Loops

Prof. Darrell Long

CSE 13S

© 2021 Darrell Long

What is a loop?

- A loop allows you to repeat a sequence of code.
- Programs spend the vast majority of their execution time in loops.
- We will focus on structured loops: `while`, `for`, and `do-while`.
- You can also create loops with `goto`: but don't, it's *ugly*.
  - Ghostbusters don't cross streams, and good programmers don't cross loops.

© 2021 Darrell Long

while ()

- It is called a top-test loop
  - The test is evaluated *before* entering the loop.
- Executes the statement as long as the Boolean condition remains *true*.
- Executes the statement *zero* or more times.

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


© 2021 Darrell Long

Equivalent `goto` Code

- You can implement it with the `goto` statement.
- Just because you *can* does not mean that you *should*.


```
i = 1;
loop: if (!(i <= 10)) goto skip;
  printf("%d\n", i);
  i = i + 1;
  goto loop;
skip:
```


© 2021 Darrell Long

for ()

- Also a top-test loop.
- Puts:
  - Initialization,
  - Test, and
  - Increment all together.
- By convention they are related, but nothing in **C** *requires* them to be.

```
for (int i = 1; i <= 10; i = i + 1) {
  printf("%d\n", i);
}
```

© 2021 Darrell Long

Equivalent `while()`

- This is the equivalent `while` statement.
- The `while` statement is *complete*
  - Which means you can implement any loop using it.

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

© 2021 Darrell Long

`do { } while ()`

- This is a bottom-test loop.
- Used when you want to perform the statement at least once.
- Continues to execute the enclosed statement as long as the Boolean condition remains *true*.

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


© 2021 Darrell Long

Equivalent `goto` Code

- This is the equivalent code using `goto`.
- You should *never* write code like this unless your programming language lacks the equivalent statement.

```
i = 1;
loop: printf("%d\n", i);
  i = i + 1;
  if (i <= 10) goto loop;
```


© 2021 Darrell Long

Infinite Loops

- All of these execute *forever*.
- The one you choose is a matter of *style*, not of substance.

- How do you ever escape?
  - Use the `break` statement.

```
while (1) {
  statement; ...
}

do {
  statement; ...
} while (1)

for (;;) {
  statement; ...
}
```


© 2021 Darrell Long

break

- Immediately exits the enclosing loop.
- Allows for middle-exit loops.

- This is still considered structured programming, but it should be used in moderation.

```
while (1) {
   stmt; ...
   if (exit condition) break;
   stmt; ...
}
```


© 2021 Darrell Long

Equivalent `goto` Code

- It goes without saying that you should not write code like this…

```
loop: {
  stmt; ...
  if (exit condition) goto leave;
  stmt; ...
  goto loop;
}
leave:
```


© 2021 Darrell Long

Factorial Example

- $n! = n \times (n-1)!$
- This code will print from 0! to the largest that will fit in an `int`.
- We use the fact that numbers are stored in two's complement (and so turn negative when they exceed the positive numbers).
- We are trying to be perhaps a bit *too* clever.

```c
#include <stdio.h>

int main(void) {
  int f = 1, n = 0;
  while (1) {
    printf("f(%d) = %d\n", n, f);
    n = n + 1;
    f = f * n;
    if (n < 0 || f < 0)
      break;
  }
}
```

← Our attempt at cleverness

© 2021 Darrell Long

When can I use `goto`?

- *One place*: non-local error handling.
- This is when an exceptional condition—an error—that you cannot handle occurs.
- *It is not pretty.*
- More modern languages like C++ provide an exception handling mechanism.

```
{
  { // ... stuff ...
    { // ... stuff ...
      if (terrible thing) goto run_away;
    }
  }
  // ... stuff ...

run_away: // Try to fix the problem?
```


© 2021 Darrell Long

```
bool busy = true;
while (busy = true) {
  // do something
  if (finished) {
    busy = false;
  }
}
```

What's wrong with this code?

This!

- It's an infinite loop!
- How do I know that?
- *Never forget* that
  - = is assignment and
  - == is equality!

© 2021 Darrell Long

continue

```
#include <stdio.h>

int main(void) {
  for (int i = 0; i < 10; i += 1) {
    if (i % 3 == 0)
      continue;
    printf("%d\n", i);
  }
  return 0;
}
```

```
$ cc ex.c -o ex
$ ./ex
1
2
4
5
7
8
```

- You may have times when you want to skip the remainder of a loop.
- For this, there is `continue`.
- Please use it sparingly.

© 2021 Darrell Long

Let's Compute!

```
1.000000
1.500000
1.250000
1.375000
1.437500
1.406250
1.421875
1.414062
```

We'll compute $\sqrt{2}$.

But, can't I just call a library routine?

No!

There is no magic — someone has to write the code.

The subfield of writing numerical programs is called *numerical analysis*.

$\sqrt{2}$ is that same as solving the equation:

$$x^2 - 2 = 0.$$

What else do we know?

We know $0 \leq x \leq 2$, so we'll start looking in the middle.

$\sqrt{2} \approx 1.41421356237309504880168872420969807856967187537694807 3$

© 2021 Darrell Long

Bisection Method

- Start in the middle.
- We have two intervals:

`(low, mid)` and `(mid, high)`

- If we guess to low them we choose the right interval,
- If we guess too high then we choose the left interval.
- We repeat until we're within our error bound.

```
pascal:~ darrell$ cc -o sqrt sqrt.c
pascal:~ darrell$ ./sqrt
sqrt(2) = 1.414207 took 18 steps
```

```c
#include <math.h>
#include <stdio.h>
#define SGN(x) (x < 0 ? -1 : 1)

int main(void) {
  float low = 0.0, high = 2.0, mid, epsilon = 0.00001;
  int steps = 0;
  while (fabs(high - low) > epsilon) {
    mid = (low + high) / 2.0;
    float fm = (mid * mid) - 2.0;
    float fa = (low * low) - 2.0;
    if (SGN(fm) == SGN(fa)) {
      low = mid;
    } else {
      high = mid;
    }
    steps = steps + 1;
  }
  printf("sqrt(2) = %lf took %d steps\n", mid, steps);
  return 0;
}
```

What's this?

This requires thinking though!

© 2021 Darrell Long

What is that thing?!

```
// Give me the next odd number

n = (n % 2 == 1) ? n + 2 : n + 1;
```

```
#define MIN(x,y) (x < y ? x : y)
#define MAX(x,y) (x > y ? x : y)
#define ABX(x)   (x < 0 ? -x : x)
```

This defines a macro, more on that later.

- `?:` is a *ternary* operator.
- It's like an *if-else* statement, but it can be part of an expression.
- If the first part is *true*, it's value is the second part.
- If the first part is *false*, it's value is the third part.
- Use it with care, it can lead to unreadable code if abused.

© 2021 Darrell Long

Vade Mecum

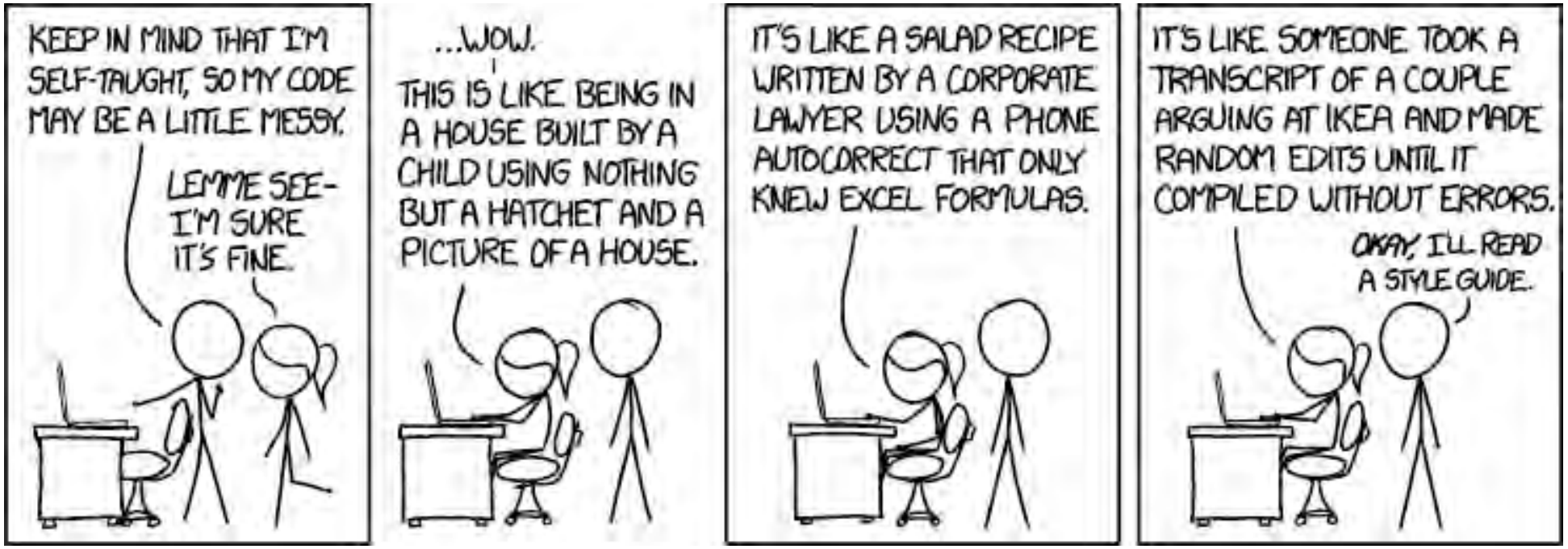
```c
#include <stdbool.h>
#include <stdint.h>

#define LARGEST 22 // log(2^64)/log(10) ~ 19 + sign +

char *itoa(int n) {
  static char b[LARGEST];
  char *t = b + LARGEST;
  bool negative = false;
  if (n < 0) {
    n = -n;
    negative = true;
  }
  *--t = '\0';
  do {
    *--t = n % 10 + '0';
    n /= 10;
  } while (n > 0);
  if (negative) {
    *--t = '-';
  }
  return t;
}
```

© 2021 Darrell Long

Do not be *That Guy*...



© 2021 Darrell Long

Some Advice…

- You should take time to *think* before you code.
- Work out examples on paper or on a whiteboard.
- Pounding on the keyboard is unlikely to produce quality code.

- Quality code requires that you *rewrite* it, just like you rewrite drafts of an essay.

© 2021 Darrell Long