“Time is Nature’s way of preventing everything from happening at once.”

There are at least seven manifestations:

1. Sequencing
   
   ```
   foo(); bar();
   ```

2. Selection
   
   ```
   if (a) foo();
   ```

3. Iteration
   
   ```
   while (i<10) foo(i);
   ```

4. Procedures
   
   ```
   foo(10,20);
   ```

5. Recursion
   
   ```
   foo(int i) {
   foo(i-1);
   }
   ```

6. Concurrency
   
   ```
   foo() || bar()
   ```

7. Nondeterminism
   
   ```
   do a -> foo(); [ ] b -> bar();
   ```
Ordering Within Expressions

What code does a compiler generate for

\[ a = b + c + d; \]

Most likely something like

\[ \text{tmp} = b + c; \]
\[ a = \text{tmp} + d; \]

(Assumes left-to-right evaluation of expressions.)
Order of Evaluation

Why would you care?

Expression evaluation can have side-effects.

Floating-point numbers don’t behave like numbers.

Mayan numbers
Side-effects

```c
int x = 0;

int foo() { x += 5; return x; }

int a = foo() + x + foo();

What's the final value of a?
```
Side-effects

```c
int x = 0;

int foo() { x += 5; return x; }

int a = foo() + x + foo();

GCC sets a=25.
Sun’s C compiler gave a=20.
C says expression evaluation order is implementation-dependent.
```
Side-effects

Java prescribes left-to-right evaluation.

class Foo {
    static int x;
    static int foo() { x += 5; return x; }
    public static void main(String args[]) {
        int a = foo() + x + foo();
        System.out.println(a);
    }
}

Always prints 20.
Number Behavior

Basic number axioms:

\[ a + x = a \text{ if and only if } x = 0 \quad \text{Additive identity} \]
\[ (a + b) + c = a + (b + c) \quad \text{Associative} \]
\[ a(b + c) = ab + ac \quad \text{Distributive} \]
Misbehaving Floating-Point Numbers

$1e20 + 1e-20 = 1e20$

$1e-20 ≪ 1e20$

$(1 + 9e-7) + 9e-7 \neq 1 + (9e-7 + 9e-7)$

$9e-7 ≪ 1$, so it is discarded, however, $1.8e-6$ is large enough

$1.00001(1.000001 − 1) \neq 1.00001 \cdot 1.000001 − 1.00001 \cdot 1$

$1.00001 \cdot 1.000001 = 1.000011\textbf{00001}$ requires too much intermediate precision.
What’s Going On?

Floating-point numbers are represented using an exponent/significand format:

\[
\begin{align*}
1 & \quad 10000001 & \quad 0110000000000000000000000000000 \\
& \quad \text{8-bit exponent} & \quad \text{23-bit significand} \\
= & \quad -1.011_2 \times 2^{129-127} = -1.375 \times 4 = -5.5.
\end{align*}
\]

What to remember:

\[ \underline{1363.456846353963456293} \]

represented rounded
What’s Going On?

Results are often rounded:

\[ 1.0000100000 \times 1.0000010000 \]

\[ \Rightarrow 1.00001100001 \]

rounded

When \( b \approx -c \), \( b + c \) is small, so \( ab + ac \neq a(b + c) \) because precision is lost when \( ab \) is calculated.

Moral: Be aware of floating-point number properties when writing complex expressions.
Short-Circuit Evaluation

When you write

```c
if (disaster_could_happen)
    avoid_it();
else
    cause_a_disaster();
```

cause_a_disaster() is not called when disaster_could_happen is true.

The if statement evaluates its bodies lazily: only when necessary.
Short-Circuit Evaluation

The section operator \(?\) : does this, too.

\[
cost =
\begin{align*}
disaster\_possible & \quad ? \quad avoid\_it() & \quad : \quad cause\_it(); \\
\end{align*}
\]

\textit{cause\_it} is not called if \textit{disaster\_possible} is true.
Logical Operators

In Java and C, Boolean logical operators “short-circuit” to provide this facility:

```java
if (disaster_possible || case_it()) { ... }
```

`cause_it()` only called if `disaster_possible` is false.

The `&&` operator does the same thing.

Useful when a later test could cause an error:

```java
int a[10];

if (i => 0 && i < 10 && a[i] == 0) { ... }
```
Short-Circuit Operators

Not all languages provide short-circuit operators. Pascal does not.

C and Java have two sets:

Logical operators || && short-circuit.

Boolean (bitwise) operators | & do not.
Unstructured Control-Flow

Assembly languages usually provide three types of instructions:

Pass control to next instruction:

    add, sub, mov, cmp

Pass control to another instruction:

    jmp rts

Conditionally pass control next or elsewhere:

    beq bne blt
Unstructured Control-Flow

So-called because it’s easy to create spaghetti:

```
beq A
B:
  jmp C
A:
  beq D
C:
  beq B
D:
  bne B
```
Structured Control-Flow

The “object-oriented languages” of the 1960s and 70s.
Structured programming replaces the evil goto with structured (nested) constructs such as

if-then-else
for
while
do .. while
break
continue
return
Gotos vs. Structured Programming

A typical use of a goto is building a loop. In BASIC:

10 print I
20 I = I + 1
30 IF I < 10 GOTO 10

A cleaner version in C using structured control flow:

do {
    printf("%d\n", i);
    i = i + 1;
} while ( i < 10 )

An even better version

for (i = 0 ; i < 10 ; i++) printf("%d\n", i);
Gotos vs. Structured Programming

Break and continue leave loops prematurely:

```c
for ( i = 0 ; i < 10 ; i++ ) {
    if ( i == 5 ) continue;
    if ( i == 8 ) break;
    printf("%d\n", i);
}

Again: if (!(i < 10)) goto Break;
    if ( i == 5 ) goto Continue;
    if ( i == 8 ) goto Break;
    printf("%d\n", i);
Continue: i++; goto Again;
Break:
```
Escaping from Loops

Java allows you to escape from labeled loops:

```java
a: for (int i = 0 ; i < 10 ; i++)
    for ( int j = 0 ; j < 10 ; j++) {
        System.out.println(i + "","" + j);
        if (i == 2 && j == 8) continue a;
        if (i == 8 && j == 4) break a;
    }
```

Gotos vs. Structured Programming

Pascal has no “return” statement for escaping from functions/procedures early, so goto was necessary:

```pascal
procedure consume_line(var line : string);
begin
  if line[i] = '%' then goto 100;
  (* .... *)
100:
end
```

In C and many others, return does this for you:

```c
void consume_line(char* line) {
  if (line[0] == '%' ) return;
}
```
Multi-way Branching

switch (s) {
    case 1: one(); break;
    case 2: two(); break;
    case 3: three(); break;
    case 4: four(); break;
}

Switch sends control to one of the case labels. Break terminates the statement.
Implementing multi-way branches

switch (s) {
    case 1: one(); break;
    case 2: two(); break;
    case 3: three(); break;
    case 4: four(); break;
}

Obvious way:

if (s == 1) { one(); }
else if (s == 2) { two(); }
else if (s == 3) { three(); }
else if (s == 4) { four(); }

Reasonable, but we can sometimes do better.
Implementing multi-way branches

If the cases are dense, a branch table is more efficient:

```c
switch (s) {
    case 1: one(); break;
    case 2: two(); break;
    case 3: three(); break;
    case 4: four(); break;
}

labels l[] = { L1, L2, L3, L4 }; /* Array of labels */
if (s>=1 && s<=4) goto l[s-1]; /* not legal C */
L1: one(); goto Break;
L2: two(); goto Break;
L3: three(); goto Break;
L4: four(); goto Break;
Break:
```
Recursion and Iteration

Consider computing

\[ \sum_{i=0}^{10} f(i) \]

In C, the most obvious evaluation is iterative:

```c
double total = 0;
for ( i = 0 ; i <= 10 ; i++ )
    total += f(i);
```
Recursion and Iteration

\[ \sum_{i=0}^{10} f(i) \]

But this can also be defined recursively:

def double sum(int i)
{
    double fi = f(i);
    if (i <= 10) return fi + sum(i+1);
    else return fi;
}

sum(0);
Recursion and Iteration

Grammars make a similar choice:

Iteration:

clist : item ( ""," item )* ;

Recursion:

clist : item tail ;

tail : ""," item tail
       | /* nothing */
       ;
Tail-Recursion and Iteration

```c
int gcd(int a, int b) {
    if (a == b) return a;
    else if (a > b) return gcd(a-b, b);
    else return gcd(a, b-a);
}
```

Notice: no computation follows any recursive calls.

Stack is not necessary: all variables “dead” after the call.

Local variable space can be reused. Trivial since the collection of variables is the same.
Tail-Recursion and Iteration

```c
int gcd(int a, int b) {
    if ( a==b ) return a;
    else if ( a > b ) return gcd(a-b,b);
    else return gcd(a,b-a);
}
```

Can be rewritten into:

```c
int gcd(int a, int b) {
    start:
    if ( a==b ) return a;
    else if ( a > b ) a = a-b; goto start;
    else b = b-a; goto start;
}
```
Tail-Recursion and Iteration

Good compilers, especially those for functional languages, identify and optimize tail recursive functions. Less common for imperative languages. But gcc -O was able to rewrite the gcd example.
Applicative- and Normal-Order Evaluation

```c
int p(int i) { printf("%d ", i); return i; }

void q(int a, int b, int c)
{
    int total = a;
    printf("%d ", b);
    total += c;
}

What is printed by

q( p(1), 2, p(3) );
```
Applicative- and Normal-Order Evaluation

```c
int p(int i) { printf("%d ", i); return i; }
void q(int a, int b, int c)
{
    int total = a;
    printf("%d ", b);
    total += c;
}
q( p(1), 2, p(3) );
```

Applicative: arguments evaluated before function is called.
Result: 1 3 2

Normal: arguments evaluated when used.
Result: 1 2 3
Applicative- vs. and Normal-Order

Most languages use applicative order.

Macro-like languages often use normal order.

#define p(x) (printf("%d ",x), x)
#define q(a,b,c) total = (a), \
printf("%d ", (b)), \
total += (c)

q( p(1), 2, p(3) );

Prints 1 2 3.

Some functional languages also use normal order evaluation to avoid doing work. “Lazy Evaluation”
Argument Order Evaluation

C does not define argument evaluation order:

```c
int p(int i) { printf("%d ", i); return i; }
int q(int a, int b, int c) {}

q( p(1), p(2), p(3) );
```

Might print 1 2 3, 3 2 1, or something else.

This is an example of *nondeterminism*. 
Nondeterminism

Nondeterminism is not the same as random:
Compiler usually chooses an order when generating code.
Optimization, exact expressions, or run-time values may affect behavior.
Bottom line: don’t know what code will do, but often know set of possibilities.

```c
int p(int i) { printf("%d ", i); return i; } 
int q(int a, int b, int c) {} 
q( p(1), p(2), p(3) );
```

Will *not* print 5 6 7. It will print one of
1 2 3, 1 3 2, 2 1 3, 2 3 1, 3 1 2, 3 2 1
Nondeterminism

Nondeterminism lurks in most languages in one form or another.

Especially prevalent in concurrent languages.

Sometimes it’s convenient, though:

```
if a >= b -> max := a
[] b >= a -> max := b
fi
```

Nondeterministic (irrelevant) choice when a=b.

Often want to avoid it, however.
Implementing Inheritance

Simple: Add new fields to end of the object
Fields in base class always at same offset in derived class
Consequence: Derived classes can never remove fields

C++
```cpp
class Shape {
    double x, y;
};

class Box : Shape {
    double h, w;
};
```

Equivalent C
```c
struct Shape {
    double x, y;
};

struct Box {
    double x, y;
    double h, w;
};
```
Virtual Functions

class Shape {
    virtual void draw(); // Invoked by object’s class
    // not its compile-time type.
};
class Line : public Shape {
    void draw();
};
class Arc : public Shape {
    void draw();
};

Shape *s[10];
s[0] = new Line;
s[1] = new Arc;
s[0]->draw(); // Invoke Line::draw()
s[1]->draw(); // Invoke Arc::draw()
Virtual Functions

The Trick: Add a "virtual table" pointer to each object.

```cpp
struct A {
    int x;
    virtual void Foo();
    virtual void Bar();
};

struct B : A {
    int y;
    virtual void Foo();
    virtual void Baz();
};

A a1, a2; B b1;
```
Virtual Functions

```c
struct A {
    int x;
    virtual void Foo();
    virtual void Bar()
        { do_something(); }
};

struct B : A {
    int y;
    virtual void Foo();
    virtual void Baz();
};

A *a = new B;
a->Bar();
```
Virtual Functions

```cpp
struct A {
    int x;
    virtual void Foo();
    virtual void Bar();
};

struct B : A {
    int y;
    virtual void Foo()
    { somethingelse(); }  
    virtual void Baz();
};

A *a = new B;
a->Foo();
```
Multiple Inheritance

Rocket Science, and nearly as dangerous

Inherit from two or more classes

class Window { ... };

class Border { ... };

class BWindow : public Window,
           public Border {
           ... 
           };

Multiple Inheritance Ambiguities

class Window {
    void draw();
};

class Border {
    void draw(); // OK
};

class BWindow : public Window,
                public Border {
};

BWindow bw;
bw.draw(); // Compile-time error: ambiguous
Resolving Ambiguities Explicitly

class Window { void draw(); };

class Border { void draw(); };

class BWindow : public Window,
    public Border {
    void draw() { Window::draw(); }
};

BWindow bw;
bw.draw();  // OK
Duplicate Base Classes

A class may be inherited more than once

class Drawable { ... };
class Window : public Drawable { ... };
class Border : public Drawable { ... };
class BWindow : public Window, public Border { ... };

BWindow gets two copies of the Drawable base class.
Virtual Base Classes

Virtual base classes are inherited at most once

```cpp
class Drawable { ... };
class Window : public virtual Drawable { 
  ... 
};
class Border : public virtual Drawable { 
  ... 
};
class BWindow : public Window, public Border { 
  ... 
};
```

BWindow gets one copy of the Drawable base class
Implementing Multiple Inheritance

A virtual function expects a pointer to its object

```cpp
struct A { int x; virtual void f(); };
struct B { int y; virtual void f(); };
struct C : A, B { int z; void f(); };

B *b = new C;    // Calls C::f()
b->f();           // “this” expected by C::f()
```

“obj” is, by definition, a pointer to a B, not a C. Pointer must be adjusted depending on the actual type of the object. At least two ways to do this.
Implementation using Offsets

```cpp
struct A { int x; virtual void f(); }
struct B { int y; virtual void f();
    virtual void g(); }
struct C : A, B { int z; void f(); }
B *b = new C;
b->f(); // Call C::f()
```

adjust from offset
Implementation using Thunks

```cpp
struct A { int x; virtual void f(); }
struct B { int y; virtual void f();
    virtual void g(); }
struct C : A, B { int z; void f(); }
B *b = new C;
b->f(); // Call C::f()

void C::f_in_B(void *this)
{
    this = this - 2;
goto C::f;
}
```
## Offsets vs. Thunks

<table>
<thead>
<tr>
<th>Offsets</th>
<th>Thunks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offsets to virtual tables</td>
<td>Helper functions</td>
</tr>
<tr>
<td>Can be implemented in C</td>
<td>Needs “extra” semantics</td>
</tr>
<tr>
<td>All virtual functions cost more</td>
<td>Only multiply-inherited functions cost</td>
</tr>
<tr>
<td>Tricky</td>
<td>Very Tricky</td>
</tr>
</tbody>
</table>
Exceptions

A high-level replacement for C’s setjmp/longjmp.

```c
struct Except { }; 

void baz() { throw Except; } 
void bar() { baz(); } 

void foo() { 
  try {
    bar(); 
  } catch (Except e) {
    printf("oops");
  }
}
```
One Way to Implement Exceptions

```
try {
    push(Ex, Handler);
    throw Ex;
    throw(Ex);
    pop();
    goto Exit;
}
```

```
catch (Ex e) {
    Handler:
    foo();
    foo();
}
```

```
Exit:
```

**push()** adds a handler to a stack

**pop()** removes a handler

**throw()** finds first matching handler

Problem: imposes overhead even with no exceptions
Implementing Exceptions Cleverly

Real question is the nearest handler for a given PC.

```
1 void foo() {
2     try {
3         bar();
4     } catch (Ex1 e) { H1: a(); }
5 }
6
7 void bar() {
8     try {
9         throw Ex1();
10     } catch (Ex2 e) { H2: b(); }
11 }
12}
```

<table>
<thead>
<tr>
<th>Lines</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1–2</td>
<td>Reraise</td>
</tr>
<tr>
<td>3–5</td>
<td>H1</td>
</tr>
<tr>
<td>6–9</td>
<td>Reraise</td>
</tr>
<tr>
<td>10–12</td>
<td>H2</td>
</tr>
<tr>
<td>13–14</td>
<td>Reraise</td>
</tr>
</tbody>
</table>

1. look in table
2. H2 doesn't handle Ex1, reraise
3. look in table