Comparative Programming Languages  
CM20318

Russell Bradford

2023/24

### 1. Variables

Some languages, e.g., Rust and F# take the opposite point of view: variables are constant by default and require the programmer to indicate mutability

In Rust:

let x = 7;  
x = x + 1;

will not compile

[-1]let mut x = 7;  
x = x + 1;

to declare a *mutable* variable

### 2. Variables

Roughly speaking, Rust treats T and mut T as different types, allowing lots of interesting type-based things

Including type checking on mutability

fn inc(n: &mut i32) -> ... will fail to compile a call on an argument that is not mutable

Using the type system to prevent “accidental” mistakes by the programmer

### 3. Variables

Other languages, functional languages in particular, say that *all* variables are constant, no exception

So in Haskell, say, x = 1 sets the value of x to be 1 and that cannot now change

Writing x = x + 1 is always an error

Thus making Haskell code more like mathematics

The “=” should be thought of as a declaration of identity, not as an assignment

### 4. Variables

So x = x + 1 is like telling Haskell that x is a value that equals itself plus 1

**Exercise** The ghci Haskell interpreter behaves differently from the compiler when given this. Why?

### 5. Variables

The consequences of this immutability choice are many, for example you can’t have for loops in the same way as C or Java does, as a loop variable must vary

Instead you use more powerful constructs like recursion or iterators or maps

If you come from a language with mutable variables this seems a problem, but a flexible programmer will realise this can be a good thing and you can write better code because of it

### 6. Variables

In practice, for loops are actually quite limited as an idea: try traversing a tree with a for loop — it’s trivial with recursion

And then you can take such ideas back to non-functional languages such as Python and C++

### 7. Iterators

For example, iterators, an idea pioneered in functional languages that is now appearing in many other languages

Consider this loop that adds 1 to each element of a vector

for (i = 0; i < 10; i++) {  
 v[i] = v[i] + 1;  
}

What happens when you run the code and the vector is only of length 4?

### 8. Iterators

A “safe” language such as Java, will take time to check the indices i in v[i] as it runs to make sure they are within the range of the size of the vector v

This avoids the code trying to access beyond the end of the vector

It will produce some kind of error message when run

### 9. Iterators

An “unsafe” language, like C, doesn’t check and your code will happily access whatever happens to be in memory after the vector

Your code may work and give the correct results, it may crash, or — much worse — it may *seem* to work and silently give incorrect results

This is one of the common sources of memory bugs in production software (recall the statement from Microsoft)

### 10. Iterators

The trade-off here is that runtime checks like this are expensive, meaning they slow down the code a lot

C: no checks, fast code, easy bugs

Java: checks, slower code, catching more bugs

### 11. Iterators

Sometimes — just sometimes — the compiler can analyse such code and deduce when it is safe and then it can compile the code with no checking

The checking has been done by the compiler, so doesn’t need to be done at runtime

Unfortunately, for a lot of code this is not even theoretically possible to deduce

### 12. Iterators

Another very common coding error:

for (i = 0; i < len(a); i++) {  
 print(a[i] + a[i+1]);  
}

which is much less “visible” to the programmer, and to the compiler

### 13. Iterators

So we are led to think about iterators (actual syntax varies according to the language):

for val in v do { val = val + 1; }

Here val takes successive references to the elements in v, namely v[0], then v[1] (and then adds 1 to each)

And note that (a) it never goes off the end of the vector, (b) it doesn’t need runtime checks

The iterator goes though exactly and only the elements in the vector, so there is nothing that needs checking

**Exercise** For C geeks. Why can’t C support iterators like this?

### 14. Iterators

Iterators give both safety and speed, in code that is often simpler and closer to the way we think

Compare:  
“add 1 to each value in the vector”  
and  
“for val in v do val = val + 1; ”

And we know the loop is “safe”, e.g., we won’t access beyond the ends of vectors

### 15. Iterators

Is there a runtime cost from using these more abstract iterators?

Depending on the language and compiler, iterators might be

* slower than the equivalent for loop
* the same speed (even might compile to identical code)
* faster (by not needing array bound checks, for example)

These days 2+3 are quite common, but not guaranteed

### 16. Iterators

So why do people still use loops?

Many reasons:

* they are unfamiliar with iterators
* the language they are using does not support them
* they are doing something a bit more complicated so iterators are not so straightforward to use

### 17. Iterators

Consider the Python loop

for i in range(1, len(a)):  
 print(a[i] + a[i-1]);  
}

against an iterator version

for (ai, ai1) in zip(a, a[1:]) {  
 print(ai + ai1);  
}

Which is “better”?

### 18. Iterators

Note that for loops give you indices

Not so useful when your datastructure is not indexed

Iterators give you the values in your datastructure, and don’t care what the datastructure is

For a lot of use cases iterators are neater and simpler to use

### 19. Iterators

**Exercise** For C++ hackers. C++ has recently added iterators under the guise of a special for loop. Read about this

**Exercise** For Java hackers. Read about Java’s support for iterators

**Exercise** Look at Python iterators and enumerate

### 20. Iterators

**Exercise** Advanced Python. Consider the difference in the code generated for

def sum1(vec):  
 s = 0  
 for i in range(len(vec)):  
 s + vec[i]  
 return s

and

def sum2(vec):  
 s = 0  
 for v in vec:  
 s += v  
 return s

(import dis to disassemble code)

### 21. Iterators

**Exercise** Iterators can work on *any* datastructure where there is a clear way of going through the elements one-by-one. For example, trees, lists and hash tables. Think about the code you need to write to add 1 to every element in a tree (a) using for loops; (b) using recursion; (c) using an iterator

**Exercise** Read about maps, a functional version of an iterator

**Advanced Exercise** Read about *internal* vs. *external* iterators

### 22. Aside

#### Zero Cost Abstraction

Iterators (in some languages) can be an example of a *zero cost abstraction*

A Zero Cost Abstraction:

* no runtime overhead when you don’t use it
* no runtime overhead in using it, as compiled code produced is as good as what a programmer could have done directly

But they allow the programmer to work at a higher, more abstract level and possibly they are more likely to write correct code

### 23. Aside

#### Zero Cost Abstraction

More colloquially:

* what you don’t use, you don’t pay for
* what you do use, you couldn’t hand code any better.

### 24. Aside

#### Zero Cost Abstraction

For example: iterators in some languages compile down to code equivalent to (or better than) a for loop, but provide a “higher-level” way of coding

Meaning your code is not going to run slower due to your use of this higher-level abstraction

But is possibly less prone to bugs

### 25. Aside

#### Zero Cost Abstraction

In contrast, exceptions (for example) in some languages have an overhead (of doing extra stuff to deal with possible exceptions) even if you don’t use them, so these would not be zero cost

Cost in space as well as time: e.g., in some OO languages objects are not zero cost on data as they have identifying headers on values taking up space

### 26. Aside

#### Zero Cost Abstraction

Zero cost abstractions often do have a cost in slower compilations!

And it’s *using* the abstraction that has no cost, as the thing you are abstracting still might be expensive, regardless of how you write the code

Perhaps a better description is “zero *additional* cost abstraction”

It’s about the ability to write code at a higher level, but not pay a cost in slower execution

### 27. Evaluation

Next: different ways values are passed into function calls

You might think that when you see a function definition and call like

int f(int p, int q) { ...p...q... }  
...  
z = f(x+y, x-y);

you understand what is happening!

### 28. Aside

Before we start, some words that are often mis-used

* *variable*: a symbol or name that refers to a memory location, e.g., x and cos
* *expression*: a combination of variables, operators and the like that describe a computation, e.g., 42 and x and x + cos(y)
* *value*: some instance of a datatype, e.g., 42 and "hello world"
* *parameters* (of a function): the variables in the function definition, e.g., p and q above
* *arguments* (of a function call): the values passed in when calling a function, e.g., in the above the values of the expressions x+y and x-y

### 29. Aside

* *declaration*: where we indicate a symbol is a variable, often combined with a type declaration, e.g., var y; or int x; or int inc(int x);
* *definition*: the first time we give a variable a value, mostly referring to functions (for non-functions we tend to say *initialisation*). Often combined with a declaration, e.g.,  
  int inc(int x) return x+1; or int x = 42;

### 30. Aside

In particular, be clear on the difference between variables and values, even though we often use lazy language, e.g., “x is 0” rather than the more correct “the variable x has the value 0”

Also distinguish between *function* and *function call*

cos is a (variable that names a) function (some complicated bit of code); while cos(1.0) is a function call (that will be evaluated to produce a value)

Hint. A good way to spot bad programmers at an interview is when they confuse these concepts

**Exercise** Look up *output* parameter and *inout* parameter

**Exercise** Find a language where a variable can be a value

### 31. Aside

**Exercise** In

int f(int p, int q) { ...p...q... }  
...  
z = f(x+y, x-y);

identify the variables, expressions, parameters, arguments, declarations, definitions, functions and function calls

### 32. Evaluation

So we have the code:

int f(int p, int q) { ...p...q... }  
...  
z = f(x+y, x-y);

with a function definition and a corresponding function call

### 33. Evaluation

#### Call by Value

In most languages you are familiar with you expect it to:

* evaluate the arguments x+y and the x-y (in some order…)
* pass those values into f as the values of its parameters p and q
* execute the body of f with p and q having those values

This is *call by value*, where the *values* of the argument expressions are passed to the function call

### 34. Evaluation

#### Call by Value

This is so very common that everyone thinks this is how it is always done

And computer hardware is built in the expectation this is how it is done (stacks, etc.)

Example. C, Java. And most others

### 35. Evaluation

#### Call by Reference

Some languages can do things very differently: in C++, for example, we can write

void inc(int &n)   
{   
 n++;   
}   
...   
int m = 1;   
inc(m);

and the value of m is incremented

The argument declaration is read as “int *reference* n” or “int *ref* n” for short

### 36. Evaluation

#### Call by Reference

This is a *call by reference*

It’s not the *value* of m that gets passed into the function, but (a *reference* to) the variable m itself

Meaning, within the function, operations on n are “really” operations on m

Call by reference passes in the variables, not their values

### 37. Evaluation

#### Call by Reference

So the body of inc is effectively evaluated as

{  
 m++;  
}

Though, in practice, it is implemented in a somewhat different way

### 38. Evaluation

#### Call by Reference

C++ allows both call by value and call by reference, with by value the default

Call by reference allows simple looking code like the above that manipulates variables out of the scope of the function body

Used wisely, it makes for simpler code, potentially more efficient when than call by value, when those values are large structures that are slow to copy around

Used unwisely, it is a source of subtle bugs

Note it is generally regarded as bad practice for code to affect non-local state, e.g., non-local variables, and CBR makes this easy to do by accident

### 39. Evaluation

#### Call by Reference

In the example above calling

inc(a[3]);

is fine as a[3] refers to a memory location; now n in the function is simply a reference to a[3]

But

[-1]inc(2\*m);

is a code bug, and will not compile! Here, 2\*m is an expression that does not refer to a memory location, which is what inc expects