GC

Some languages have Garbage Collection, some don't

GC

Some languages have Garbage Collection, some don't

In code

```
bigclass x = new bigclass(1);
x = new bigclass(2);
```

or

```
(setq x (make <bigclass> 1))
(setq x (make <bigclass> 2))
```

the memory allocated to the new class 1 is no longer accessible to the program

GC

Some languages have Garbage Collection, some don't

In code

```
bigclass x = new bigclass(1);
x = new bigclass(2);
```

or

```
(setq x (make <bigclass> 1))
(setq x (make <bigclass> 2))
```

the memory allocated to the new class 1 is no longer accessible to the program

It is garbage, so we need a garbage collector to search out inaccessible memory and reclaim it for the system

GC

Languages with integral GC include Lisp, Haskell, Java, Perl

GC

Languages with integral GC include Lisp, Haskell, Java, Perl

Languages without integral GC include C, C++

GC

Languages with integral GC include Lisp, Haskell, Java, Perl

Languages without integral GC include C, C++

In languages without GC, if you drop all references to an object, that's the programmer's problem

GC

Languages with integral GC include Lisp, Haskell, Java, Perl

Languages without integral GC include C, C++

In languages without GC, if you drop all references to an object, that's the programmer's problem

The Java-like code above is also valid C++

GC

Languages with integral GC include Lisp, Haskell, Java, Perl

Languages without integral GC include C, C++

In languages without GC, if you drop all references to an object, that's the programmer's problem

The Java-like code above is also valid C++

Thus it is buggy C++ with a *memory leak*

GC

A program with a memory leak will gradually use more and more memory until the OS says it's had enough

GC

A program with a memory leak will gradually use more and more memory until the OS says it's had enough

And then the program probably crashes as the programmer only tested it on small examples

GC

A program with a memory leak will gradually use more and more memory until the OS says it's had enough

And then the program probably crashes as the programmer only tested it on small examples

In Java it is arguably not buggy, but it is definitely poor code as it wastes time creating useless objects

GC

Code written in non-GC languages must be very careful on their use of memory (e.g., use of `malloc` and `free`, or `new` and `delete`)

GC

Code written in non-GC languages must be very careful on their use of memory (e.g., use of `malloc` and `free`, or `new` and `delete`)

Code written in GC languages can let the GC take care of things

GC

Code written in non-GC languages must be very careful on their use of memory (e.g., use of `malloc` and `free`, or `new` and `delete`)

Code written in GC languages can let the GC take care of things

GC: no memory worries, but generally less efficient and encourages sloppy programming

GC

Code written in non-GC languages must be very careful on their use of memory (e.g., use of `malloc` and `free`, or `new` and `delete`)

Code written in GC languages can let the GC take care of things

GC: no memory worries, but generally less efficient and encourages sloppy programming

Non GC: allows accurate memory management, but also encourages buggy programming

GC

A garbage collector can be added to C and C++ (etc.) as a library

GC

A garbage collector can be added to C and C++ (etc.) as a library

Not as precise as an in-built GC as it might miss occasional bits of garbage

GC

A garbage collector can be added to C and C++ (etc.) as a library

Not as precise as an in-built GC as it might miss occasional bits of garbage

Is this the best of both worlds? Unclear

GC

A garbage collector can be added to C and C++ (etc.) as a library

Not as precise as an in-built GC as it might miss occasional bits of garbage

Is this the best of both worlds? Unclear

Better is to write correct code in the first place

GC

A garbage collector can be added to C and C++ (etc.) as a library

Not as precise as an in-built GC as it might miss occasional bits of garbage

Is this the best of both worlds? Unclear

Better is to write correct code in the first place

> *If Java had true garbage collection, most programs would delete themselves upon execution*
> Robert Sewell

Types

We can classify according to how types are treated

Types

We can classify according to how types are treated

NB. Some vagueness over nomenclature in this area

Types

We can classify according to how types are treated

NB. Some vagueness over nomenclature in this area

Note we are *not* specifically talking about OO languages here

Types

We can classify according to how types are treated

NB. Some vagueness over nomenclature in this area

Note we are *not* specifically talking about OO languages here

Types and OO have an interesting relationship, but we shall mostly talking about types as a separate concept from classes or objects

Types

We can classify according to how types are treated

NB. Some vagueness over nomenclature in this area

Note we are *not* specifically talking about OO languages here

Types and OO have an interesting relationship, but we shall mostly talking about types as a separate concept from classes or objects

So the following applies to non-OO languages like C

Types

Static typing: C, Haskell, Java, . . .

Types

Static typing: C, Haskell, Java, . . .

- expressions and types checked at **compile time** for correctness

Types

Static typing: C, Haskell, Java, . . .

- expressions and types checked at **compile time** for correctness
- typed variables

Types

Static typing: C, Haskell, Java, . . .

- expressions and types checked at **compile time** for correctness
- typed variables
- the type of a value is determined by the type of the variable it came from

Types

Static typing: C, Haskell, Java, ...

- expressions and types checked at **compile time** for correctness
- typed variables
- the type of a value is determined by the type of the variable it came from

Most modern languages have some element of static typing, sometimes optionally (Maple, Common Lisp)

Types

Dynamic typing: Lisp, Perl, JavaScript, . . .

Types

Dynamic typing: Lisp, Perl, JavaScript, . . .

- expressions and types checked at **run time**

Types

Dynamic typing: Lisp, Perl, JavaScript, . . .

- expressions and types checked at **run time**
- untyped variables

Types

Dynamic typing: Lisp, Perl, JavaScript, . . .

- expressions and types checked at **run time**
- untyped variables
- values have intrinsic types independent of where they come from

Types

Dynamic typing: Lisp, Perl, JavaScript, . . .

- expressions and types checked at **run time**
- untyped variables
- values have intrinsic types independent of where they come from

Often scripting and prototyping languages are dynamically typed

Types

Strong typing: (very vague) a thing has a definite type and no implicit conversions between types

Types

Strong typing: (very vague) a thing has a definite type and no implicit conversions between types

- expressions checked for type correctness at compile or runtime

Types

Strong typing: (very vague) a thing has a definite type and no implicit conversions between types

- expressions checked for type correctness at compile or runtime
- little to no automatic type conversions, e.g., integer to floating point

Types

Strong typing: (very vague) a thing has a definite type and no implicit conversions between types

- expressions checked for type correctness at compile or runtime
- little to no automatic type conversions, e.g., integer to floating point

Python does no static checking, but does check types at runtime

Types

Strong typing: (very vague) a thing has a definite type and no implicit conversions between types

- expressions checked for type correctness at compile or runtime
- little to no automatic type conversions, e.g., integer to floating point

Python does no static checking, but does check types at runtime

“Strong” seems to cover different ideas in different peoples’ minds, and possibly ought to be avoided as a concept

Types

Perhaps “strong” is better used as a comparator, e.g., “this language is more strongly typed than that one”

Types

Perhaps “strong” is better used as a comparator, e.g., “this language is more strongly typed than that one”

E.g., “Rust is more strongly typed than C”

Types

Perhaps “strong” is better used as a comparator, e.g., “this language is more strongly typed than that one”

E.g., “Rust is more strongly typed than C”

Weak typing: not strongly typed

Types

Untyped: assembler, BCPL, Forth, . . .

Types

Untyped: assembler, BCPL, Forth, . . .

- up to the programmer how to interpret a value

Types

Untyped: assembler, BCPL, Forth, . . .

- up to the programmer how to interpret a value
- all values are just presented as a machine byte or word

Types

Untyped: assembler, BCPL, Forth, . . .

- up to the programmer how to interpret a value
- all values are just presented as a machine byte or word

Not much used as these days types are seen as an essential aid to the programmer

Types

Untyped: assembler, BCPL, Forth, . . .

- up to the programmer how to interpret a value
- all values are just presented as a machine byte or word

Not much used as these days types are seen as an essential aid to the programmer

Though assembler is still more widely used than you might expect

Types

Feet

- BCPL: You shoot yourself somewhere in the leg—you can't get any finer resolution than that

Types

Feet

- BCPL: You shoot yourself somewhere in the leg—you can't get any finer resolution than that
- Forth: Foot yourself in the shoot

Types

Orthogonal to the kinds of types is how a typed language indicates its types

Types

Orthogonal to the kinds of types is how a typed language indicates its types

Manifest Typing: where the program code includes the types of variables, e.g., C

```
int inc(int n)
{
  return n+1;
}
```

Types

Implicit Typing: where the compiler infers any types it needs (as much as it can), e.g., Haskell

```
inc x = x + 1
```

which Haskell determines to be `Num a => a -> a`

Types

Implicit Typing: where the compiler infers any types it needs (as much as it can), e.g., Haskell

```
inc x = x + 1
```

which Haskell determines to be `Num a => a -> a`

Quite often a statically typed, implicit typed language will also have type variables

Types

Implicit Typing: where the compiler infers any types it needs (as much as it can), e.g., Haskell

```
inc x = x + 1
```

which Haskell determines to be `Num a => a -> a`

Quite often a statically typed, implicit typed language will also have type variables

And allow (or require, in ambiguous code) the programmer to include type annotations

Types

But implicit typing is also used in dynamic languages, too, e.g., Lisp

```
(defun inc (n)
  (+ n 1))
```

Types

Comparing these kinds of types:

Types

Comparing these kinds of types:

- Dynamic: flexibility for the programmer, particularly in prototyping where fast coding through few restrictions is important

Types

Comparing these kinds of types:

- Dynamic: flexibility for the programmer, particularly in prototyping where fast coding through few restrictions is important
- Static: types checked at compile time, catching some bugs in the source before the program is run

Types

Comparing these kinds of types:

- Dynamic: flexibility for the programmer, particularly in prototyping where fast coding through few restrictions is important
- Static: types checked at compile time, catching some bugs in the source before the program is run
- Untyped: no type errors possible

Types

We can look at what each do when presented with code like `a+b`

Types

We can look at what each do when presented with code like `a+b`

- what a compiler needs to do

Types

We can look at what each do when presented with code like `a+b`

- what a compiler needs to do
- what happens when the compiled program is running

Types

We can look at what each do when presented with code like `a+b`

- what a compiler needs to do
- what happens when the compiled program is running

An interpreter would need to do both stages above while executing

Types

A compiler for a dynamic language will need to output code that

- checks if `a` is a number
- checks if `b` is a number
- if so call the appropriate add function
- else does some coercions then adds; or just signals an error, as appropriate

Types

At runtime this code will be executed

Types

At runtime this code will be executed

There must be a runtime check (in the absence of clever optimisations)

Types

At runtime this code will be executed

There must be a runtime check (in the absence of clever optimisations)

Thus a lot of checking overhead before actually doing the expected operation

Types

Static. The compiler will determine the types of `a` and `b` and output code for the appropriate add operation

Types

Static. The compiler will determine the types of `a` and `b` and output code for the appropriate add operation

It does not need to include code to check the values of `a` and `b` as they *must* be numbers

Types

Static. The compiler will determine the types of `a` and `b` and output code for the appropriate add operation

It does not need to include code to check the values of `a` and `b` as they *must* be numbers

At runtime this simple operation will be executed

Types

Static. The compiler will determine the types of `a` and `b` and output code for the appropriate add operation

It does not need to include code to check the values of `a` and `b` as they *must* be numbers

At runtime this simple operation will be executed

There's no runtime check

Types

Untyped. The compiler will output code to add the values (presumably an integer add) regardless of what the programmer thinks they happen to be

Types

Untyped. The compiler will output code to add the values (presumably an integer add) regardless of what the programmer thinks they happen to be

At runtime this simple operation will be executed

Types

Untyped. The compiler will output code to add the values (presumably an integer add) regardless of what the programmer thinks they happen to be

At runtime this simple operation will be executed

There's nothing to check!

Types

If I have a box marked “Socks” I don’t need to check what comes out of it before I put them on my feet

Types

If I have a box marked “Socks” I don’t need to check what comes out of it before I put them on my feet

If I have an unmarked box, I need to look at what I get, first

Types

In the case of OO method lookup we can also see significant differences

Types

In the case of OO method lookup we can also see significant differences

Suppose we have code `a.foo()`

Types

Dynamic. The compiler will output code to determine the current value of `a`, plus code to look up a method that matches this, then code to call the method

Types

Dynamic. The compiler will output code to determine the current value of `a`, plus code to look up a method that matches this, then code to call the method

At runtime this complex code is executed

Types

Dynamic. The compiler will output code to determine the current value of `a`, plus code to look up a method that matches this, then code to call the method

At runtime this complex code is executed

Again, a lot of overhead before the method can be run

Types

Static. The compiler will determine the type of `a`, find the appropriate method, and output code to directly call that method

Types

Static. The compiler will determine the type of `a`, find the appropriate method, and output code to directly call that method

At runtime the code of the method is called directly (the lookup has already been done by the compiler)

Types

Untyped. No OO possible!

Types

It seems that static is always faster to run and is therefore better

Types

It seems that static is always faster to run and is therefore better

But the hidden point in dynamic is “the current value of `a`”

Types

It seems that static is always faster to run and is therefore better

But the hidden point in dynamic is “the current value of `a`”

The type of the object held in `a` can vary at runtime, so the appropriate method can vary at runtime

Types

It seems that static is always faster to run and is therefore better

But the hidden point in dynamic is “the current value of `a`”

The type of the object held in `a` can vary at runtime, so the appropriate method can vary at runtime

The same piece of code might need a different method each time you come to it

Types

It seems that static is always faster to run and is therefore better

But the hidden point in dynamic is “the current value of `a`”

The type of the object held in `a` can vary at runtime, so the appropriate method can vary at runtime

The same piece of code might need a different method each time you come to it

This is the essence of the flexibility of dynamic languages

Types

It seems that static is always faster to run and is therefore better

But the hidden point in dynamic is “the current value of `a`”

The type of the object held in `a` can vary at runtime, so the appropriate method can vary at runtime

The same piece of code might need a different method each time you come to it

This is the essence of the flexibility of dynamic languages

The cost is the speed

Types

Duck typing is a particular kind of dynamic: examples are Python, JavaScript, Common Lisp, Ruby

Types

Duck typing is a particular kind of dynamic: examples are Python, JavaScript, Common Lisp, Ruby

To evaluate `a.foo()` the interpreter/compiler examines the current value of `a` to see if there is a `foo` method defined on it and calls it if it find one

Types

Duck typing is a particular kind of dynamic: examples are Python, JavaScript, Common Lisp, Ruby

To evaluate `a.foo()` the interpreter/compiler examines the current value of `a` to see if there is a `foo` method defined on it and calls it if it find one

It is a runtime error if no method is found

Types

Duck typing is a particular kind of dynamic: examples are Python, JavaScript, Common Lisp, Ruby

To evaluate `a.foo()` the interpreter/compiler examines the current value of `a` to see if there is a `foo` method defined on it and calls it if it find one

It is a runtime error if no method is found

The same line of code may or may not work depending on the current value of `a`!

Types

Duck typing is a particular kind of dynamic: examples are Python, JavaScript, Common Lisp, Ruby

To evaluate `a.foo()` the interpreter/compiler examines the current value of `a` to see if there is a `foo` method defined on it and calls it if it find one

It is a runtime error if no method is found

The same line of code may or may not work depending on the current value of `a`!

“If it walks like a duck and talks like a duck, then it is a duck”

Types

Exercise. Consider the Python

```
def two10(n):
    for i in range(10):
        n = 2*n
    return n

two10(1)
two10("1")
```

Types

- Manifest: allows for a simpler compiler as it doesn't have to work so hard. Requires the programmer to think explicitly about the types. For example, Rust requires type annotations on function declarations, even though it is mostly implicit and could infer the types itself: the language designers thought that it would be good practice to get the programmer thinking

Types

- Manifest: allows for a simpler compiler as it doesn't have to work so hard. Requires the programmer to think explicitly about the types. For example, Rust requires type annotations on function declarations, even though it is mostly implicit and could infer the types itself: the language designers thought that it would be good practice to get the programmer thinking
- Implicit: allows for simpler code, but (in dynamic languages) also allows for more trivial type errors

Types

Types are often further divided:

Types

Types are often further divided:

- Monomorphic/Lexical: types are determined by the variables and checked by comparing variable names

```
int f(int x) { ... x ... }
```

Types

Types are often further divided:

- Monomorphic/Lexical: types are determined by the variables and checked by comparing variable names

  ```
  int f(int x) { ... x ... }
  ```

- Polymorphic: types are identified by variables and by type schema using type variables

  ```
  cons: a * [a] -> [a]
  ```

  Type inference is needed here

Types

Many people separate the ideas of *polymorphic* and *overloading*

Types

Many people separate the ideas of *polymorphic* and *overloading*

Overloading

Some languages (e.g., C++, not C) allow:

```
int f(int x) { return -x; }
double f(double x) { return 2.0*x; }
```

Types

Many people separate the ideas of *polymorphic* and *overloading*

Overloading

Some languages (e.g., C++, not C) allow:

```
int f(int x) { return -x; }
double f(double x) { return 2.0*x; }
```

Multiple different functions with the same name. The compiler can distinguish which we mean by the argument types

Types

Many people separate the ideas of *polymorphic* and *overloading*

Overloading

Some languages (e.g., C++, not C) allow:

```
int f(int x) { return -x; }
double f(double x) { return 2.0*x; }
```

Multiple different functions with the same name. The compiler can distinguish which we mean by the argument types

Different chunks of code are compiled for each function

Types

Many people separate the ideas of *polymorphic* and *overloading*

Overloading

Some languages (e.g., C++, not C) allow:

```
int f(int x) { return -x; }
double f(double x) { return 2.0*x; }
```

Multiple different functions with the same name. The compiler can distinguish which we mean by the argument types

Different chunks of code are compiled for each function

The function bodies can be completely different: it's almost incidental that the functions have the same name

Types

`f(2)` is compiled as a call to the first
`f(2.0)` is compiled as a call to the second

Types

`f(2)` is compiled as a call to the first
`f(2.0)` is compiled as a call to the second

In fact, typically the compiler internally renames ("name mangling") the two functions as (something like) `f_int` and `f_double`, so giving them distinct names

Types

`f(2)` is compiled as a call to the first
`f(2.0)` is compiled as a call to the second

In fact, typically the compiler internally renames ("name mangling") the two functions as (something like) `f_int` and `f_double`, so giving them distinct names

`f(2)` is compiled as `f_int(2)`
`f(2.0)` is compiled as `f_double(2.0)`

Types

`f(2)` is compiled as a call to the first
`f(2.0)` is compiled as a call to the second

In fact, typically the compiler internally renames ("name mangling") the two functions as (something like) `f_int` and `f_double`, so giving them distinct names

`f(2)` is compiled as `f_int(2)`
`f(2.0)` is compiled as `f_double(2.0)`

Overloading is very widespread and appears (in a limited way) in lots of languages: common functions like + are often overloaded

Types

Polymorphic

```
cons x y = x:y
```

The *same* function code works on many types

Types

Polymorphic

```
cons x y = x:y
```

The *same* function code works on many types

There is just one chunk of code that works on multiple types

Types

Polymorphic

```
cons x y = x:y
```

The *same* function code works on many types

There is just one chunk of code that works on multiple types

`cons 1 [2]` runs the *same code* as `cons 1.0 [2.0]`

Types

Polymorphic

```
cons x y = x:y
```

The *same* function code works on many types

There is just one chunk of code that works on multiple types

`cons 1 [2]` runs the *same code* as `cons 1.0 [2.0]`

`cons` doesn't care about the types of its arguments

Types

Beware of overloading disguised as polymorphism:

```
template <class T>          // T is a type variable
T f(T x) { return -x; }
... f(2) ...
... f(2.0) ...
```

in C++

```
fn f<T>(x: T) -> T
  where T: Neg<Output=T> { // T implements negation
  -x
}
... f(2) ...
... f(2.0) ...
```

in Rust

Types

The programmer writes out the same source code for a function that will work on many types: superficially this looks like polymorphism

Types

The programmer writes out the same source code for a function that will work on many types: superficially this looks like polymorphism

Here, the compiler also rewrites the code for the individual `int` and `double` versions and compiles those (or does the equivalent)

Types

The programmer writes out the same source code for a function that will work on many types: superficially this looks like polymorphism

Here, the compiler also rewrites the code for the individual `int` and `double` versions and compiles those (or does the equivalent)

```
int f(int x) { return -x; }
double f(double x) { return -x; }
```

This is called *monomorphization*: replacing something apparently polymorphic with multiple monomorphic bits of code

Types

The programmer writes out the same source code for a function that will work on many types: superficially this looks like polymorphism

Here, the compiler also rewrites the code for the individual `int` and `double` versions and compiles those (or does the equivalent)

```
int f(int x) { return -x; }
double f(double x) { return -x; }
```

This is called *monomorphization*: replacing something apparently polymorphic with multiple monomorphic bits of code

This is actually overloading as the underlying code to negate an integer is different from the code to negate a floating point value

Types

Be aware that some people classify overloading as a particular kind of polymorphism, even though overloading uses different pieces of code for each type

Types

Be aware that some people classify overloading as a particular kind of polymorphism, even though overloading uses different pieces of code for each type

They sometimes call it *ad hoc polymorphism*, in contrast with true polymorphism, *parametric polymorphism*

Types

Note that polymorphism and overloading are not reliant on OO: in fact they both predate OO

Types

Note that polymorphism and overloading are not reliant on OO: in fact they both predate OO

A large number of languages overload the arithmetic functions like + and $*$

Types

Note that polymorphism and overloading are not reliant on OO: in fact they both predate OO

A large number of languages overload the arithmetic functions like + and $*$

Lisp has function polymorphism (`cons`, `length`, etc.)

Types

Note that polymorphism and overloading are not reliant on OO: in fact they both predate OO

A large number of languages overload the arithmetic functions like + and $*$

Lisp has function polymorphism (`cons`, `length`, etc.)

Also note that method overriding is merely an example of overloading

Types Conclusion

These days types are considered to be an essential part of a language

Types Conclusion

These days types are considered to be an essential part of a language

And so appear in many different kinds of ways

Types Conclusion

These days types are considered to be an essential part of a language

And so appear in many different kinds of ways

They are intended to reduce errors, or find errors more quickly

Types Conclusion

These days types are considered to be an essential part of a language

And so appear in many different kinds of ways

They are intended to reduce errors, or find errors more quickly

Even in early untyped languages there was a recommendation that the intended type of a value be reflected in the name of a variable

Types Conclusion

These days types are considered to be an essential part of a language

And so appear in many different kinds of ways

They are intended to reduce errors, or find errors more quickly

Even in early untyped languages there was a recommendation that the intended type of a value be reflected in the name of a variable

iIndex, fSalary. See *Hungarian notation*

Types Conclusion

There are many places to check for errors

Types Conclusion

There are many places to check for errors

- compile time: mostly type errors

Types Conclusion

There are many places to check for errors

- compile time: mostly type errors
- run time: e.g., division by 0, null pointers

Types Conclusion

There are many places to check for errors

- compile time: mostly type errors
- run time: e.g., division by 0, null pointers

The Rust type system is so strong it can check for null pointers at compile time and so can avoid this kind of run time error

Types Conclusion

There are many places to check for errors

- compile time: mostly type errors
- run time: e.g., division by 0, null pointers

The Rust type system is so strong it can check for null pointers at compile time and so can avoid this kind of run time error

Haskell has no (explicit) pointers, and avoids this, too

Types Conclusion

There are many places to check for errors

- compile time: mostly type errors
- run time: e.g., division by 0, null pointers

The Rust type system is so strong it can check for null pointers at compile time and so can avoid this kind of run time error

Haskell has no (explicit) pointers, and avoids this, too

Java has no explicit pointers, but still manages to get null pointer exceptions

Types Conclusion

There are many places to check for errors

- compile time: mostly type errors
- run time: e.g., division by 0, null pointers

The Rust type system is so strong it can check for null pointers at compile time and so can avoid this kind of run time error

Haskell has no (explicit) pointers, and avoids this, too

Java has no explicit pointers, but still manages to get null pointer exceptions

I don't think there could reasonably be a language that checks for 0 division at compile time!

Types Conclusion

There are other places for errors we often forget about

Types Conclusion

There are other places for errors we often forget about

- link time, load time: making sure libraries are present and correctly called

Types Conclusion

There are other places for errors we often forget about

- link time, load time: making sure libraries are present and correctly called
- coding time: getting it right in the first place

Types Conclusion

“Strong types are for weak minds”
Anon.

Evaluation

Next: different ways values are passed into function calls

Evaluation

Next: different ways values are passed into function calls

You might think that when you see a function call like

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

you understand what is happening!

Evaluation

Call by Value

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

Evaluation

Call by Value

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

- evaluate the `x+y` and the `x-y` (in some order...)

Evaluation

Call by Value

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

- evaluate the `x+y` and the `x-y` (in some order...)
- pass those values into `f` as the values of its parameters `p` and `q`

Evaluation

Call by Value

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

- evaluate the `x+y` and the `x-y` (in some order...)
- pass those values into `f` as the values of its parameters `p` and `q`

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

Evaluation

Call by Value

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

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.)

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. And most others

Evaluation

Call by Reference

In C++ we can write

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

and the value of `m` is incremented

Evaluation

Call by Reference

In C++ 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`"

Evaluation

Call by Reference

This is a *call by reference*

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`

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`

Meaning, within the function, operations on `n` are "really" operations on `m`

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`

Meaning, within the function, operations on `n` are "really" operations on `m`

Call by reference passes in the variables, not their values

Evaluation

Call by Reference

C++ allows both call by value and call by reference

Evaluation

Call by Reference

C++ allows both call by value and call by reference

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

Evaluation

Call by Reference

C++ allows both call by value and call by reference

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

Evaluation

Call by Reference

C++ allows both call by value and call by reference

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

Used unwisely, it is a source of subtle bugs

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]`

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

```
inc(2*m);
```

is a bug, and will not compile!

Evaluation

Call by Reference

Note that in C and other languages we can use pointers

```
void inc(int *n)
{
  *n++;
}
...
int m;
...
inc(&m);
```

will update the value of `m`

Evaluation

Call by Reference

This looks like call by reference, but C is purely call by value

Evaluation

Call by Reference

This looks like call by reference, but C is purely call by value

It's just that the value is a reference!

Evaluation

Call by Reference

This looks like call by reference, but C is purely call by value

It's just that the value is a reference!

Exercise. Using & in the function declaration in C++ is a hint on how C++ implements call by reference. Read about this

Evaluation

Call by Name

Call by name takes this a bit further, lifting the restriction that the arguments are variables

Evaluation

Call by Name

Call by name takes this a bit further, lifting the restriction that the arguments are variables

For example the function

```
integer procedure sumsq(n, m)
integer n, m;
begin
  sumsq := (n + m)*(n + m);
end;
```

that squares the sum of the arguments

Evaluation

Call by Name

Then

```
sumsq(x+1, y+2)
```

is evaluated as

```
((x+1) + (y+2)) * ((x+1) + (y+2))
```

i.e., the whole *expressions* in the call are substituted in the function body, which is then evaluated

Evaluation

Call by Name

Then

```
sumsq(x+1, y+2)
```

is evaluated as

```
((x+1) + (y+2)) * ((x+1) + (y+2))
```

i.e., the whole *expressions* in the call are substituted in the function body, which is then evaluated

Exercise. Compare with *inlining* code

Evaluation

Call by Name

Care is taken over name clashes so that local variables in the function body will never coincide with variables passed in

Evaluation

Call by Name

Care is taken over name clashes so that local variables in the function body will never coincide with variables passed in

```
integer procedure foo(n)
integer n;
begin integer m;
  m := 1;
  foo := n + m;
end;
```

Evaluation

Call by Name

Care is taken over name clashes so that local variables in the function body will never coincide with variables passed in

```
integer procedure foo(n)
integer n;
begin integer m;
  m := 1;
  foo := n + m;
end;
```

And then `foo(m + 1)` is *not* evaluated as

```
begin integer m;
  m := 1;
  foo := (m + 1) + m;
end;
```

as there is inadvertent *capture* of the global `m` by the local `m`

Evaluation

Call by Name

Rather, something more like

```
begin integer m001;
  m001 := 1;
  foo := (m + 1) + m001;
end;
```

where the local `m` is renamed

Evaluation

Call by Name

Rather, something more like

```
begin integer m001;
  m001 := 1;
  foo := (m + 1) + m001;
end;
```

where the local `m` is renamed

Example. Algol 60

Evaluation

Call by Name

Rather, something more like

```
begin integer m001;
  m001 := 1;
  foo := (m + 1) + m001;
end;
```

where the local `m` is renamed

Example. Algol 60

Exercise. Read about *Jensen's Device*

Evaluation

Call by Name

This is an interesting evaluation strategy that is sometimes more efficient than call by value:

```
integer procedure k(x, y)
integer x, y;
begin
  k := x;
end
...
n = k(1+1, 1+2+3+4+5+6+7);
```

Evaluation

Call by Name

This is an interesting evaluation strategy that is sometimes more efficient than call by value:

```
integer procedure k(x, y)
integer x, y;
begin
  k := x;
end
...
n = k(1+1, 1+2+3+4+5+6+7);
```

Here the second argument is not used in the function body, so will not be substituted in, and therefore not evaluated

Evaluation

Call by Name

This is an interesting evaluation strategy that is sometimes more efficient than call by value:

```
integer procedure k(x, y)
integer x, y;
begin
  k := x;
end
...
n = k(1+1, 1+2+3+4+5+6+7);
```

Here the second argument is not used in the function body, so will not be substituted in, and therefore not evaluated

(Note: of the millions of functions I have written, only vanishingly few of them have had unused arguments. . .)

Evaluation

Call by Name

On the other hand, the call by name substitution mechanism is usually quite expensive, so we don't often win overall

Evaluation

Call by Name

On the other hand, the call by name substitution mechanism is usually quite expensive, so we don't often win overall

And in the example above, the `x+1` and `y+2` are both evaluated *twice*, less efficient than a call by value

Evaluation

Call by Name

On the other hand, the call by name substitution mechanism is usually quite expensive, so we don't often win overall

And in the example above, the `x+1` and `y+2` are both evaluated *twice*, less efficient than a call by value

Algol 60 also allows call by value, for this reason

Evaluation

Call by Name

On the other hand, the call by name substitution mechanism is usually quite expensive, so we don't often win overall

And in the example above, the `x+1` and `y+2` are both evaluated *twice*, less efficient than a call by value

Algol 60 also allows call by value, for this reason

Exercise. Compare with non-strict evaluation

Evaluation

Call by Need

Call by need, also called *lazy evaluation*

Evaluation

Call by Need

Call by need, also called *lazy evaluation*

A form of call by name that tries to get closer to the efficiency of call by value, where you only evaluate a given argument once

Evaluation

Call by Need

Call by need, also called *lazy evaluation*

A form of call by name that tries to get closer to the efficiency of call by value, where you only evaluate a given argument once

Now

```
sumsq(x+1, y+2)
```

would evaluate as call by name, but now the `x+1` and the `y+2` are only evaluated at most *once* each

Evaluation

Call by Need

The argument evaluations are *memoised*, i.e., remembered, so when the same expression is seen again (within the function body), the previously computed valued can simply be reused

Evaluation

Call by Need

The argument evaluations are *memoised*, i.e., remembered, so when the same expression is seen again (within the function body), the previously computed valued can simply be reused

The trade-off here is single evaluation of the arguments against a more complicated evaluation mechanism

Evaluation

Call by Need

The argument evaluations are *memoised*, i.e., remembered, so when the same expression is seen again (within the function body), the previously computed valued can simply be reused

The trade-off here is single evaluation of the arguments against a more complicated evaluation mechanism

Example. Haskell

Evaluation

Call by Need

Proponents of languages like Haskell claim the compiler can analyse the code, spot the actual use of an expression, and compile it in “the normal way” so avoiding the cost of lazy and memoisation

Evaluation

Call by Need

Proponents of languages like Haskell claim the compiler can analyse the code, spot the actual use of an expression, and compile it in “the normal way” so avoiding the cost of lazy and memoisation

This is true, *if* the compiler is good enough

Evaluation

Call by Need

Proponents of languages like Haskell claim the compiler can analyse the code, spot the actual use of an expression, and compile it in “the normal way” so avoiding the cost of lazy and memoisation

This is true, *if* the compiler is good enough

There has been a long history of language design predicated on the future existence of a “sufficiently clever compiler”

Evaluation

Call by Need

Proponents of languages like Haskell claim the compiler can analyse the code, spot the actual use of an expression, and compile it in “the normal way” so avoiding the cost of lazy and memoisation

This is true, *if* the compiler is good enough

There has been a long history of language design predicated on the future existence of a “sufficiently clever compiler”

Mostly, that compiler was never created

Evaluation

Call by Need

Proponents of languages like Haskell claim the compiler can analyse the code, spot the actual use of an expression, and compile it in “the normal way” so avoiding the cost of lazy and memoisation

This is true, *if* the compiler is good enough

There has been a long history of language design predicated on the future existence of a “sufficiently clever compiler”

Mostly, that compiler was never created

Perhaps, in the last couple of years, such compilers are just about beginning to appear

Evaluation

Examples. Suppose we have

```
struct Big {
  int stuff[1000];
  int things[1000];
};
```

This structure might occupy 8000 bytes

Evaluation

Examples. Suppose we have

```
struct Big {
  int stuff[1000];
  int things[1000];
};
```

This structure might occupy 8000 bytes

(Be careful about saying “a big value”: if you have `int n = 100000000;` then the value of `n` is big, but the variable `n` occupies maybe just 4 bytes)

Evaluation

Then for `struct Big b = ...` we get

call by value
`foo(b);` slow, copies 8000 bytes of `b` into the function
`bar(&b);` fast, copies 8 (perhaps) bytes of pointer into the function

Evaluation

call by reference

`foo(b);` fast, copies 8 bytes of reference to `b` (a pointer) into the function

Evaluation

call by name

`foo(b);` expression `b` is substituted into function; cost likely high without a good optimiser

Evaluation

call by need

`foo(b);` as call by name, but with extra cost of the memoisation check

Evaluation

Exercise. Many other evaluation strategies have been thought about. Read about them

Exercise. Is Java call by value or call by reference? Explain.

Evaluation

Exercise.

```
func foo(n) {
   if (n < 2) { return 1; }
   return n*foo(n-1);
}
```

Trace the evaluation of this function in a call by need language

Application

In contrast to the “generic” languages, several applications areas have languages specifically designed for that area

Application

In contrast to the “generic” languages, several applications areas have languages specifically designed for that area

Maple: maths. The basic datatypes are numbers, polynomials, matrices, functions (trig, exp, etc.) and the like. The basic operations are arithmetics of all these things, integration, differentiation, and so on

Application

```
 expand((x+1)^100);
     100        99         98           97            96             95
1 + x    + 100 x   + 4950 x   + 161700 x   + 3921225 x   + 75287520 x

                   94                93                  92                   91
     + 1192052400 x   + 16007560800 x   + 186087894300 x   + 1902231808400 x

                     90                    89                      88
     + 17310309456440 x   + 141629804643600 x   + 1050421051106700 x

                      87                      86                        85
     + 7110542499799200 x   + 44186942677323600 x   + 253338471349988640 x

                            84                          83
     + 1345860629046814650 x   + 6650134872937201800 x

                             82                             81
     + 30664510802988208300 x   + 132341572939212267400 x

                              80                               79
     + 535983370403809682970 x   + 2041841411062132125600 x

                               78                                77
     + 7332066885177656269200 x   + 24865270306254660391200 x

                                76                                  75
     + 79776075565900368755100 x   + 24251926972033712101550 4 x

     ...
```

Application

Cobol: business. Data on employees, payroll and so on

Application

Cobol: business. Data on employees, payroll and so on

Fortran: numerical computation. Numbers and almost nothing else

Application

Visual Basic: interfaces, teaching

Application

Visual Basic: interfaces, teaching

Postscript and its compact cousin, PDF: printing and display

Application

Visual Basic: interfaces, teaching

Postscript and its compact cousin, PDF: printing and display

Cisco IOS (Internetwork Operating System): Network hardware

Application

Visual Basic: interfaces, teaching

Postscript and its compact cousin, PDF: printing and display

Cisco IOS (Internetwork Operating System): Network hardware

Actionscript (derived from JavaScript): Flash Player

Application

Visual Basic: interfaces, teaching

Postscript and its compact cousin, PDF: printing and display

Cisco IOS (Internetwork Operating System): Network hardware

Actionscript (derived from JavaScript): Flash Player

And so on

Application

Visual Basic: interfaces, teaching

Postscript and its compact cousin, PDF: printing and display

Cisco IOS (Internetwork Operating System): Network hardware

Actionscript (derived from JavaScript): Flash Player

And so on

It is so easy to create new language these days, people rarely stop to consider whether they should: is there an existing language that would suit this application well?

Application

Exercise. Go, Rust and Apple Swift are new languages presently being developed. Look at them and decide what is new and different in each language (if anything)