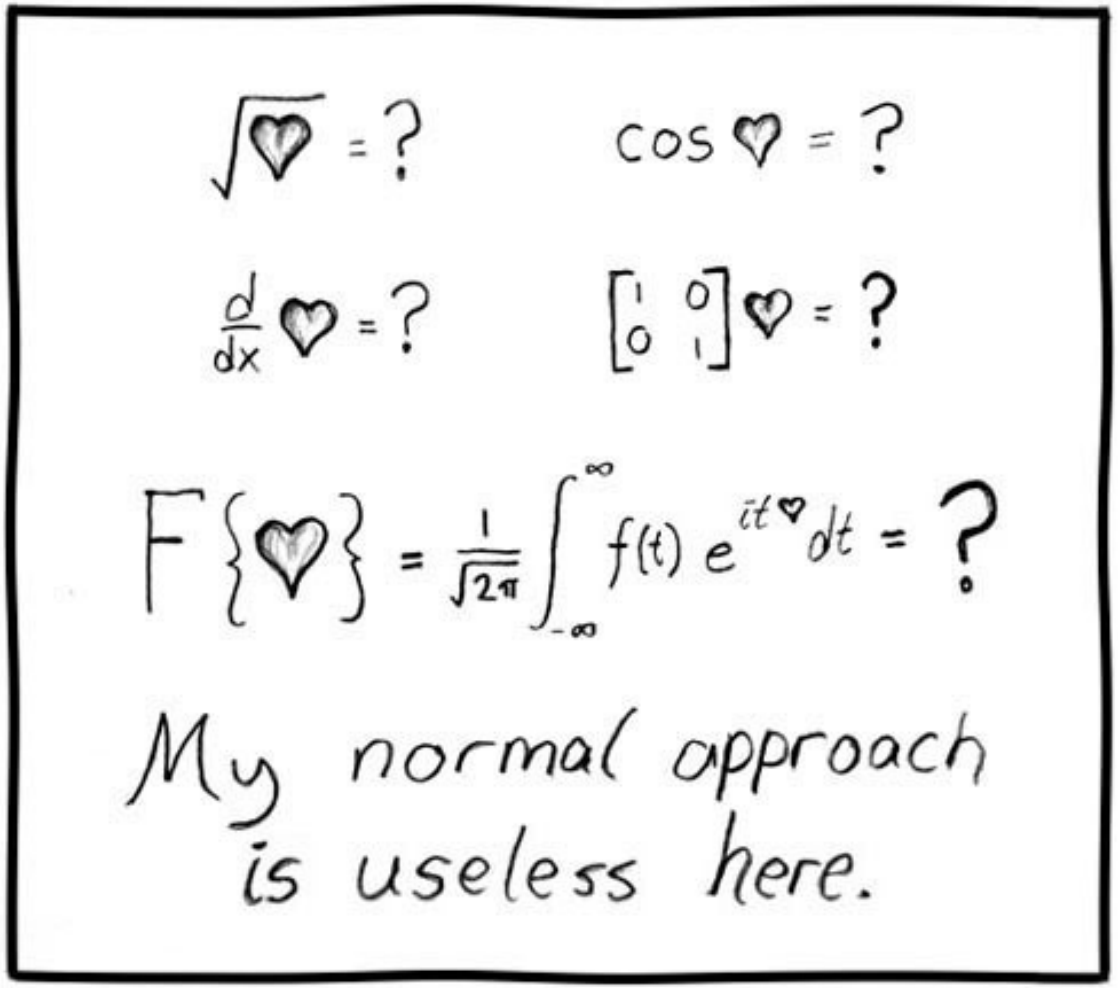
Welcome to Section 1!

This is CS50.

CS 50 Resources

- Office Hrs(http://www.cs50.net/ohs/)
- Bulletin Board
- help@cs50.net
- Walkthroughs every Sunday
- After section Q&A

Functions (?)

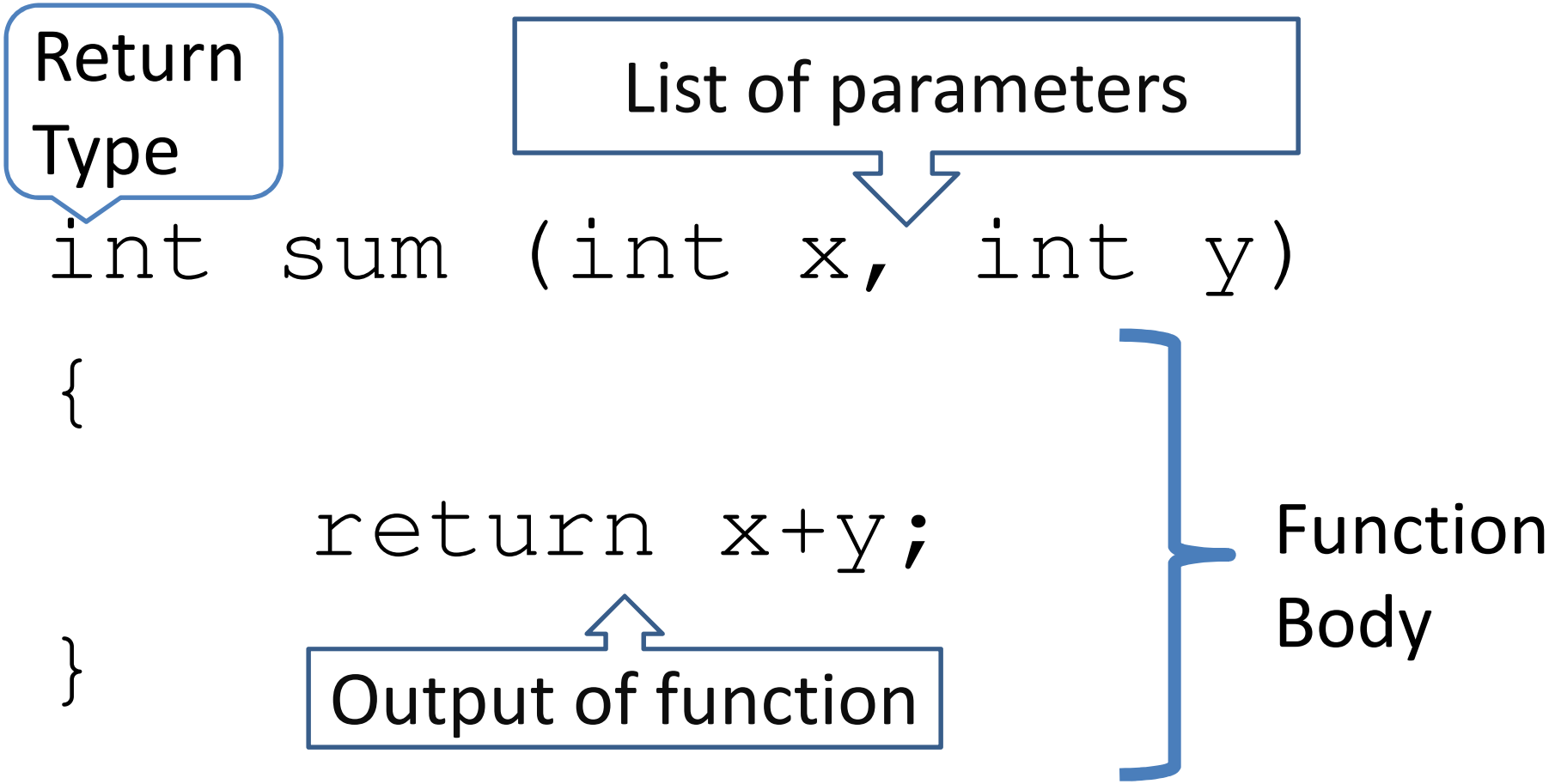


http://xkcd.com/55/

Functions in C

For **main()**, the contents of this function are like a black box.

Anatomy of a function



Anatomy (2)

1. Return type:

   void, int, float, double,
   char, etc.

2. Function name: foo, bar, foo_bar, ~~foo~~ ~~bar~~

3. Parameter or argument list:

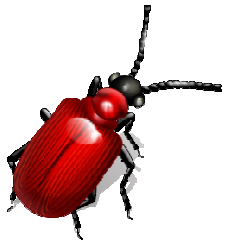
   (), (int x), (char c), (float sum)

4. Function body – *local* variables, loops, statements, side effects, *return statement*

Simplest function

```
void hello_world (void)
{
  printf(“Hallo! \n ”);
}
```

This function causes a **side-effect.**



Side effects

```
int avg (float x, float y)
{
   float sum = x+y;
   printf(“\n %f”, sum);
   return sum / 2;
}
```

Why use functions?

1. **Logical flow** and **organization** of code: helps the reader and helps you debug.

2. **Reusability**: library functions, cs50.h, or your very own set of tools

Why use functions? (2)

3. **Simplification**:

   -break down a large problem into subproblems;

   -isolate that bug!

   -better design

Design Decisions (1)

- Break down problems into smaller puzzle pieces (like you intuitively did in Scratch!)
- Make a function for every logically distinct block of code
- Don’t repeat code; make functions for repeating sections of code

Design Decisions (2)

```
double
interest(double balance, double
rate)
{
    double accrued, updated;
    accrued = balance * rate;
    updated = balance + accrued;

    return updated;
}
```

Design Decisions (3)

```
double
interest (double balance, double
rate)
{
    return balance+(balance*rate);
}
```

What resource is used less here?

Function Declaration

- Remember the black box analogy?

- The compiler needs to know what type of function it should expect (what it returns) and how many parameters it has.

Function Calls

- Call a function using its name and arguments, respecting the type and order of the arguments.
- Any function, not just main(), can call any other function.

Scope of Variables

- *Global Variables* (seen and accessed by any function)

Vs

Local Variables (seen only within the function in which they were created)

Pitfalls

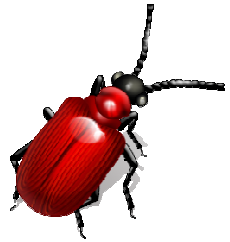
- Overuse of globals (bad design, waste memory)

- Naming identically locals used in separate functions, or for locals and globals

Pitfalls – Local Variables

- Arguments are *passed by value*: each function call will make a *local* copy of its arguments, which disappears after its execution.
- Function parameters are *local variables* to the function.

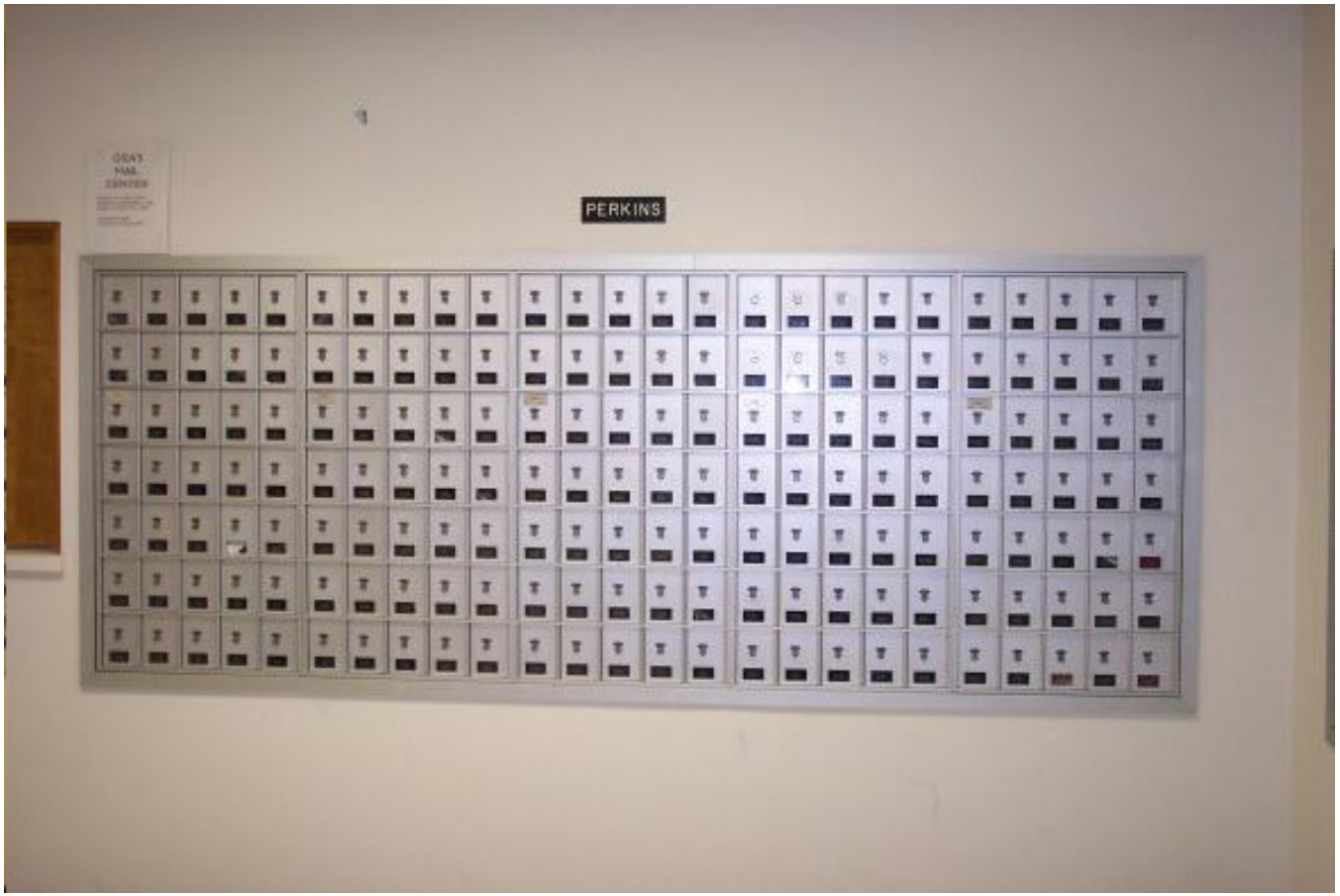
```
int increment(x);

int main ()
{
    int x = 1;
    int y = increment(x);
    printf(" %d %d \n", x, y);
}
int increment(int x)
{
    x++;
    return x;
}
```



Take-away(s)

- The local copy of x in increment() got changed, not the one in main().
- Globals can also be tricky, since you need to track which functions change your globals and how.
- Better solutions than *pass by value*?

Array = block of contiguous space in memory, partitioned in identical smaller boxes:



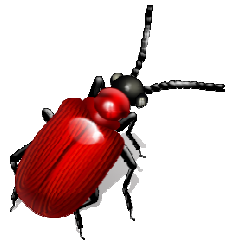
http://www.hcs.harvard.edu/~gdc/index.php?n=Reshall.Perkins

Arrays

- Arrays are formed of **identical individual** blocks, which can be accessed through an index number (your mailbox #, for instance).
- Want an array of size n? Array indices start at 0 and end at n-1.

Declaring and Initializing Arrays

```
//array of size n
int student_grade[n];


for(int i = 0; i <= n; i++)
    student_grade[n] = 0;
```



Declaring and Initializing Arrays

- Bug 1: If accessing out-of-bounds indices (i.e, student[n] where bounds are 0 to n-1), you access data that's not yours

=>

catastrophic errors

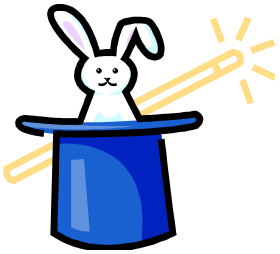
(or malicious programmer).

- Bug 2: Array names aren't variables. You need loops to copy the contents of one array to another:

```
int mail[n], post_office[3*n];
//fill to 1/3 of capacity
for (int i = 0; i < n; i++)
        post_office[i] = mail[i];
```

NOT ~~`post_office = mail;`~~

Magic…



…and bad design!

- Remember the number 23 in mario.c?
- What if a reader 10 years from now stumbles upon your code? Will 23 have meaning to that reader?

Magic Numbers

- Like 23 make your code non-portable. What happens if a different terminal window size is used?
- C provides a construct to deal with such constants:

```
# define const_name const_value
```

Most useful for arrays!

define

- `const_name` is NOT a variable. It is treated by the compiler as replacing every textual instance of const_name in your code by what followed it in the #define.

- Not a variable = > Can't be changed by any function in your program

- **Excellent for array bounds info!**

#define and arrays

```
#define NUM_TFS 30
int main(int argc, char* argv[])
{
   string tf_array[NUM_TFS];
   ... //populate array
   //print out array contents
   for(int i=0; i<NUM_TFS; i++)
      printf("%s\n", tf_array[i]);
}
```

Multidimensional arrays

```
#define x_max 600
#define y_max 800
//values of tones of grey (0…16)
int pixel[x_max][y_max];
for(int i=0; i<x_max; i++)
   for(int j=0; j<y_max; j++)
       pixel[x_max][y_max]=GetInt();
```

Imagine your screen is just like the mailbox grid

Arrays as function arguments

- When you pass an array to a function, the function doesn't get its own copy of that, it's the same array.
- This means that the contents of the actual array can be changed by a function (unlike when you pass a variable like increment(x)).
- Takeaway:

ARRAYS ARE NOT VARIABLES [DEMO]

Command Line Arguments

- The mysterious code we've been using:

```
int main (int argc, char* argv[])
```

Can be explained through the use of arrays:

```
argc = # of strings that make up
  command line, including command
  itself (ie, mario executable)
argv[] = array that contains those
  strings
```

Hint-hint!

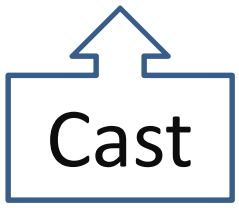
- Error checking:

```
if(argc!=2)
{
   printf(“Error: too few args”);
   return 1;
}
//hint-hint!
int height = atoi(argv[1]);
//don’t forget to include stdlib.h
```

Typecasting

- The operator *cast* forces a conversion from one type of variable to another (ie, int to float). Useful for division:

```
int i, j;
double a, e, f;
a = 100 / (double) i;
j = (int) (e+f);
```



Typecasting (2)

- NOT useful for strings, though:

```
int foo;
string bar;
foo = (int) “246”;
bar = (string) 246;
```

are **NOT** equivalent; use functions like atoi() (ASCII string to Int) and **look up the man pages** ☺

Conditional Operator ? :

- Ternary (takes three arguments):

  `<condition>?<true_expr>:<false_expr>`

- It's like a short-circuit: only one of the two expressions following ? is executed. Think of it similar to an if-else.

- If `<condition>` is true, only `<true_expr>` exaluates. Otherwise, only `<false_expr>` evaluates.

```
//example for ?: operator
int max (int x, int y)
{
    int greater;

    greater = (x > y ? x : y);

    return greater;
}
```

Bug opportunity: Make sure the two expressions are of the same type!

Style

- **White space =** proper tabs + newlines to structure your code;
- **Consistency =** use indentation and format similar pieces of code the same way. Function definitions, if-elses, nested loops should be consistent.
- **Comments** = header comments for functions; use nontrivial parts of code should always be commented; with moderation, though.
- **For examples: this week's lecture source code.**

Crypto and you

- The hacker dates back to Caesar…

- Caesar cypher:

$$c_i = (p_i + k)\ mod\ 26$$

Remember ROT13?

- Key here is the constant *k* that goes from 0 to 25 (there are 26 letters in the alphabet).

Crypto and you (2)

- You just shift by *k* positions the letters in the message to cypher it;
- Problem here?
- 44 BC…

Better than Caesar: Vigènere

- Vigenère cypher uses a different key *k* for every letter of the message:

$$c_i = (p_i + k_i) \; mod \; 26$$

T	H	I	S		C	L	A	S	S		R	O	C	K	S	!
C	*O*	*M*	*P*		*U*	*T*	*E*	*R*	*C*		*O*	*M*	*P*	*U*	*T*	
V	**V**	**U**	**H**		**W**	**E**	**E**	**J**	**U**		**F**	**A**	**R**	**E**	**L**	**!**

Keyword:
Computer

Vigènere Tableau

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

That’s all folks!

http://i-love-cartoons.com