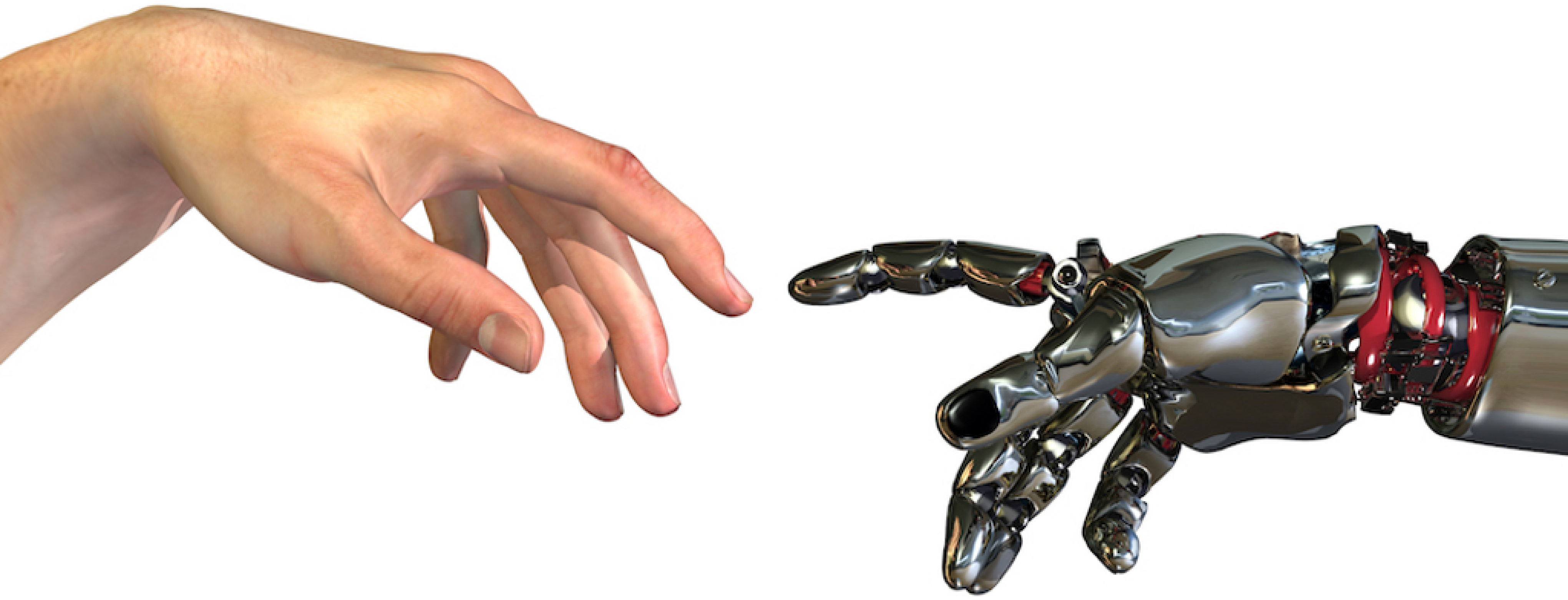
Statements



The four laws of programs

These are like Isaac Asimov's 4 laws of robotics:

- `0:` programs must work properly
- `1:` programs must be readable, provided this does not conflict with the previous law
- `2:` programs must be compact, provided this does not conflict with the previous laws
- `3:` programs must be efficient, provided this does not conflict with the previous laws

What statements do

A statement in a function tells the computer to do something

Example: grade

grade.c

```
/* Find the grade for a mark. */
#include <stdio.h>

int grade(int mark) {
    int grade;
    if (mark >= 70) grade = 1;
    else if (mark >= 50) grade = 2;
    else if (mark >= 40) grade = 3;
    else grade = 4;
    return grade;
}

int main() {
    printf("My grade is %d\n", grade(66));
    return 0;
}
```

Compiling and running

To compile and run in a terminal window:

```
$ clang -std=c11 -Wall grade.c -o grade
$ ./grade
2
$
```

Scope

Each name has a limited *scope*

grade.c

```
int grade(int mark) {
    int grade;
    ...
}
```

The scope of the local integer variable `grade` is the function body, between the curly brackets

It temporarily hides the global `grade` function

Declarations and definitions

Things can be declared first, then defined later

```
int grade;                                          grade.c
...
grade = 1;
```

The statement `int grade;` declares the variable without defining it

The assignment `grade = 1;` defines it later

Function declarations

You can also declare functions before defining them

```
int grade(int mark);

int main() {
    ...
}

int grade(int mark) {
    ...
}
```

This is to tell the compiler about functions defined (a) later or (b) in other modules (often via header files)

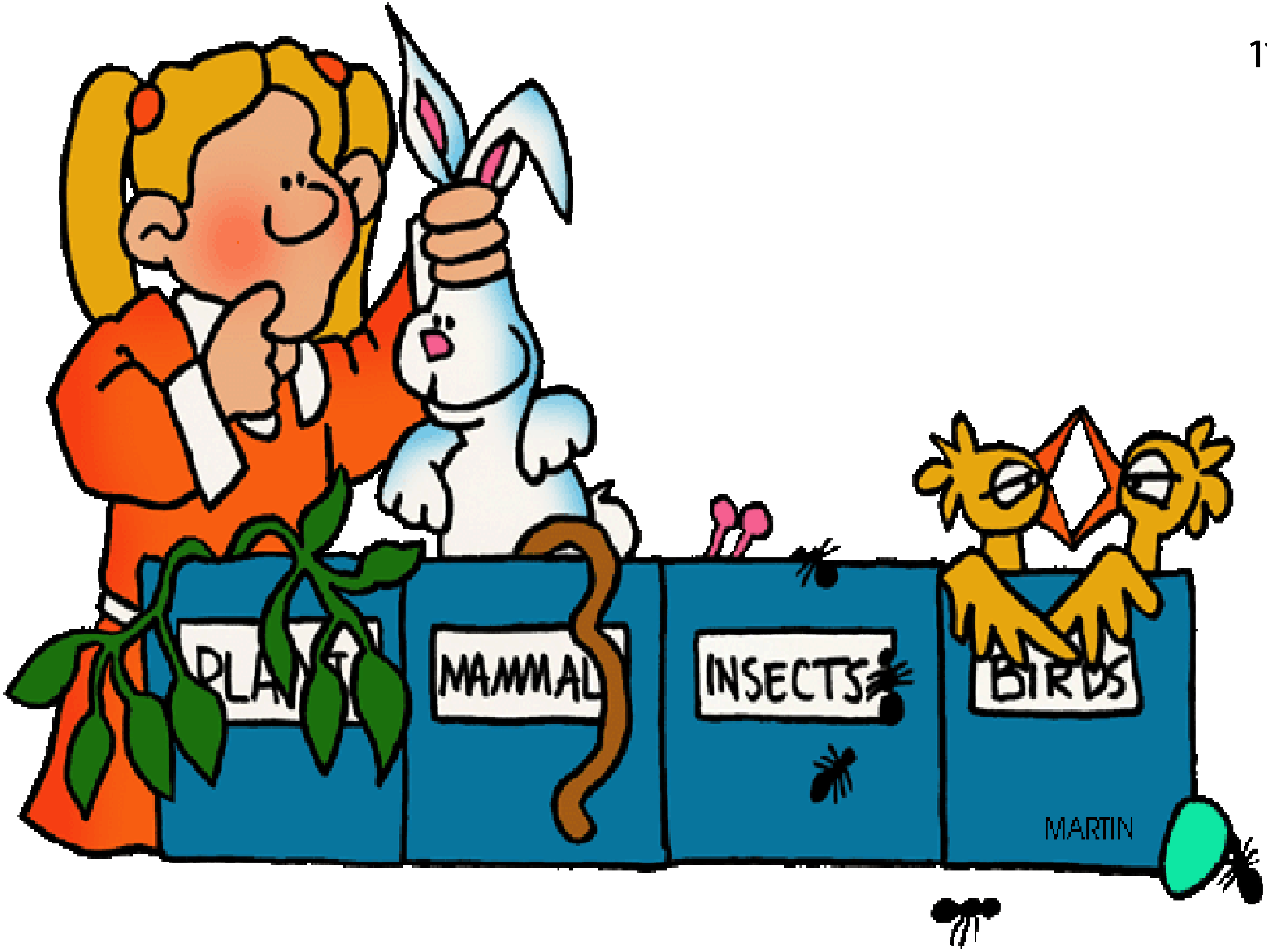
These declarations are called ***signatures*** or ***prototypes***

Decisions

Simple decisions can be made using `if` and `else`

grade.c

```c
if (mark >= 70) grade = 1;
else if (mark >= 50) grade = 2;
else if (mark >= 40) grade = 3;
else grade = 4;
```


PLANTS
MAMMALS
INSECTS
BIRDS
MARTIN

Example: Sort

max.c

```c
/* Sort two numbers. */
#include <stdio.h>

// Sort two numbers into ascending order and print
int main() {
    int a = 42, b = 21;
    if (a > b) {
        int save = a;
        a = b;
        b = save;
    }
    printf("%d %d\n", a, b);
    return 0;
}
```

With no arrays yet, it is difficult to sort lots of things, or separate the calculation from the input/output

Blocks

An `if` statement can be followed by a block

```
if (a > b) {
    int save = a;
    a = b;
    b = save;
}
```

`else` is optional - the default is 'else do nothing'

A block is a sequence of statements between curly brackets, the same as a function body

The scope of `save` is the block - it doesn't exist outside

printf

Use `%d` in `printf` to print out ints

```
printf("%d %d\n", a, b);
```

max.c

Variables

An `int` variable in C is a 'box' with an integer in it

```
int a = 42;                                       max.c
...
a = b;
```

A variable can be re-used by putting different numbers in the box

Swapping needs a third variable because after `a = b` the old number in `a` has been forgotten

Incrementing

In C, you often see this:

```
n = n + 1;
```

As an equation, it is unsolvable, as an executable definition, it defines n as an infinite loop

But in C, the = operator means "becomes"

So this means "get the number out of the box called n, add one to it, and put the result back in the box n, throwing away the old number"

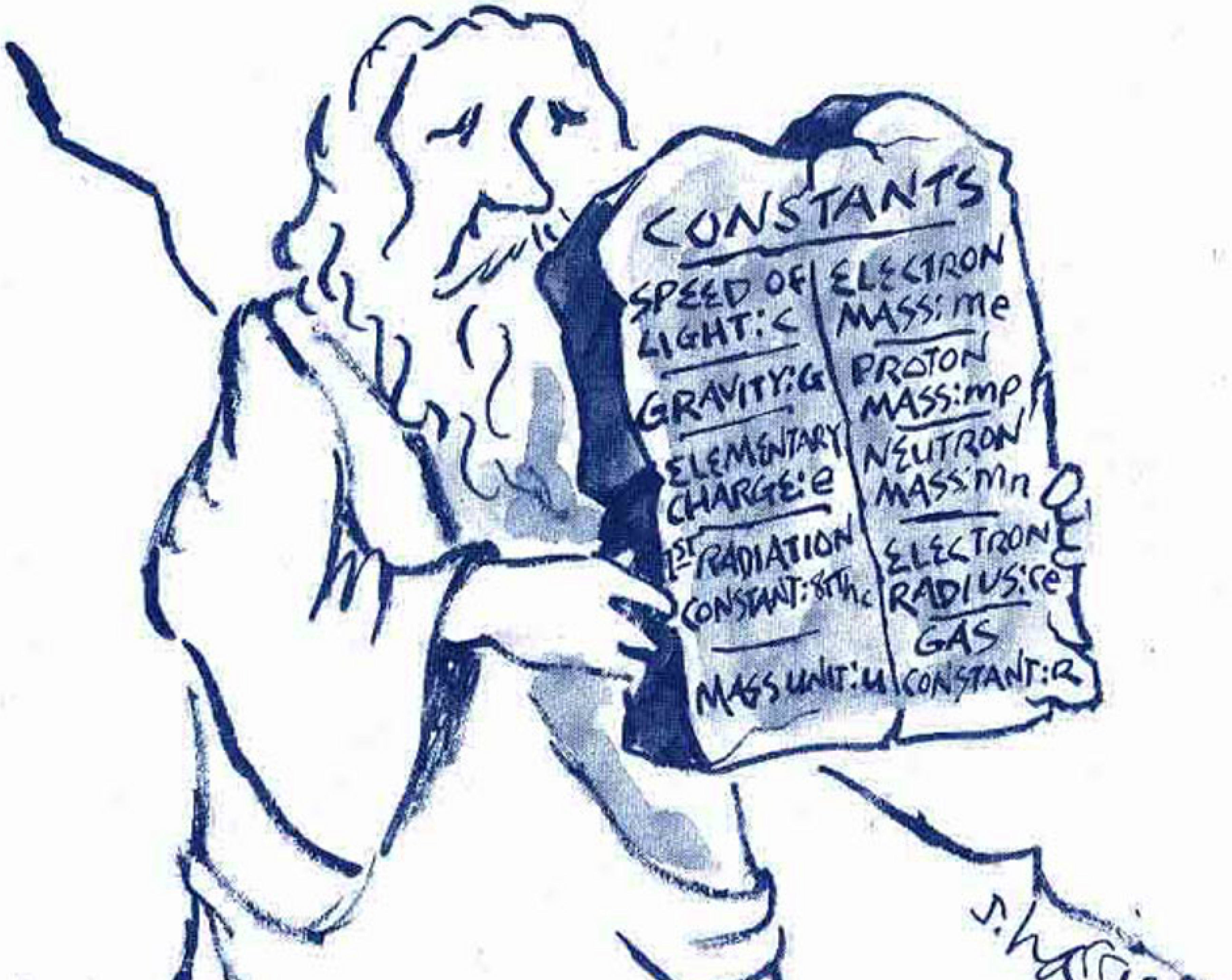
In other words, increase n by one, also written as n++

Tracing

An important skill is to be able to trace execution in your head, or on paper, or with a tool like gdb

```
                     a   b   save
                    42  21   ?
int save = a;
                    42  21   42
a = b;
                    21  21   42
b = save;
                    21  42   42
```

Use gdb only in emergencies, otherwise it soaks up too much time - judicious printfs are usually better


CONSTANTS
SPEED OF LIGHT: c
GRAVITY: G
ELEMENTARY CHARGE: e
1st RADIATION CONSTANT: 8πhc
MASS UNIT: u
ELECTRON MASS: me
PROTON MASS: mp
NEUTRON MASS: mn
ELECTRON RADIUS: re
GAS CONSTANT: R
S. Harris

Constants

If you are writing a chess program, the constant 8 is likely to appear all over the place in your program

```
... for (i=0; i<8; i++) ...
```

Even though you will probably never want to change it, it makes programs more readable to give it a name

```
const int size = 8;
... for (i=0; i<size; i++) ...
```

`const` doesn't mean constant, just "check that the variable is never explicitly updated"

Global constants

It is possible to define a constant outside of any functions, so that it is available everywhere:

```
#include ...
const int size = 8;
int doSomething() { ... size ... }
int main() { ... size ... }
```

This is not as bad as global variables (which we are going to avoid) but it is restrictive (e.g. prevents multiple boards of different sizes) and unnecessary (as we will see when we reach structures)

Const arguments

It is possible to declare arguments as `const`:

```
int grade(const char mark[]) { ... }
```

This only applies to pointer and array arguments where the function *could* make changes to the original

It documents the fact that the function will not in fact change the argument, and the compiler checks

Doing this is optional (especially if it causes restrictions elsewhere)

Const order

Technically, the keyword `const` comes ***after*** the thing you want to be readonly (except that it can come before if it is the first thing)

So some programmers ***always*** put `const` after, e.g.

```
int grade(char const mark[]) { ... }
```

In more complex situations, it matters (e.g. `char const *` means pointer to read-only characters, `char * const` means read-only pointer to characters, `char const * const` means read-only pointer to read-only characters)

Enumerated constants

Another way of defining constants is enumeration:

```
enum { First, Second, Third, Fail };
```

This is (almost) the same as:

```
const int First=0, Second=1, Third=2, Fail=3;
```

These are integer constants, and `enum` is most often used where it doesn't matter what the constants are

It is very common (but not necessary) to use capital first letters, or all-capitals, for the names of constants

Example

A grade program might contain:

```
...
if (mark >= 70) grade = First;
...
if (grade == First) printf("First");
...
```

The constant `First` has been used as a code for a ***concept***, without ever caring what its value is

Values

You can set the values of some or all of the constants:

```
enum { First=1, Second=2, Third=3, Fail=4 };
```

Or, you can write code which takes advantage of the fact that the constants are successive integers

```
enum { Mon, Tue, Wed, Thu, Fri, Sat, Sun };

int next(int day) {
    if (day == Sun) day = Mon;
    else day = day + 1;
    return day;
}
```

Switch

```
switch (day) {
case Mon: ... ; break;
case Tue: ... ; break;
...
case Sat: case Sun: ... break;
default: ... ; break;
}
```

A *switch* is a direct jump, more efficient than sequential tests

Beware: (a) don't forget the `break` ('fall through') (b) no agreement on indenting (c) can make functions big: consider one line per case, maybe a function call

Enum versus const

One difference between `enum` and `const` is that an `enum` is always an `int` (or something compatible like `char`) whereas a `const` can be any type

Another difference is that an `enum` can be used in a switch statement, but a `const int` can't

That's because `const` means readonly, not constant, e.g. a temperature reading from a device might be declared `const`, to prevent the ***program*** from updating it

Isolated constants

Sometimes, `enum` is used to define isolated constants, not sequences:

```
enum { size=8 };
enum { Width=80, Height=24 };
```

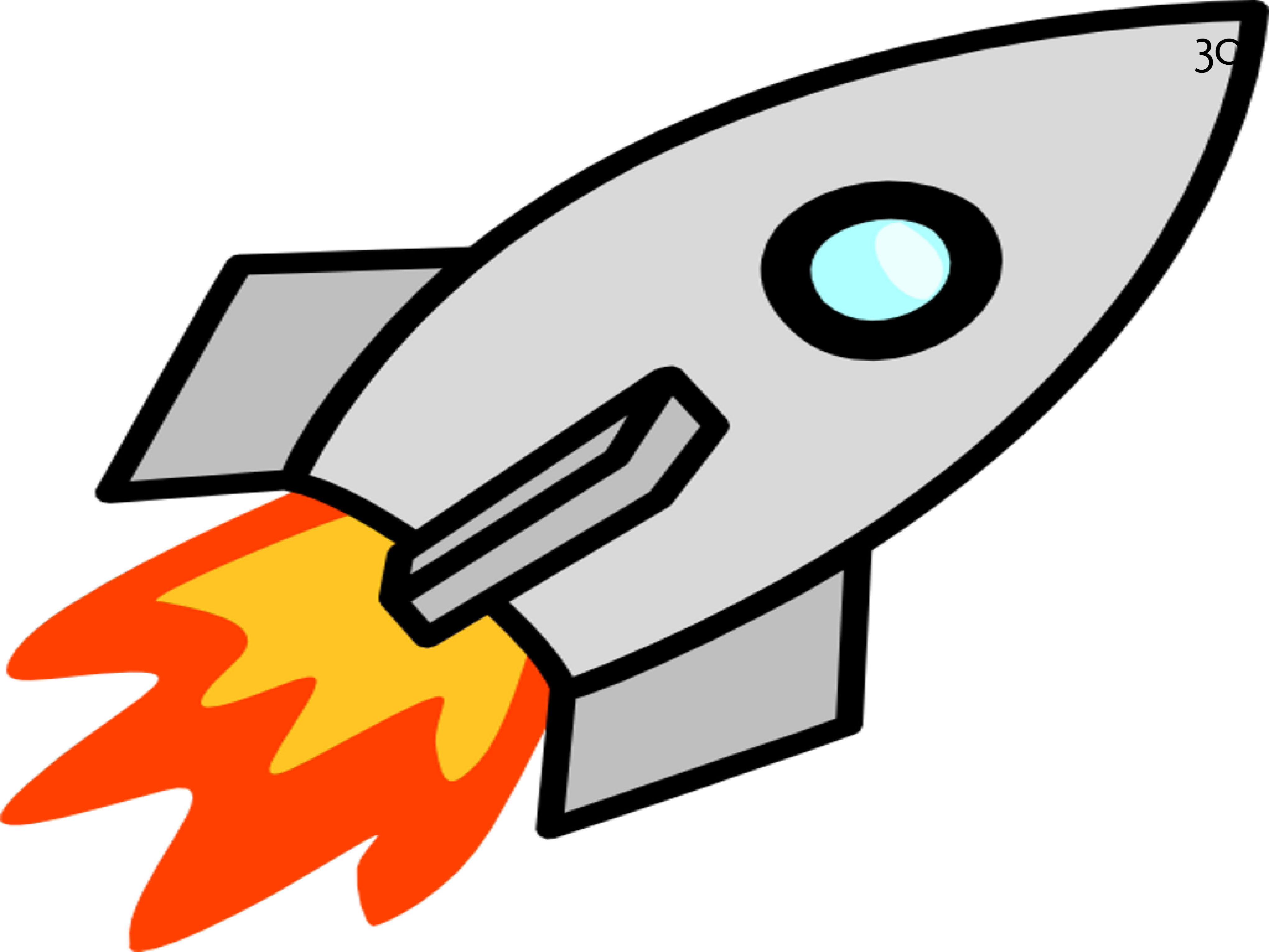
The advantage over using `const` is that the compiler ***knows*** that they are truly constant, and can optimize better or allow them to be used in some special contexts

Enumerated types

You can pretend that an enumeration is a different type:

```
enum Grade { First, Second, Third, Fail };
...
enum Grade grade = First;
```

The type `enum Grade` is just a synonym for `int`, but the code may be more readable, by making intentions clearer



While loops

A while loop allows code to be repeated: it is basically a conditional backward jump in the code

countdown.c

```
/* Print a countdown. */
#define _POSIX_C_SOURCE 200809L
#include <unistd.h>
#include <stdio.h>

int main() {
    int t = 10;
    while (t >= 0) {
        sleep(1);
        printf("%d\n", t);
        t = t - 1;
    }
    return 0;
}
```

Portability

C claims to be portable, not platform independent

To go beyond the minimal standard libraries, you need to find libraries which are available across platforms

For coursework, ***don't*** use platform specific libraries, e.g.

```
#include <windows.h>
```

You ***can*** use the Posix cross-platform libraries, e.g.

```
#define _POSIX_C_SOURCE 200809L
#include <unistd.h>
```

Abbreviations

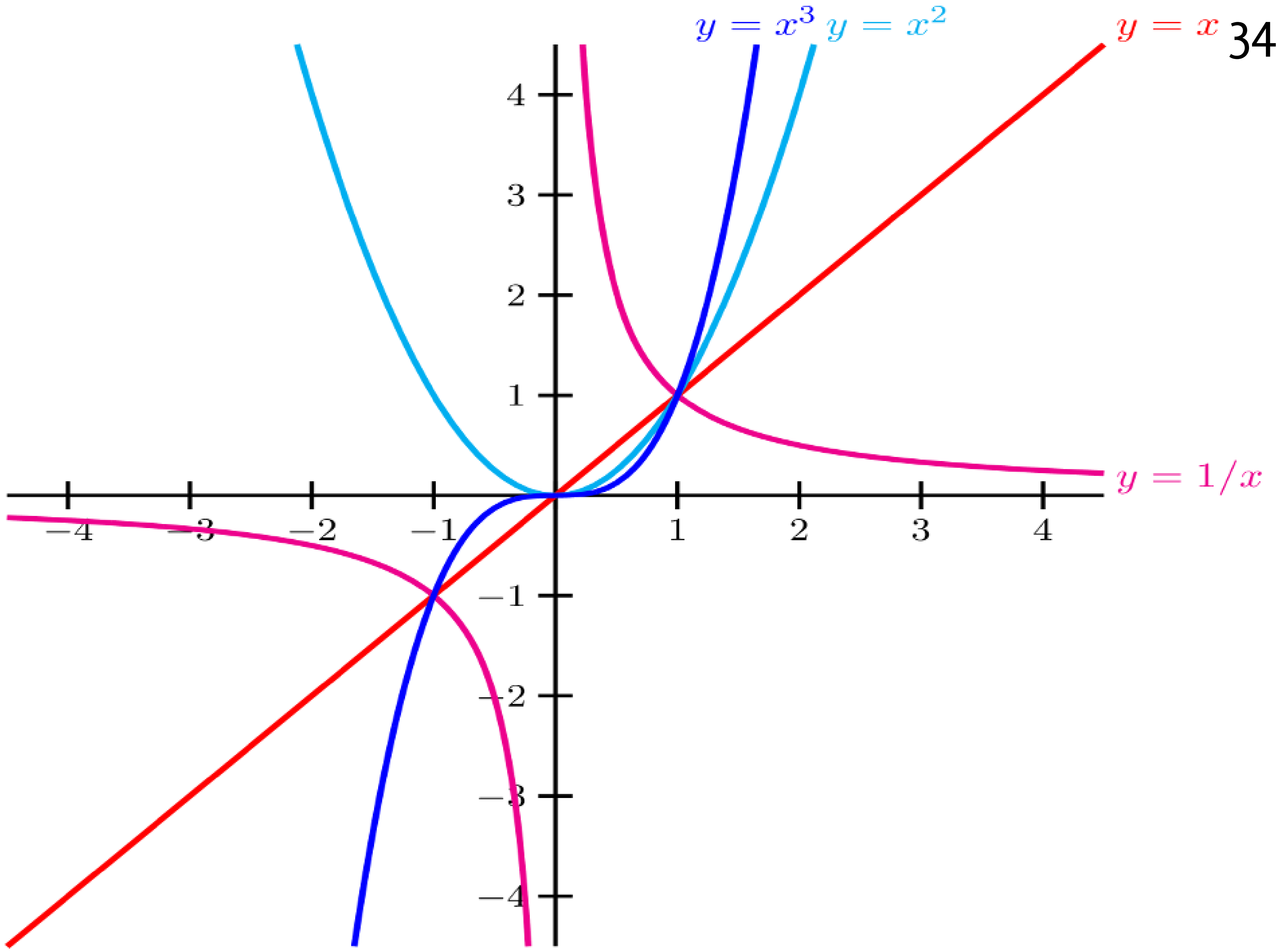
There are increment and decrement abbreviations:

```
n++;          means   n = n + 1;
++n;          means   n = n + 1;
n--;          means   n = n - 1;
--n;          means   n = n - 1;

m = n++;      means   m = n; n++;
m = ++n;      means   n++; m = n;

n = n++;      is a bug
```

Use `++` sparingly, and avoid the bug `n = n++;`


$y = x^3$
$y = x^2$
$y = x$
$y = 1/x$
4
3
2
1
−1
−2
−3
−4
−4
−3
−2
−1
1
2
3
4

Example: square root

Here's a square root function

```c
// Find square roots (like sqrt).                              root.c

double root(double x) {
    double r = x / 2.0;
    double eps = 1e-15;
    while (fabs(r - x/r) > eps) {
        r = (r + x/r) / 2;
    }
    return r;
}
```

double

The `double` type is usually used for floating point

It is stored in 8 bytes, has about 15 decimal significant digits of precision, and has a range of about $\pm 10^{\pm 308}$

When compactness is needed and precision/range requirements are low, e.g. graphics, you can use `float`

A `double` is not exact, even `0.1` can't be stored exactly - try `printf("%.18f\n", 0.1);`

Errors accumulate at an average rate of `sqrt(n)` for `n` operations, when there is no bias (rounding alternates)

Newton's algorithm

The algorithm is essentially Newton's:

```
r = (r + x/r) / 2;                                    root.c
```

If `r` is less than the real root, then `x/r` is greater, and vice versa, so the average is a better approximation, and the gap tells you how close you are

It is easy to understand, convergence is rapid (***faster*** than halving the gap) but libraries use even faster special-purpose techniques

Edge cases

One edge case is when `x = 4.0`

Then the initial guess is exactly correct

A while loop is repeated `0` times if the test starts false:

```
r = 2;
while (r != 2) {
    ...
}
```

This is almost always what you want, and minimizes edge cases

For loops

The countdown loop could be rewritten like this:

```
int t;
for (t = 10; t >= 0; t--) {
    ...
}
```

It is completely equivalent to

```
int t = 10;
while (t >= 0) {
    ...
    t--;
}
```

It gathers the three pieces into one place

Another for loop

Here is another variation, and its equivalent:

```
for (int t = 10; t >= 0; t--) {
    ...
}
```

```
{
  int t = 10;
  while (t >= 0) {
      ...
      t--;
  }
}
```

The outer block limits the scope of `t`

Stylised for loops

Because of their logical complexity, you should only use for loops in a few familiar stylised special cases:

```
for (int i = 0; i < n; i++) ...

for (int i = n-1; i >= 0; i--) ...

for (item *p = list; p != NULL; p = p->next) ...
```

The last example, scanning a linked list, we'll see later

If your situation isn't a simple one like these, it is probably better to use a while loop

Expression statements

Sometimes you want to evaluate an expression (with side effects) and there is no result or you don't need it:

```
n++;
printResults();
```

The first is an increment, where you don't need the value of n for anything

The second is a function call, where nothing is returned, or you don't need the returned value

Do and goto statements

There is a `do..while` loop, with the test at the end, where the loop is always executed at least once

It's visually and semantically confusing - don't use it

There is a `goto` statement (left over from long ago) - don't use it

Clever functions

Kernighan (co-inventor of C) said *“debugging is twice as hard as writing the code in the first place - therefore, if you write the code as cleverly as possible, you are not smart enough to debug it”*

So it is important to avoid writing functions which are 'too clever'

A good strategy is to write a function ***as if*** you are going to have to prove that it is correct

Functions which are ***logically*** simpler are usually also ***intuitively*** simpler, and more likely to be correct

JUMP!
JUMP!
JUMP!
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
5.5
6.0

Controlled jumps

The more your code jumps about, the harder it is to debug

So it pays to make the jumps as controlled as possible

Function calls, if statements, and loops are the most controlled statements

The statements in the next few slides should be used 'sparingly', i.e. not at all, or restricted to a few familiar stylised special cases

Early return

The `return` statement doesn't have to be at the end

```
int abs(int n) {
    if (n >= 0) return n;
    return -n;
}
```

No `else` needed here - if `n>=0`, execution returns from the function before reaching the second line

One stylised use is to dispose of an exceptional case, and avoid an extra indent for the general case

The disadvantage is it may be unclear what property holds on return, or how to add extra end-code

Early loop exit

The `break` statement exits from a loop early:

```
// Search for first prime in a range
while (i < last) {
    if (isprime(i)) break;
    i++;
}
```

Again, it can help to separate special cases from the general case, without mangling the general case

Searching is the most common use

One disadvantage is that the loop can end while the test expression is still true

Searching

Programmers often say they ***must*** use `break` for efficient searching, to avoid unnecessary work

But, arguably, it is logically clearer and cleaner to write

```
// Search for first prime in a range
while (i < last && ! isprime(i)) {
    i++;
}
```

Now the test tells you exactly what must be true each time round the loop, and false when it ends (making it easier to prove correctness)

Early loop restart

The `continue` statement restarts a loop early:

```
// process the even numbers
for (int i = 0; i < n; i++) {
    if (odd(i)) continue;
    ...
}
```

Some people would say that using an `if..else` inside a loop is always better than using `continue`

Commas

What if you want to print things out with commas?

The number of commas is ***one less*** than the number of times round the loop

There are solutions using `break` or `continue`, but my favourite is:

```
// print s, n times with commas
for (int i = 0; i < n; i++) {
    if (i > 0) printf(", ");
    printf("%s", s);
}
printf("\n");
```

Double loop search

What if you have a search involving a double loop?

```
// Search for table entry
for (r = 0; r < m; r++) {
    for (c = 0; c < n; c++) {
        if (table[r][c] ...) ...
    }
}
```

You can't use `break`, because it only breaks out of the inner loop, not the outer one

Some programmers say this is the sort of exception where you must use `goto`, but don't do it!

Extra loop variable

A good readable approach to the problem is to use an extra variable

```
// Search for table entry
bool found = false;
for (r = 0; r < m && !found; r++) {
    for (c = 0; c < n && !found; c++) {
        if (table[r][c] ...) found = true;
    }
}
```

The ternary operator

There is one operator in C which has three arguments instead of the usual one or two

It has the form `t ? x : y` and is equal to `x` or `y` according to the test `t`:

```
int max = m > n ? m : n;
int abs = n >= 0 ? n : -n;
```

It is the same as the Haskell feature `if..then..else..`, but it should be used more 'sparingly' than in Haskell

See list of operators

Feature creep

Examples where the advice is "don't use" or "use sparingly" illustrate that adding features to a language is not necessarily a good idea

But inevitably features do get added (C has resisted more than most)

This tendency is called feature creep, and partly explains why languages go out of fashion