

## Destructor

The last and final member function called in the lifecycle of a class is the destructor.

Purpose of a **destructor**:

The **automatic destructor**:

- Like a constructor and copy constructor, an automatic destructor exists only when no custom destructor is defined.
- [Invoked]:  
never invoke directly  
*Assignment Operator*  
similar to copy constructor: copy contents of an obj
- [Functionality]:  
call all member destructors  
different from copy constructor: must delete current state of the obj

## Custom Destructor:

sphere.h	
5	class Sphere {
6	public:
7	Sphere(); // custom default ctor
8	Sphere(double r); // 1-param ctor
9	Sphere(const Sphere & other); // custom copy ctor
10	<b>~Sphere();</b> // destructor, or dtor
11	...

...necessary if you need to delete any heap memory!

## Overloading Operators

C++ allows custom behaviors to be defined on over 20 operators:

<b>Arithmetic</b>	+ - * / % ++ --
<b>Bitwise</b>	&   ^ ~ << >>
<b>Assignment</b>	=
<b>Comparison</b>	== != > < >= <=
<b>Logical</b>	! &&
<b>Other</b>	[] () ->

General Syntax: `ReturnType operator* ...`

use \* as example here

Adding overloaded operators to Sphere:

sphere.h		sphere.cpp	
1	#ifndef SPHERE_H	...	/* ... */
2	#define SPHERE_H	10	
3		11	
4	class Sphere {	12	
5	public:	13	
...	// ...	14	
17		15	
18		16	
19		17	
20		18	
...	// ...	...	/* ... */

## One Very Powerful Operator: Assignment Operator

sphere.h	
	Sphere & operator=(const Sphere & other);
sphere.cpp	
	Sphere & Sphere::operator=(const Sphere & other) { ... }

## Functionality Table:

	Copies an object	Destroys an object
Copy constructor	✓	
Assignment operator	② replace with copied state	① delete the current state
Destructor		✓

## The Rule of Three

If it is necessary to define any one of these three functions in a class, it will be necessary to define all three of these functions:

- copy constructor
- assignment operator
- destructor

## Inheritance

In nearly all object-oriented languages (including C++), classes can be extended to build other classes. We call the class being extended the **base class** and the class inheriting the functionality the **derived class**.

### Derived Class: Planet

Planet.h	
1	#ifndef PLANET_H_
2	#define PLANET_H_
3	
4	#include "Sphere.h"
5	
6	class Planet : public cs225::Sphere {
7	// Empty!
8	};
9	
10	#endif

In the above code, **Planet** is derived from the base class **Sphere**:

- All public functionality of **Sphere** is part of **Planet**:

planet-main.cpp	
5	int main() {
6	Planet p;
7	p.getRadius(); // Returns 1, the radius init'd
8	// by Sphere's default ctor
...	...
	}

- [Private Members of **Sphere**]: cannot access  
e.g. p.r\_ -> error

### Adding New Functionality:

Planet.h	
6	class Planet : public cs225::Sphere {
7	public:
8	Planet(std::string name, double radius); // ctor
9	private:
10	std::string name_;
11	};

## Functions: non-virtual, virtual, and pure virtual

- The **virtual** keyword:

allow for a class function to be replaced by derived class

Sphere.cpp	Planet.cpp
Sphere::print_1() { cout << "Sphere" << endl; }	// No print_1() defined // in Planet
Sphere::print_2() { cout << "Sphere" << endl; }	Planet::print_2() { cout << "Earth" << endl; }
virtual Sphere::print_3() { cout << "Sphere" << endl; }	// No print_3() defined // in Planet
virtual Sphere::print_4() { cout << "Sphere" << endl; }	Planet::print_4() { cout << "Earth" << endl; }
// .h: <b>pure virtual</b> virtual Sphere::print_5() = 0;	Planet::print_5() { cout << "Earth" << endl; }

	Sphere obj;	Planet obj;	Planet r; Sphere &obj = r;
obj.print_1();	Sphere	Sphere	Sphere
obj.print_2();	Sphere	Earth	Sphere
obj.print_3();	Sphere	Sphere	Sphere
obj.print_4();	Sphere	Earth	
obj.print_5();			

### CS 225 – Things To Be Doing:

- Theory Exam #1 is ongoing – ensure you're signed up!
- Attend and complete lab\_memory (*due Sunday*)
- MP2 is ongoing (*extra credit due Monday*)
- Daily POTDs every M-F for daily extra credit!