
Because C does not use structural equivalence in this situation, only name equivalence. x
and y are of literally the same type, but z is not, though structurally it is identical.

Two ways to fix it are to declare z with x and y

```c
struct
{ int i;
  double j;
} x,y,z;
```

and to define the struct as a typedef

```c
typedef struct A
{ int i;
  double j;
} A;
```

A x,y;

A z;

The second way is preferable since this not only enforces z's compatibility, it also enforces
compatibility on any other variables (local, global, parametric) of that structural type
wherever they may reside in the program.

2. Problem 6.6 in Louden, page 251.

In this situation, C does use structural equivalence. Enums are just integers with user-
defined name; C realizes this and allows the assignment. C++ is less forgiving, though;
C++ insists the assignment not work since x and y are enums of different types.

3. Problem 6.13 in Louden, page 252. This problem asks why C does certain things. I don't
insist that you know exactly what was running through the minds of Kernighan and Ritchie;
instead, just give sound reasons.
a) Direct comparisons in C are allowed between types containing a basic numeric value: ints, longs, floats, doubles, etc., pointers and enumerated types can all be freely assigned to each other.

b) Direct comparisons are not allowed between arrays and structs even if the arrays or structs are of the exact same type. This is because == was designed to be a simple numeric comparison. Arrays and structs are not basic numbers. Comparing them would require more sophisticated methods (loops, etc.) to compare them, and C prefers to have only simple operations as part of the basic language. More complicated comparisons can be done with strcmp() and memcmp(), both of which are in the string.h library.


Unions allow completely different types to occupy the same physical space in memory. If an int and a double (or a long and a float, or two other different types) were stored in the same place, they would not be compatible. The bit structure between an int and a double is so wildly different that the double would bear no noticeable resemblance to the int!

5. C/C++ has a operation called ->? Is such an operation necessary to give C/C++ maximum power? If so, what would be lost if -> were removed? If not, what advantage is there to using ->? (Note: this problem only concerns itself with removing -> from the language, not with removing the concepts of class and pointer from the language.)

This operation is NOT necessary to give C maximum power: a->b is the same thing as (*a).b; thus, it can be removed from the language without harm. The reason -> is in the language is that it is clearer and easier to read, especially in pointer chains. a->b->c->d is the same as (*(*(*a).b).c).d, except that the latter is a horrible mess.