Problem 1: Fish pointers

For each of the following parts, write a single C++ statement that performs the indicated task. For each part, assume that all previous statements have been executed (e.g., when doing part e, assume the statements you wrote for parts a through d have been executed).

a. Declare a pointer variable named fp that can point to a variable of type string.

b. Declare fish to be a 5-element array of strings.

c. Make the fp variable point to the last element of fish.

d. Make the string pointed to by fp equal to "salmon", using the * operator.

e. Without using the fp pointer, and without using square brackets, set the element at index 3 of the fish array to have the value "yellowtail".

f. Move the fp pointer back by three strings.

g. Using square brackets, but without using the name fish, set the element at index 2 of the fish array to have the value "eel".

h. Without using the * operator, but using square backets, set the string pointed to by fp to have the value "tuna".

i. Declare a bool variable named d and initialize it with an expression that evaluates to true if fp points to the string at the start of the fish array, and false otherwise.

j. Using the * operator in the initialization expression, declare a bool variable named b and initialize it to true if the string pointed to by fp is equal to the string immediately following the string pointed to by fp, and false otherwise.

Problem 2: Déjà vu Pointers (This is different problem)
Suppose you're tasked with fixing a function definition that does not work as intended. The function is supposed to compare two strings and set the count to the number of identical characters, two characters are identical if they are the same character and are in the same position in the cstring. This function will be case sensitive so the character 'a' is not the same as 'A'. Note that cstrings are just character arrays that have '\0' as their last character, for example

```
char name[7] = "harry";
```

might looks like this in memory:

h	a	r	r	y	\0	

Usage of this function might look like:

```
int count = 0;
compareCstrings("tacocat", "TACOCAT", count);    // should set count to 0
compareCstrings("Harry", "Malfoy", count);       // should set count to 1
compareCstrings("SMC","SBCC", count);            // should set count to 2
```

Currently the function definition is:

```
void compareCstrings(const char *str1, const char *str2, int &count) {
       *count = 0;
       while (str1 != '\0' || str2 != '\0') {
              if (*str1 == *str2)
                     *count++;
              str1++;
              str2++;
       }
}
```

Identify the errors in the above implementation and rewrite the function so that it satisfies specification. Try to keep the general form of the original code, you should not have to add or remove any lines of code, just modify the existing ones.

Problem 3: Delete All the Things

Write delete statements that correctly delete the following dynamically allocated entities. Hint: draw out the memory layout on scratch paper.

```
int *p1 = new int[10];
int *p2[15];
for (int i = 0; i < 15; i++)
       p2[i] = new int[5];
int **p3 = new int*[5];
for (int i = 0; i < 5; i++)
       p3[i] = new int;
int *p4 = new int;
int *temp = p4;
p4 = p1;
p1 = temp;
```

Problem 4: Build it up, Break it down

Consider the following 7 classes and a main function. What is printed to the console with the complete execution of main?

```
class Hey {
public:
       Hey() { cout << "!"; }
       ~Hey() { cout<<"~!"; }
};

class Snap {
public:
       Snap() { cout << "Snap "; }
       ~Snap() { cout << "~Snap "; }

       Hey hey[3];
};

Class Crackle {
public:
       Crackle() { cout << "Crackle "; }
       ~Crackle() { cout << "~Crackle ";}
};

class Pop {
public:
       Pop() { cout << "Pop "; }
       ~Pop() { cout << "~Pop "; }
};
```

```
class Rice : public Pop {
public:
       Rice() { cout << "Rice "; }
       ~Rice() { cout << "~Rice "; }
};

class Kris :public Crackle{
public:
       Kris() { cout << "Kris "; }
       ~Kris() { cout << "~Kris "; }

       Rice rice;
};

class Pies : public Snap {
public:
       Pies() { cout << "Pies "; }
       ~Pies() { cout << "~Pies "; }

       Kris kris;
};

void main(){
       Pies pies;
       cout << endl << "===" << endl;

}
```

Problem 5: Apples and Oranges

Consider the following program:

```
class A {
public:
	A() :m_msg("Apple") {}
	A(string msg) : m_msg(msg) {}
	virtual ~A() {cout << "A::~A "; message();}
	void message() const {
		cout << "A::message() ";
		cout << m_msg << endl;
	}
private:
	string m_msg;
};
```

```
class B: public A {
public:
	B() :A("Orange") {}
	B(string msg): A(msg), m_a(msg) {}
	~B() { cout << "B::~B "; }
	void message() const {
		cout << "B::message() ";
		m_a.message();
	}

private:
	A m_a;
};
```

```
int main() {
	A *b1 = new B;
	B *b2 = new B;
	A *b3 = new B("Apple");
	b1[0].message();
	b2->message();
	(*b3).message();
	delete b1;
	delete b2;
	delete b3;
}
```

How many times will you see the word Apple in the output? ____

How about Orange? ____

Now make A's message() virtual, i.e.,

```
	virtual void message() const;
```

How many times will you see the word Apple in the output? ____

How about Orange? ____

Problem 6:
Consider the following three classes; Legs, Animal and Bear. Animals have legs and Bears are a kind of Animal. You may assume that Legs and Animal are completely and correctly implemented.

```
class Legs {
public:
	void move() { cout << "B"; }
};

class Animal {
public:
	Animal(const int nlegs) { num_legs = nlegs; legs = new Legs[num_legs];}

	Animal(const Animal &other){ /*Assume Complete*/}

	virtual ~Animal() { delete[] legs; }

	Animal &operator=(const Animal &other) { /*Assume Complete*/ }

	void walk() { for (Legs* leg = legs; leg < legs + num_legs; leg++) leg->move(); }

	void play() { cout << "Herpa Derp" << endl; };

	virtual void eat() = 0;

	virtual void dance() = 0;

private:
	int num_legs;
	Legs *legs;
};

class Bear : public Animal {
public:
	Bear()  { num_honey = 99; honey = new int[num_honey]; }

	Bear(const Bear &other)  { /*TO DO*/ }

	virtual ~Bear() { delete [] honey; }

	Bear &operator=(const Bear &other) { /*TO DO*/ }

	void play() { cout << "Doo Bee Doo" << endl; }

	virtual void eat() { cout << "Yum Salmon" << endl; }

	virtual void hibernate() { cout << "ZZZZ" << endl; }

private:
	int *honey;
	int num_honey;

};
```

a. Consider the following main function, there are two unique issues preventing this from compiling, what are they?

```
int main() {
	Bear b;
	return 0;
}
```

b. Assuming the issues above are resolved what does the following print?

```
int main() {

	Animal* b = new Bear();

	b->walk();
	cout<<endl;
	b->play();
	b->eat();

	return 0;
}
```

c. Point out the ways this problem illustrates the three properties of inheritance.

Problem 7:
Assuming the issues in problem 6 are resolved:

a. Implement the copy constructor for Bear

b. Overload the assignment operator for Bear

Problem 8:
In addition to the classes from problem 6, consider this Panda class that inherits from Bear. You may assume at this point that all the syntax issues in problem 5 are resolved and completely implemented.

```
class Panda : public Bear {
public:

	Panda() {};
	virtual ~Panda() {};

	virtual void eat() { cout << "Yum Bamboo" << endl; }
	virtual void dance() { cout << "Pop and Lock" << endl; }

};
```

a. What does the following print?

```
int main() {
	Panda p;
	p.walk();
	cout<<endl;
	p.play();
	p.eat();
	p.dance();
	p.hibernate();
	return 0;
}
```

b. What does the following print?

```
int main() {
	Animal* p = new Panda();
	p->walk();
	cout<<endl;
	p->play();
	p->eat();
	p->dance();
	return 0;
}
```

c. Continuing form the main in part b, what happens if we try to execute: `p->hibernate();`