1. For each of the following IC constructs, state whether it is well-typed in some typing context. If so, give the most general typing context in which the construct is well-typed and write the corresponding proof tree. If the construct is not well-typed in any type context, explain why.

   (a) `2 + (new boolean[2]).length`

   (b) `while (x < v.length && y == true) x = y;`

   (c) `(x == 1) && (("foo" + 1) == "fool")`

   (d) `f(s)[s.length] = s[2]`

2. Consider a C-like language that manipulates pointers. Statements have the following syntax:

$$S ::= \mathtt{x} = \mathtt{NULL} \;\mid\; \mathtt{x} = \&\mathtt{y} \;\mid\; \mathtt{x} = \mathtt{y} \;\mid\; \mathtt{x} = *\mathtt{y} \;\mid\; *\mathtt{x} = \mathtt{y}$$

   where `x` and `y` are arbitrary variables.

   Initially, we consider that the only types for variables are integers and pointer types. If T is an arbitrary type, then $\mathsf{T}*$ is the type for pointers to variables of type T. This allows to create multi-level pointers when T itself is a pointer type. The syntax of types for variables is:

$$\mathsf{T} ::= \mathsf{int} \;\mid\; \mathsf{T}*$$

   (a) Write typing rules for all of the assignment statements. Use unit to denote the type of statements.

   We next refine the pointer types in the language to distinguish between variables variables that are pointed to by unique pointers or by multiple pointers. We extend the pointer types as follows:

$$\mathsf{T} ::= \mathsf{int} \;\mid\; \mathsf{T} * \mathsf{unique} \;\mid\; \mathsf{T} * \mathsf{multi}$$

   If T is an arbitrary type, $\mathsf{T} * \mathsf{unique}$ is the type of a pointer to a variable of type T which is not pointed to by any other variable; and $\mathsf{T} * \mathsf{multi}$ is a pointer type where the target variable has type T and may be pointed to by multiple variables. Using this information, the compiler can conclude that `*x` and `*y` are not aliased, whenever `x` and `y` are unique and distinct pointers.

   We want to derive typing rules for assignments, to statically check unique and multiple pointers. For instance, there are two rules for NULL assignments:

$$\frac{A \vdash \mathtt{x} : \mathsf{T} * \mathsf{unique}}{A \vdash \mathtt{x} = \mathtt{NULL} : \mathsf{unit}} \qquad \frac{A \vdash \mathtt{x} : \mathsf{T} * \mathsf{multi}}{A \vdash \mathtt{x} = \mathtt{NULL} : \mathsf{unit}}$$

   (b) For the assignment `x = &y` we must conservatively require `x` to be a multiple pointer. Explain why and write the typing rule for this statement.

   (c) Write the typing rule(s) for pointer assignments `x = y` and informally explain your choice of unique or multiple pointer types in the premise(s).

   (d) Write the typing rule(s) for the assignment `x = *y`.

   (e) Assume we add a new statement to our our language:

$$S ::= \ldots \;\mid\; \mathtt{x <-> y}$$

   This statement swaps the values of `x` and `y`. It atomically executes the sequence `t=x; x=y; y=t; t=NULL`. Write the most precise typing rule(s) for the statement `x <-> y`.

3. Consider the pointer language from the problem 2(a), extended with classes and with the standard inheritance mechanism. Object variables refer to the object structures themselves, while references to objects are constructed using explicit pointer types. There is no new statement in this language, so all of the objects are allocated on stack. The types include integers, classes C, and pointers to classes:

$$\mathsf{T} = \mathsf{int} \;\mid\; \mathsf{C} \;\mid\; \mathsf{T}*$$

We assume that inheritance yields a subtype relation, and the typing rules for assignments use the subsumption rule for object values.

   (a) We claim that covariant subtyping for pointer types is unsound. Show the corresponding subtyping rule and a counterexample program which would typecheck with that rule, but would produce a type error at run-time. You can assume two classes A and B such that B is a subclass of A, and B has an integer field x, which A doesn't.

   (b) Contravariant subtyping for pointer types is also unsound. Write the contravariant subtyping rule and a program which would typecheck with that rule, but would produce a type error at run-time.

   (c) Assume that the language supports multiple inheritance. Show that field conflicts may occur even if the classes in the program all have different field names. Write a class hierarchy that shows such conflicts.