CS 222: Control Flow (Conditionals and Iteration)

Chris Kauffman

Week 2-2

Logistics

Reading

- Ch 5 (iteration)
- Ch 6 (arrays)

Exam 1

- Next Week Thursday
- Zyante Ch 1-6
- This week's Material Included

Stack Demonstrations

- Will post a long example Friday (haven't had time since Tuesday)
- First exercise reviews

HW 1 & 2

- Last day for HW 1 (-2 late tokens)
- HW 2 up now
- Conditionals, loops, arrays, natural log

Goals

- Function wrap-up
- Conditionals
- Iteration
- Basic arrays

Headers and Prototypes

Prototypes often stored in *header* files, `something.h` and used via `#include`.

- HW 2 should do

  `#include "plane_design.h"`

  or something similar in most of your files
- Means "look in current directory for `plane_design.h` header"
- If you do

  `#include <plane_design.h>`

  will mean "look in system locations for `plane_design.h`" and will probably not work
- At the top of your `wireless.c` file do

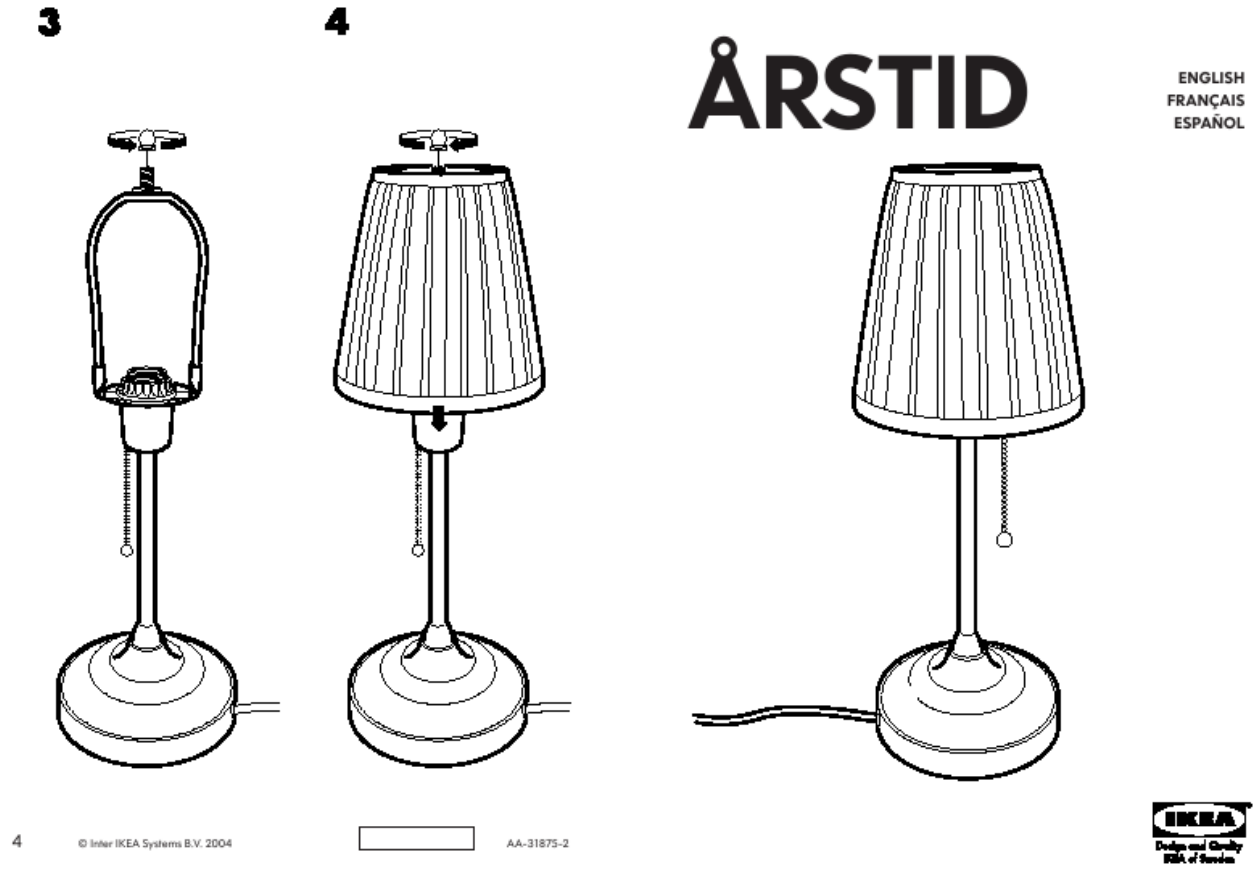
  `#include "wireless.h"`
- etc.

Exercise: Swap?

`try_swap.c`: what is printed? Show why using the call stack.

```
/* Demonstration of call-by value and call stack  */
#include <stdio.h>
void swap_ints(int a, int b){
  int tmp = a;
  a = b;
  b = tmp;
}
int main(){
  int x=20, y=50;
  printf("x=%d  y=%d\n",x,y);
  swap_ints(x,y);
  /* What gets printed here? */
  printf("x=%d  y=%d\n",x,y);
  return 0;
}
```

Making Choices



Straightline code is about as interesting as Ikea instructions: rigid.

Simplest Form of `if`

```
Always do this;
if(condition)
  sometimes do this;
Always do this;

Always;
if(condition){
  sometimes this;
  and this;
  and this;
}
Always;
```

See `if_test.c`

CK's preference - always use

```
if(...){
  ...
  ...
}
```

Do what works for you

- Or what your boss forces you to do

Comparing things

=	Assignment, NOT comparison
==	Equality test
!=	Inequality
< >	Less / Greater
<= >=	Less than equal / Greater than equal

See `comparisons.c`

Consequence and Alternative

Often have 2 cases, C provides nice syntax

```
Always;
if(cond){
  do when true;
}
else{
  do when false;
}
Always;
```

Boolean Combinations

To combine conditions

Test more than one thing at once

`&&`	and
`||`	or
`!`	not

See `booleancomb.c`

Truthy/Falsey

Which things are truthy and falsy in C again?

Combining if/elses

Nesting Arbitrary nesting of conditionals, `nesting.c`

Chaining Mutually exclusive cases, `chaining.c`

Comparison Gotchyas

Two very common errors

```
// Different meaning than intended
if(cond)
  do me;
  do me too;
always;
```

```
// Not accepted by compiler
if( 0 <= i <= 10)
```

Exercises: Conditionals in Functions

Define an absolute value function for single integers

```
int a = abs(7);  // 7
int b = abs(-2); // 2
int c = abs(0);  // 0
```

Define an *absolute minimum* function for three real numbers

```
double x = absmin3( 1.4, 0.5, -2.8); // 0.5
double y = absmin3(-1.4, 0.5, -0.1); // 0.1
double z = absmin3(-1.4, 5.5, -6.1); // 1.4
```

Iteration

Repeat something

- # of Repetitions is conditional
- Zyante Chapter 4

while

A way to repeat

```
always do this once;
while(this is true){
  do this;
  and this;
}
always do this once;
```

while.c

What does it do?
Modify to

- Print up to 20?
- Print to specified limit?
- Print only odds?
- Run forever?
- Ask for number of iterations?

Nesting

Loops can nest, works as expected: `nestwhile.c`
Modify to

- Print in a "matrixy" way
- Print lower triangle
- Print user-specified size

Looped input

Common to get input in a loop until quit command

- See `sodaloop.c`

A little sugar

Sometimes want to *always* do one iteration

```
Do this once;
do {
  do this once, maybe more;
  and do this once, maybe more;
} while(condition);
```

See `guessing_game.c`

The Other Loop

Counting loops extremely common, thus `for` is born

```
do this once;
for(Initialize; Condition; Update){
  do this til condition is false;
  do this til condition is false;
}
do this once;
...
do this once;
Initialize;
while(Condition){
  do this til condition is false;
  do this til condition is false;
  Update;
}
do this once;
```

See `for_v_while.c` and `nestfor.c`

Detour: Compound assignment

- Frequently want increase or decrease the value of a variable
- Shorthand assignment operators for this purpose

```
double d = 10.0;

d = d + 5.0;        // Increase d by 5
d += 5.0;           // Also increase d by 5

d = d - 5.0;        // Decrease d by 5
d -= 5.0;           // Also decrease d by 5

d = d * 2.0;        // Double d
d *= 2.0;           // Also double

d = d / 2.0;        // Halve d
d /= 2.0;           // Also halve d
```

Works for numeric types: `double`, `int`, etc.

Increment and Decrement

- Very frequently need to increase or decrease a variable by 1
- Shorthand *increment* and *decrement*

```
int i = 0;

i = i + 1;      // Increase i by 1
i++;            // Same
i++;            // Again
++i;            // The same (in this case)

i = i - 1;      // Decrease i by 1
i--;            // Same
--i;
```

Where you see it

```
for(i=0; i<10; i++){
  printf("Counting loop %d\n",i);
}
```

Detour: Why i++ and ++i?

Value of assignment is the assigned value

```
int i,j,k;
i = j = k = 0;
```

Post increment

```
int i=0
int j = i++;
// j is now 0, i is 1
```

Pre increment

```
int i=0;
int j = ++i;
// j is now 1, i is 1
```

Syntatic Sugar

Not strictly necessary but there for convenience (and confusion)

Composing elements

- Conditionals in Loops
- Loops in Conditionals
- Function calls in loops and conditionals
- Nested Conditionals
- Nested Loops

What's missing?

Exercise

Classic: define factorial functions

$$factorial(n) = n! = 1 \times 2 \times \cdots (n-1) \times n$$

- Write a `while` loop version
- Write a `for` loop version
- Write a main that tests the function

Examples:

```
int f3 = factorial_for(3);      // 3*2*1 = 6
int f6 = factorial_while(6);    // 6*5*4*3*2*1 = 720
int f4 = factorial_for(4);      // 4*3*2*1 = 24
```

```
lila [w02-2-code]% gcc factorial.c
lila [w02-2-code]% a.out
input n: 12
12! = 479001600 (for)
12! = 479001600 (while)
```

Data Types

Define	Now there's a type, it looks like blah
Declare	Here is a variable, it's type is bleh
Access	Element foo of variable bar has value . . .
Assign	Element foo of variable bar gets value blip

Scalar Types

Only one element/value per variable

Define Done for you for `int`,`double`,`char` etc

Declare ??

Access ??

Assign ??

Aggregate Data: Two Kinds

Arrays collection of the same thing (*homogeneous*)

- Like vectors/matrices
- Indexed by number
- Elements accessed via `array[index]`

`structs` collection of different things (*heterogeneous*)

- A record
- Named elements (field)
- Elements accessed via `mystruct.fieldname`

Arrays

A block of memory

Define	Built in
Declare	`type name[size];`
Access	`x = name[index];`
Alter	`name[index] = x;`

See `arraytypes.c`

Initialize

Initial values are undefined - gabbledegook
Must initialize values, typically

- By hand
- By loop
- Immediate notation: `{el1, el2, el3}`
- By library call (later)

See `array_init.c`

Careful

Uninitialized stack memory could be anything

- See `random_initialize.c`

Exercise: Price is Right

```
int [] guesses = {45, 22, 86, 37, 1, 2, 47};
int find_closest_guess(int actual_price){
  ...
}
```

- Use a set of loops and conditionals to determine the closest value in `guesses` to `actual_price` that does not go over `actual_price`.
- Return the closest value from the function
- If all values in `guesses` are larger than `actual_price`, return -1

```
lila [w02-2-code]% gcc price_right.c
lila [w02-2-code]% a.out
Guesses: 45 22 86 37 12 13 47
Actual 42 closest_guess 37
Actual 46 closest_guess 45
Actual 22 closest_guess 22
Actual 10 closest_guess -1
```

So far

- ☒ Comments
- ☒ Statements/Expressions
- ☒ Variable Types
- ☒ Assignment
- ☒ Basic Input/Output
- ☒ Function Declarations
- ☒ Conditionals (if-else)
- ☒ Iteration (loops)
- ⊟ Aggregate data (arrays, structs, objects, etc)
- ☒ Library System