*Notes 11/29*

*This* always points to the function you are inside of. If you are implementing an element of a class, *this* always refers to the class at large.

When you run the destructor argument you don’t have to worry about array structure or anything except removing all the dynamically allocated memory. Remember, this function is only called when the class is about to go away – none of the interior aspects matter except getting rid of the memory allocations.

Project 7 – the game plays to completion, but the program crashes when it is finished. Why?

First of all – since the program made it all the way to the end, but then crashed, this means the *destructor* crashed the program. However, the problem is most likely when a robot is shot and killed, but the symptom doesn’t show up until the destructor is triggered.

If you want to check if you are destroying all the dynamically allocated memory, create a global variable equal to zero; every time you add dynamic memory increment it, every time you delete some decrement it.

Void drawRectangle(Rectangle r);  
void drawCircle(Circle c);  
  
int main()  
{  
 Rectangle rect1;  
 Circle circ1;  
 …  
 drawRectangle(rect1);  
 drawCircle(circ1);  
   
}  
  
Using 2 different function names to specify the same action (draw a shape) is required for older languages, but newer ones (like C++) allow *overloaded function names.*

This means that functions can have the same name as long as they take different types of arguments.  
  
void draw(Rectangle r);  
void draw(Circle c);  
  
int main()  
{  
 Rectangle rect1;  
 Circle circ1;  
 …  
 draw(rect1);  
 draw(circl1);  
}  
The compiler can tell the 2 “draw”s apart and this will compile in C++.  
  
However, you have to declare everything that you use – you couldn’t pass draw a “Chicken” value and expect it to figure it out.

What if the declared function takes a double, and you pass it a 3? This will work. The compiler recognizes that the int can be converted to a double and automatically accepts it.

What if there are 2 declared functions, one taking a double and the other taking an int, and you pass it a 3? The compiler will take the *best match* and use that function. In this case the best match would be the function taking an int, so it would call that one.

What about this:  
void f(double d, int j);  
void f(int d, double j);  
  
f(2,3);  
  
This will not compile because both are equally “good.” You will get an error message about “ambiguity.”

#include <cmath>  
//defines double sqrt(double d);  
//defines float sqrt(float f);  
  
… sqrt(2.4) … // 2.4 is a double and the function requiring a double will be called.  
… sqrt(2) … //2 is an int; the two functions that can be called are double and float. This won’t compile! It // will give an ambiguity error.  
… sqrt(2.0) … //Fixed it. Calls the double function successfully.  
  
Class Point  
{  
 public:  
 Point(double x, double y);   
 Point();  
 private:  
 double m\_x;  
 double m\_y;  
};  
…  
Point p1(4, -3.8); //that’s fine because an int can be converted to a double.  
Point p2; //works, because a blank constructor is provided.  
  
Summary: You can only overload a function if the passed variables differ in some way in terms of type or number.

There is no structure built into the language for complex numbers; however, we can create our own.  
  
struct Complex  
{  
 Couplex(double real; double imag)  
 double re;  
 double im;  
};  
  
bool operator==(Complex c1, Complex c2)  
{  
 return c1.re == c2.re && c1.im == c2.im;  
}  
  
Complex z1(3,4);  
Complex z2(-4,5);  
double d;  
..  
Complex z3(3,2\*d);

If ( z1 == z3 //means operator==(z1,z3))  
 …  
  
The operator word overloads the == comparison operator. Basically by declaring it as operator, you can call == to call that specific function (it’s much more natural .)