Programming Languages - Principles and Practice
3rd Edition
by Kenneth C Louden and Kenneth A. Lambert
Cengage 2011

Answers to Selected Exercises

Chapter 2

2.2 Here are some additional examples of non-regularity:

Generality

1. When passing multidimensional arrays to a function, the size of the first index can be left unspecified, but all subsequent sizes must be specified:

```
void f(int x[], int y[][10]) /* legal */
void g(int y[][]) /* illegal! */
```

The reason is that C allows pointer arithmetic on arrays, so that **`y++`** should advance to the next memory location of **`y`** in either **`f`** or **`g`** above (that is, ++ doesn't just add one!). In **`f`**, the compiler knows to interpret **`y++`** as **`y+10*sizeof(int)`**, but in **`g`** it would have no idea how much to increment **`y`**. Thus, incomplete array types are incompatible with pointer arithmetic. (So is this *really* a lack of generality??)

2. The equality test **`==`** is not general enough. Indeed, assignment can be applied to everything except arrays, but equality tests can only be applied to scalar types:

```
struct { int i; double r;} x,y;
...
x = y; /* legal */
if(x==y)... /* illegal */
```

3. Arrays lack generality in C because they must be indexed by integers, and the index must always start with 0. (In Ada and Pascal, arrays can be indexed by characters, and can have any lower index bound.)

4. The C **`switch`** statement restricts the test expressions to integers, and also allows only one constant value per case. Thus, the switch statement is not sufficiently general.

Uniformity

1. The comma operator has non-uniform semantics in C. In an expression, such as

```
y = (x++,x+1);
```

it guarantees left to right evaluation order, while in a call such as

```
y = f(x++,x+1);
```

the first argument is not guaranteed to be evaluated before the second.

2. The semicolon is not used uniformly as a terminator in C: it must be used to terminate variable and type definitions:

```
int x; <-- semicolon required
```

but it cannot be used to terminate function definitions (because compound statements are *not* terminated by a semicolon):

```
int f(void) { ... }; <-- an error here
```

Orthogonality

1. Functions can return structures and unions, but not arrays. (Is this orthogonality or generality?). Actually this is false. While it is true that functions cannot be declared with a fixed-size array return type, using the array-pointer equivalence of C allows arrays to be returned as pointers. Thus, the following function **`f`** is illegal in C, but **`g`** is fine (nevertheless, there is a problem with **`g`**!):

```
typedef int ArType[10];

ArType f(void)
{ int y[10]; return y;}

int * g(void)
{ int y[10]; return y;}
```

2.4 This is a nonorthogonality, since it is an interaction between assignment and the datatypes of the subjects of the assignment. It could possibly be viewed as a nonuniformity, although we are really talking about an interaction within a single construct rather than a comparison of constructs. It definitely cannot be viewed as a nongenerality, since it makes no sense for assignment to be so general as to apply to all cases (assignment should only apply when data types are comparable in some sense). C allows the assignment of a real to an integer (with a silent truncation or round), but this is highly questionable, since information is lost by this action, and its precise behavior is unclear from the code itself. One could call this a nonuniformity or nonorthogonality in C as well: assignment does not work the same way, with a special interaction (truncation) occurring for specific data types.

2.8

Readability. Requiring the declaration of variables forces the programmer to document his/her expectations regarding variable names, data types, and scope (the region of the program where the variable will be applicable). Thus, the program becomes much more readable to the programmer and to others.

Writability. Requiring the declaration of variables may actually decrease writability in its most direct sense, since a programmer cannot simply use variables as needed, but must write declarations in their appropriate places to avoid error messages. This increased burden on the programmer can increase programming time. On the other hand, without declarations there can be no local variables, and the use of local variables can increase writability by allowing the programmer to reuse names without worrying about non-local references. Forcing the programmer to plan the use of variables may also improve writability over the long run.

Efficiency. As we saw, readability and writability can be viewed as efficiency issues from the point of view of maintenance and software engineering, so the comments about those issues also apply here in that sense.

The use of declarations may also permit more efficient implementation of the program. Without declarations, if no assumptions are made about the size of variables, less efficient access mechanisms using pointers must be used. Also, the programmer can use declarations to specify the exact size of variable needed (such as **short int** or **long int**). Restricting scope by using local variables can also save memory space by allowing the automatic deallocation of variables. Note, however, that Fortran is a very efficient language in terms of execution speed, so it is not always true that requiring declarations must improve execution speed. Also, speed of translation may actually be decreased by the use of declarations, since more information must be kept in tables to keep track of the declarations. (It is not true, as Fortran and BASIC attest, that without declarations a translator must be multi-pass.)

Security. Requiring declarations enhances the translator's ability to track the use of variables and report errors. A clear example of this appears in the difference between ANSI C and old-style Unix C. Early C did not require that parameters to functions be declared with function prototypes. (While not exactly variable declarations, parameter declarations are closely related and can be viewed as essentially the same concept.) This meant that a C compiler could not guarantee that a function was called with the appropriate number or types of parameters. Such errors only appeared as crashes or garbage values during program execution. The use of parameter declarations in ANSI C greatly improved the security of the C language.

Expressiveness. Expressiveness may be reduced by requiring the declaration of variables, since they cannot then be used in arbitrary ways. Scheme, for example, while requiring declarations, does not require that data types be given, so that a single variable can be used to store data of any data type. This increases expressiveness at the cost of efficiency and security.

2.9 A language with dynamic typing generally does not require the programmer to specify the type of a variable, because only values are typed. This reduces finger typing for the programmer, and also provides flexibility on how variables may be used in programs. On the other hand, the absence of compile-time type checking requires the programmer to test all possible runtime uses of a variable to ensure that no type errors are present. Also, run-time type checking generally slows the execution of a program. Some languages with static type checking (C++ and Java, for example) force the programmer to specify the types of variables and functions in their source code. Other statically typed languages, such as Haskell, do not, however. Their advantage is that all type errors are normally caught before execution, and execution is not slowed down by run-time type checking.

2.10. C has the same problems with semicolons as C++ — indeed, C++ inherited them from C. Thus, in C, we must always write a semicolon after a **`struct`** declaration:

```
struct X { int a; double b; } ; /* semicolon required here */
```

but never after a function declaration:

```
int f( int x) { return x + 1; } /* no semicolon */
```

The reason is C's original definition allowed variables to be declared in the same declaration as types (something we would be very unlikely to do nowadays):

```
struct X { int a; double b; } x;
          /* x is a variable of type struct X */
```

In addition to this nonuniformity of semicolon usage, C (and C++) have at least one additional such nonuniformity: semicolons are used as *separators* inside a for-loop specifier, rather than as terminators:

```
for (i = 0; i < n; i++ /* no semicolon here! */ )
```

2.14. Readability: Ada's comment notation is difficult to confuse with other constructs, and the comment indicators are always present on each comment line. By contrast, a C comment may have widely separated comment symbols, so it may not be easy to determine what is a comment and what is not (especially noticeable if a comment extends over more than one video screen). Embedded C comments may also be confusing, since **/** and ***** are arithmetic operators:

```
2 / 3 /* this is a comment */
2 / 3 / * this is an error */
```

Nested comments can also present readability problems in C:

```
/* A comment
       /* a nested comment
  ...
  but only one comment closer */
```

Thus Ada comments may be judged more readable than C's.

Writability: Ada's comments require extra characters for each new line of comments. This makes it more difficult to write an Ada comment, if only from a count of the number of extra characters required. C's comments, on the other hand, can be written more easily with a single opening and closing character sequence.

Reliability: A more readable comment convention is likely to be more reliable, since the reader can more easily determine errors, so Ada is likely to be more reliable in its comment convention. The main feature of Ada comments that perhaps increases their reliability is their *locality of reference:* all comments are clearly indicated locally, without the need for a proper matching symbol farther on. The nested comment issue in C, mentioned above, is also a source of errors, since more than one comment closer will result in compiler errors that are difficult to track down. Thus, C's comment convention is less reliable than Ada's.

C++ Comment Convention: C++ cannot use the Ada convention of a double-dash, since it is already in use as a decrement operator, and a translator would have no way of guessing which use was meant.

2.15 The principle of locality implies that sufficient language constructs should be available to allow variables to be declared close to their use, and also to allow the restriction of access to variables in areas of the program where they are not supposed to be used. There are several constructs in C that promote the principle of locality. First, C allows blocks containing declarations (surrounded by curly brackets {...}) to occur anywhere a statement might occur, thus allowing local declarations a great deal of freedom. For example, if a temporary variable is only needed for a few lines of code, then we can write in C:

```
{ int temp = x;
  x = y;
  y = temp;
}
```

and **temp** is restricted to the region within the curly brackets. Ada, C++, and Java all permit this as well; additionally C++, Java, and (recently) C also allow declarations to occur *anywhere* in a block (Ada does not). C also allows local variables to be declared **static**, which allows static allocation while preserving restriction of scope. The **static** attribute can also be applied to external variables, where it restricts access to the compilation unit (other separately compiled code files cannot access it). This also promotes the principle of locality. On the other hand, C only allows global function declarations -- no local functions are allowed. Thus, a function used in only one small part of a program must still be declared globally. This compromises the principle of locality. C also lacks a module mechanism to clearly distinguish what should

and should not be visible among separately compiled files. C++ and Java offer the class, which allows a much finer control over access. Ada has the **`package`** construct, which allows significant control over access as well (though not as fine as C++ and Java).

2.21. An obvious advantage of arbitrary-precision integers is that it frees the behavior of integers from any dependence on the (implementation-dependent) representation of the integers, including elimination of the need for considering overflow in the language definition. The disadvantage is that the size of memory needed for an integer is not static (fixed prior to execution), and therefore memory for an integer must be dynamically allocated. This has serious consequences for a language like C. For example, in the following code,

```
struct X { int i; char c; } x;
   ...
x.i = 100;
x.c = 'C';
  ...
x.i = 100000000000000000;
...
```

the allocation of new storage for **`x`** on the second assignment to **`x.i`** means **`x.b`** must also be reallocated and copied, unless indirection is used. Indeed, a reasonable approach would be to make integer variables into pointers and automatically allocate and deallocate them on assignment. This means that the runtime system must become "fully dynamic" (with a garbage collector), substantially complicating the implementation of the language. The arithmetic operators, such as addition and multiplication, also become much less efficient, since a software algorithm must be used in place of hardware operations.

In principle, a real number with arbitrary precision can be represented in the same way as an arbitrary-precision integer, with the addition of a distinguished position (the position of the decimal point). For example, 33.256 could be represented as (33256,2), the 2 expressing the fact that the decimal point is after the second digit. (Note that this is like scientific notation, with the 2 representing a power of 10: $33.256 = .33256 * 10^2$.) The same comments hold for such reals as for arbitrary-precision integers. However, there is a further complication: while integer operations *always* result in a finite number of digits, real operations can result in infinitely many digits (consider the result of 1.0/3.0 or sqrt(2.0)). How many digits should these results get? Any answer is going to have to be arbitrary. For this reason, even systems with arbitrary-precision integers often place restrictions on the precision of real numbers. (Scheme calls any number with a decimal point *inexact*, and any time an integer—which is exact—is converted to a real, it becomes inexact, and some of its digits may be lost.)

2.22 We answer parts (a) and (b) together in the following.

1. **Design, don't hack**. Basically this means: have some design goals and a plan for achieving them. A set of clear goals was in fact a great strength of the C++ design effort; whether Stroustrup ever had an extensive plan for achieving them is less clear, especially since one of the goals was to allow the language to expand based on practical experience. However, the C++ design effort can be said to have substantially met this criterion.
2. **Study other designs**. Clearly, this means: know what other languages have done well and badly, and choose the appropriate mix of features from them. Definitely this criterion was met, as Stroustrup knew fairly early what features of Simula and C he wanted in the core of C++. Later, he also borrowed carefully from ML, CLU, Ada, and even Algol68.
3. **Design top-down**. Quote from FOLDOC (http://foldoc.doc.ic.ac.uk/foldoc/): "The software design technique which aims to describe functionality at a very high level, then partition it repeatedly into more detailed levels one level at a time until the detail is sufficient to allow coding." For the design of a programming language, this means: start with the most general goals and criteria, then collect a general set of features that will meet these goals, finally refine the feature descriptions into an actual language.

The C++ design effort met this criterion probably the least of all, since it was mostly designed from the bottom up, adding one feature at a time beginning with C as the basic language.

4. **Know the application area**. Stroustrup clearly based his design effort on his own and others' practical experience, and he knew extremely well the particular simulation applications that he wanted C++ to solve. So the C++ design effort met this criterion very well.
5. **Iterate**. This means: don't try to make all the design decisions at once, but build up carefully from a core, expanding the target goals as subgoals are met. Clearly the C++ design effort met this goal, as Stroustrup made no attempt to add everything at once, but waited to see how one major feature would work out before adding another.