Memory Management

GC

GC can be problematic in systems that require real-time behaviour, e.g., car controls, video streaming

Memory Management

GC

GC can be problematic in systems that require real-time behaviour, e.g., car controls, video streaming

The code to do a GC quite often needs the main computation to pause (“Stop the world” collection) while it searches for inaccessible chunks of memory

Memory Management

GC

GC can be problematic in systems that require real-time behaviour, e.g., car controls, video streaming

The code to do a GC quite often needs the main computation to pause ("Stop the world" collection) while it searches for inaccessible chunks of memory

These pauses can be considerable fractions of a second

Memory Management

GC

GC can be problematic in systems that require real-time behaviour, e.g., car controls, video streaming

The code to do a GC quite often needs the main computation to pause (“Stop the world” collection) while it searches for inaccessible chunks of memory

These pauses can be considerable fractions of a second

Leading to glitches in your video; or your car crashing into a wall

Memory Management

GC

GC can be problematic in systems that require real-time behaviour, e.g., car controls, video streaming

The code to do a GC quite often needs the main computation to pause (“Stop the world” collection) while it searches for inaccessible chunks of memory

These pauses can be considerable fractions of a second

Leading to glitches in your video; or your car crashing into a wall

A GC pause will inevitably happen at the most inconvenient point in time

Memory Management

GC

GC can be problematic in systems that require real-time behaviour, e.g., car controls, video streaming

The code to do a GC quite often needs the main computation to pause (“Stop the world” collection) while it searches for inaccessible chunks of memory

These pauses can be considerable fractions of a second

Leading to glitches in your video; or your car crashing into a wall

A GC pause will inevitably happen at the most inconvenient point in time

Normally when an object needs to be allocated and memory is full

Memory Management

GC

Stop the World is increasingly bad when you have multiple parallel threads: you have to stop them all

Memory Management

GC

Stop the World is increasingly bad when you have multiple parallel threads: you have to stop them all

There are versions of GC (ephemeral GC, generational GC, concurrent GC, etc.) that try to minimise this disruptive behaviour, but it is very hard to get a guaranteed good behaviour (no inconvenient or dangerous glitches)

Memory Management

GC

Stop the World is increasingly bad when you have multiple parallel threads: you have to stop them all

There are versions of GC (ephemeral GC, generational GC, concurrent GC, etc.) that try to minimise this disruptive behaviour, but it is very hard to get a guaranteed good behaviour (no inconvenient or dangerous glitches)

The more complex *concurrent GC* uses an extra thread to do the GC while the other threads continue to run

Memory Management

GC

Stop the World is increasingly bad when you have multiple parallel threads: you have to stop them all

There are versions of GC (ephemeral GC, generational GC, concurrent GC, etc.) that try to minimise this disruptive behaviour, but it is very hard to get a guaranteed good behaviour (no inconvenient or dangerous glitches)

The more complex *concurrent GC* uses an extra thread to do the GC while the other threads continue to run

But it (a) uses a thread that could be working and (b) messes up memory accesses (caching) for the working threads causing them to slow down

Memory Management

GC

You can find many papers describing GC implementations that strive towards decreasing GC pause times, e.g., for Java and Go

Memory Management

GC

You can find many papers describing GC implementations that strive towards decreasing GC pause times, e.g., for Java and Go

But they all increase other things, e.g., memory usage or allocation time or overall CPU usage

Memory Management

GC

You can find many papers describing GC implementations that strive towards decreasing GC pause times, e.g., for Java and Go

But they all increase other things, e.g., memory usage or allocation time or overall CPU usage

So comparing GC algorithms purely by pause time is an incomplete picture

Memory Management

GC

So real-time systems usually avoid automatic GC, and only employ languages using manual memory management (like C and C++)

Memory Management

GC

So real-time systems usually avoid automatic GC, and only employ languages using manual memory management (like C and C++)

Similarly, close-to-the-machine applications (e.g., operating systems) avoid GC as a garbage collector may well move live values around in memory as part of their tidying up to avoid memory fragmentation

Memory Management

GC

So real-time systems usually avoid automatic GC, and only employ languages using manual memory management (like C and C++)

Similarly, close-to-the-machine applications (e.g., operating systems) avoid GC as a garbage collector may well move live values around in memory as part of their tidying up to avoid memory fragmentation

Low level programs often need to keep a close control on where things are located in memory and this would be a huge complicating factor

Memory Management

GC

So real-time systems usually avoid automatic GC, and only employ languages using manual memory management (like C and C++)

Similarly, close-to-the-machine applications (e.g., operating systems) avoid GC as a garbage collector may well move live values around in memory as part of their tidying up to avoid memory fragmentation

Low level programs often need to keep a close control on where things are located in memory and this would be a huge complicating factor

Plus they don't like the potentially non-deterministic behaviour of GC (when will the GC happen?), and again prefer the deterministic behaviour of manual memory management

Memory Management

GC vs. Manual

Note that manual memory management does have a cost (`malloc` and `free` are not free!) but that cost is spread throughout the running of the program, and not in big pauses when the GC runs

Memory Management

GC vs. Manual

Note that manual memory management does have a cost (`malloc` and `free` are not free!) but that cost is spread throughout the running of the program, and not in big pauses when the GC runs

Furthermore, GC generally has more work to do overall as it needs to search throughout memory to discover inaccessible memory, while `free` doesn't

Memory Management

GC vs. Manual

Note that manual memory management does have a cost (`malloc` and `free` are not free!) but that cost is spread throughout the running of the program, and not in big pauses when the GC runs

Furthermore, GC generally has more work to do overall as it needs to search throughout memory to discover inaccessible memory, while `free` doesn't

GCs ignore information in the code that could help, e.g., when an object is no longer accessible might be possible to be determined from the code. But common language design means you can't *always* determine this, thus the need for GC in such languages

Memory Management

GC vs. Manual

GC code tends to have a larger memory footprint as it might need up to twice the space that data would normally require: the space for the data plus an equal amount of space for the allocation and GC mechanisms

Memory Management

GC vs. Manual

GC code tends to have a larger memory footprint as it might need up to twice the space that data would normally require: the space for the data plus an equal amount of space for the allocation and GC mechanisms

Also it typically fills memory before starting a GC sweep through memory, while manual management (when used correctly) tends to keep memory sprawl in check

Memory Management

GC vs. Manual

GC code tends to have a larger memory footprint as it might need up to twice the space that data would normally require: the space for the data plus an equal amount of space for the allocation and GC mechanisms

Also it typically fills memory before starting a GC sweep through memory, while manual management (when used correctly) tends to keep memory sprawl in check

Sprawl can be a problem in memory-limited embedded systems

Memory Management

GC vs. Manual

But on the plus side, a GC deallocates many objects in one sweep which might be less overhead than deallocating many objects individually in manual system (amortisation of deallocation costs, particularly in parallel systems where memory allocation might need to acquire locks)

Memory Management

GC vs. Manual

But on the plus side, a GC deallocates many objects in one sweep which might be less overhead than deallocating many objects individually in manual system (amortisation of deallocation costs, particularly in parallel systems where memory allocation might need to acquire locks)

And GC applications can need less *development* time, as the programmer has less “to get right”

Memory Management

GC vs. Manual

But on the plus side, a GC deallocates many objects in one sweep which might be less overhead than deallocating many objects individually in manual system (amortisation of deallocation costs, particularly in parallel systems where memory allocation might need to acquire locks)

And GC applications can need less *development* time, as the programmer has less “to get right”

In a world of big memory and “fast enough” processors development time might be the most important factor!

Memory Management

GC vs. Manual

These days business are often more worried about code development time (get the product out the door) than code execution speed (it goes fast enough)

Memory Management

GC vs. Manual

These days business are often more worried about code development time (get the product out the door) than code execution speed (it goes fast enough)

Or code correctness (get the customer to find the bugs)

Memory Management

GC vs. Manual

These days business are often more worried about code development time (get the product out the door) than code execution speed (it goes fast enough)

Or code correctness (get the customer to find the bugs)

But there are a *lot* of cases (embedded systems) where the processor is not particularly fast, or where there is not a huge amount of memory

Memory Management

GC vs. Manual

These days business are often more worried about code development time (get the product out the door) than code execution speed (it goes fast enough)

Or code correctness (get the customer to find the bugs)

But there are a *lot* of cases (embedded systems) where the processor is not particularly fast, or where there is not a huge amount of memory

Or where customer-discovered bugs are not acceptable (safety-critical systems)

Memory Management

Exercise Find out the overhead of `malloc` and `free` in C, or the equivalent in your favourite language

Exercise Read about the concurrent GC in Go

Note for Gamers: at 60fps you only have 16ms per frame. So even 1ms for a GC is troublesome

Exercise Using Java for high-frequency trading (in banks) has problems with GC latency. Read about this

Exercise There has been talk of building specific hardware support for GC: a “GC coprocessor”. Read about this.

Other Memory Management

So the problem is we need something to keep track of values (objects/data) so that the bytes they occupy can be reclaimed and reused when that value is no longer needed by the program

Other Memory Management

So the problem is we need something to keep track of values (objects/data) so that the bytes they occupy can be reclaimed and reused when that value is no longer needed by the program

With manual memory management it is the programmer's responsibility

Other Memory Management

So the problem is we need something to keep track of values (objects/data) so that the bytes they occupy can be reclaimed and reused when that value is no longer needed by the program

With manual memory management it is the programmer's responsibility

With GC it is the GC subsystem's responsibility

Other Memory Management

So the problem is we need something to keep track of values (objects/data) so that the bytes they occupy can be reclaimed and reused when that value is no longer needed by the program

With manual memory management it is the programmer's responsibility

With GC it is the GC subsystem's responsibility

While these two cases cover a lot of existing languages, there are other approaches the try to tackle the above two's shortcomings

Other Memory Management

So the problem is we need something to keep track of values (objects/data) so that the bytes they occupy can be reclaimed and reused when that value is no longer needed by the program

With manual memory management it is the programmer's responsibility

With GC it is the GC subsystem's responsibility

While these two cases cover a lot of existing languages, there are other approaches the try to tackle the above two's shortcomings

Namely relying on the programmer is not a good idea and GCs have unpredictable or undesirable behaviour

Other Memory Management

Runtime Tracking of Allocations

Other Memory Management

Runtime Tracking of Allocations

Swift and Python (amongst others) use runtime *reference counting* to keep track of values

Other Memory Management

Runtime Tracking of Allocations

Swift and Python (amongst others) use runtime *reference counting* to keep track of values

Each value has an internal counter that is incremented every time a new reference is made to that value; and decremented when a reference is dropped

Other Memory Management

Runtime Tracking of Allocations

Swift and Python (amongst others) use runtime *reference counting* to keep track of values

Each value has an internal counter that is incremented every time a new reference is made to that value; and decremented when a reference is dropped

When the counter reaches zero, the value's memory is deallocated

Other Memory Management

Runtime Tracking

So in

```
y = x
```

the count of the value that `y` used to refer to is decremented while the count of the value that `x` refers to is incremented

Other Memory Management

Runtime Tracking

So in

```
y = x
```

the count of the value that `y` used to refer to is decremented while the count of the value that `x` refers to is incremented

If there are no other references to the old value of `y`, its count will now be zero and so can be deallocated: its memory is freed by the runtime

Other Memory Management

Runtime Tracking

So in

```
y = x
```

the count of the value that `y` used to refer to is decremented while the count of the value that `x` refers to is incremented

If there are no other references to the old value of `y`, its count will now be zero and so can be deallocated: its memory is freed by the runtime

If there *are* other references, its count will be non-zero, so the value is not deallocated

Other Memory Management

Runtime Tracking

Advantages: no GC, no `free`, as the runtime does this automatically

Other Memory Management

Runtime Tracking

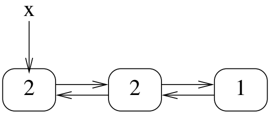
Advantages: no GC, no `free`, as the runtime does this automatically

Disadvantages: a runtime overhead on every assignment (and most programs have lots of assignments!); reference loops (a circle of values, each containing a reference to the next) can hold on to inaccessible memory

Other Memory Management

Runtime Tracking

For example, a doubly-linked list:

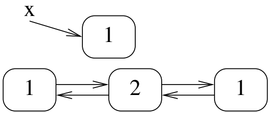


Initial refcounts

Other Memory Management

Runtime Tracking

For example, a doubly-linked list:



Inaccessible after `x` is reassigned

Other Memory Management

Runtime Tracking

In certain cases, a good compiler might be able to eliminate some of these overheads if it can determine statically (i.e., by analysis of the code) it is not needed

Other Memory Management

Runtime Tracking

In certain cases, a good compiler might be able to eliminate some of these overheads if it can determine statically (i.e., by analysis of the code) it is not needed

But this is very hard in general, often even impossible, so not often well supported

Other Memory Management

Runtime Tracking

Exercise Look up `sys.getrefcount()` in Python

Exercise Python also has a GC. Why?

Exercise Swift doesn't have a GC, but uses *weak references* to fix the doubly-linked list problem. Read about this

Advanced Exercise The new M1 chip from Apple has special support in its instruction set to support Swift's reference counting. Read about this

Other Memory Management

Compiletime Tracking of Allocations

Other Memory Management

Compiletime Tracking of Allocations

Rust uses compiler analysis of the source code to keep track of values

Other Memory Management

Compiletime Tracking of Allocations

Rust uses compiler analysis of the source code to keep track of values

The compiler can determine exactly where in the code a value has no more references to it and so can compile exactly the deallocations needed

Other Memory Management

Compiletime Tracking of Allocations

Rust uses compiler analysis of the source code to keep track of values

The compiler can determine exactly where in the code a value has no more references to it and so can compile exactly the deallocations needed

Various language restrictions on the use of values mean this is possible to do in Rust, while being impossible in other languages

Other Memory Management

Compiletime Tracking

For example, many languages can make references to values with no restriction and that makes this kind of compile time analysis impossible for those languages (so, e.g., Swift has to do this tracking at runtime by the reference count)

Other Memory Management

Compiletime Tracking

For example, many languages can make references to values with no restriction and that makes this kind of compile time analysis impossible for those languages (so, e.g., Swift has to do this tracking at runtime by the reference count)

```
if random() > 0.5 {
    y = x;
}
else {
    y = w;
}
```

Other Memory Management

Compiletime Tracking

For example, many languages can make references to values with no restriction and that makes this kind of compile time analysis impossible for those languages (so, e.g., Swift has to do this tracking at runtime by the reference count)

```
if random() > 0.5 {
    y = x;
}
else {
    y = w;
}
```

How many references to the value in `x` are there now?

Other Memory Management

Compiletime Tracking

Rust has a strict notion of which variable *owns* a value that makes this tracking possible by the compiler

Other Memory Management

Compiletime Tracking

Rust has a strict notion of which variable *owns* a value that makes this tracking possible by the compiler

This (very roughly) is ensuring that there is only ever *one* reference to a value

Other Memory Management

Compiletime Tracking

Rust has a strict notion of which variable *owns* a value that makes this tracking possible by the compiler

This (very roughly) is ensuring that there is only ever *one* reference to a value

Exercise The above is a huge simplification. Read about what Rust actually does

Other Memory Management

Compiletime Tracking

Advantages: no GC; no `free`; no runtime overhead for tracking; makes many kinds of memory errors impossible; also makes certain kinds of parallelism errors impossible

Other Memory Management

Compiletime Tracking

Advantages: no GC; no `free`; no runtime overhead for tracking; makes many kinds of memory errors impossible; also makes certain kinds of parallelism errors impossible

Disadvantages: the language has the concepts of *lifetimes* and *ownership* for values that programmers coming from other languages find hard to grasp and which (correctly) makes certain kinds of things you would do in those other languages impossible in Rust

Other Memory Management

Compiletime Tracking

Exercise For advanced C++ programmers: compare the idea of C++ `move` semantics

Exercise For advanced Swift programmers: compare the idea of exclusivity enforcement on variables

Advanced Exercise Read about *linear* and *affine types*

Memory Management

On the use of memory-safe (non-MMM) languages in Android (Rust, Java, Kotlin):

Memory Management

On the use of memory-safe (non-MMM) languages in Android (Rust, Java, Kotlin):

“With less memory-unsafe code (C, C++) entering Android, memory safety flaws went from 76% of Android vulnerabilities in 2019 to 35% in 2022

Memory Management

On the use of memory-safe (non-MMM) languages in Android (Rust, Java, Kotlin):

“With less memory-unsafe code (C, C++) entering Android, memory safety flaws went from 76% of Android vulnerabilities in 2019 to 35% in 2022

2022 was the first year where memory safety vulnerabilities did not form a majority of Android’s vulnerabilities

Memory Management

On the use of memory-safe (non-MMM) languages in Android (Rust, Java, Kotlin):

“With less memory-unsafe code (C, C++) entering Android, memory safety flaws went from 76% of Android vulnerabilities in 2019 to 35% in 2022

2022 was the first year where memory safety vulnerabilities did not form a majority of Android’s vulnerabilities

And because memory safety flaws accounted for most of the critical issues, the vulnerabilities that have surfaced have proven to be less severe”

Jeffrey Vander Stoep, Google Security Blog, December 2022

Memory Management

Before we conclude memory management: note that “garbage collection” is another phrase used in many ways. Some people regard malloc/free as “manual GC”

Memory Management

Before we conclude memory management: note that “garbage collection” is another phrase used in many ways. Some people regard malloc/free as “manual GC”

Similarly reference counting is GC for some people

Memory Management

Before we conclude memory management: note that “garbage collection” is another phrase used in many ways. Some people regard malloc/free as “manual GC”

Similarly reference counting is GC for some people

We shall restrict ourselves to the most common usage of GC that means “subsystem that scans and collects inaccessible memory”

Memory Management

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

Memory Management

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

[C# is] sort of Java with reliability, productivity and security deleted
James Gosling

Types

We now look at how types are treated in various very different ways

Types

We now look at how types are treated in various very different ways

Types are very important to modern programming languages: some people spend their lives purely thinking about types

Types

We now look at how types are treated in various very different ways

Types are very important to modern programming languages: some people spend their lives purely thinking about types

Types are seen as an essential aid to the programmer, to help them write larger, correct programs

Types

We now look at how types are treated in various very different ways

Types are very important to modern programming languages: some people spend their lives purely thinking about types

Types are seen as an essential aid to the programmer, to help them write larger, correct programs

We can classify according to how types are treated

Types

We now look at how types are treated in various very different ways

Types are very important to modern programming languages: some people spend their lives purely thinking about types

Types are seen as an essential aid to the programmer, to help them write larger, correct programs

We can classify according to how types are treated

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

Types vs. Classes

Types and Classes have an interesting relationship

Types vs. Classes

Types and Classes have an interesting relationship

They are sometimes the same, but sometimes different

Types vs. Classes

Types and Classes have an interesting relationship

They are sometimes the same, but sometimes different

For example, Common Lisp has both and they are different things, though connected

Types vs. Classes

Types and Classes have an interesting relationship

They are sometimes the same, but sometimes different

For example, Common Lisp has both and they are different things, though connected

Java has both `int` and `Integer`. Primitive types in Java are not classes (often regarded as a flaw in the design of Java!)

Types vs. Classes

Types and Classes have an interesting relationship

They are sometimes the same, but sometimes different

For example, Common Lisp has both and they are different things, though connected

Java has both `int` and `Integer`. Primitive types in Java are not classes (often regarded as a flaw in the design of Java!)

And, of course, C and Fortran and Pascal, etc., have types, but no classes

Types vs. Classes

We shall be talking about types as a separate concept from classes

Types vs. Classes

We shall be talking about types as a separate concept from classes

So the following also 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 and functions

Types

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

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

Types

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

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

Static typing is quite common in modern languages, and sometimes optional (Maple, Common Lisp)

Types

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

Types

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

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

Types

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

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

Types

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

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

Types

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

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

Often scripting and prototyping languages are dynamically typed

Types

Take care over this: in a dynamically typed language the **variables** do not have types associated

Types

Take care over this: in a dynamically typed language the **variables** do not have types associated

Values have types, encoded into themselves in various ways (e.g., in the header of the value structure)

Types

Take care over this: in a dynamically typed language the **variables** do not have types associated

Values have types, encoded into themselves in various ways (e.g., in the header of the value structure)

A given variable may hold values of different types at different times

Types

Take care over this: in a dynamically typed language the **variables** do not have types associated

Values have types, encoded into themselves in various ways (e.g., in the header of the value structure)

A given variable may hold values of different types at different times

```
x = 1;
x = "hello";
```

is valid in such a language, though arguably poor code as the programmer is clearly confused on the role of `x` in this program

Types

Exercise And the classifications are not exclusive: read about *gradual typing* that mixes static and dynamic

Exercise Find out how much overhead Java, Python, etc., have on each (non-primitive) value to encode its type (and other things)

Types

Strong typing: (a bit of a fuzzy concept) a value has a definite type and no implicit conversions between types

Types

Strong typing: (a bit of a fuzzy concept) a value has a definite type and no implicit conversions between types

- expressions checked for type correctness at compile or runtime

Types

Strong typing: (a bit of a fuzzy concept) a value has a definite type and no implicit conversions between types

- expressions checked for type correctness at compile or runtime
- little (preferably: no) automatic type conversions, e.g., between integer and floating point

Types

Strong typing: (a bit of a fuzzy concept) a value has a definite type and no implicit conversions between types

- expressions checked for type correctness at compile or runtime
- little (preferably: no) automatic type conversions, e.g., between integer and floating point

Python does no static checking, but does enforce moderately strong typing at runtime

Types

Strong typing: (a bit of a fuzzy concept) a value has a definite type and no implicit conversions between types

- expressions checked for type correctness at compile or runtime
- little (preferably: no) automatic type conversions, e.g., between integer and floating point

Python does no static checking, but does enforce moderately strong typing at runtime

Perl is more forgiving on mixing types in an expression, e.g., `'2' + 3`

Types

Strong typing: (a bit of a fuzzy concept) a value has a definite type and no implicit conversions between types

- expressions checked for type correctness at compile or runtime
- little (preferably: no) automatic type conversions, e.g., between integer and floating point

Python does no static checking, but does enforce moderately strong typing at runtime

Perl is more forgiving on mixing types in an expression, e.g., `'2' + 3`

"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

For example, C is fairly weakly typed and it is common (and bad) to write:

```
double x = 1;     // poor code!
```

Here, the `int` value is automatically coerced to a `double` value before being bound to `x`

Types

For example, C is fairly weakly typed and it is common (and bad) to write:

```
double x = 1;     // poor code!
```

Here, the `int` value is automatically coerced to a `double` value before being bound to `x`

Exercise Compare `sqrt(3/2)` with `sqrt(3.0/2.0)` in C. Then do the same in Python2, Python3, Haskell and other languages

Types

Remember: even though C `float` and `int` might both use 32 bits to represent a number, they are very different things: the integer 1 might be represented by

00000000 00000000 00000000 00000001

while the float 1.0 might be represented in IEEE by

00111111 10000000 00000000 00000000

Types

Remember: even though C `float` and `int` might both use 32 bits to represent a number, they are very different things: the integer 1 might be represented by

00000000 00000000 00000000 00000001

while the float 1.0 might be represented in IEEE by

00111111 10000000 00000000 00000000

In programming languages, floating point and integers are different types that behave very differently and you shouldn't casually mix them

Types

The confusion possibly arises because in Mathematics we have the natural inclusion of integers within the reals, where the first are regarded as a subset of the second

Types

The confusion possibly arises because in Mathematics we have the natural inclusion of integers within the reals, where the first are regarded as a subset of the second

Mathematicians do this as it is convenient, but in CS we have to be more careful

Types

Exercise Consider the difference between:

- Converting an `int` to a `float` where we want to preserve the “meaning”, e.g., `1` becomes `1.0`
- Converting an `int` to a `float` where we want to preserve the bits, e.g.,
  `00000000 00000000 00000000 00000001` as an `int`
  becomes
  `00000000 00000000 00000000 00000001` as a `float`