University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 220: Computer Systems & Programming

Dynamic Allocation in C++: New and Delete

Avoid C's Dynamic Allocation Routines in C++

`malloc` does not call constructors.

`free` does not call destructors.

Do not use these functions to allocate or deallocate class instances!

In general,
- **when writing C++** code,
- it's best to **avoid using `malloc` and `free`** directly at all.

Use **`new`** to Create New Instances

To create a new object, write

```
MyClass* m = new MyClass
                 (arg1, arg2, ...);
```

Returns pointer to a constructed **`MyClass`** instance (a **`MyClass*`**).

arguments passed to constructor; if omitted (no parentheses), no arguments passed

`new` Throws an Exception on Failure

On failure,
- **new throws an exception**,* which (by default)
- **terminates your program**.

If you want failure to return NULL (no constructor is called in that case), use

```
#include <new>
MyClass* m = new (std::nothrow)
         MyClass (arg1, arg2, ...);
```

*We will not have time to discuss exception handling in our class.

new Also Used to Allocate Arrays

To allocate an array, write

```
MyClass* m = new MyClass[42];
```

The number of elements is an arbitrary expression.

The **constructor with no arguments**
- (which must exist)
- is **used to construct each element**.

Remember What Type of Allocation is Used

As with **C**, the **programmer is responsible for deallocating** all dynamically-allocated instances.

In **C++**, the **programmer must also remember**
- **whether each allocation was an instance**
- **or an array**.

There are two kinds of deallocation.
- If you choose the wrong one,
- good luck finding the bug.

Use `delete` to Deallocate Instances, `delete[]` for Arrays

Given `MyClass* m`,
- `delete m; // deletes an instance`
- `delete[] m; // deletes an array`

Before the memory is freed, **destructors** (with no arguments) **are called** on all instances.

As with modern **C**,
- **deleting NULL has no effect**, but
- deleting a "pointer" of uninitialized bits is problematic.

Initialization Rules Can Be Convoluted

Did you notice that I said that parentheses had to be omitted to get the constructor with no arguments?

In certain cases, **C++** applies "value-initialization:"

```
int32_t i{};
int32_t i = int32_t (); // avoid
MyClass* m = new MyClass ();
  // iff default no args constructor
  // is available; user-def'd is called
```

Value-initialization zeroes all non-instance fields, then calls constructors for base classes and instance fields.

