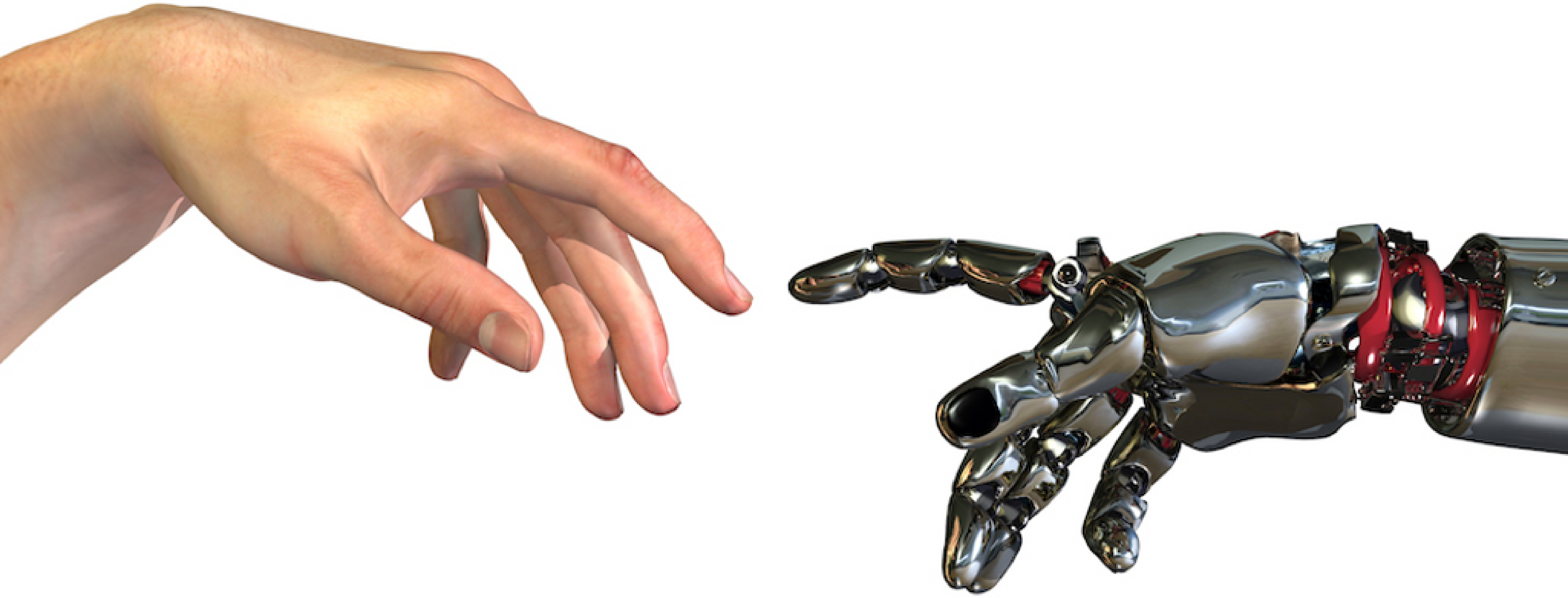
Statements



The four laws of programs

These are like 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;
}
```

Scope

Each name has a limited scope

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

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)

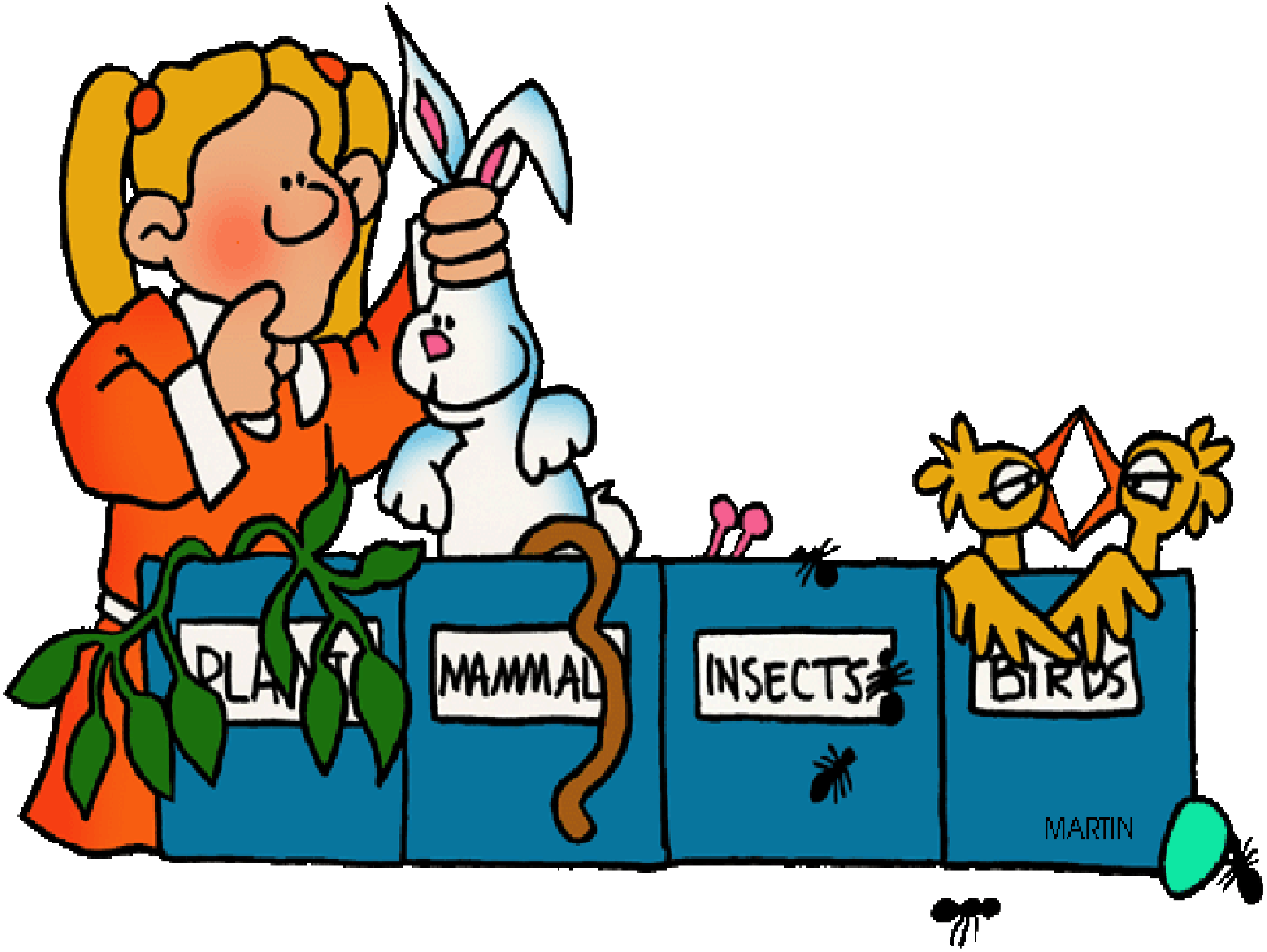
These declarations are called ***signatures*** or ***prototypes***

Decisions

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

grade.c

```
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

```
/* 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 old = a;
        a = b;
        b = old;
    }
    printf("%d %d\n", a, b);
}
```

With no arrays yet, it is difficult to be cleverer

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

max.c

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

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"

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

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

Blocks

An `if` statement can be followed by a block

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

`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 `old` is the block - it doesn't exist outside

Example: Sort words

sort.c

```c
/* Sort two words. */

#include <stdio.h>
#include <string.h>

// Sort two words into ascending order and print
int main() {
    char *a = "cat", *b = "bat";
    if (strcmp(a,b) > 0) {
        char *old = a;
        a = b;
        b = old;
    }
    printf("%s %s\n", a, b);
}
```

String library and printing

There is a `string` library module

```
#include <string.h>                    sort.c
```

Use `%s` in `printf` formats to print out strings

```
printf("%s %s\n", a, b);               sort.c
```

Strings

In C, strings have type `char *`

```
char *a = "cat", *b = "bat";                     sort.c
```

The type is `char *` and the variable names are `a` and `b`

It is conventional to put the `*` next to the variable, because `char* a, b;` means `char* a; char b;`

Typedefs

For greater readability, you could add this at the top:

```
typedef char* string;
```

This defines `string` as a synonym for `char*`, but not many programmers do this, because it is not possible to keep it up indefinitely, and in the end it is better not to 'hide' the pointers in C

Arrays

When you declare `char *a`, the variable `a` is a string, referring to some characters ***stored somewhere else***

When you want to reserve some space for the characters, you can declare `a` as an array instead:

```
char a[100];
```

After this, you can fill in the characters (e.g. using `strcpy` or `strcat` or `sprintf`) and use `a` as a string in exactly the same way as before

String functions

The `*` in `char*` is a warning that strings are not primitive (they are arrays of characters)

So they must be compared using a function `strcmp` and not `==`

C has a 'policy' that all the primitive operations provided take essentially one instruction (no loops)

`strcmp` has a rubbish name because, originally, only the first 6 characters of a name 'counted' (now it's 31)

Pointer comparisons

A string test `s1 == s2` checks whether the strings are identical, i.e. stored at the same place in memory

It is a 'pointer' comparison (we'll do pointers later)

```
char *s1 = "cat";
char *s2 = "cat";
if (s1 == s2) printf("same\n");
else printf("different\n");
```

What would these statements print out?

Pointer comparisons

A string test `s1 == s2` checks whether the strings are identical, i.e. stored at the same place in memory

It is a 'pointer' comparison (we'll do pointers later)

```
char *s1 = "cat";
char *s2 = "cat";
if (s1 == s2) printf("same\n");
else printf("different\n");
```

What would these statements print out?

Answer: it depends on how well the C compiler optimizes the code

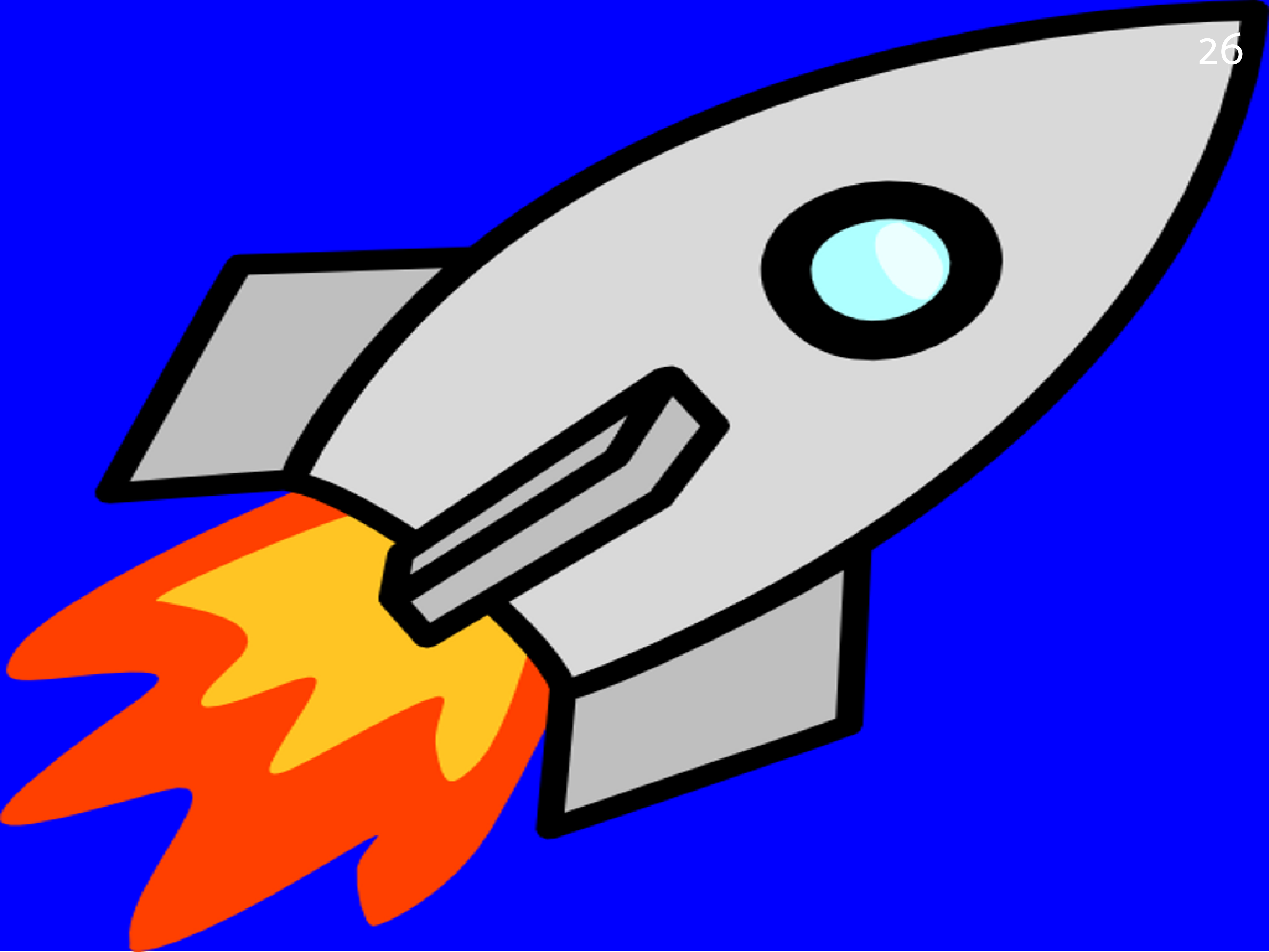
strcmp

The `strcmp` function is one of many string functions in the `string` library module

`strcmp(s1,s2)` returns a negative, zero, or positive result, so `strcmp(s1,s2) < 0` means "`s1 < s2`"

Why doesn't it return -1, 0, 1?

Because it uses subtraction, it would cost an instruction or two to convert, C tries to guarantee to be efficient, and the programmer only ever needs to test the sign anyway (programmer beware)



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. */

#include <stdio.h>
#include <unistd.h>

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

Portability

C claims to be portable, not platform independent

The `unistd` module is Unix-specific

For Windows, replace `unistd.h` by `windows.h` and `sleep(1)` by `Sleep(1000)`

Don't: use non-standard libraries for coursework

Later: find a portable library, make the program platform independent, for each platform bundle the platform-version of the library with the program to create an installation package

Abbreviations

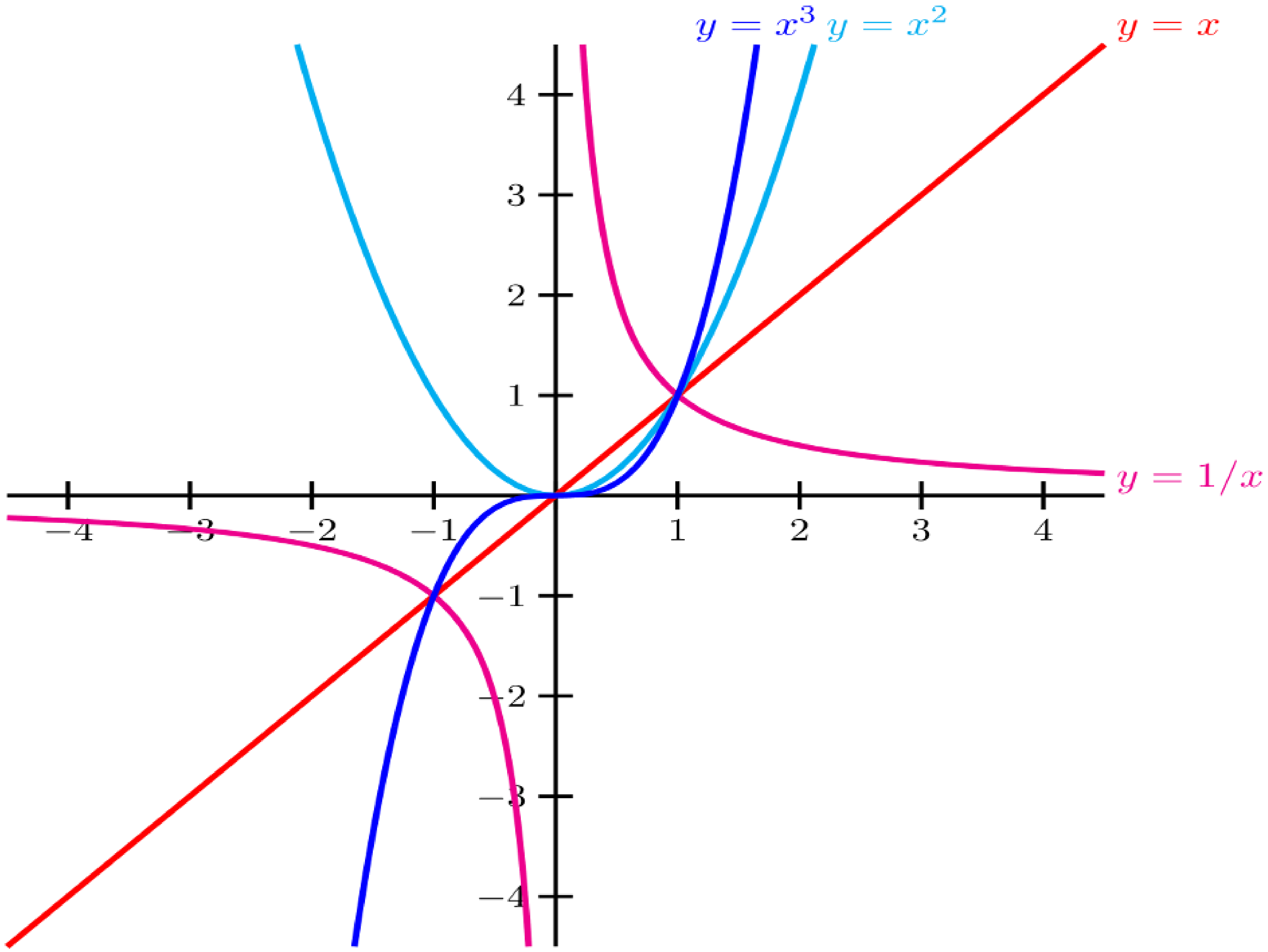
There are increment and decrement abbreviations:

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

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

n = n++;
```

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

```
// 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:

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

It is completely equivalent to

```
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--;
  }
}
```

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

Switch

```
switch (opcode) {
case 0: ... ; break;
case 1: ... ; break;
case 2: case 3: ... break;
...
default: ... ; break;
}
```

It is a jump, so more efficient than sequential tests

Beware: (a) forgetting the `break` ('fall through') (b) no agreement on indenting (c) can make functions big: consider one line per case, maybe a function call

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 the Basic language, maybe) - don't use it

Clever functions

It has been said that '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

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

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 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
(it is the 'loop invariant' needed to prove correctness)

Early loop restart

The `continue` statement restarts a loop early:

```
// print s, n times with commas
for (int i = 0; i < n; i++) {
    printf("%s", s);
    if (i == n-1) continue;
    printf(", ");
}
printf("\n");
```

As it happens, this example can be done with `break` or just `else`

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

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;
    }
}
```

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