COMP20005 Workshop Week 5

```
1  Functions. Discuss: Ex. 5.01 5.02, 5.03, 5.05

2  (time permitted) Recursive Functions, Ex 5.13


3    implement 5.05, 5.06, 5.14


4    while (having_time()){
       review/ask questions on MST
       implement("a new exercise from grok C05");
     }
```

Functions

x= 2.0, need to print out √x

How?

(Library) functions are black boxes

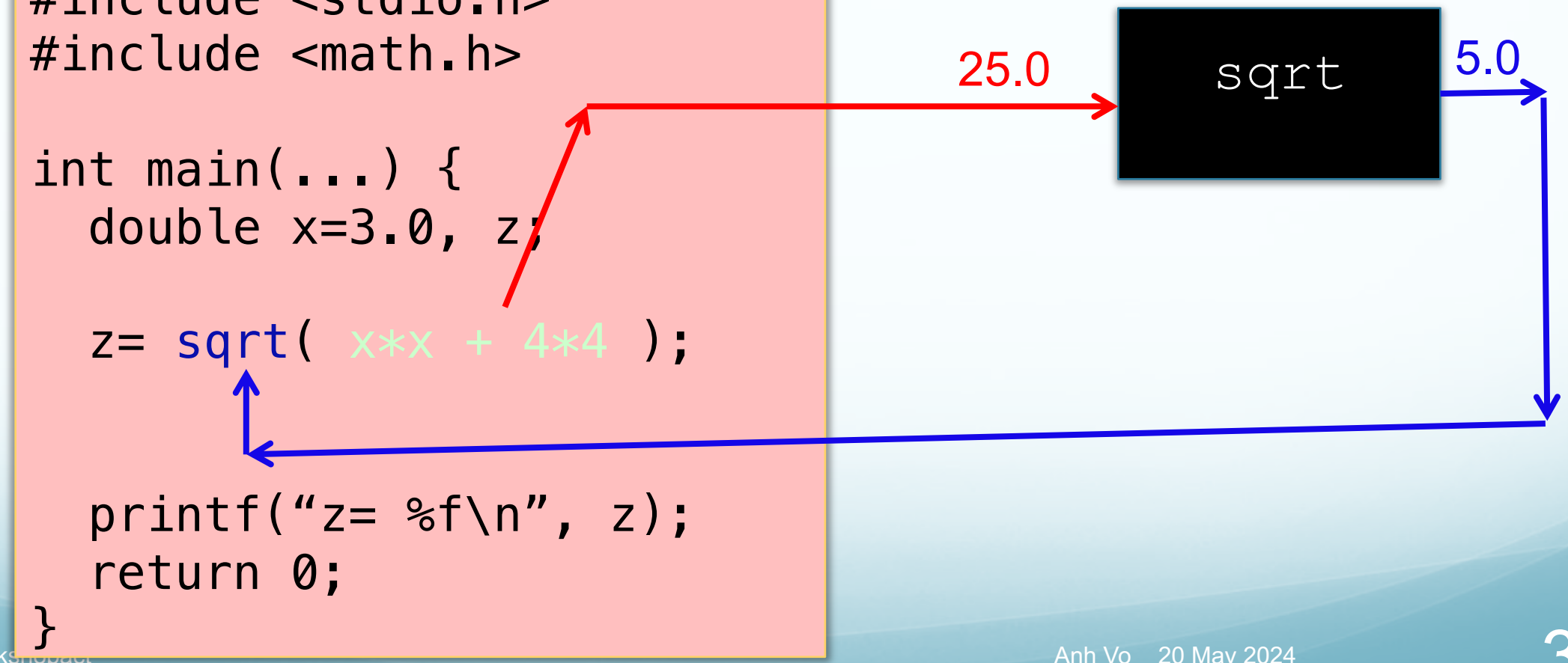
my_prog.c

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

int main(...) {
  double x=3.0, z;

  z= sqrt( x*x + 4*4 );

  printf("z= %f\n", z);
  return 0;
}
```



Last week solution

```
... scanf("%d", &n) ...
while (1) {
  n++;
```

input: **n**

```
  int isprime = 1, div;
  for (div = 2; div * div <= n; div++) {
    if (n % div == 0) {
      isprime = 0;
      break;
    }
  }
  // - if isprime == 1 ➔ prime
  // - if isprime == 0 ➔ not prime
```

output: **isprime**

```
  if (isprime) {
     break;
  }
}
printf("next prime= %d\n", n);
```

Functions - Useful Abstraction

Exercise 4.09: Write a program nextprime that calculates the *next* prime number after a given value.

Alistair's code for checking if n is a prime number

Last week solution

```
... scanf("%d", &n) ...
while (1) {
  n++;

  int isprime = 1, div;
  for (div = 2; div * div <= n; ...
    if (n % div == 0) {
      isprime = 0;
      break;
    }
  }
  // - if isprime == 1 ➔ prime
  // - if isprime == 0 ➔ not prime

  if (isprime) {
     break;
  }
}
printf("next prime= %d\n", n);
```

input: **n**

output: **isprime**

Using a Function as an Abstraction

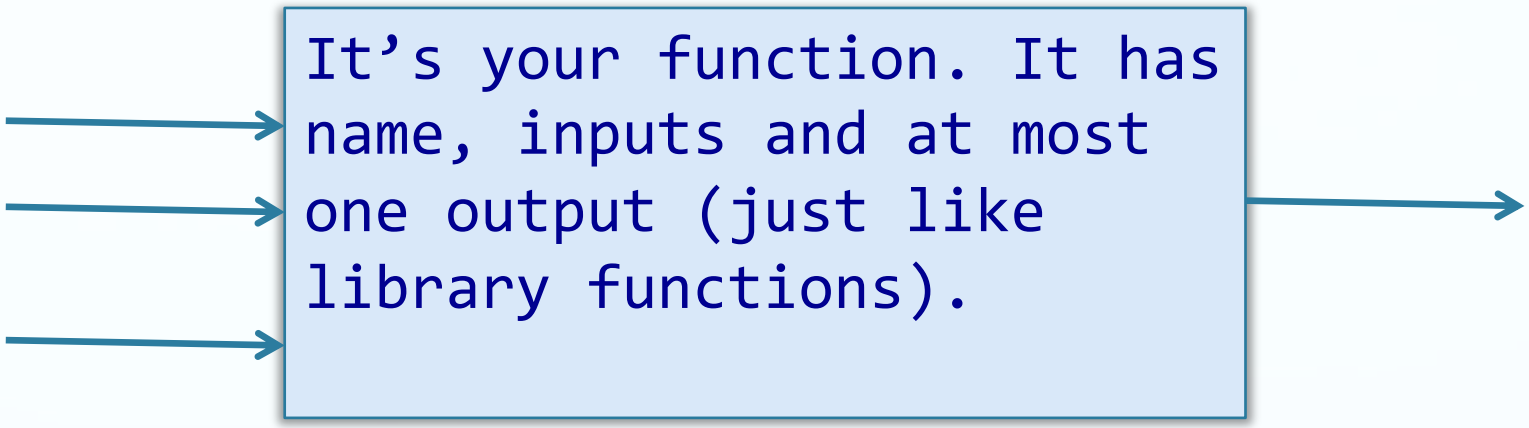
```
  ... scanf("%d", &n) ...
  while (1) {
    n++;
    if (is_prime_number(n)) {
       break;
    }
  }
  printf("next prime= %d\n", n);
  return 0;
}

int is_prime_number(int n) {

  int isprime = 1, div;
  for (div = 2; div * div <= n; div++){
    if (n % div == 0) {
      isprime = 0;
      break;
    }
  }

  return isprime;
}
```

How to write our own *functions*



User-defined functions & program structure

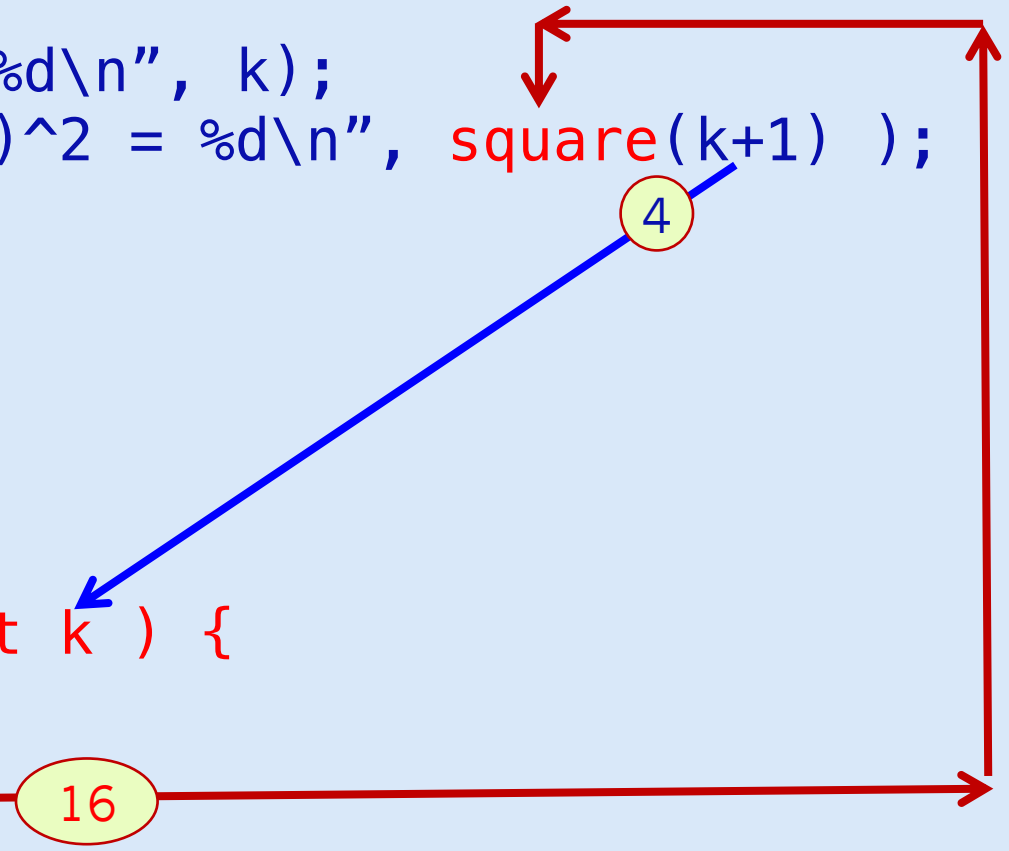
my_prog.c

```
#include <stdio.h>
int square(int);  // function prototype

int main(int argc, char *argv[]) {
   int k= 3;
   printf("k = %d\n", k);
   printf("(k+1)^2 = %d\n", square(k+1) );

   return 0;
}

// input: n
// output: n^2
int square( int k ) {
    return k*k;
}
```



how to write a function

First, write the function header, therefore determine:

```
int        square      ( int n )
```

data type of function result	name of the function	list of inputs, for each: • name • data type

Then, design the algorithm and write the function body: use the *inputs*, compute and *return* the result.

	my_program.c

#include & #define
function prototypes
 	
main() function –start
 	
-end
	`#include ...`
	`int factorial( int ) ;`
	
	`int main(int argc, char *argv[]) {`
	`  int n;`
	`  ...  // including scanf/set value for n`
	`  printf( "%d! = %d\n", n, factorial(n));`
	`  ...`
	`  return 0;`
	`}`

implementation of
declared functions
	`int factorial( int k ) {`
	`  ...`
	`  return ...;`
	`}`

Do-Together Ex 5.01, 5.02, 5.03

For each task, spend a few minutes *before writing* to discuss with your peers:

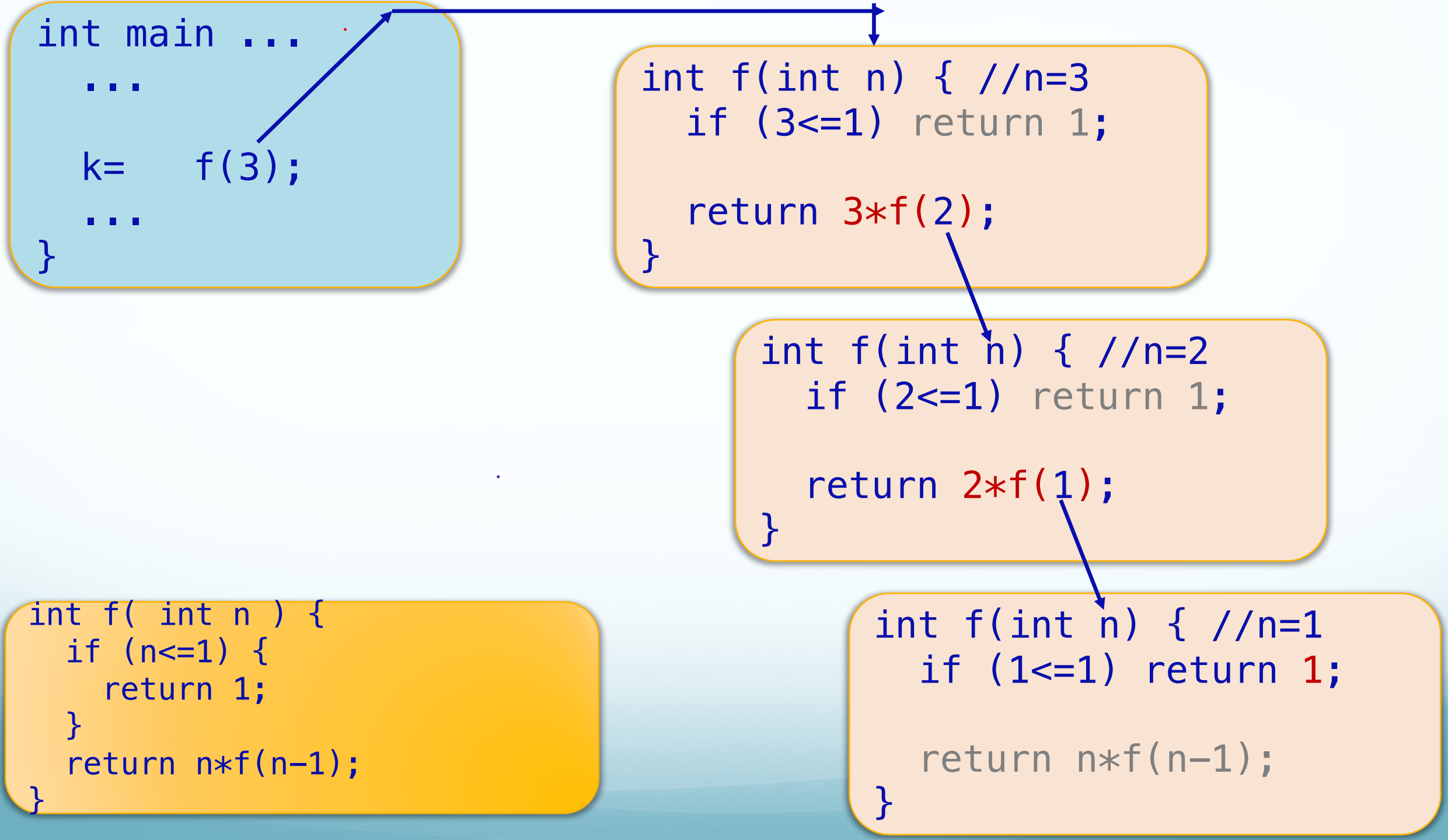
- WHAT are possible approach(es)?
- WHICH APROACH is simpler for us to write?

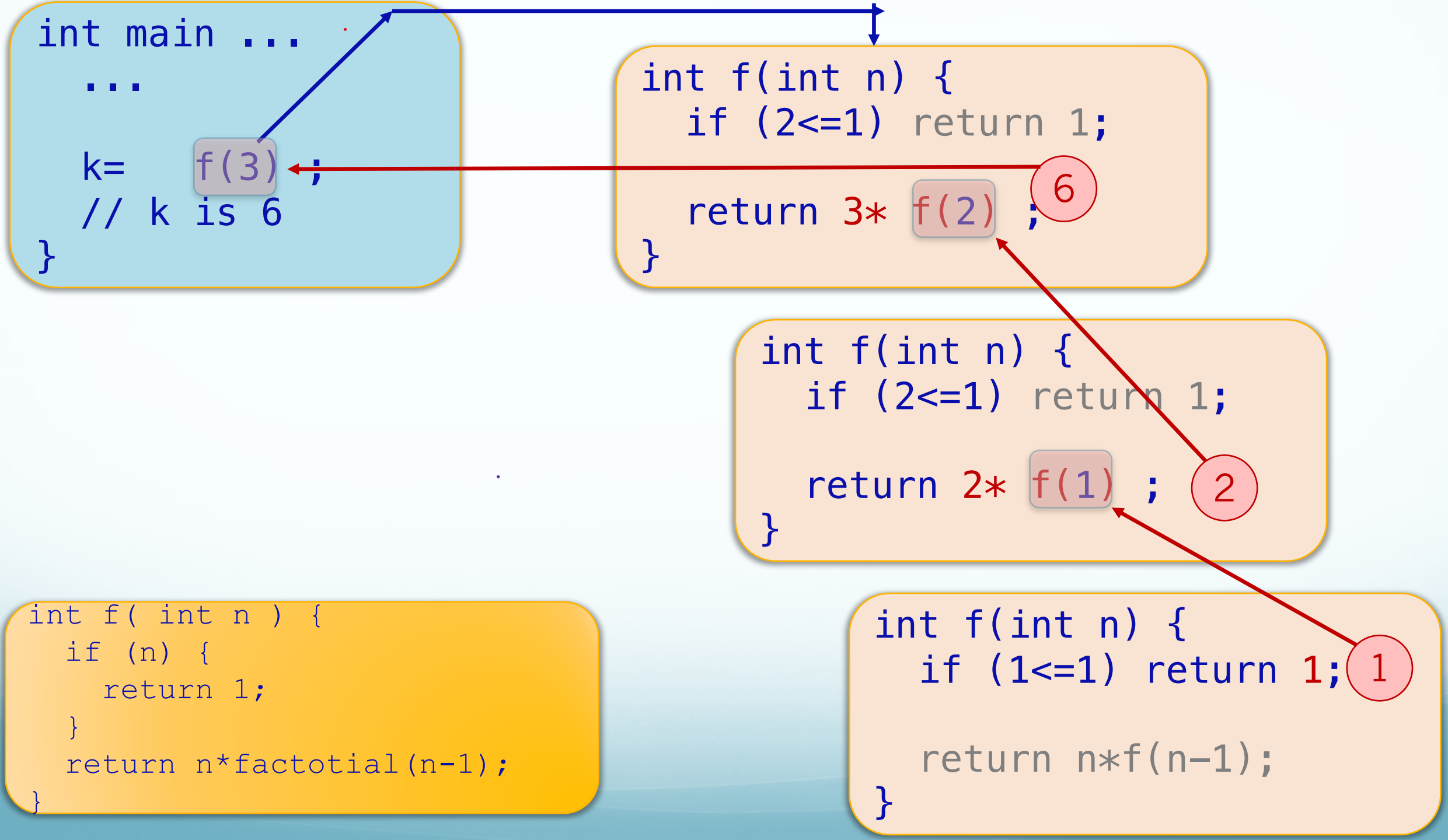
Recursive functions [time permits]

Recursive function: function that calls itself

```
int fib( int n ) {
  if (n==1) return 1;
  if (n==2) return 1;
  return fib(n-1) + fib(n-2);
}
```

```
int factorial( int n ) {
  if (n==1) {
    return 1;
  }
  return n * factorial(n-1);
}
```


int main ...
...
k= f(3);
...
}
int f(int n) { //n=3
if (3<=1) return 1;
return 3*f(2);
}
int f(int n) { //n=2
if (2<=1) return 1;
return 2*f(1);
}
int f(int n) { //n=1
if (1<=1) return 1;
return n*f(n-1);
}
int f(int n) {
if (n<=1) {
return 1;
}
return n*f(n-1);
}


int main ...
...
k= f(3) ;
// k is 6
}
int f(int n) {
if (2<=1) return 1;
return 3* f(2) ;
}
6
int f(int n) {
if (2<=1) return 1;
return 2* f(1) ;
}
2
int f(int n) {
if (n) {
return 1;
}
return n*factotial(n-1);
}
int f(int n) {
if (1<=1) return 1;
return n*f(n-1);
}
1

Recursive functions: How

- reduce the task of size n to the same tasks of smaller sizes
- clearly describe the base cases where the solutions are trivial
- when writing code, start with solving the base cases first

Examples: factorial (n):

- base case: when n==1 the solution is 1
- general case: factorial(n) can be computed from factorial(n-1)

```
int factorial( int n ) {
  if (n==1) { // base case
    return 1;
  }
  // general case   [note: else is normally not needed]
  return   factorial(n-1)*n;
}
```

- Do-Together **Exercise 5.13**

Lab

- Implement `5.05` and `5.06`
- Extras: 5.11, 5.6b, 5.14, and other exercises in C05

For each task, spend a few minutes *before writing* to discuss with your peers:

- WHAT are possible approach(es)?
- WHICH APROACH is simpler for us to write?

- Week 6 Workshop:
  - Review for MST & Ask Questions
  - Try sample tests several times before the workshop

5.05 & 5.06

5.05: A number is perfect if it equal to the sum of its factors, excluding itself. An example is 6 (=1+2+3).

- Write function `int isperfect(int)` *that returns true if its argument is a perfect number, and false otherwise.*

- Write function `int nextperfect(int)` *that returns the next perfect number greater than its argument.*

You need a `main()` *function that reads a number. If the number is perfect, print out it; if not, print out the next perfect number.*

WARNING: the first 6 perfect numbers are

6, 28, 496, 8 128, 33 550 336, 8 589 869 056

5.06: Two numbers are an amicable pair if their factors (excluding themselves) add up to each other. The first such pair is 220, which has the factors [1, 2, 4, 5, 10, 11, 20, 22, 44,55, 110], adding to 284; and 284, which has the factors [1, 2, 4, 71, 142], the sum of which is 220. The next pairs are 1,184 and 1,210; and then 2,620 and 2,924.

Write a function that takes two `int` *arguments and return true if they are an amicable pair. Then, test your function by writing a simple main program that inputs 2 integers and prints out if they are an amicable pair.*

Remember

Review chapters 1-5 for the MST:

- program structures, data types, expression, precedence order
- `scanf` and `printf`
- `if ...`
- loops: `for ...` , `while ...`
- functions, recursive functions
- others, better not to use: `switch ...`, `do ... while`