scope of the variable's declaration!
  - When the variable goes out of scope, the object dies and can't be used.

- But a dynamic object isn't created from a declaration that has any particular scope!
- We keep track of it using a pointer that holds its address.

```
int *ptr = new int(3);
```

The `delete` Operator

- Use `delete` to destroy a dynamic object.
- The operand is the address of the dynamic object (i.e. a pointer to it).

```
int *ptr = new int(3);
delete ptr;
```

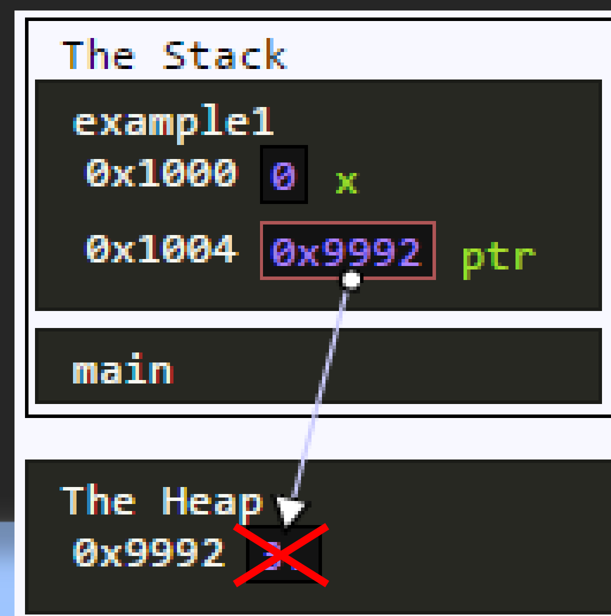
- Here's what happens:
  - Follow the pointer to a dynamic object.
  - Destroy whatever object was there.

Using `delete`

- `delete` follows the pointer and kills the object found.
- Kills the object pointed to by `ptr`, not `ptr` itself.

```cpp
void example1() {
  int x = 0;
  double *ptr = new double(3.0);
  delete ptr;
}

int main() {
  example1();
  ...
}
```



Simple Pattern for new/delete

```
void rotate_left(Image* img) {

  int width = Image_width(img);
  int height = Image_height(img);

  // auxiliary image to temporarily store rotated image
  Image *aux = new Image;
  Image_init(aux, height, width); // width and height switched

  // iterate through pixels and place each where it goes in temp
  for (int r = 0; r < height; ++r) {
    for (int c = 0; c < width; ++c) {
      Image_set_pixel(aux, width - 1 - c, r, Image_get_pixel(img, r, c));
    }
  }

  // Copy data back into original
  *img = *aux;
  delete aux;
}
```

The Factory Pattern

- A factory function creates and returns objects to be used elsewhere.
- Dynamic memory allows the object to live beyond the scope of the factory function.

```
// REQUIRES: Color must be blue or black
// EFFECTS:  Creates a bird with the given color
//           and returns a pointer to it.
Bird * Bird_factory(const string &color, const string &name) {
  if (color == "blue") {
    return new BlueBird(name);
  }
  else if (color == "black") {
    return new Raven(name);
  }
  else {
    assert(false);
  }
}
```

```
int main() {
  string color;
  cin >> color; // User enters "black"
  Bird *bird = Bird_factory(color, "Poe");
  bird->talk(); // Prints "nevermore!"
  delete bird;
}
```

Remember to delete when you're done with it!

Reprise: Class-Type Objects

- When a class-type object is created, its **constructor** runs.
  - For a **local** object, when its declaration runs.
  - For a **dynamic** object, when it is created with `new`.
- When a class-type object dies, its **destructor** runs.
  - For a **local** object, when it goes out of scope.
  - For a **dynamic** object, when it is destroyed with `delete`.
- The lifetimes of the members of a class-type object are tied to the lifetime of the whole object itself.
  - After its body executes, the destructor of a class-type object automatically runs the destructors of its members that are of class type.

Exercise: Whac-a-Mole

http://bit.ly/3eiYhnl

```
class Mole {
public:
  Mole(int id_in)
    : id(id_in) {
    cout << "Mole ctor: "
         << id << endl;
  }

  ~Mole() {
    cout << "Mole dtor: "
         << id << endl;
  }

private:
  string id;
};
```

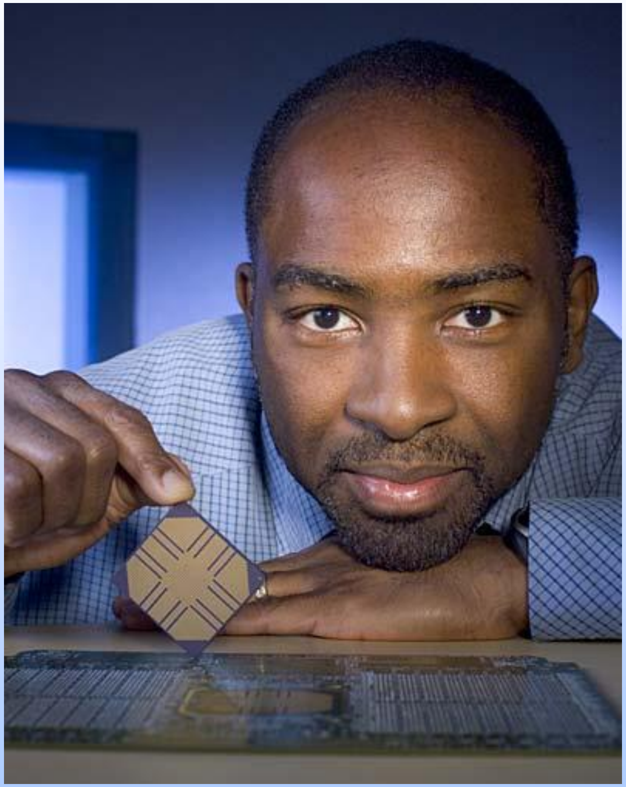
```
Mole * func() {
  Mole m(123);
  return new Mole(456);
}
```

Question

At each line, how many `Mole` objects are alive?

```
int main() {
  Mole m1(1);
  Mole *mPtr;
  // Line 1
  mPtr = func();
  // Line 2
  delete mPtr;
  // Line 3
  mPtr = new Mole(2);
  func();
  // Line 4
  delete mPtr;
  // Line 5
  cout << "all done!" << endl;
}
// Line 6 – after main returns
```

Kunle Olukotun



Designed the First
General-Purpose Multicore CPU

Tim Berners-Lee

Inventor of the World Wide Web

Break Time!

We’ll start again in one minute.

Managing Dynamic Memory

- We're used to the compiler taking care of everything for us.
- Now, we have to make sure we clean up any dynamic memory when we're done with it.
- There are a few traps you might fall into…

Put on our evil villain hats…

http://bit.ly/2MXQUGx

Question

Assume 8KB of stack space and 4MB of heap space. Assume an int is 4 bytes.

Which of these programs run out of memory and crash?

"I find your lack of memory disturbing..."

A

```
int main() {
  int *ptr;
  for (int i = 0; i < 1000000000; ++i) {
    ptr = new int(i);
  }
  delete ptr;
}
```

B

```
void helper() {
  int *ptr = new int(10);
  ptr = new int(20);
  delete ptr;
}

int main() {
  for (int i = 0; i < 1000000000; ++i) {
    helper();
  }
}
```

C

```
int main() {
  int x = 10000;
  for (int i = 0; i < 10000; ++i) {
    x = i;
  }
}
```

D

```
int main() {
  int arr[10000];
  for (int i = 0; i < 10000; ++i) {
    arr[i] = i;
  }
}
```

E

```
int main() {
  int *arr = new int[10000];
  for (int i = 0; i < 10000; ++i) {
    arr[i] = i;
  }
}
```

Memory Leaks

- **Memory Leaks**

  Part of your code allocates dynamic memory, but neglects to free up the space when it's done.

- **Orphaned Memory**

  You lose the address of a heap object, meaning it is inevitably leaked.

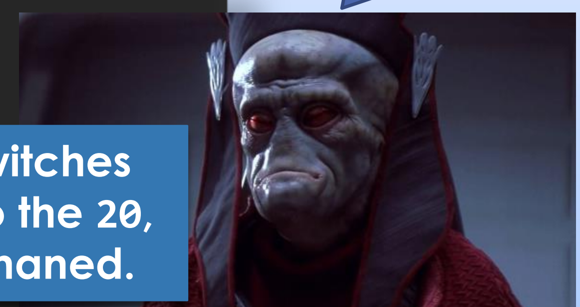
This is getting out of hand. Now there are two of them!

The `helper()` function leaks memory. It allocates 2 ints, but only cleans up 1.

```
void helper() {
  int *ptr = new int(10);
  *ptr = new int(20);
  delete ptr;
}

int main() {
  for (int i = 0; i < 1000000000; ++i) {
    helper(i);
  }
}
```

When `ptr` switches from the `10` to the `20`, the `10` is orphaned.



If leaky code is used frequently, your program can run out of memory!

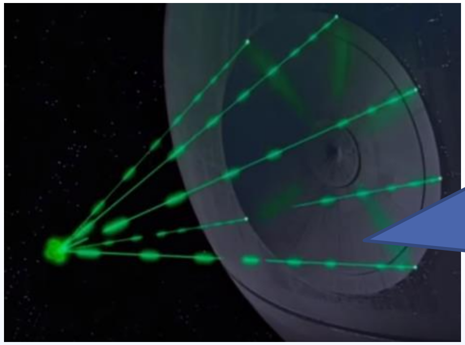
http://bit.ly/3sZU9fV

Question

Deleting an object twice **usually results in a crash.**

Deleting a non-heap object **usually results in a crash.**

Which of these program contain those runtime errors?



"I think it is time we demonstrate the full power of this station! ... Twice!"

A
```
int main() {
  int *ptr1 = new int(1);
  delete ptr1;
  ptr1 = new int(2);
  delete ptr1;
}
```

B
```
int main() {
  int *ptr1 = new int(1);
  ptr1 = new int(2);
  delete ptr1;
  delete ptr1;
}
```

C
```
int main() {
  int x = 0;
  int *ptr1 = &x;
  delete ptr1;
}
```

D
```
int main() {
  int *ptr1 = new int(1);
  delete &ptr1;
}
```

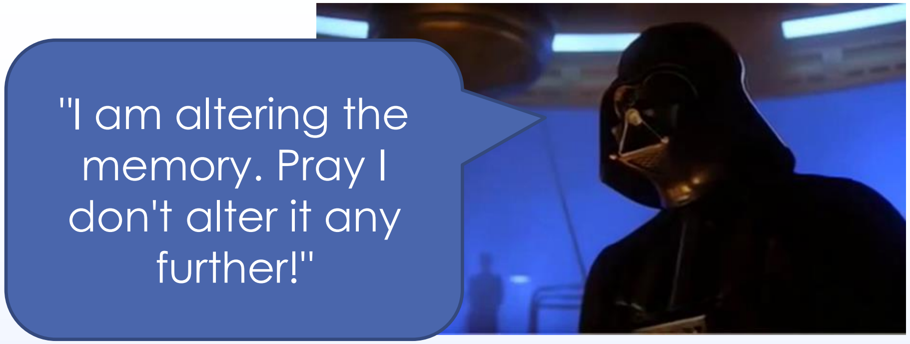
E
```
int main() {
  int *ptr1 = new int(1);
  int *ptr2 = ptr1;
  delete ptr1;
  delete ptr2;
}
```

F
```
int main() {
  int *ptr;
  for (int i = 0; i < 10; ++i) {
    ptr = new int(i);
  }
  for (int i = 0; i < 10; ++i) {
    delete ptr;
  }
}
```

Fun Fact

- If you delete a null pointer (i.e. `nullptr`), nothing will happen.
  - A null pointer often represents "nothing there".
  - So `delete ptr;` means "delete the object there, if there is one".

Dangling Pointers



- Deleting an object allows its memory to potentially be reused for other objects.
- As soon as an object's lifetime has ended, its data is no longer safe to access.
- However, pointers to that dead object may still be hanging around. We call these **dangling pointers**.

```
void example() {
  int x = 0;
  int *ptr = new int(42);
  delete ptr;
  int *ptr1 = new int(3);
  cout << *ptr << " "; // oops, we meant to type ptr1
  delete ptr1;
}
```

Oops, we accidentally used a dangling pointer!

Dangling Pointers

- We can help ourselves out with a bit of defensive programming. Set dangling pointers to null[1].

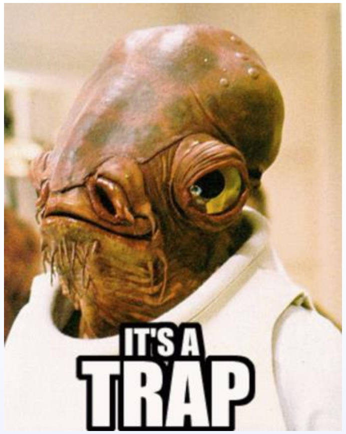
```
void example4() {
  int x = 0;
  int *ptr = new int(42);
  cout << *ptr << endl;
  delete ptr; ptr = nullptr;

  int *ptr1 = new int(3);
  cout << *ptr << endl;
  delete ptr1;
}
```

Now this will probably crash.
But that's much easier to debug!

1 If the pointer object itself is about to die, there is no need to set it to null, since it won't be around to be dereferenced.

Dynamic Memory Errors



- **Memory Leaks**

  Part of your code allocates dynamic memory, but neglects to free up the space when it's done.

- **Orphaned Memory**

  You lose the address of a heap object, meaning it is inevitably leaked.

- **Double Free**

  Attempt to delete a heap object more than once.

- **Bad Delete**

  Give delete an address not pointing to a heap object.

- **Dangling Pointers**

  Be careful not to dereference any pointers to dead objects!

Don't worry!

- Dynamic memory is a bit tricky at first, but with some general rules you can totally manage it!
  - We'll see some more next time…
- It's also really useful!

Factory Pattern

- Dynamic memory is necessary when you don't know what you want until runtime.
- Example: **Factory Pattern**
  - Can create whatever subtype on the fly.
  - The object is passed between scopes.

```
Bird * Bird_factory(const string &color,
                    const string &name) {
  if (color == "blue") {
    return new BlueBird(name);
  }
  else if (color == "black") {
    return new Raven(name);
  }
}
```

```
int main() {
  Bird *birdPtr =
    Bird_factory("blue",
                 "Myrtle");

  // Use birdPtr

  delete birdPtr;
}
```

Dynamic Arrays

- The size of a statically allocated array must be known at **compile time**.

```
int main() {
  const int NUM_ELEMENTS = 10;
  int arr[NUM_ELEMENTS];
}
```

- The size of a dynamically allocated array may be determined at **runtime**.

```
int main() {
  cout << "How many elements? ";
  int howMany;
  cin >> howMany;
  int *arrPtr = new int[howMany];
}
```

Using Dynamic Arrays

- The result of a `new` expression that creates a dynamic array is a **pointer to the first element**.
- You can still use the `[]` operator to access elements!

```
int main() {
  cout << "How many elements? ";
  int howMany;
  cin >> howMany;
  int *arrPtr = new int[howMany];
  for (int i = 0; i < howMany; ++i) {
    arrPtr[i] = 42; // set each element to 42
  }
}
```

Dynamic Arrays and `delete[]`

- Problem: The runtime environment is not able to distinguish between…
  - An `int*` pointing to a single `int` on the heap.
  - An `int*` pointing to the first element of a dynamic array on the heap.
- Solution: Use a special `delete[]` syntax for arrays.

```
int main() {
  ...
  int *ptr1 = new int(42);
  int *ptr2 = new int[howMany];

  delete ptr1;
  delete[] ptr2;
}
```

Growable Containers

- We'll see this next time – the basic idea is to dynamically allocate more space when you need it!