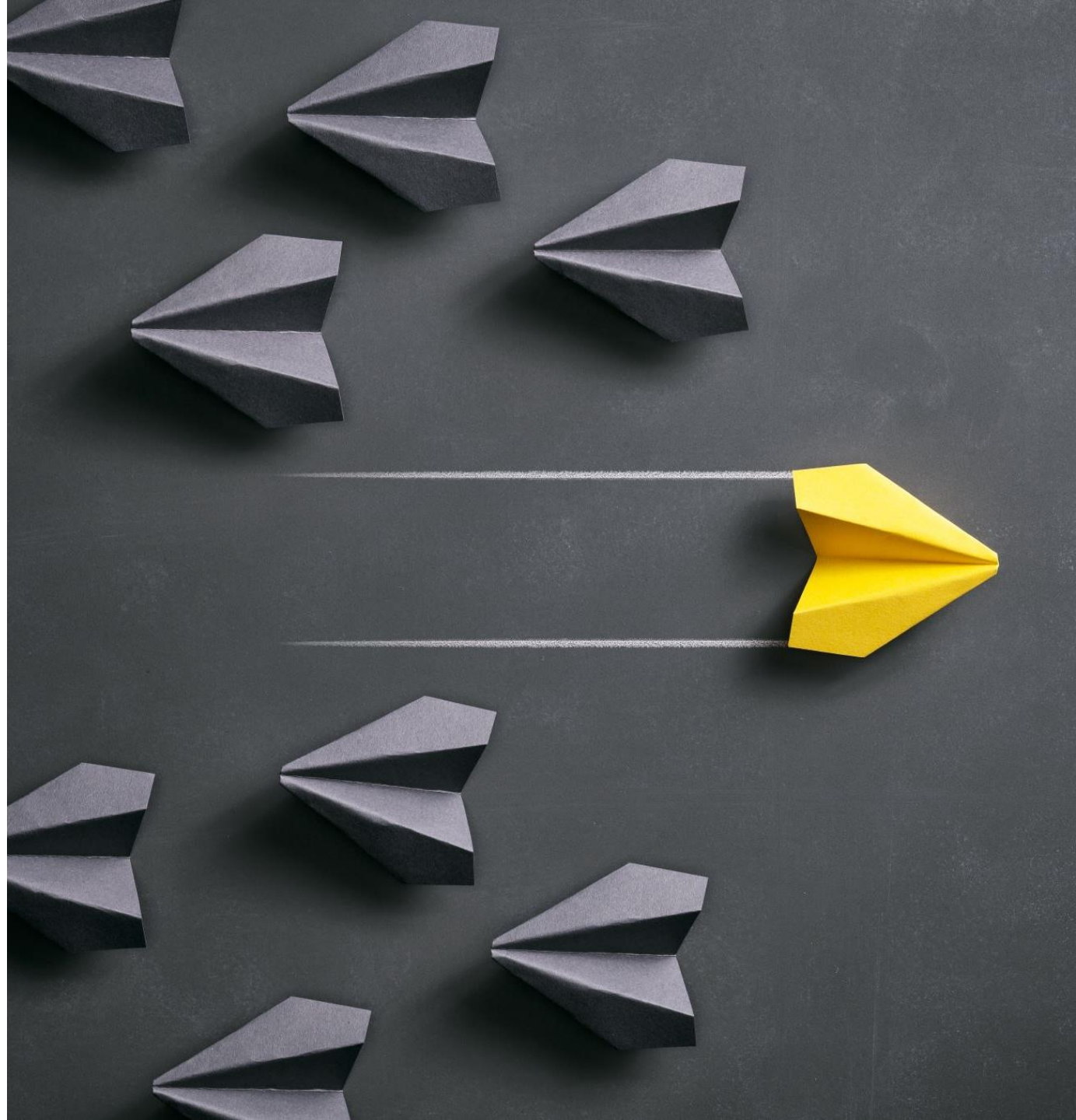


SCC.111 Software Development – Lecture 4: Functions & Flow

Adrian Friday, Nigel Davies, Hansi Hettiarachchi, Saad Ezzini

Today: Flow into functions

- Extending our knowledge on how code flows
- How and when to declare our own functions
- Flow of data through functions (parameter passing)
- How function calls change program flow





Recall we
said that, our
code moves
at different
scales



Between programs... (later courses...)



Between blocks of code (*this* lecture)



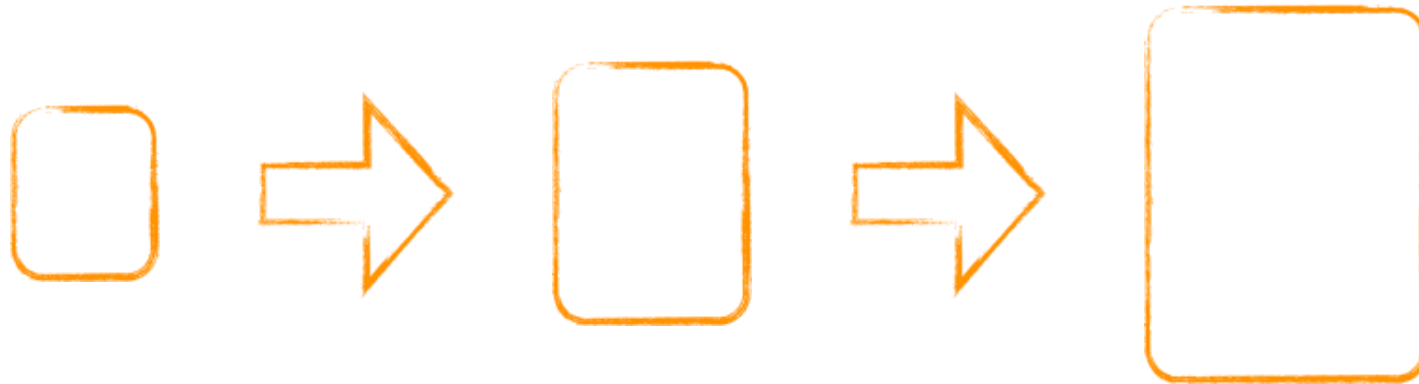
Between statements of the program
(*last* lecture)



Even, within expressions within the
statements as they're evaluated (*last*
lecture)

Solving by problem decomposition

- We solve pretty much all problems by repeatedly decomposing it into a series of smaller steps until the problem is tractable.



Iterative refinement

Sometimes it makes sense

- To package up sets of statements into functional unit
- Or blocks of statements we would otherwise repeat
- Makes our code easier to maintain and reuse (we write less code and have fewer places to change it!)





An example

A top-down view of a wooden desk with various items. In the upper center is a yellow mug filled with dark liquid. To its left is a black smartphone. Below the mug is an open sketchbook with pencil drawings of geometric shapes, including a hexagonal grid and a cube. A closed black notebook lies on top of the sketchbook. To the right of the notebook are two pens, one silver and one black. On the far right is a white architectural model made of foam blocks, showing a stepped, terraced structure. The text "Let's design a solution on paper" is overlaid in white, centered on the image.

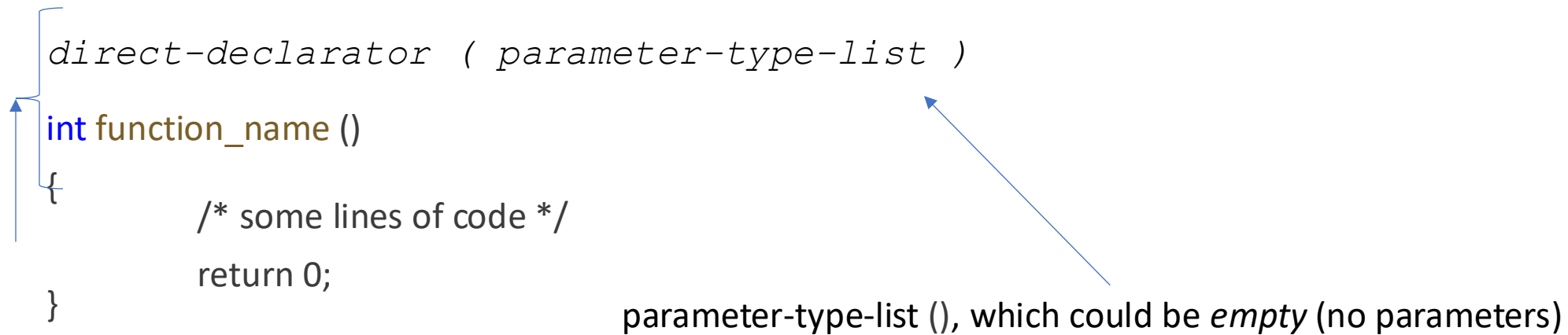
Let's design a solution on
paper

Functions in C

- Most programming languages give us some support for this:
 - C lets us define functions which package and name specific functionality
 - And libraries – sets of pre-compiled functions we can call (defined by header files!)
 - Later, we might even bundle up our functions to create our own libraries!



C functions follow the form:



```
direct-declarator ( parameter-type-list )
{
    /* some lines of code */
    return 0;
}
```

parameter-type-list (), which could be *empty* (no parameters)

The diagram illustrates the syntax of a C function. A blue bracket on the left groups the return type 'int' and the function name 'function_name' as the 'direct-declarator'. A blue arrow points from the text 'parameter-type-list (), which could be empty (no parameters)' to the empty parentheses '()' in the function signature 'int function_name ()'.

- A function may only return a simple *arithmetic type* (e.g. an 'int'), or nothing 'void' (no return)
- They are followed by a *compound-statement*, i.e. {...}, the function body

Beep!

```
#include <stdio.h>

void beep(int howManyTimes)
{
    // Create 'beeps', set to 0, loop until beeps reaches 'howManyTimes' value

    for (int beeps = 0; beeps < howManyTimes; beeps++) {
        printf("Beep!\n");
        sleep(1);
    }
}

int main()
{
    // Make bigtrak beep some number of times

    beep(5);

    return 0;
}
```

main () as a function

/* I'm a function, the 'entry point' to the program
I take no parameters and return a code to the shell
Strictly, main is: int main(int argc, char *argv[])
*but more on this later! */*

```
int main()  
{  
    // do something, then return  
}
```




Where do we declare functions?

We can declare functions before use

```
#include <stdio.h>
```

```
void beep(int howManyTimes)
{
    for (int beeps = 0; beeps < howManyTimes; beeps++) {
        printf("Beep!\n");
        sleep(1);
    }
}
```

```
int main()
{
    // Make bigtrak beep some number of times

    beep(5);

    return 0;
}
```



Compiler 'sees' beep declaration



So can call it here

The new 'flow'

```
#include <stdio.h>
```

```
void beep(int howManyTimes)
```

```
{  
  3 for (int beeps = 0; beeps < howManyTimes; beeps++) {  
    printf("Beep!\n");  
    sleep(1);  
  }  
}
```

```
int main()  
{
```

```
  // Make bigtrak beep some number of times
```

```
  beep(5);
```

```
  return 0;  
}
```

Jump to here 2

Still start here! 1

Return to here 4



Data flows as well!



The flow of data

```
int main()  
{  
    printf("Hello, world\n");  
}
```

"Hello, world\n" is data that 'gets passed' into
printf

A variable inside the function (the argument)
becomes "Hello, world\n"

Similarly

```
#include <stdio.h>
```

```
void beep(int howManyTimes)
```

```
{  
    for (int beeps = 0; beeps < howManyTimes; beeps++) {  
        printf("Beep!\n");  
        sleep(1);  
    }  
}
```

```
int main()
```

```
{  
    // Make bigtrak beep some number of times
```

```
    beep(5);
```

```
    return 0;  
}
```

howManyTimes set to 5

2

Parameter is a variable we can use

3

5 passed in as argument

1

Results are returned from functions too

```
#include <stdio.h>
```

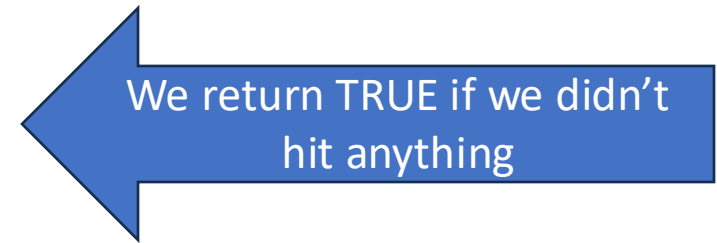
```
int move_forward(int howManyTimes)
{
    while (howManyTimes > 0 && sense_obstacle() == 0) {
        motor_on();
        sleep(1);
        motor_off();
        howManyTimes--;
    }

    return howManyTimes == 0;
}
```

```
int main()
{
    // Make bigtrack move forward

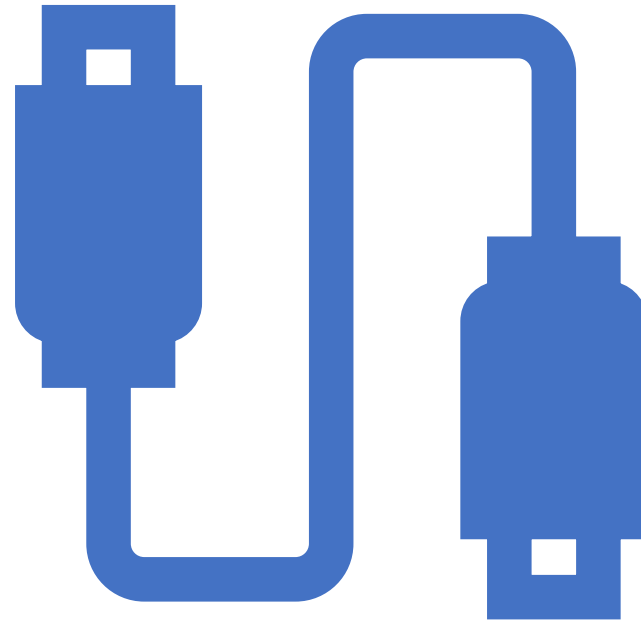
    if (move_forward(5))
        printf("Success!\n");
    else
        printf("Oh no!\n");

    return 0;
}
```



The function effectively 'evaluates' to the returned result

A function declaration is
like a 'socket' with a
precise specification.
We can 'plug' into (or
'call') it, iff our
arguments match



‘Formal’ and ‘actual’ parameters

- The function **declaration** specifies type information for each parameter (formal parameter list), and the caller must pass actual parameters that match these in **both** type and position!

Declaration `float buy_coffees(int howMany, float cost) { ... }`

Call `float totalCost = buy_coffees(10, 4.50);`

In C, the parameter values in the function are **only ever a copy** of what's passed in (*pass by value*)!

Even if the formal and actual parameters are variables with the same name, they exist in '**different scopes**' (*more later*)

Summary

- You should understand
 - That we can create reusable sub-units of code called functions
 - That programs flow into and out of functions
 - How information (variables) take information in and out of a function