BLG 102E
Introduction to Scientific Computing and Engineering

SPRING 2025

WEEK 5

ISTANBUL TECHNICAL UNIVERSITY

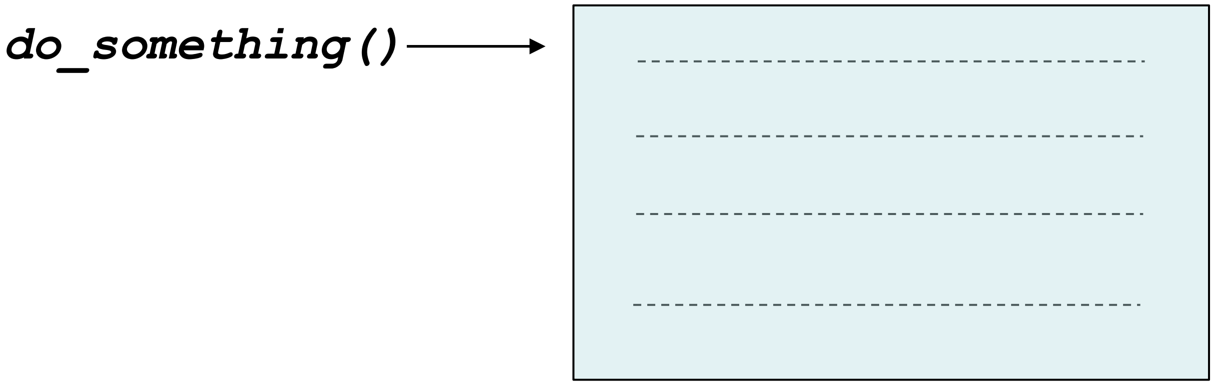
Functions

Slides by Evan Gallagher

What Is a Function? Why Functions?

A function is a sequence of instructions with a name.

A function packages a computation into a form that can be easily understood and reused.



Program Modules in C

- Functions
  - C functions are subprogram modules
  - Function categories:
    - Built-in library functions (ready-made C standard libraries)
    - Programmer-defined functions
- Function calls
  - Invoking functions
    - Provide function name and arguments (data)
    - Function performs operations or manipulations
    - Function returns a result

Implementing Functions

Write the function that will do this:

(***any*** cube)

Compute the volume of *a* cube with a given side length

Implementing Functions

When writing this function, you need to:

- Pick a good, descriptive name for the function

Implementing Functions

When writing this function, you need to:

- Pick a good, descriptive name for the function

(What else would a function named **`cube_volume`** do?)

`cube_volume`

Implementing Functions

When writing this function, you need to:

- Pick a good, descriptive name for the function

- Give a type and a name for each parameter.

```
cube_volume
```

Implementing Functions

When writing this function, you need to:

- Pick a good, descriptive name for the function

- Give a type and a name for each parameter.
  There will be one parameter for each piece
  of information the function needs to do its job.

(And don't forget the parentheses)

```
cube_volume(double side_length)
```

Implementing Functions

When writing this function, you need to:

- Pick a good, descriptive name for the function

- Give a type and a name for each parameter.
  There will be one parameter for each piece
  of information the function needs to do its job.

- Specify the type of the return type

```
cube_volume(double side_length)
```

Implementing Functions

When writing this function, you need to:

- Pick a good, descriptive name for the function

- Give a type and a name for each parameter.
  There will be one parameter for each piece
  of information the function needs to do its job.

- Specify the type of the return type

```
double cube_volume(double side_length)
```

Implementing Functions

When writing this function, you need to:

- Pick a good, descriptive name for the function
- Give a type and a name for each parameter. There will be one parameter for each piece of information the function needs to do its job.
- Specify the type of the return type

Now write the *body* of the function:

the code to do the cubing

Implementing Functions

The code the function names must be in a block:

```
double cube_volume(double side_length)
{
   double volume = side_length * side_length * side_length;
   return volume;
}
```

Implementing Functions

The parameter allows the caller to give the function information it needs to do it’s calculating.

```
double cube_volume(double side_length)
{
    double volume = side_length * side_length * side_length;
    return volume;
}
```

Implementing Functions

The code calculates the volume.

```
double cube_volume(double side_length)
{
   double volume = side_length * side_length * side_length;
   return volume;
}
```

Implementing Functions

The **`return`** statement gives the function’s result to the caller.

```
double cube_volume(double side_length)
{
   double volume = side_length * side_length * side_length;
   return volume;
}
```

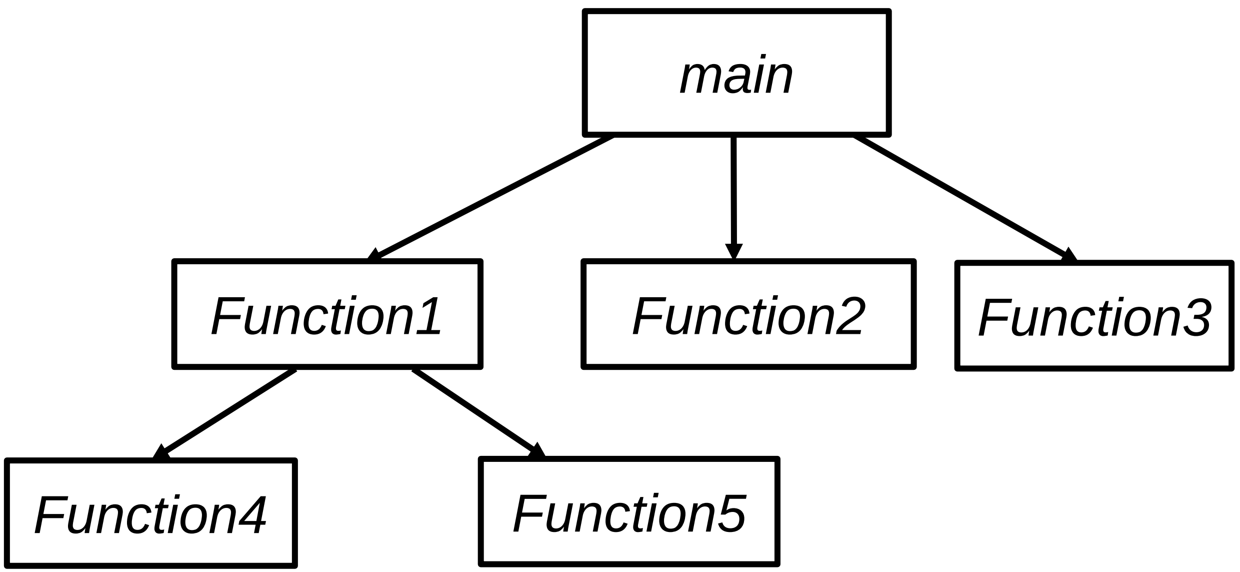
Test Your Function

You should always test the function.

You'll write a **`main`** function to do this.

Calling Functions

Consider the order of activities when a function is called.



Calling (i.e., using) a function

```
#include <stdio.h>

double cube_volume(double side_length)
{
   double volume = side_length * side_length * side_length;
   return volume;
}

int main()
{
   double result1 = cube_volume(2);
   double result2 = cube_volume(10);
   printf("A cube with side length 2 has volume %f\n", result1);
   printf("A cube with side length 10 has volume %f\n", result2);

   return 0;
}
```

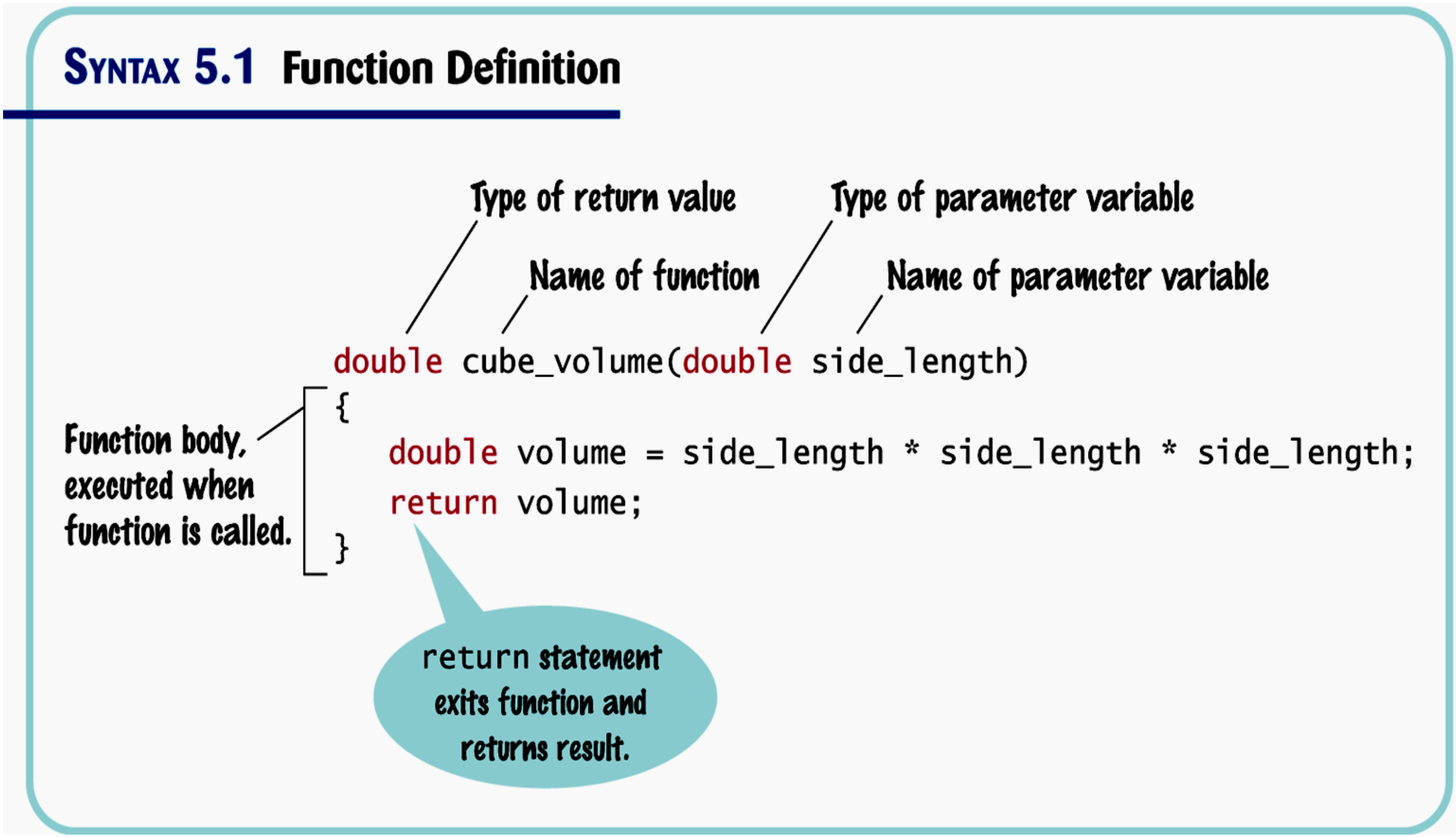
Example: Calling a function

- *When **f** function is called from main program, value of the **a** variable is automatically copied to the **x** parameter variable of the function.*
- *The returned result is assigned to the **b** variable in main.*

```c
// Caller program
#include <stdio.h>
int main()
{
   float a = 1.5;
   float b;
   // Send the argument value and
   // get the returned value

   b = f (a);

   printf( "%f", b);
   return 0;
}
```

```c
// Called function

float f (float x)
{
    float result;
    result = x * 2;
    return result;
}
```

Implementing Functions



An Output Statement Does Not Return a Value

output on terminal ≠ return

If a function needs to display something for a user to see, it cannot use a **`return`** statement.

An output statement using **`printf`** communicates *only* with the user running the program.

The Return Statement Does Not Display (Good!)

output on terminal ≠ return

If a programmer needs the result of a calculation done by a function, the function *must* have a **`return`** statement.

An output statement using printf does *not* communicate with the calling programmer

The Return Statement Does Not Display (Good!)

```
int main()
{
    double z = cube_volume(2);

    // display result of calculation
    // stored in variable z
    printf("%f",z);

    // return from main - no output here!!!
    return 0;
}
```

Main function

- The main is a function called by the operating system.

```
int main()
{
  return 0;
}
```

- In Windows command-line, the returned value of main can be shown as following:

```
C:\>myprog.exe

C:\>echo  %ERRORLEVEL%
 0
```

Return Values

The **`return`** statement yields the function result.

Also,

The **`return`** statement

- terminates a function call

- immediately

Return Values

This behavior can be used to handle unusual cases.

What should we do if the side length is negative?
We choose to return a zero and not do any calculation:

```
double cube_volume(double side_length)
{
   if (side_length < 0)
      return 0;
   double volume = side_length * side_length * side_length;
   return volume;
}
```

Return Values

The **`return`** statement can return the value of any expression.

Instead of saving the return value in a variable and returning the variable, it is often possible to eliminate the variable and return a more complex expression:

```
double cube_volume(double side_length)
{
   return side_length * side_length * side_length;
}
```

Common Error – Missing Return Value

Your function always needs to return something.

Consider putting in a guard against negatives
and also trying to eliminate the local variable:

```
double cube_volume(double side_length)
{
   if (side_length >= 0)
   {
      return side_length * side_length * side_length;
   }
}
```

Common Error – Missing Return Value

Consider what is returned if the caller *does* pass in a negative value!

```
double cube_volume(double side_length)
{
   if (side_length >= 0)
   {
      return side_length * side_length * side_length;
   }
}
```

Common Error – Missing Return Value

Every possible execution path
should return a meaningful value:

```
double cube_volume(double side_length)
{
   if (side_length >= 0)
   {
      return side_length * side_length * side_length;
   }
}
```

?

Common Error – Missing Return Value

Depending on circumstances, the compiler might flag this as an error, or the function might return a random value.

This is always bad news, and you must protect against this problem by returning some safe value.

Functions Without Return Values – The `void` Type

void

```
void square_display( int y )
 {
   printf("%d  ", y * y ); // print the result
 }
```

Notice the return type of this function

Functions Without Return Values – The `void` Type

This kind of function is called a ***`void`*** *function*.

```
void square_display( int y )
```

Use a return type of **`void`** to indicate that a function does not return a value.

`void` functions are used to

- simply do a sequence of instructions
  - – They do not return a value to the caller.
    - > Returning a value will lead to an error.

Functions Without Return Values – The `void` Type

Because there is no return value, you
cannot use **`square_display`** in an expression.

You can make this kind of call:

```
square_display(5);
```

but not this kind:

```
result = square_display(5);
// Error: square_display doesn't
//        return a result.
```

Functions Without Return Values – The `void` Type

If you want to return from a **`void`** function before reaching the end, you use a **`return`** statement without a value. For example:

```
void square_display( int y )
{
  if (n < 0) {
      return; // Return immediately
  }
  // The following lines will not be
  // executed if n < 0

  printf("%d  ", y * y ); // print the result
}
```

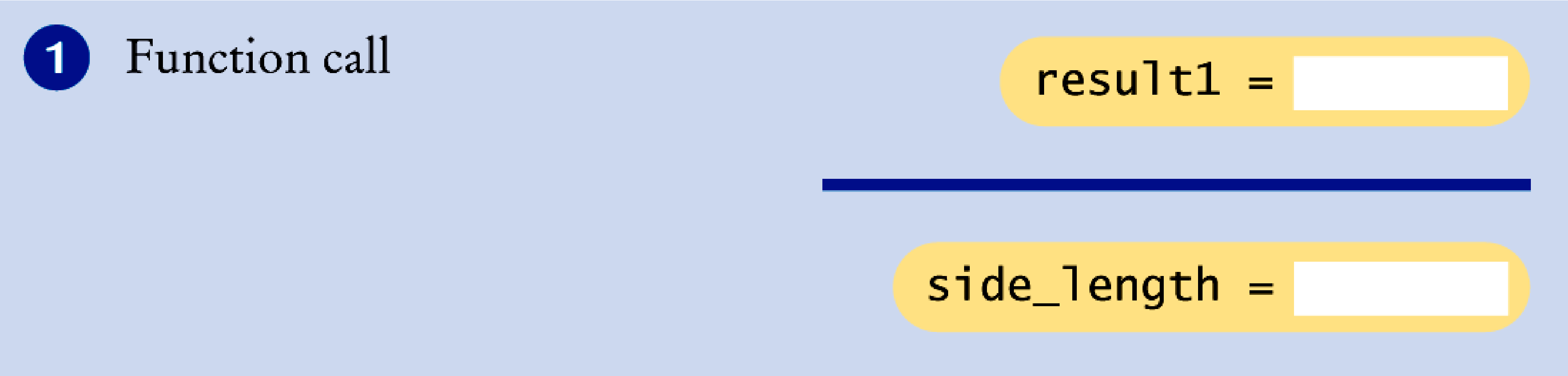
Calling (i.e., using) a function

```
#include <stdio.h>

double cube_volume(double side_length)
{
   double volume = side_length * side_length * side_length;
   return volume;
}

int main()
{
   double result1 = cube_volume(2);
   double result2 = cube_volume(10);
   printf("A cube with side length 2 has volume %f\n", result1);
   printf("A cube with side length 10 has volume %f\n", result2);

   return 0;
}
```

Parameter Passing

1. In the calling function, the local variable **`result1`** already exists. When the **`cube_volume`** function is called, the parameter variable **`side_length`** is created.

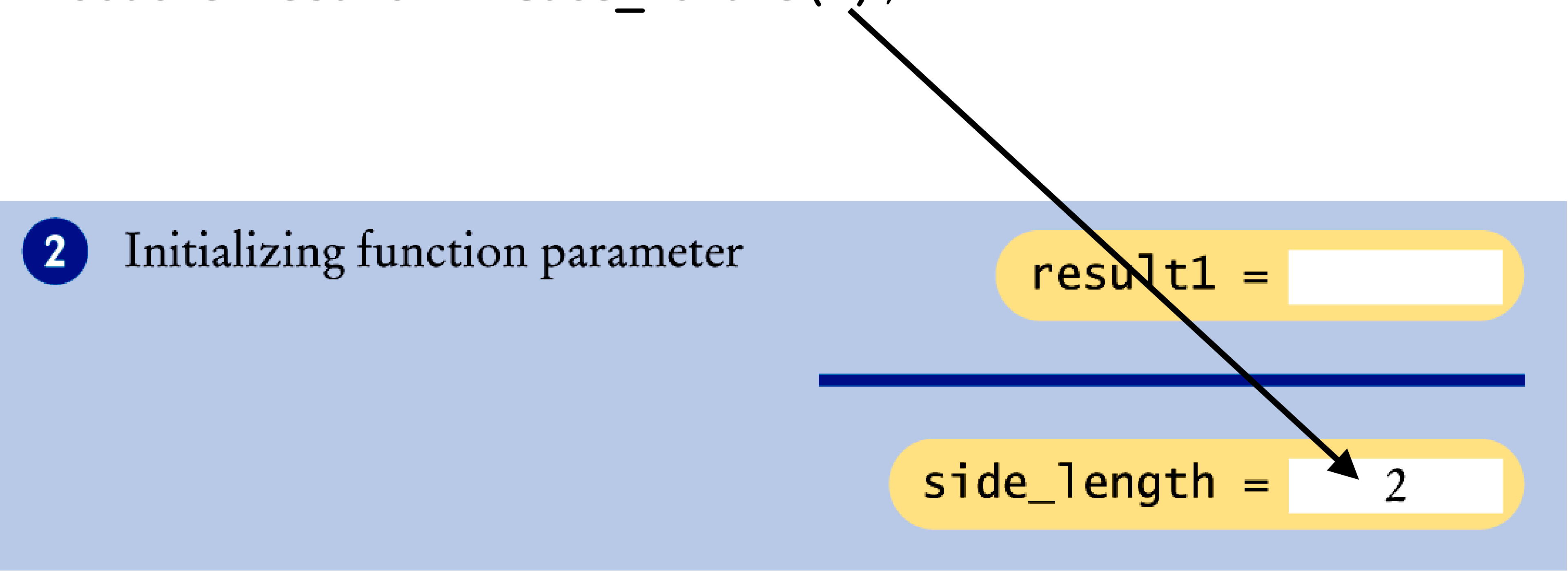
```
double result1 = cube_volume(2);
```



Parameter Passing

2. The parameter variable is initialized with the value that was passed in the call. In our case, **`side_length`** is set to 2.

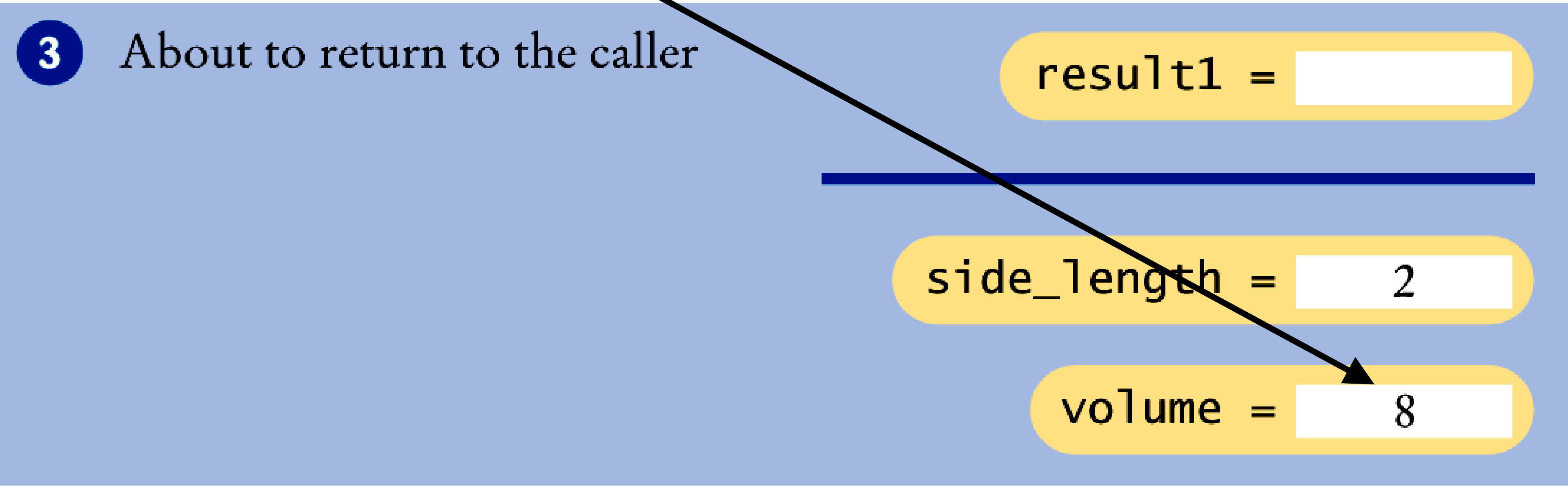
```
double result1 = cube_volume(2);
```



Parameter Passing

3. The function computes the expression **`side_length * side_length * side_length`**, which has the value 8. That value is stored in the local variable **`volume`**.

```
[inside the function]
double volume = side_length * side_length * side_length;
```



Parameter Passing

4. The function returns. All of its variables are removed. The return value is transferred to the caller, that is, the function calling the **`cube_volume`** function.

```
double result1 = cube_volume(2);
```

The function executed: **`return volume;`**
which gives the caller the value **`8`**

Parameter Passing

4. The function returns. All of its variables are removed. The return value is transferred to the caller, that is, the function calling the **`cube_volume`** function.

```
double result1 = cube_volume(2);
```

the returned 8 is about to be stored

The function is over.
`side_length` and **`volume`** are gone.

Parameter Passing

The caller stores this value in their local variable **`result1`**.

```
double result1 = cube_volume(2);
```

Parameter Passing

In the function call,
a value is supplied for each parameter,
called the *parameter value*.
(Other commonly used terms for this value
are: *actual parameter* and *argument*.)

```
int hours = read_value_between(1, 12);

. . .
```

Parameter Passing

When a function is called,
a *parameter variable* is created for each value passed in.

```
int hours = read_value_between(1, 12);
. . .
int read_value_between(int low, int high)
```

Parameter Passing

Each parameter variable is *initialized* with the corresponding parameter value from the call.

```
int hours = read_value_between(1, 12);
. . .
int read_value_between(int low, int high)
                                  1         12
```

Parameter Passing

```
int hours = read_value_between(1, 12);



int read_value_between(int low, int high)
{                                          1          12
    int input;
    do
    {
        printf("Enter a value between %d and
                %d\n", low, high);
        scanf("%d", &input);
    } while (input < low || input > high);
    return input;
}
```

Parameter Passing

Here is a call to the **cube_volume** function:

```
double result1 = cube_volume(2);
```

Here is the function definition:

```
double cube_volume(double side_length)
{
  double volume = side_length * side_length * side_length;
  return volume;
}
```

We'll keep up with their variables and parameters:

```
result1
side_length
volume
```

Commenting Functions

- Whenever you write a function, you should comment its behavior.
- Comments are for human readers, not compilers
- There is no universal standard for the layout of a function comment.
  - The layout used in the previous program is borrowed from the Java programming language and is used in some C++ tools to produce documentation from comments.

Commenting Functions

Function comments do the following:

- explain the purpose of the function
- explain the meaning of the parameters
- state what value is returned
- state any special requirements

```
/**
   Computes the volume of a cube.
   @param side_length the side length of the cube
   @return the volume
*/
double cube_volume(double side_length)
```

Comments state the things a developer who wants to use your function needs to know.

Function Prototypes

- Function prototype
  - Function definition line without function body.
  - Consists of followings:
    - Function name
    - Parameters – what the function takes in
    - Return type – data type function returns (default `int`)
  - Important: Prototype only needed if function definition comes after use in program.

  - Example: The function with the prototype

    ```
    int maximum( int x, int y, int z );
    ```

    - Takes in 3 `ints` (function input parameters)
    - Returns an `int` (function output result)

Example: Function that returns square of a number

```
/* Creating and using a programmer-defined function */
#include <stdio.h>

int square( int y ); // function prototype line

int main() {
   int x; // counter

   // loop 10 times and calculate and output square of x each time
   for ( x = 1; x <= 10; x++ ) {
      printf( "%d  ", square(x) ); // function call
   }
} // end main

// Square function body : Calculates the square of parameter
// and returns the result.
int square( int y ) // y is a copy of argument to function
{
   return y * y; // returns square of y as an int
} // end function square
```

Control is transferred to the function.

Control is transferred back to main.

Program Output

```
1  4  9  16  25  36  49  64  81  100
```

Alternative : Without writing a prototype line

- If a function body is placed at the beginning of program, before it is called the first time by main program (or by other functions), then there is no need to write a prototype line.
- Execution always begins at main program.

```
#include <stdio.h>

int square( int y )
{
   return y * y;
}


int main()
{
  int x;

  for ( x = 1; x <= 10; x++ ) {
      printf( "%d  ", square(x) );
   }
}
```

Functions Without Return Values

```c
#include <stdio.h>

void square_display( int y )
{
   printf("%d  ", y * y ); // function prints the result
}


int main() {
   int x;

   for ( x = 1; x <= 10; x++ ) {
       square_display( x ); // function call
   }
}
```

Program Output

```
1  4  9  16  25  36  49  64  81  100
```

BUILT-IN FUNCTIONS

Math Library Functions

- Math library functions (built-in)
  - perform common mathematical calculations
  - `#include <math.h>`

- Format for calling functions
  - `FunctionName(`*`argument`*`);`
    - If multiple arguments, use comma-separated list
  - `printf( "%f \n", sqrt(900.0) );`
    - Calls function `sqrt`, which returns the square root of its argument
    - All math functions return data type <u>`double`</u>
  - Arguments may be constants, variables, or expressions

Some Math Library Functions

C Function	Description	Examples
`sqrt(x)`	square root of *x*	`sqrt(900.0) is 30.0` `sqrt(9.0) is 3.0`
`exp(x)`	exponential function e^x	`exp(1.0) is 2.718282` `exp(2.0) is 7.389056`
`log(x)`	natural logarithm of *x* (base *e*)	`log(2.718282) is 1.0` `log(7.389056) is 2.0`

`log10(x)`	logarithm of *x* (base 10)	`log10(1.0) is 0.0`
		`log10(10.0) is 1.0`
		`log10(100.0) is 2.0`

`fabs(x)`	absolute value of float *x*	`fabs(5.0) is 5.0`
		`fabs(0.0) is 0.0`
		`fabs(-5.0) is 5.0`

`ceil(x)`	rounds *x* to the smallest integer not less than *x*	`ceil(9.2) is 10.0` `ceil(-9.8) is -9.0`
`floor(x)`	rounds *x* to the largest integer not greater than *x*	`floor(9.2) is 9.0` `floor(-9.8) is -10.0`

Additional Math Library Functions

C Function	Description	Examples
`pow(x, y)`	*x* raised to power *y* (x^y)	`pow(2, 7) is 128.0` `pow(9, 0.5) is 3.0`
`fmod(x, y)`	remainder of *x*/*y* as a floating point number	`fmod(13.657, 2.333) is 1.992`
`sin(x)`	trigonometric sine of *x* (*x* in **radians**)	`sin(0.0) is 0.0`
`cos(x)`	trigonometric cosine of *x* (*x* in **radians**)	`cos(0.0) is 1.0`
`tan(x)`	trigonometric tangent of *x* (*x* in **radians**)	`tan(0.0) is 0.0`

Calling a Function

```
#include <math.h>
int main()
{
   double z = pow(2, 3);
   ...
}
```

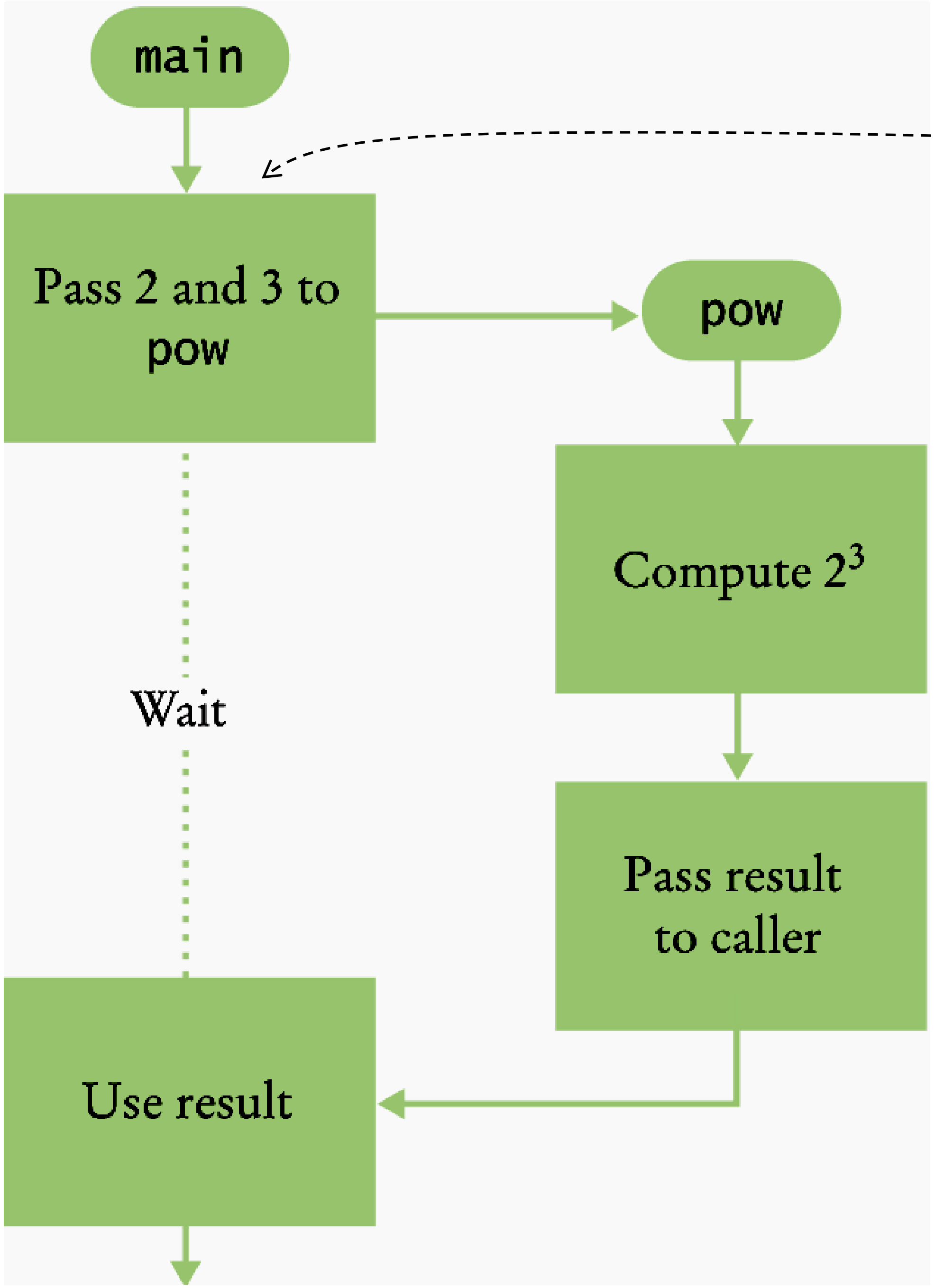
By using the expression: **`pow(2, 3)`**
`main` *calls* the **`pow`** function, asking it to compute 2^3.

The **`main`** function is temporarily suspended.

The instructions of the **`pow`** function execute and compute the result.

The pow function *returns* its result back to **`main`**, and the **`main`** function resumes execution.

Calling a Function




```
int main()
{
    double z = pow(2, 3);
    ...
}
```

Execution flow during a function call

Parameters

```
int main()
{
    double z = pow(2, 3);
    ...
}
```

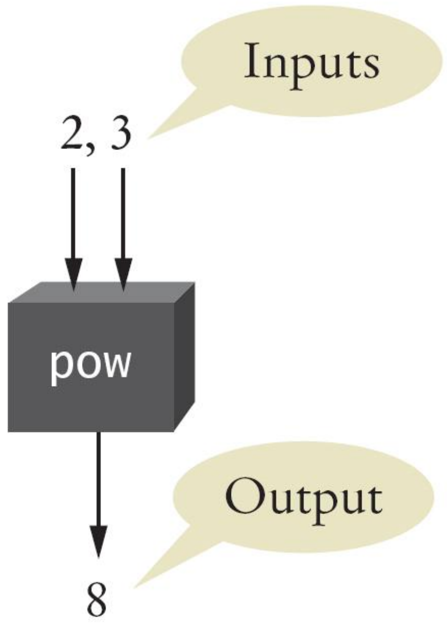
When another function calls the **`pow`** function, it provides "inputs", such as the values 2 and 3 in the call **`pow(2, 3)`.**

In order to avoid confusion with inputs that are provided by a human user, these values are called *parameter values*.

The "output" that the **`pow`** function computes is called the return value.

The Black Box Concept

- You can think of it as a “black box” where you can’t see what’s inside but you know what it does.



- How did the **`pow`** function do its job?
- You don’t need to know.
- You only need to know its *specification*.

Example: Trigonometric functions

- Program calls the built-in cos and sin functions.

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

#define  PI  3.14159265  // Symbolic constant

int main()
{
  printf("Cos(90) = %f\n",  cos(90 * PI / 180.0));
  printf("Cos(0)  = %f\n",  cos(0));
  printf("Sin(90) = %f\n",  sin(90 * PI / 180.0));
  printf("Sin(0)  = %f\n",  sin(0));
}
```

Formula for converting degree to radian :

$$Radian = \frac{Degree.\pi}{180}$$

DESIGNING FUNCTIONS

Designing Functions – Turn Repeated Code into Functions

When you write nearly identical code multiple times, you should probably introduce a function.

Designing Functions – Turn Repeated Code into Functions

Consider how similar the following code blocks are:

```
int hours;
do
{
   printf( "Enter a value between 0 and 23:");
   scanf("%d",&hours);
} while (hours < 0 || hours > 23);
```

```
int minutes;
do
{
   printf("Enter a value between 0 and 59: ");
   scanf("%d",&minutes);
} while (minutes < 0 || minutes > 59);
```

Designing Functions – Turn Repeated Code into Functions

The values for the high end of the range are different.

```
int hours;
do
{
   printf( "Enter a value between 0 and 23:");
   scanf("%d",&hours);
} while (hours < 0 || hours > 23);

int minutes;
do
{
   printf("Enter a value between 0 and 59: ");
   scanf("%d",&minutes);
} while (minutes < 0 || minutes > 59);
```

Designing Functions – Turn Repeated Code into Functions

The names of the variables are different.

```
int hours;
do
{
   printf( "Enter a value between 0 and 23:");
   scanf("%d",&hours);
} while (hours < 0 || hours > 23);

int minutes;
do
{
   printf("Enter a value between 0 and 59: ");
   scanf("%d",&minutes);
} while (minutes < 0 || minutes > 59);
```

Designing Functions – Turn Repeated Code into Functions

But there is common behavior.

```
int hours;
do
{
   printf( "Enter a value between 0 and 23:");
   scanf("%d",&hours);
} while (hours < _ || hours > _);
int minutes;
do
{
   printf("Enter a value between 0 and 59: ");
   scanf("%d",&minutes);
} while (minutes < _ || minutes > _);
```

Designing Functions – Turn Repeated Code into Functions

Move the *common behavior* into *one* function.

```
int read_int_up_to(int high)
{
    int input;
    do
    {
        printf("Enter a value between
                0 and %d:", high);
        scanf("%d", &input);
    } while (input < 0 || input > high);
    return input;
}
```

Designing Functions – Turn Repeated Code into Functions

Here we read one value, making sure it's within the range.

```
int read_int_up_to(int high)
{
    int input;
    do
    {
        printf("Enter a value between
             0 and %d", high: ");
        scanf("%d", &input);
    } while (input < 0 || input > high);
    return input;
}
```

Designing Functions – Turn Repeated Code into Functions

Then we can use this function as many times as we need:

```
int hours = read_int_up_to(23);
int minutes = read_int_up_to(59);
```

Note how the code has become much easier to understand.

And we are not rewriting code

– code reuse!

Designing Functions – Turn Repeated Code into Functions

Perhaps we can make this function even better:

```
int months = read_int_up_to(12);
```

Can we use this function to get a valid month?

Months are numbered starting at 1, not 0.

We can modify the code to take two parameters:
the end points of the valid range.

Designing Functions – Turn Repeated Code into Functions

Again, consider how similar the following statements are:

```
int month;
do
{
    printf("Enter a value between 1 and 12:");
    scanf("%d",&month);
} while (month < 1 || month > 12);

int minutes;
do
{
    printf("Enter a value between 0 and 59: ");
    scanf("%d",&minutes);
} while (minutes < 0 || minutes > 59);
```

Designing Functions – Turn Repeated Code into Functions

As before, the values for the range are different.

```
int month;
do
{
   printf("Enter a value between 1 and 12:");
   scanf("%d",&month);
} while (month < 1 || month > 12);

int minutes;
do
{
   printf("Enter a value between 0 and 59: ");
   scanf("%d",&minutes);
} while (minutes < 0 || minutes > 59);
```

Designing Functions – Turn Repeated Code into Functions

But the names of the variables are different.

```
int month;
do
{
   printf("Enter a value between 1 and 12:");
   scanf("%d",&month);
} while (month < 1 || month > 12);

int minutes;
do
{
   printf("Enter a value between 0 and 59: ");
   scanf("%d",&minutes);
} while (minutes < 0 || minutes > 59);
```

Designing Functions – Turn Repeated Code into Functions

Notice the common behavior?

```
int month;
do
{
   printf("Enter a value between 1 and 12:");
   scanf("%d",&month);
} while (month < _ || month > _);
int minutes;
do
{
   printf("Enter a value between 0 and 59: ");
   scanf("%d",&minutes);
} while (minutes < _ || minutes > _);
```

Designing Functions – Turn Repeated Code into Functions

Again, move the *common behavior* into *one* function.

```
int read_value_between(int low, int high)
{
    int input;
    do
    {
        printf("Enter a value between
             %d and %d: ", low, high);
        scanf("%d", &input);
    } while (input < low || input > high);
    return input;
}
```

Designing Functions – Turn Repeated Code into Functions

A different name would need to be used, of course because it does a different activity.

```
int read_value_between(int low, int high)
{
    int input;
    do
    {
        printf("Enter a value between
            %d and %d: ", low, high);
        scanf("%d", &input);
    } while (input < low || input > high);
    return input;
}
```

Designing Functions – Turn Repeated Code into Functions

We can use this function as many times as we need, passing in the end points of the valid range:

```
int months = read_value_between(1, 12);
int hours = read_value_between(1, 23);
int minutes = read_value_between(0, 59);
```

Note how the code has become even better.

And we are still not rewriting code

– code reuse!

Stepwise Refinement

- One of the most powerful strategies for problem solving is the process of *stepwise refinement.*

- To solve a difficult task, break it down into simpler tasks.

- Then keep breaking down the simpler tasks into even simpler ones, until you are left with tasks that you know how to solve.

Stepwise Refinement

Use the process of stepwise refinement
to decompose complex tasks into simpler ones.

Stepwise Refinement

Stepwise Refinement

We will break this problem into steps

(and for then those steps that can be further broken, we'll break them)

(and for then those steps that can be further broken, we'll break them)

(and for then those steps that can be further broken, we'll break them)

(and for then those steps that can be further broken, we'll break them)

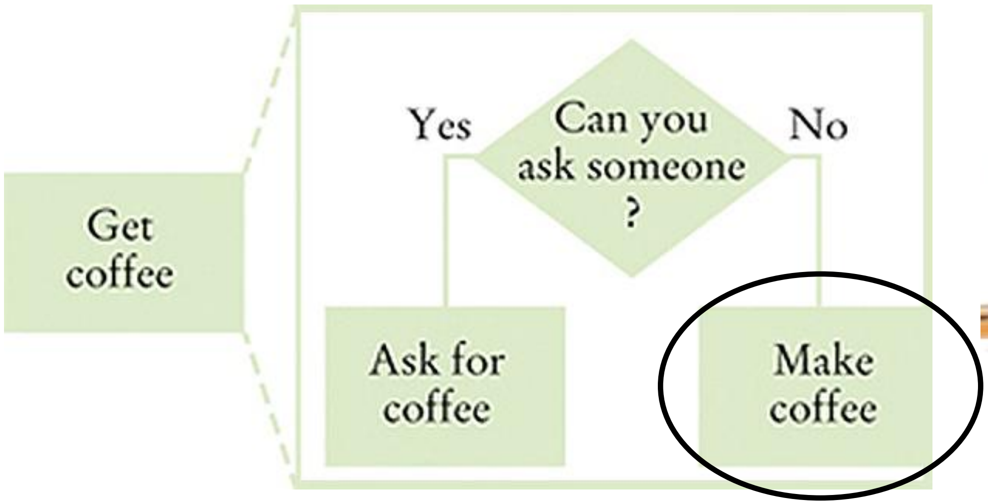
… and so on…

until the sub-problems are small enough to be just a few steps

Stepwise Refinement

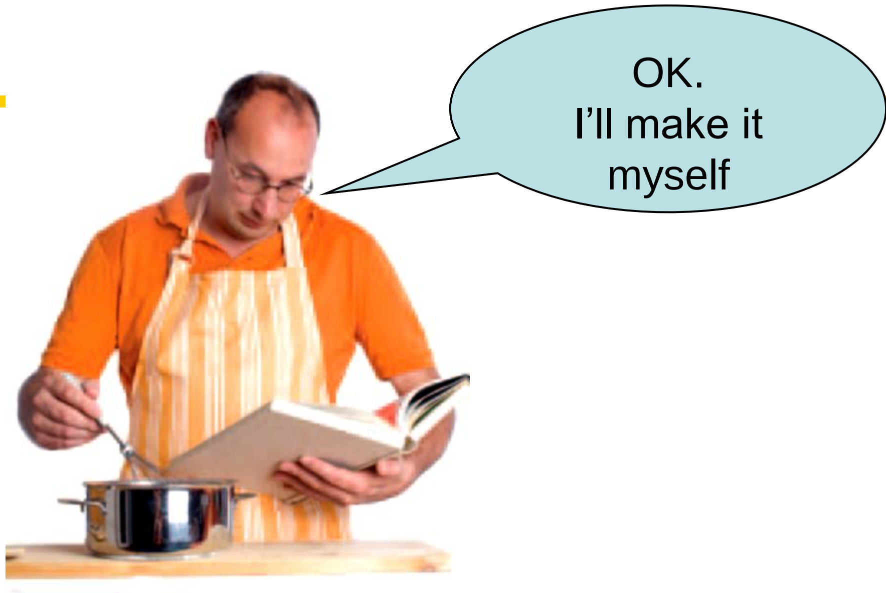
This is the whole problem: this is like **`main`**.

Stepwise Refinement



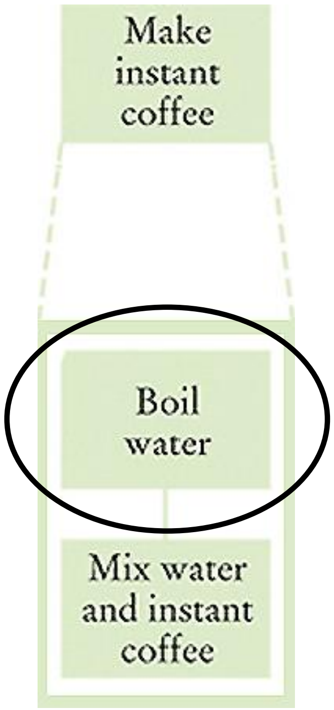
The whole problem can be broken into:
if we can ask someone to give us coffee, we are done
but if not, we can make coffee (which we will
have to break into its parts)

Stepwise Refinement



The make coffee sub-problem can be broken into:
if we have instant coffee, we can make that
but if not, we can brew coffee
(maybe these will have parts)

Stepwise Refinement

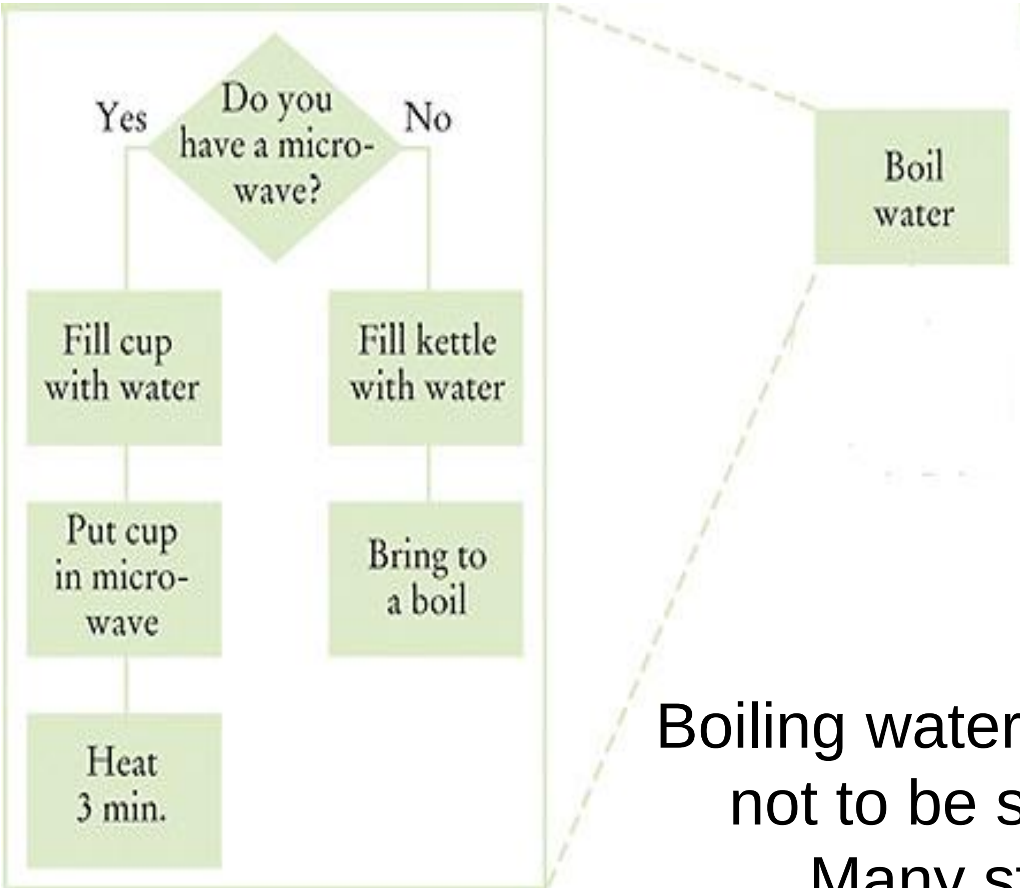


Making instant coffee breaks into:

1. Boil Water
2. Mix (stir if you wish)

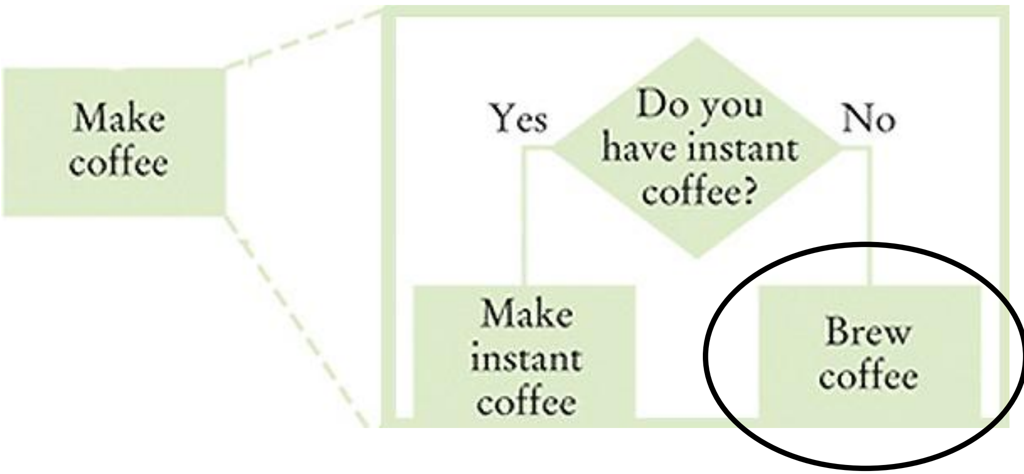
(Do these have sub-problems?)

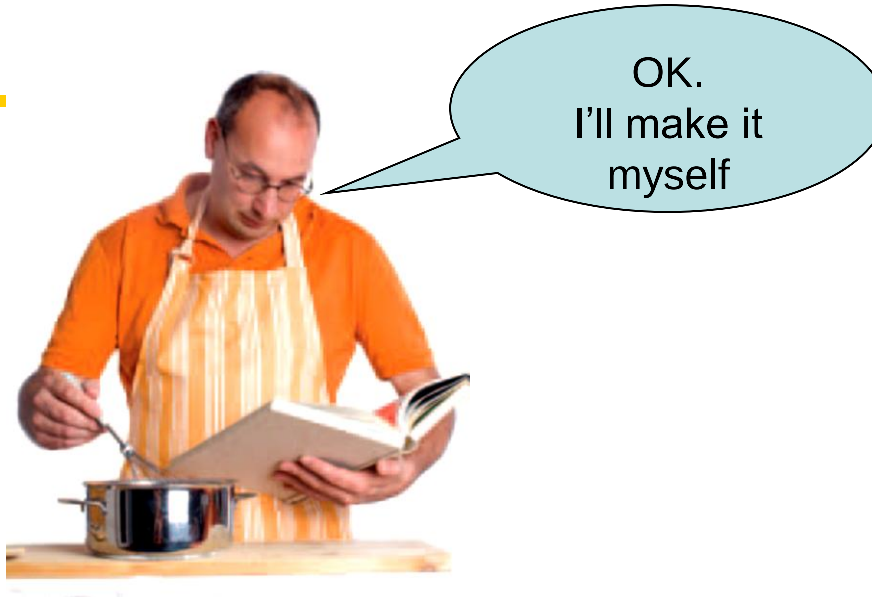
Stepwise Refinement



Boiling water appears
not to be so easy.
Many steps,
but none have sub-steps.

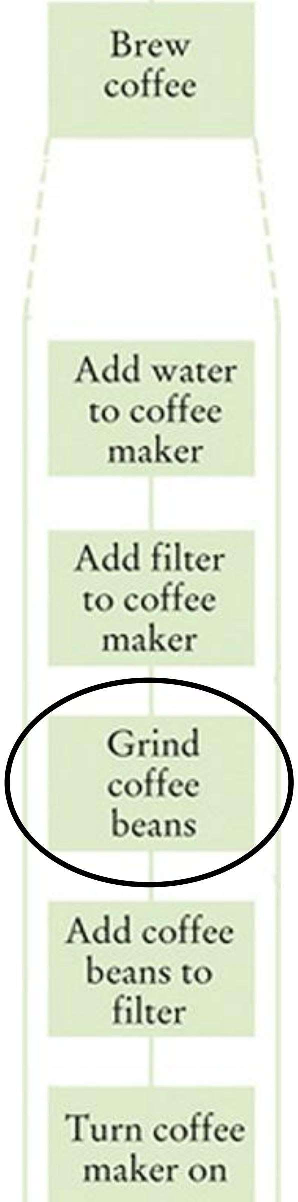
Stepwise Refinement

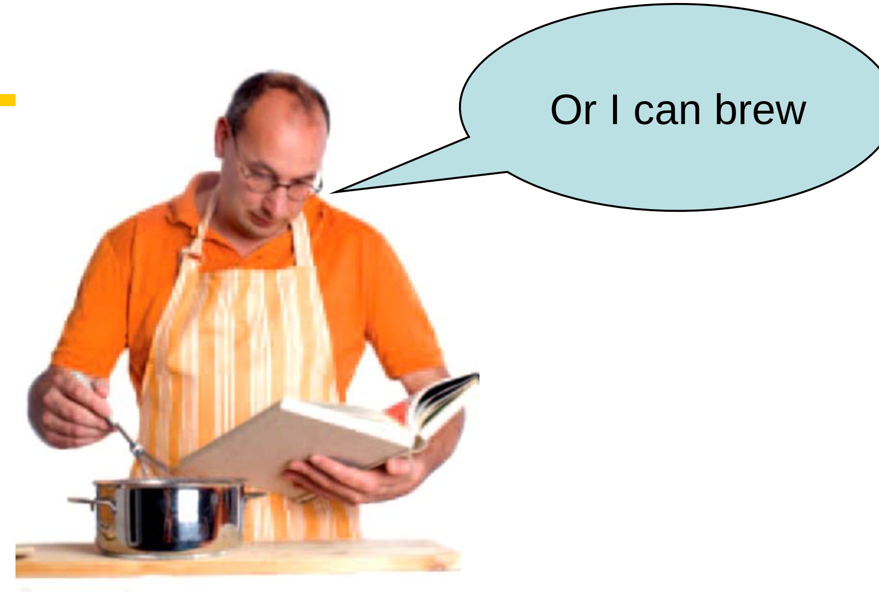




Going back to the branch between instant or brew, we need to think about brewing. Can we break that into parts?

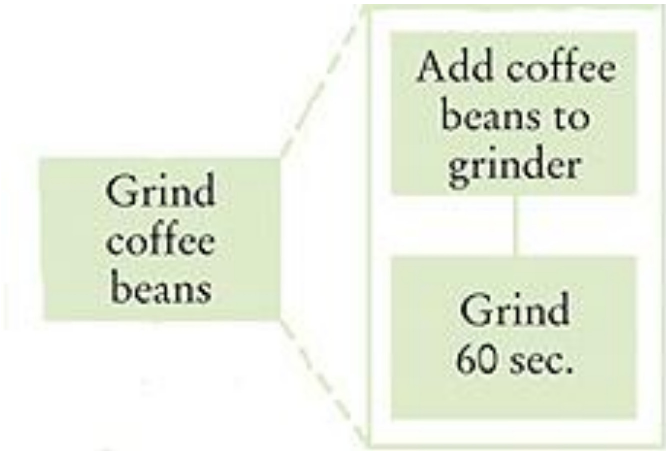
Stepwise Refinement

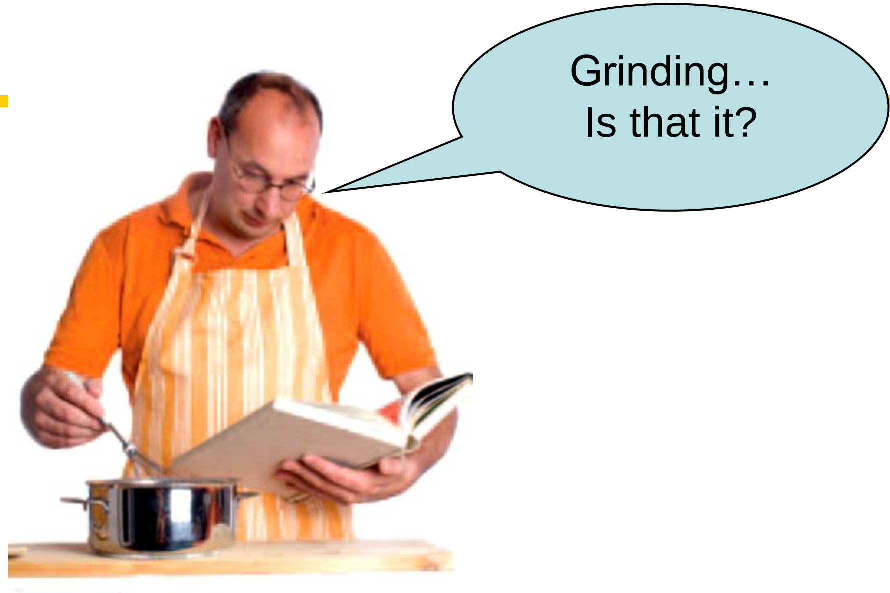




Brewing coffee has several steps.
Do any need more breakdown
(grind coffee beans)?

Stepwise Refinement

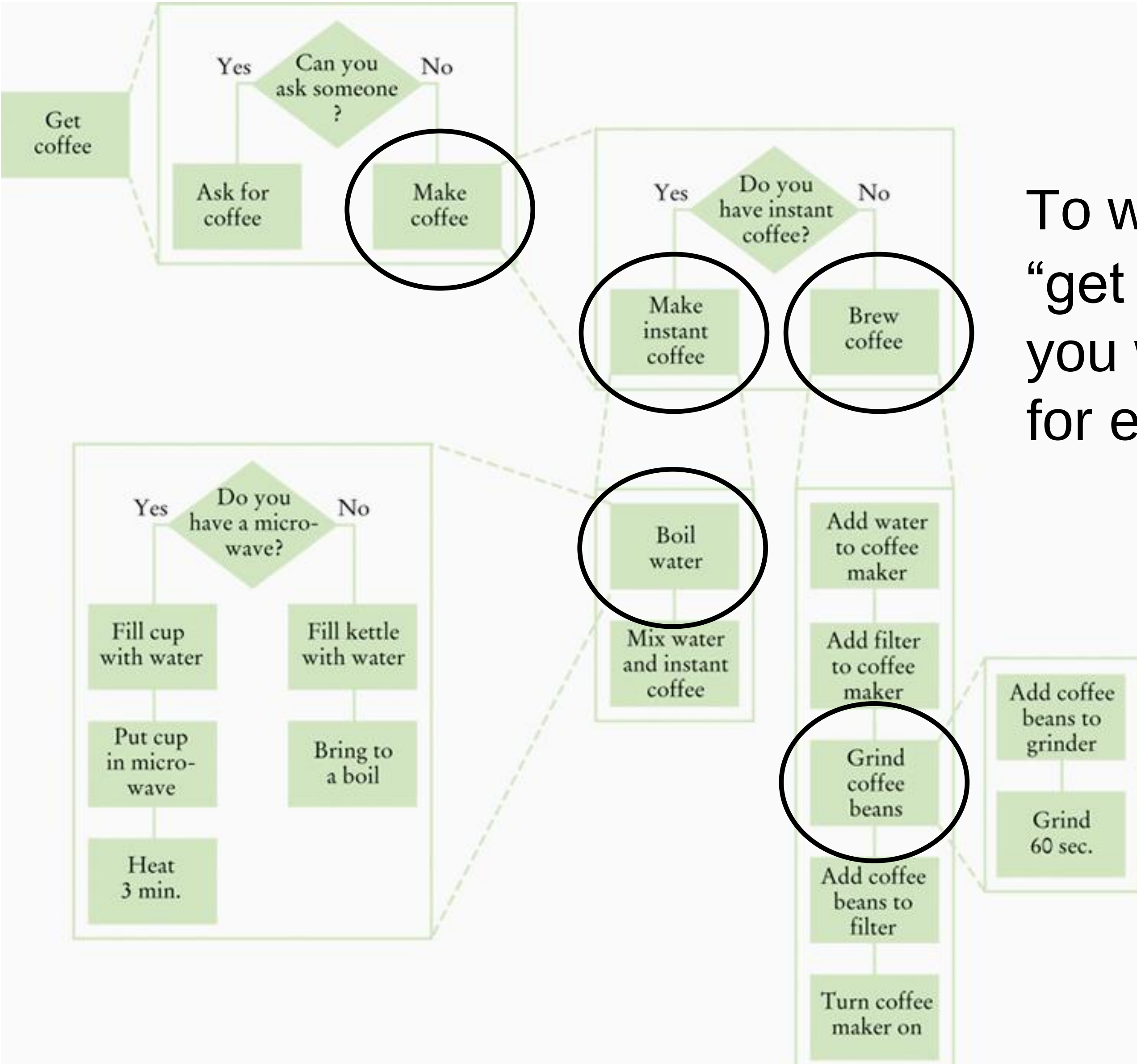




Grinding is a two step process with no sub-sub-steps.

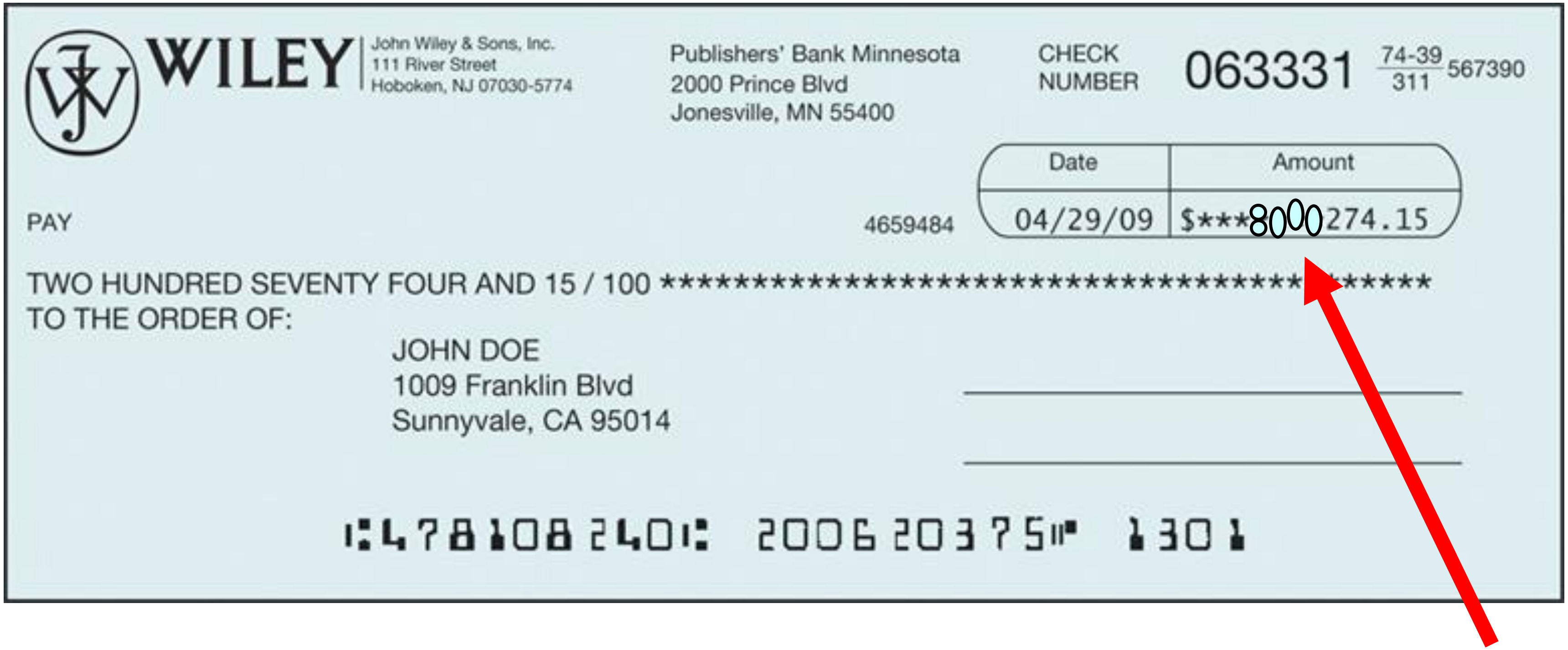
Stepwise Refinement

Stepwise Refinement – The Complete Process Shown



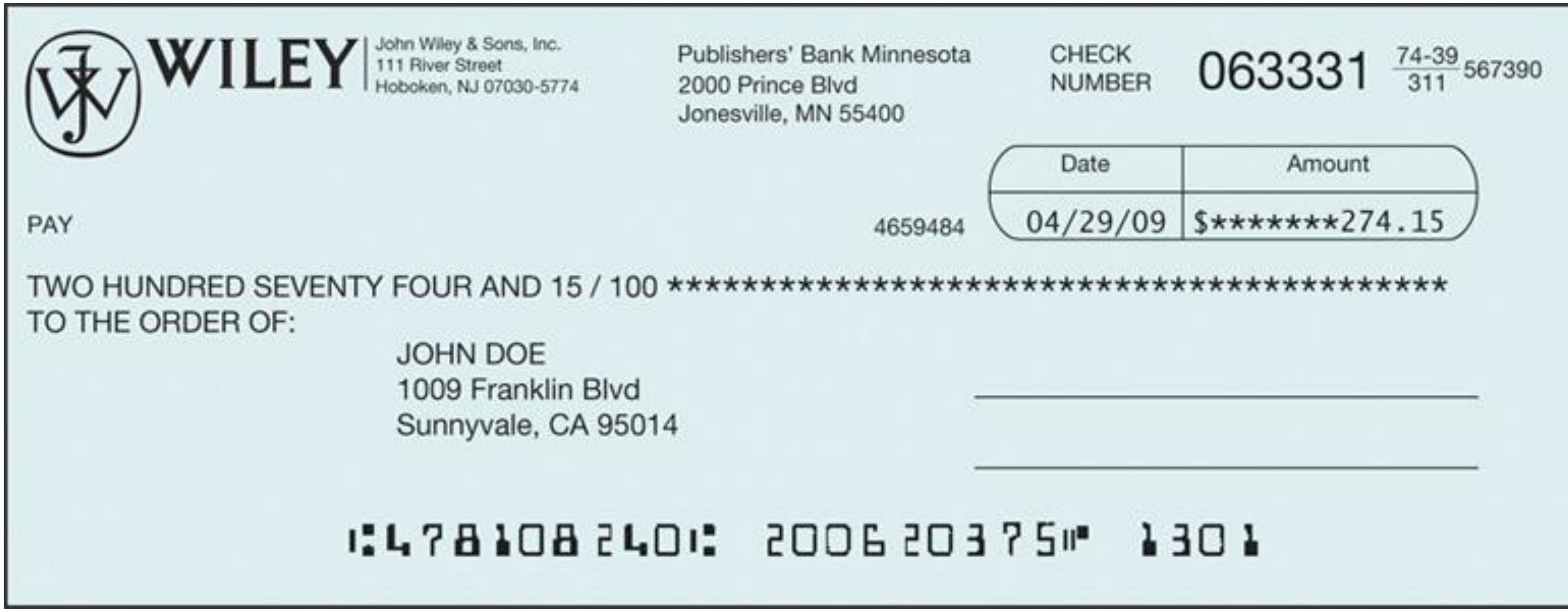
To write the "get coffee" program, you would write functions for each sub-problem.

Stepwise Refinement



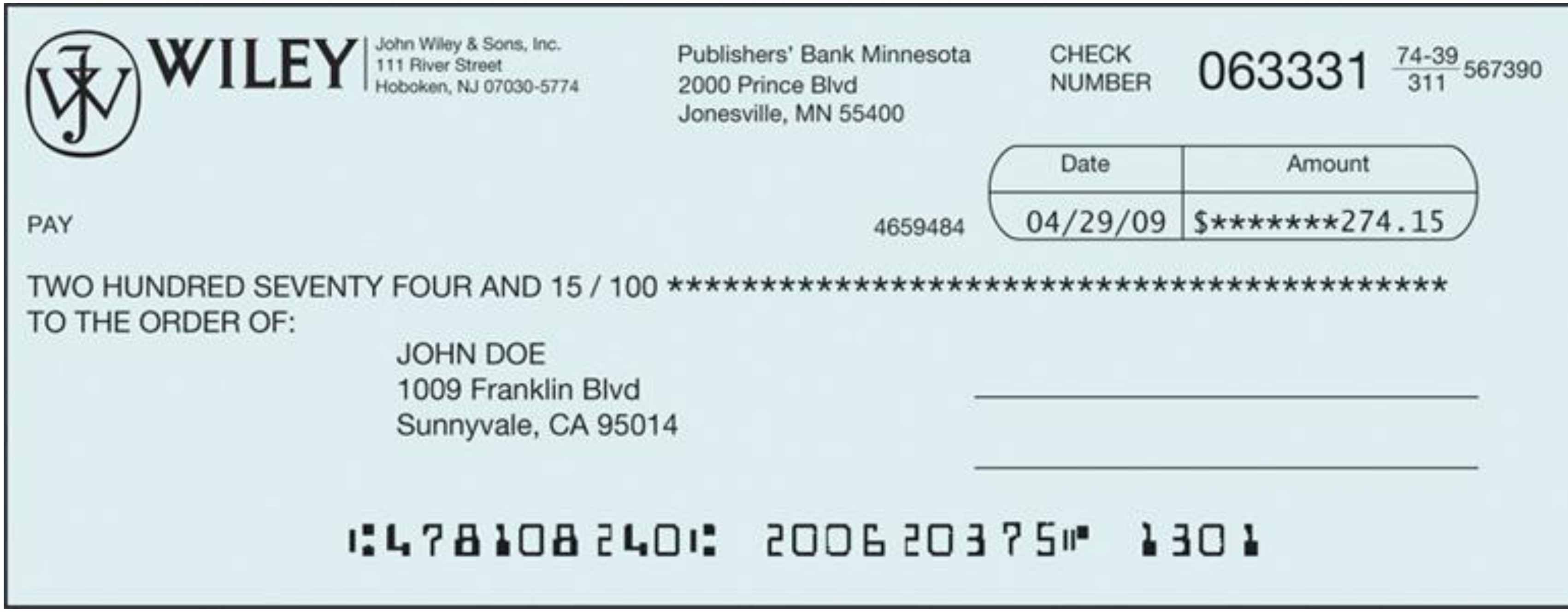
When writing a check by hand the recipient might be tempted to add a few digits in front of the amount.

Stepwise Refinement



To discourage this, when printing a check,
it is customary to write the check amount both
as a number (“$274.15”) and as a text string
(“two hundred seventy four dollars and 15 cents”)

Stepwise Refinement



We will write a program to take an amount and produce the text.

And practice stepwise refinement.

Stepwise Refinement

Sometimes we reduce the problem a bit when we start:
we will only deal with amounts less than $1,000.

Stepwise Refinement

Of course we will write a function to solve this sub-problem.

```
/**
Turns a number into its English name.
@param number a positive integer < 1,000
@return no return - prints the number in text inside the
   function (e.g., "two hundred seventy four")
*/
void int_name(int number)
```

Notice that we started by writing only the comment and the first line of the function.

Also notice that the constraint of < $1,000 is announced in the comment.

Stepwise Refinement

Before starting to write this function, we need to have a plan.

Are there special considerations?

Are there subparts?

Stepwise Refinement

If the number is between 1 and 9,
we need to compute "one" … "nine".

In fact, we need the same computation
again for the hundreds (“*two*” hundred).

Any time you need to do something more than once,
it is a good idea to turn that into a function:

Stepwise Refinement

```
/**
    Turns a digit into its English name.
    @param digit an integer between 1 and 9
    @return no return – prints the name of digit
   ("one" ... "nine")
*/
void digit_name(int digit)
```

Stepwise Refinement

Numbers between 10 and 19 are special cases.

Let’s have a separate function **`teen_name`** that converts them into strings "eleven", "twelve", "thirteen", and so on:

```
/**
Turns a number between 10 and 19 into its English
   name.
@param number an integer between 10 and 19
@return no return - prints the name of the number
   (“ten” ... “nineteen”)
*/
void teen_name(int number)
```

Stepwise Refinement

Next, suppose that the number is between 20 and 99. Then we show the tens as "twenty", "thirty", …, "ninety". For simplicity and consistency, put that computation into a separate function:

```
/**
Gives the name of the tens part of a number between 20 and 99.
@param number an integer between 20 and 99
@return no return - prints the name of the tens part of the number
  ("twenty"..."ninety")
*/
void tens_name(int number))
```

Stepwise Refinement

- Now suppose the number is at least 20 and at most 99.
  - If the number is evenly divisible by 10, we use **`tens_name`**, and we are done.
  - Otherwise, we print the tens with **`tens_name`** and the ones with **`digit_name`**.

- If the number is between 100 and 999,
  - then we show a digit, the word "hundred", and the remainder as described previously.

Stepwise Refinement – The Pseudocode

part = number (The part that still needs to be converted)

If part >= 100

    Print name of hundreds in part + " hundred“.

    Remove hundreds from part.

digit_name(…)

If part >= 20

    Print tens_name(part).

    Remove tens from part.

Else if part >= 10

    Print teen_name(part)

    part = 0

If (part > 0)

    Print digit_name(part).

Stepwise Refinement – The Pseudocode

- This pseudocode has a number of important improvements over the descriptions and comments.
  - It shows how to arrange *the order of the tests*, starting with the comparisons against the larger numbers
  - It shows how the smaller number is subsequently processed in further **`if`** statements.

Stepwise Refinement – The Pseudocode

- On the other hand, this pseudocode is vague about:
  - The actual conversion of the pieces, just referring to "name of hundreds" and the like.
  - Spaces—it would produce strings with no spaces: "**`twohundredseventyfour`**"

Stepwise Refinement – The Pseudocode

Compared to the complexity of the main problem,
one would hope that spaces are a minor issue.

It is best not to muddy the pseudocode with minor details.

Stepwise Refinement – Pseudocode to C

Now for the real code.
The last three cases are easy so let's start with them:

```
if (part >= 20)
{
    tens_name(part);
    part = part % 10;
}
else if (part >= 10)
{
    teens_name(part);
    part = 0;
}

if (part > 0)
{
    digit_name(part);
}
```

If part >= 20

Print tens_name(part).

Remove tens from part.

Else if part >= 10

Print teen_name(part)

part = 0

If (part > 0)

Print digit_name(part).

Stepwise Refinement – Pseudocode to C

Finally, the case of numbers between 100 and 999.
Because **`part < 1000`**, **`part / 100`** is a single digit, and we obtain its name by calling **`digit_name`**.
Then we add the “hundred” suffix:

```
if (part >= 100)
{
    digit_name(part / 100);
    printf(" hundred");
    part = part % 100;
}
```

Stepwise Refinement – Pseudocode to C

Now for the complete program.

The Complete Program

```
#include <stdio.h>

/**
   Prints a digit into its English name.
   @param digit an integer between 1 and 9
   @return no return- print the name of digit ("one" ... "nine")
*/
void digit_name (int digit)
{
  if (digit == 1)     printf ("one");
  else if (digit == 2)     printf ("two");
  else if (digit == 3)     printf ("three");
  else if (digit == 4)     printf ("four");
  else if (digit == 5)     printf ("five");
  else if (digit == 6)     printf ("six");
  else if (digit == 7)     printf ("seven");
  else if (digit == 8)     printf ("eight");
  else if (digit == 9)     printf ("nine");
  else {}
}
```

The Complete Program

```
/**
    Prints a number between 10 and 19 into its English name.
    @param number an integer between 10 and 19
    @return no return - print the name of the given number ("ten" ...
    "nineteen")
*/
void teens_name (int number)
{
  if (number == 10)     printf ("ten");
  else if (number == 11)    printf ("eleven");
  else if (number == 12)    printf ("twelve");
  else if (number == 13)    printf ("thirteen");
  else if (number == 14)    printf ("fourteen");
  else if (number == 15)    printf ("fifteen");
  else if (number == 16)    printf ("sixteen");
  else if (number == 17)    printf ("seventeen");
  else if (number == 18)    printf ("eighteen");
  else if (number == 19)    printf ("nineteen");
  else {}
}
```

The Complete Program

```
/**
   Gives the name of the tens part of a number between 20 and 99.
   @param number an integer between 20 and 99
   @return no return - prints the name of the tens part of the number
   ("twenty" ... "ninety")
*/
void tens_name (int number)
{
  if (number >= 90)            printf ("ninety");
  else if (number >= 80)       printf ("eighty");
  else if (number >= 70)       printf ("seventy");
  else if (number >= 60)       printf ("sixty");
  else if (number >= 50)       printf ("fifty");
  else if (number >= 40)       printf ("forty");
  else if (number >= 30)       printf ("thirty");
  else if(number >= 20)        printf ("twenty");
  else {}
}
```

The Complete Program

```
/**
   Turns a number into its English name.
   @param number a positive integer < 1,000
   @return no return - prints the name of the number (e.g. "two
   hundred seventy four")
*/
void int_name(int number)
{
   int part = number; // The part that still needs to be converted

   if (part >= 100)
   {
      digit_name (part / 100);
      printf (" hundred");
      part = part % 100;
   }
   if (part >= 20)
   {
      printf (" ");
      tens_name (part);
      part = part % 10;
   }
```

The Complete Program

```
    else if (part >= 10)
    {
       printf (" ");
       teens_name (part);
       part = 0;
    }

    if (part > 0)
    {
       printf (" ");
       digit_name (part);
    }
}

int main()
{
  int input;
  printf("Please enter a positive integer: ");
  scanf("%d", &input);
  int_name(input);
  return 0;
}
```

Good Design – Keep Functions Short

- There is a certain cost for writing a function:
  - You need to design, code, and test the function.
  - The function needs to be documented.
  - You need to spend some effort to make the function *reusable* rather than tied to a specific context.

Good Design – Keep Functions Short

- And you should keep your functions short.
- As a rule of thumb, a function that is so long that its will not fit on a single screen in your development environment should probably be broken up.
- Break the code into other functions

Tracing Functions

When you design a complex set of functions, it is a good idea to carry out a manual walkthrough before entrusting your program to the computer.

This process is called *tracing* your code.

You should trace each of your functions separately.

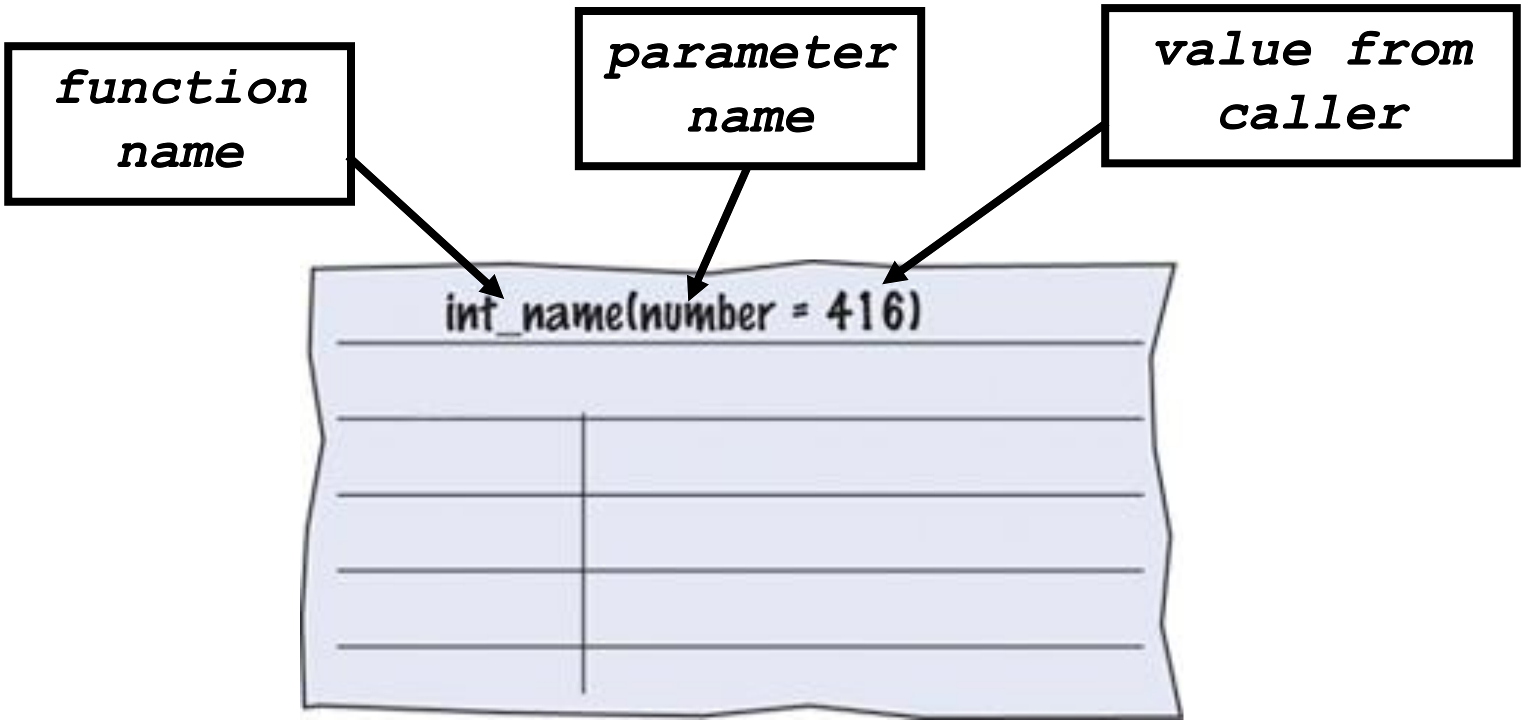
Tracing Functions

To demonstrate, we will trace the **`int_name`** function when 416 is passed in.

Tracing Functions

Here is the call: **`... int_name(416) ...`**

Take an index card (or use the back of an envelope) and write the name of the function and the names and values of the parameter variables, like this:

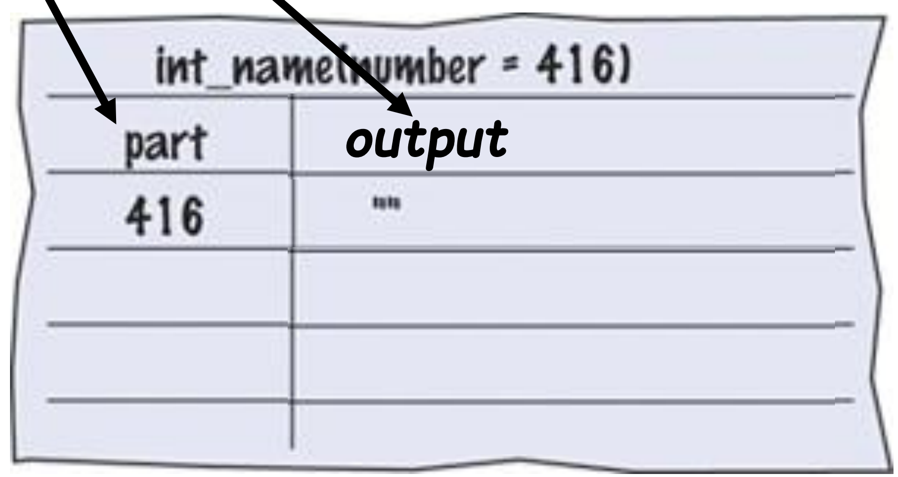


Tracing Functions

Then write the names and values of the function variables.

```
void int_name(int number)
{
   int part = number; // The part that still needs
                      // to be converted
   // Printed value, initially ""
```

Write them in a table, since you will update them as you walk through the code:



Tracing Functions

The test (**`part >= 100)`** is **`true`** so the code is executed.

```
if (part >= 100)
    {
        digit_name(part / 100)
        printf(" hundred");

        part = part % 100;
    }
```

Tracing Functions

`part / 100` is 4

```
if (part >= 100)
    {
        digit_name(part / 100)
        printf(" hundred");
        part = part % 100;
    }
```

so **`digit_name(4)`** is easily seen to be "four".

Tracing Functions

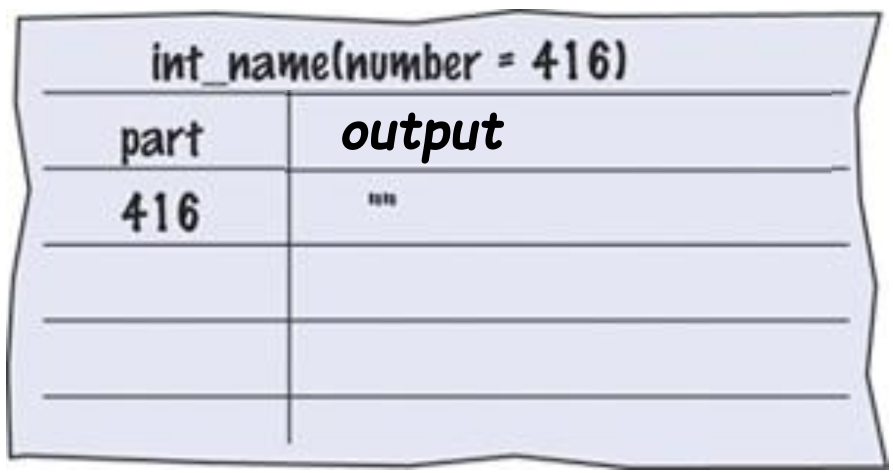
```
if (part >= 100)
   {
      digit_name(part / 100)
      printf(" hundred");
      part = part % 100;
   }
```

`part % 100` is 16.

Tracing Functions

Output has changed to "four hundred",

`part` has changed to **`part % 100`**, or 16.



Tracing Functions

Output has changed to "four hundred",

`part` has changed to **`part % 100`**, or 16.

Cross out the old values and write the new ones.

int_name(number = 416)

part	output
~~416~~	~~""~~
16	"four hundred"

Tracing Functions

Let’s continue…

Here is the status of the parameters and variables now:

int_name(number = 416)

part	output
~~416~~	~~""~~
16	"four hundred"

Tracing Functions

The test **`(part >= 20)`** is **`false`** but
the test **`(part >= 10)`** is **`true`** so that code is executed.

```
if (part >= 20)…
else if (part >= 10) {
    printf (" ");
   teens_name (part);
   part = 0;
}
```

`teens_name(16)` is “sixteen”, **`part`** is set to 0, so do this:

int_name(number = 416)

part	*output*
~~416~~	~~""~~
~~16~~	~~"four hundred"~~
0	"four hundred sixteen"

Tracing Functions

Why is **`part`** set to 0?

```
if (part >= 20)…
else if (part >= 10) {
   printf (" ");
   teens_name (part);
   part = 0;
}

if (part > 0)
{
    printf (" ");
   digit_name (part);
}
```

After the **`if-else`** statement ends, **`name`** is complete.

The test in the following **`if`** statement needs to be "fixed" so that part of the code will not be executed

- nothing should be added to **`name`**.

Stubs

- When writing a larger program, it is not always feasible to implement and test all functions at once.
- You often need to test a function that calls another, but the other function hasn’t yet been implemented.

Stubs

- You can temporarily replace the body of function yet to be implemented with a *stub*.

- A stub is a function that returns a simple value that is sufficient for testing another function.
- It might also have something written on the screen to help you see the order of execution.
- Or, do both of these things.

Stubs

Here are examples of stub functions.

```
/**
   Prints a digit as its English name.
   @param digit an integer between 1 and 9
   @return no return – prints the name of digit (“one” ...
   “nine”)
*/
void digit_name(int digit)
{
   printf("mumble");
}

/**
 Gives the name of the tens part of a number between 20 and 99.
 @param number an integer between 20 and 99
 @return no return – Prints the tens name of the number
   (“twenty” ... “ninety”)
*/
void tens_name(int number)
{
   printf("mumblety");
}
```

Stubs

If you combine these stubs with the completely written **`int_name`** function and run the program testing with the value 274, this will the result:

```
Please enter a positive integer: 274
mumble hundred mumblety mumble
```

which *everyone* knows indicates that the basic logic of the **`int_name`** function is working correctly.

(OK, only <u>you</u> know, but that is the important thing with stubs)

Now that you have tested **`int_name`**, you would “unstubify” another stub function, then another...