2001 Programming Languages Comp. Solutions

written by 2006 Ph.D. First-Years

1. *Lifetime and Scope:* Within a function, if one variable is declared `static` and the other one is declared non-static, then the two variables have the same scope (accessible only within that function) but have different lifetimes (the static variable has a lifetime lasting the duration of the program execution while the non-static variable has a lifetime lasting the duration of a particular function call).

2. *Static and Dynamic Scope:* Static scope is resolved at compile-time (use the value of the variable declared in the closest enclosing block); dynamic scope is resolved at execution-time (use the value of the variable closest to the current function's activation record on the stack). Exceptions are a language feature that uses dynamic scope.

3. *Tail Recursion:* The primary advantage of tail recursion is space savings. Time requirement is still $O(n)$ but space requirement is $O(1)$ because new activation records aren't being pushed onto the stack for each recursive call.

4. *Parameter Passing:*

   (a) At the beginning of the call to `increment()`, the value of `y` on the stack (0) is copied to the slot where `x` is allocated on the stack. Then `x` is incremented to 1, and when the function returns, the stack pointer is moved back up and the value of `x` is lost. The call doesn't increment `y` because its value was copied to another location, and the value in that location (not `y`) is incremented.

   (b) `increment(&y)`

   (c) The C++ pass-by-reference code is a bit less efficient than the pass-by-value C code in part (a) because it must perform a pointer dereference before incrementing the value of `x`. However, it actually does the correct thing, which is to increment the value of `y` in the calling function, whereas the pass-by-value code doesn't do anything useful. In general, pass-by-reference is more efficient because you can pass a pointer to a large data structure instead of passing the data structure itself by value, which requires that a copy be made on the stack.

5. *Type Conversion*

   (a) `sizeof(double)` $\geq$ `sizeof(float)` $\geq$ `sizeof(long)` $\geq$ `sizeof(int)` $\geq$ `sizeof(char)` (that's implied by the hierarchy in the passage, but we don't think it's true in general because a `long` is usually longer than a `float`)

   (b) A type higher in the hierarchy contains more bytes than a type lower in the hierarchy, so no information is lost when converting from lower to higher because all of those bytes can fit (usually with more room to spare).

   (c) `Car` probably requires a larger representation in memory because it contains at least all the fields of `Vehicle`, and most likely, some additional fields.

(d) No, it's not consistent, because converting from `Car*` to `Vehicle*` is converting a pointer referring to something with a larger representation in memory (subclass) into something with a smaller representation in memory (superclass), whereas converting from `int` to `float` is converting from something with a smaller representation to something with a larger representation.

(e) It makes more sense to convert from `Car` to `Vehicle`. A problem with converting from `Car` to `Vehicle` is that you can't access Car-specific fields. A problem with converting from `Vehicle` to `Car` is that you attempt to access Car-specific fields, which don't exist in `Vehicle`. The latter is a more serious problem because you can read junk data.