Chapter Five: Functions

Slides by Evan Gallagher

Chapter Goals

- To be able to implement functions
- To become familiar with the concept of parameter passing
- To appreciate the importance of function comments
- To develop strategies for decomposing complex tasks into simpler ones
- To be able to determine the scope of a variable
- To recognize when to use value and reference parameters

Call a Function to Get Something Done

If it's chilly in here… do something about it!

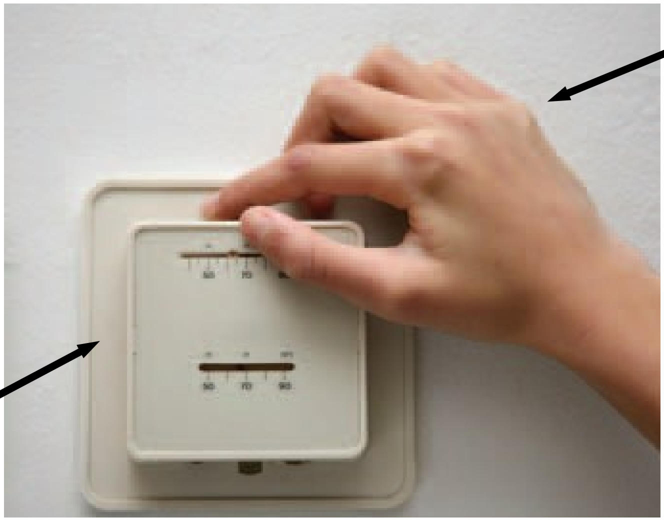
What Is a Function? Why Functions?

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.

Calling a Function

A programmer *calls* a function
to have its instructions executed.



the caller
(getting things done)

the function
(has the *modify temperature* instructions)

Calling a Function

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

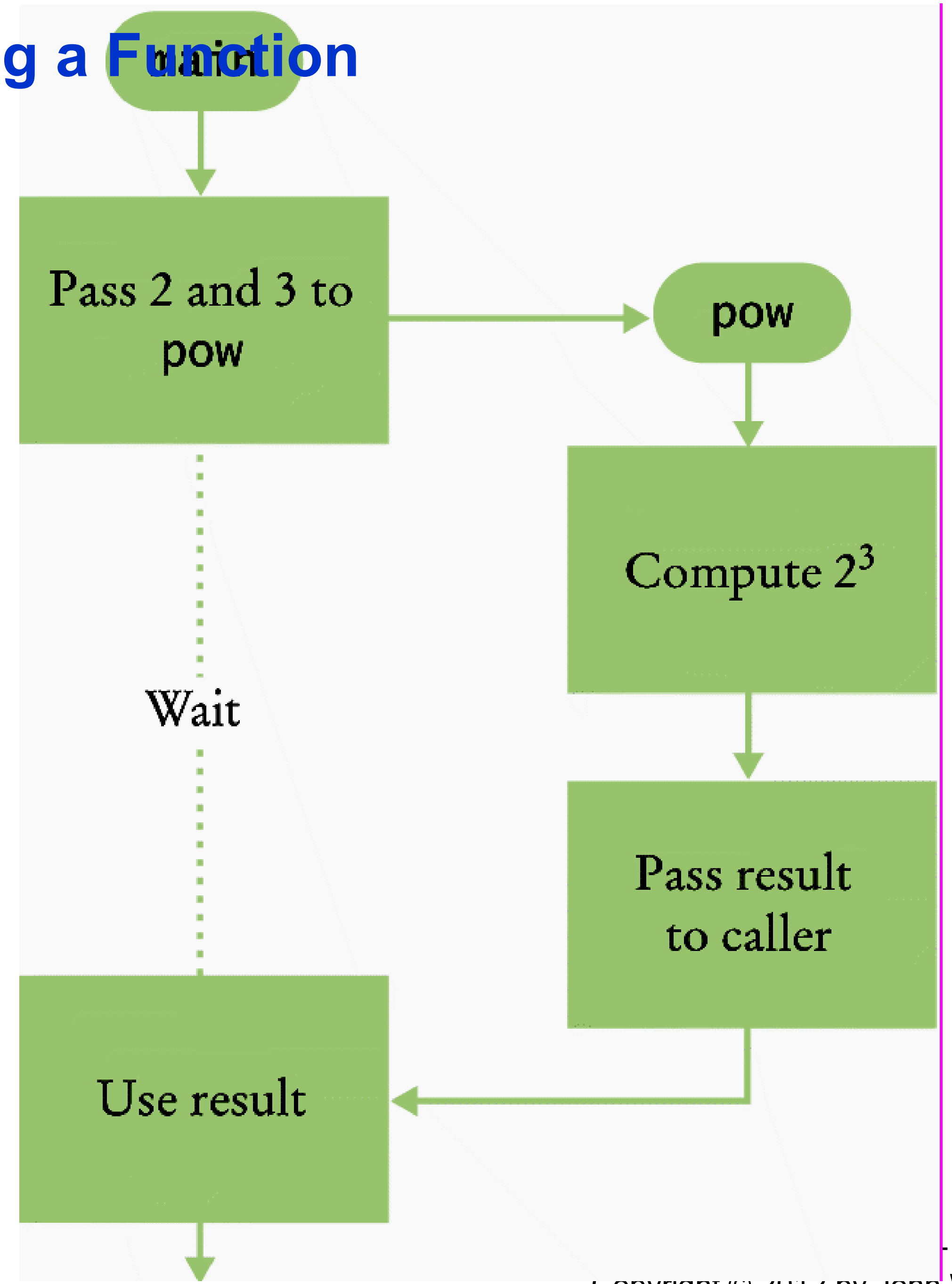
By using the expression: `pow(2, 3)`
main *calls* the **pow** function, asking it to compute 2^3.

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



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.

An Output Statement Does Not Return a Value

output ≠ 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 ≠ 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!)

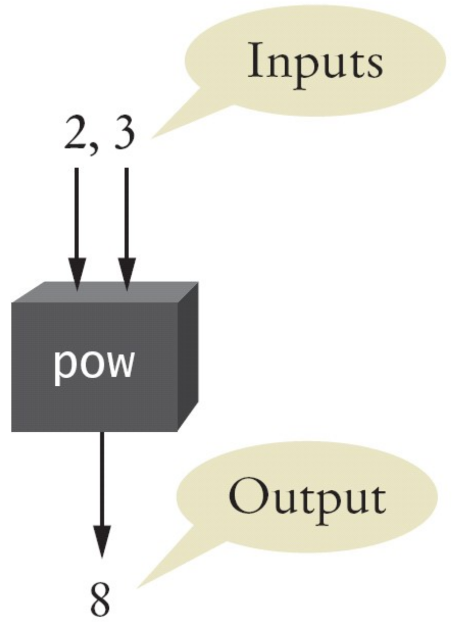
```
int main()
{
   double z = pow(2, 3);

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

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

The Black Box Concept

Do you care what's *inside* a thermostat?

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*.

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?)

<u>`cube_volume`</u>

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.

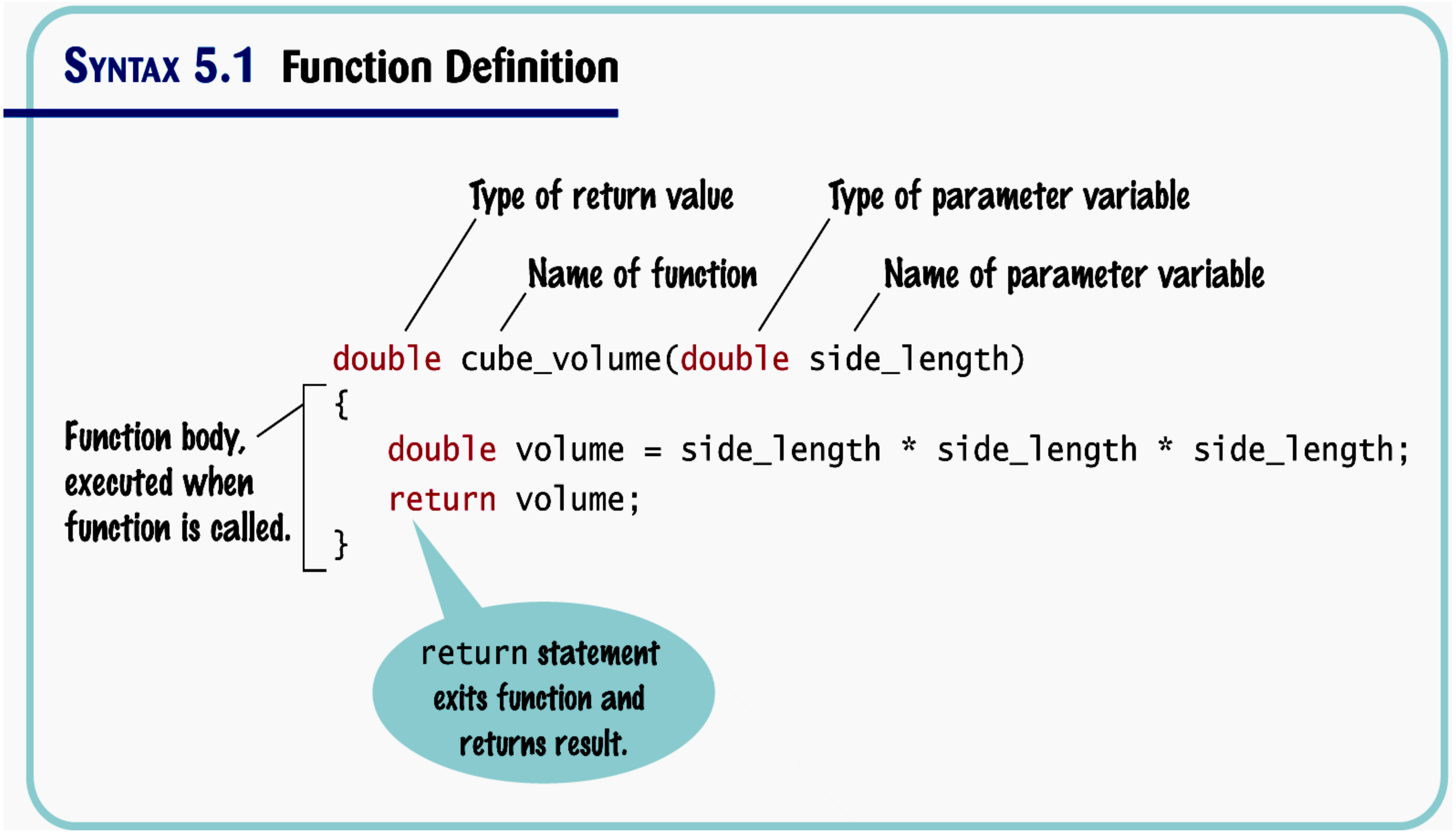
A Complete Function

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

A Complete Testing Program

```
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 EXIT_SUCCESS;
}
```

Implementing Functions



Syntax 5.1 Function Definition

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

Type of return value
Name of function
Type of parameter variable
Name of parameter variable
Function body, executed when function is called.
return statement exits function and returns result.

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 used in some 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***

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

Calling Functions

Consider the order of activities when a function is called.

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*.)

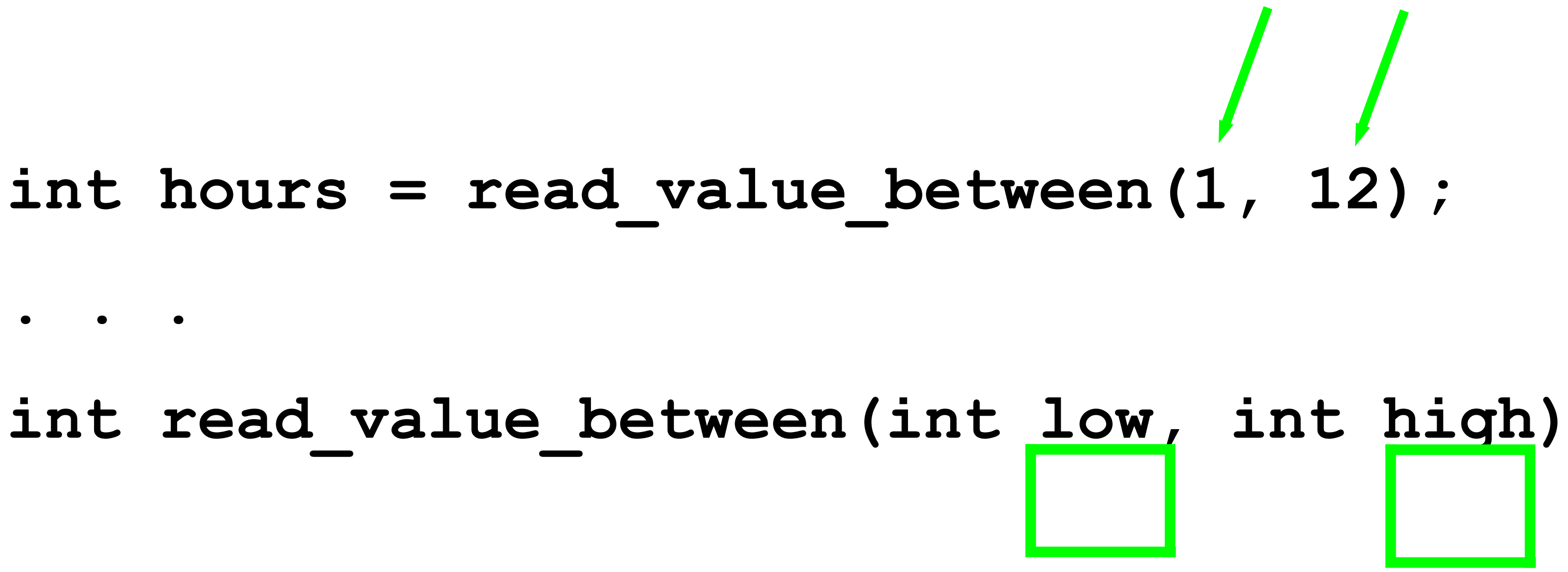
```
int hours = read_value_between(1, 12);

. . .
```

Parameter Passing

When a function is called,

a *parameter variable* is created for each value passed in.

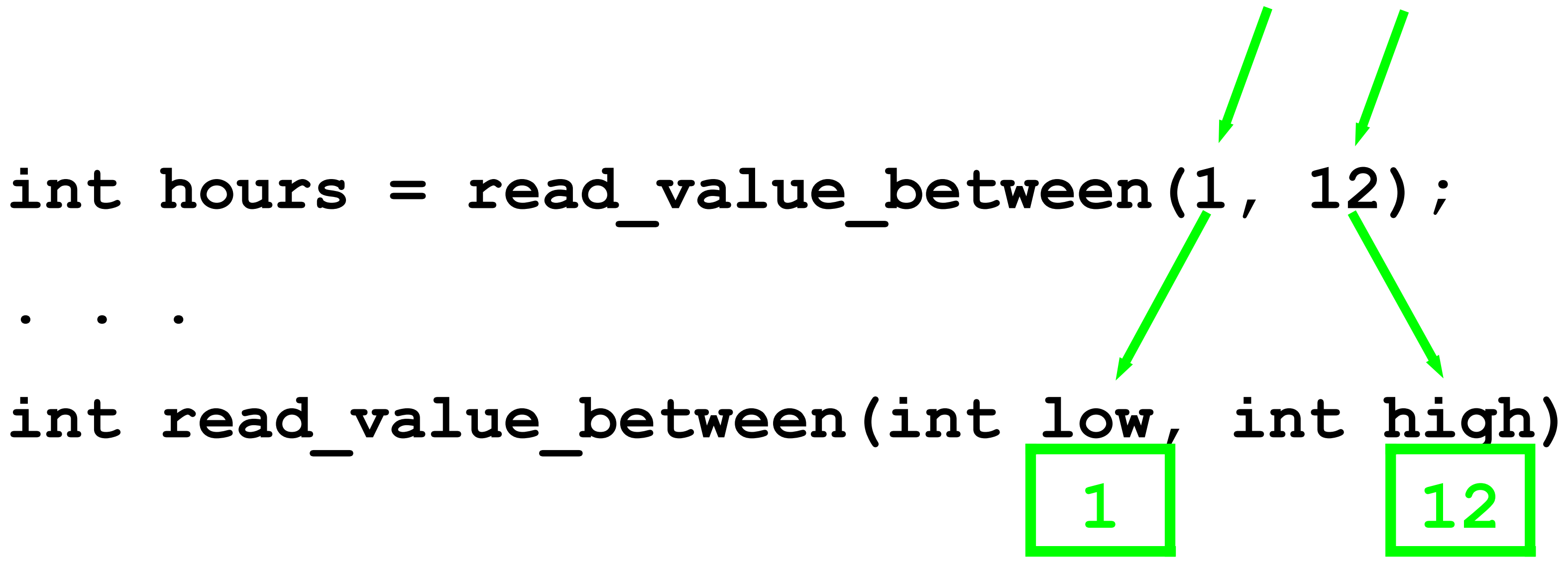
(Another commonly used term is *formal parameter*.)



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

. . .

int read_value_between(int low, int high)
```

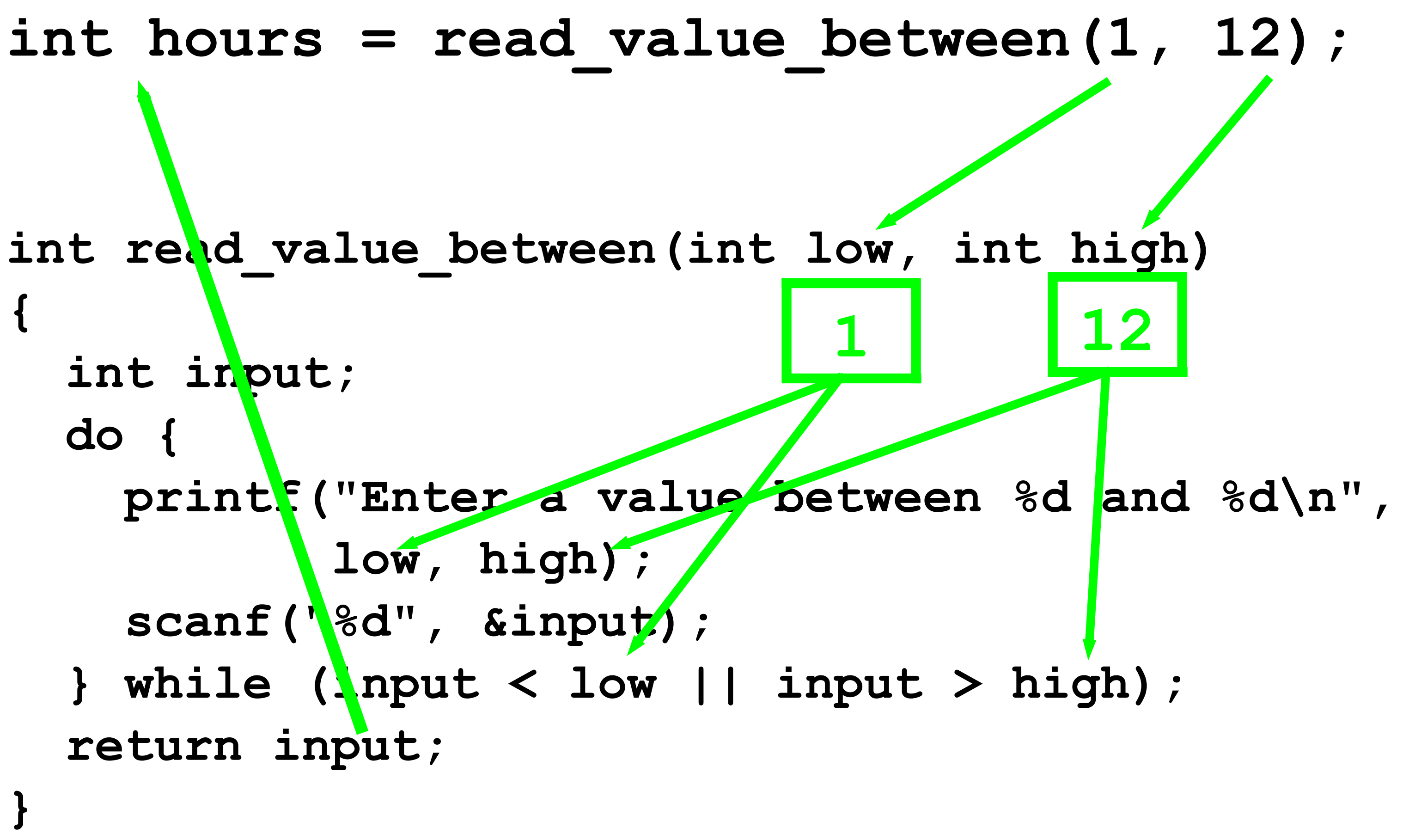
Parameter Passing

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)
```

Parameter Passing



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

```
int read_value_between(int low, int high)
{
   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;
}
```

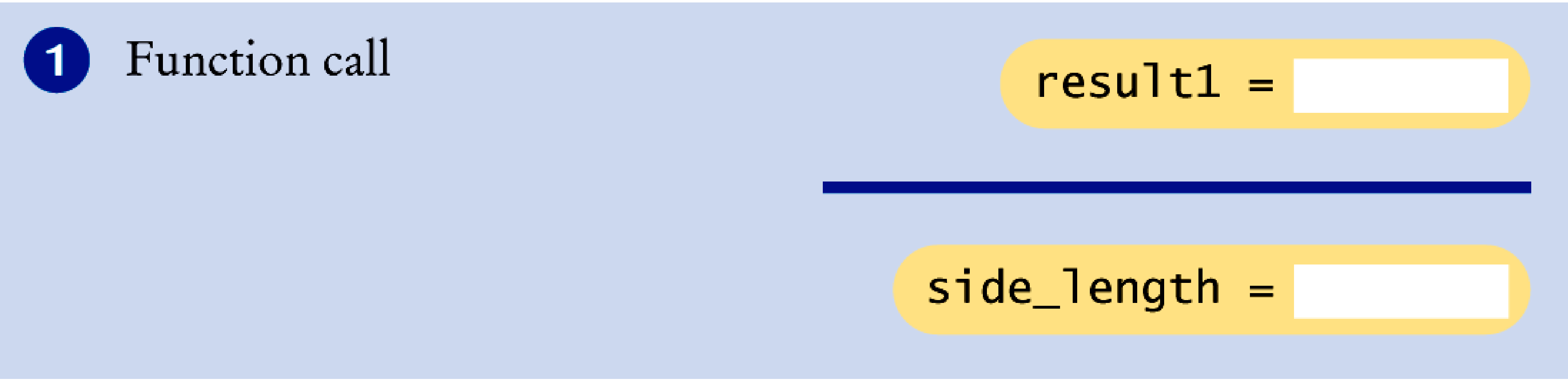
We'll keep up with their variables and parameters:

```
result1
side_length
volume
```

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.

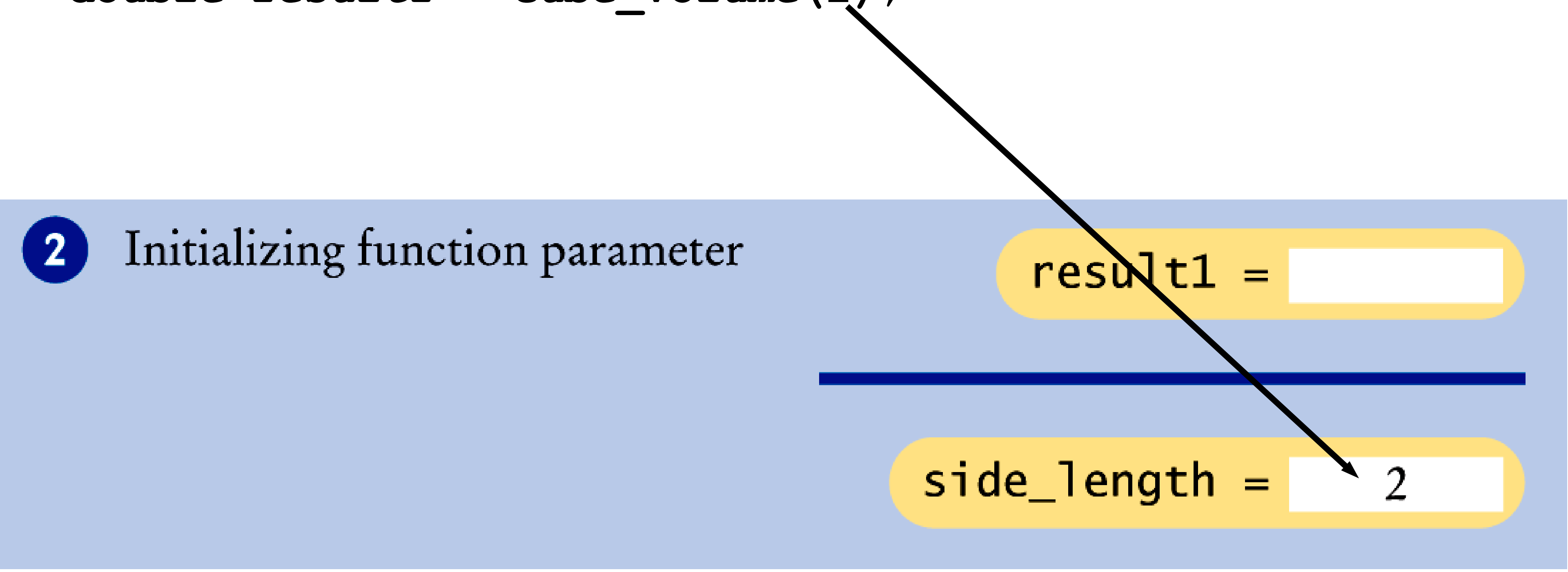
```
double result1 = cube_volume(2);
```



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.

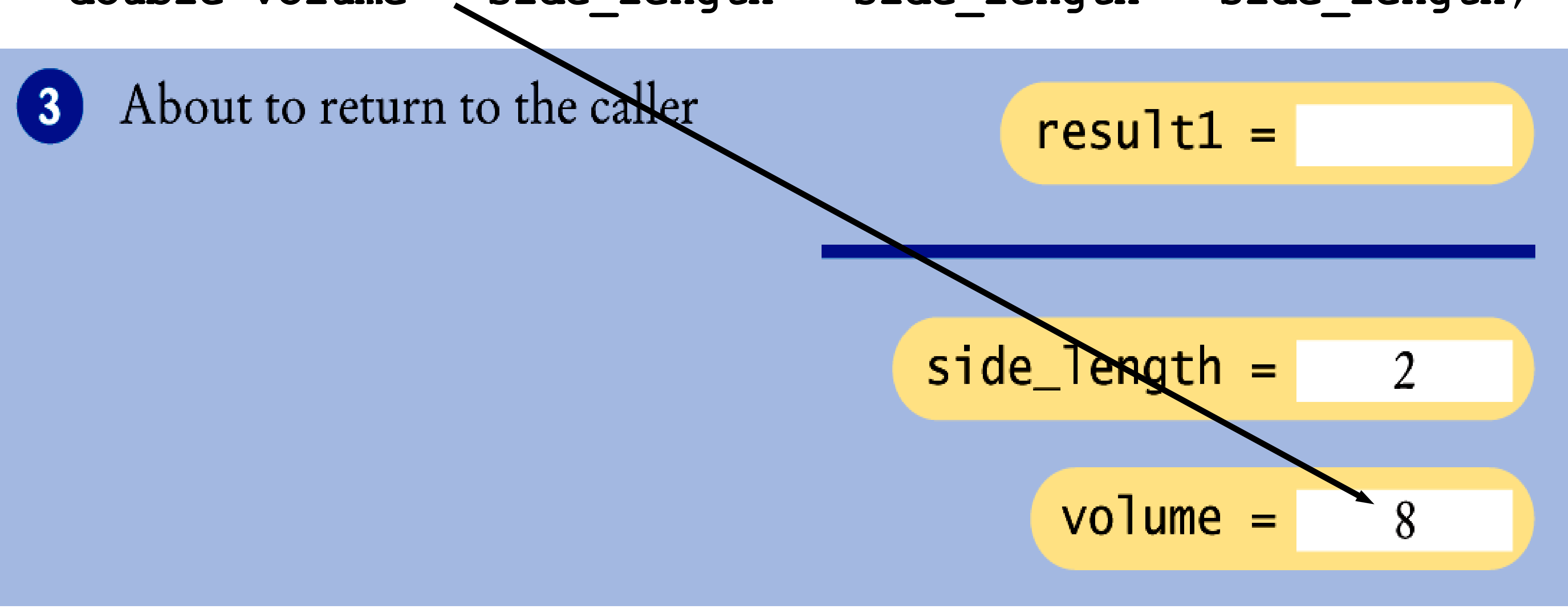
```
double result1 = cube_volume(2);
```



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`**.

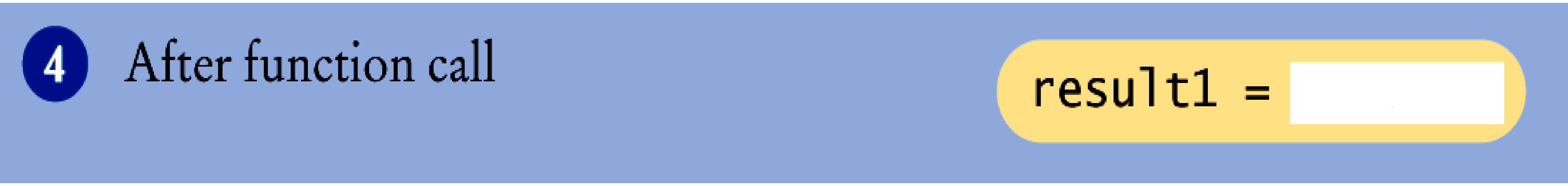
```
[inside the function]
double volume = side_length * side_length * side_length;
```



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);
```



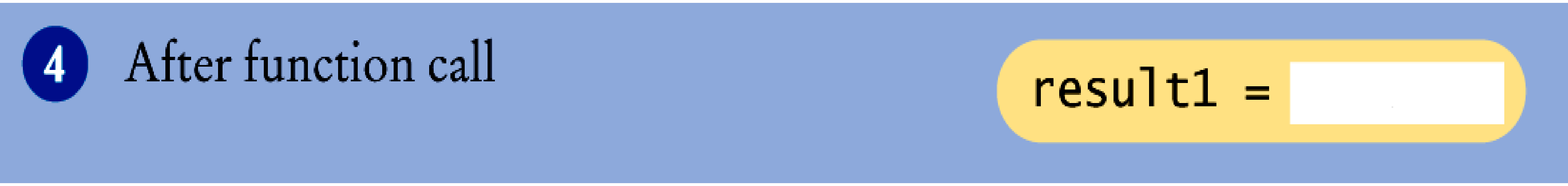
The function executed: **`return volume;`**
which gives the caller the value **`8`**

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

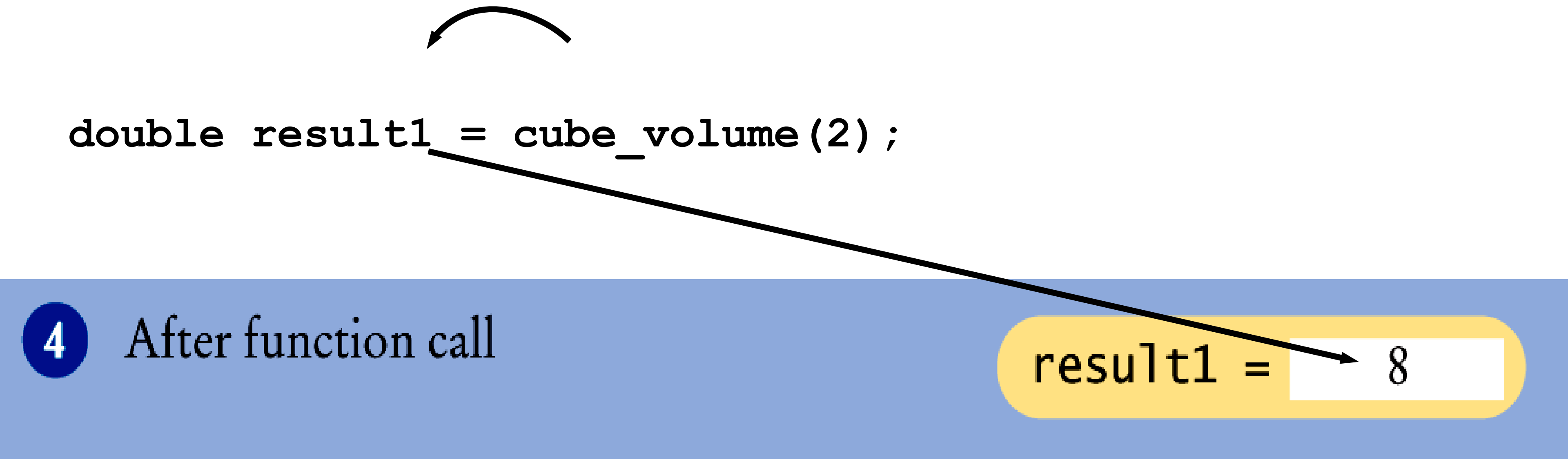


The function is over.
`side_length` and **`volume`** are gone.

Parameter Passing

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

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



Return Values

The `return` statement yields the function result.

Return Values

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

Consider the task of writing a string
with the following format around it.

Any string could be used.

For example, the string `"Hello"` would produce:

```
-------
 Hello
-------
```

Functions Without Return Values – The `void` Type

A function for this task can be defined as follows:

```
void box_string(int n)
```

Notice the return type of this function: **`void`**

Functions Without Return Values – The `void` Type

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

```
void box_string(int n)
```

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.

Functions Without Return Values – The `void` Type

void functions are used ***only*** to do a sequence of instructions.

Functions Without Return Values – The `void` Type

```
-------
 Hello
-------
```

- Print a line that contains the ‘-’ character n + 2 times, where n is the length of the string.
- Print a line containing the string,
- Print another line containing the - character n + 2 times.

Functions Without Return Values – The `void` Type

```
void box_string(int n)
{
   for (int i = 0; i < n + 2; i++) {
      printf("-");
   }
   printf("\n");
}
```

Note that this function doesn't compute any value.

It performs some actions and then returns to the caller – without returning a value.

(The return occurs at the end of the block.)

Functions Without Return Values – The `void` Type

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

You can make this call kind of call:

```
    box_string(5);
```

but not this kind:

```
    result = box_string(5);
      // Error: box_string 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 box_string(int n)
{
   if (n == 0) {
      return;
   }             // Return immediately
   . . .   // None of these statements
           // will be executed
```

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 statements 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

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 hours = read_value_between(1, 12);
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

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)

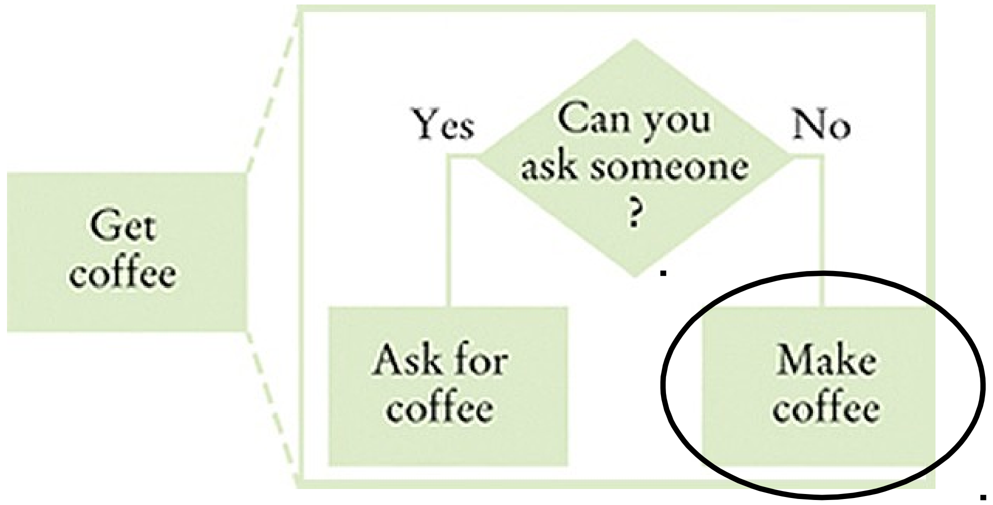
… and so on…

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

Stepwise Refinement

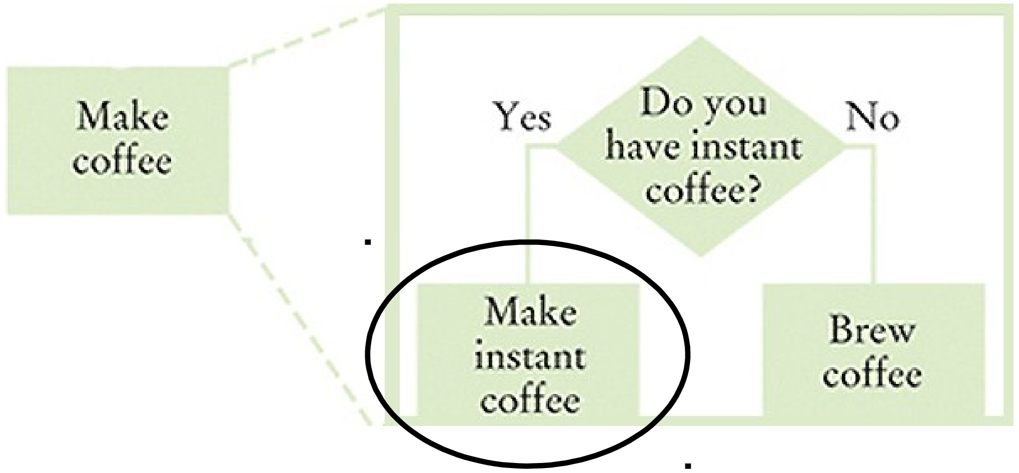
This is the whole problem: this is like **`main`**.

Stepwise Refinement



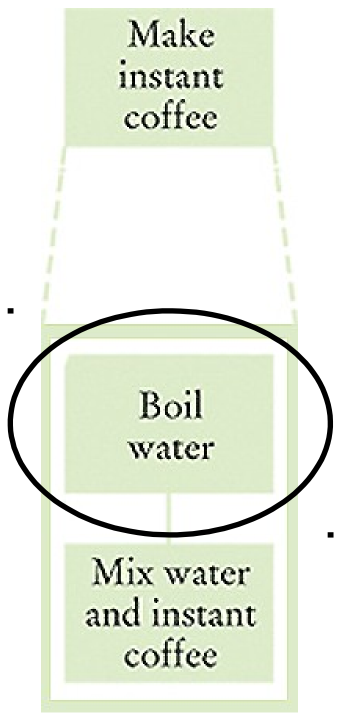
The whole problem can be broken into:
if we can ask someone to give us coffee, we are done
but if not, we can make coffee (which we will
have to break into its parts)

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

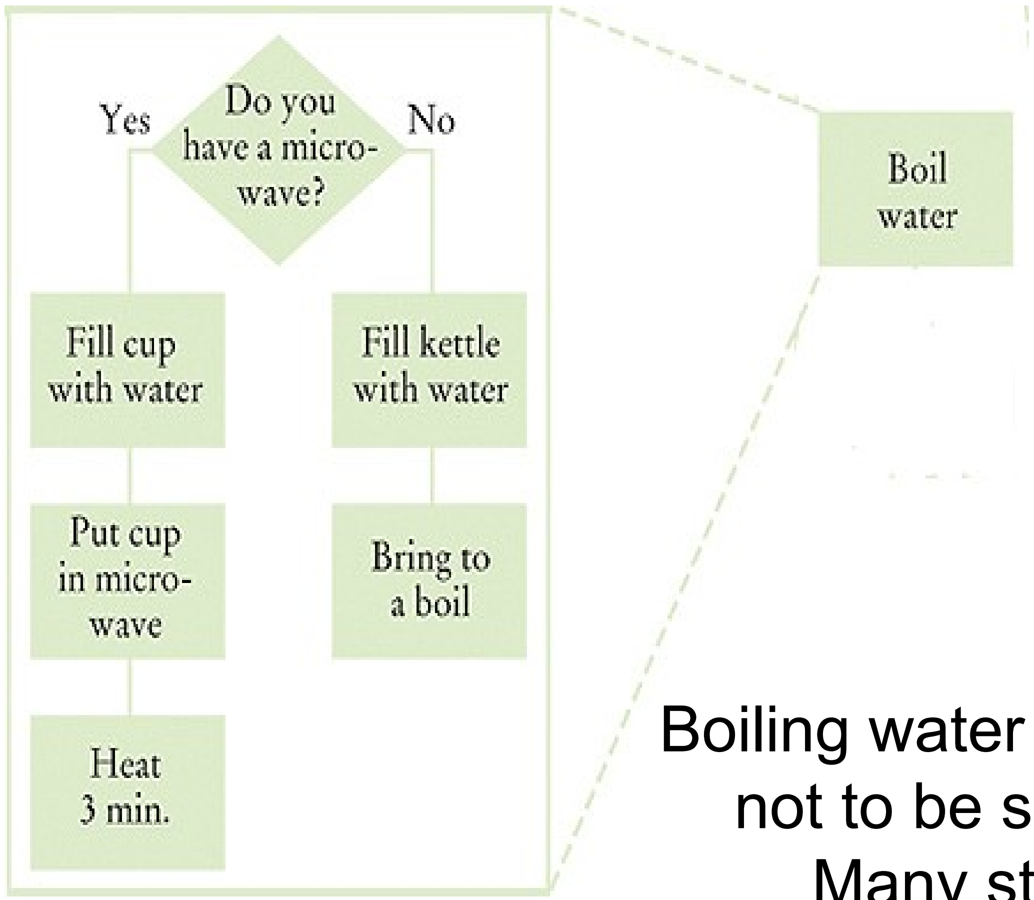


Making instant coffee breaks into:

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

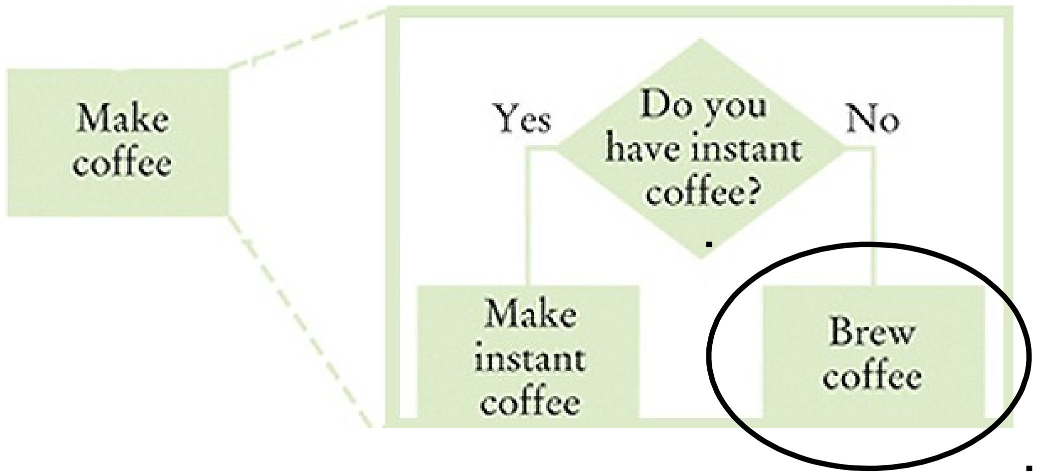
(Do these have sub-problems?)

Stepwise Refinement



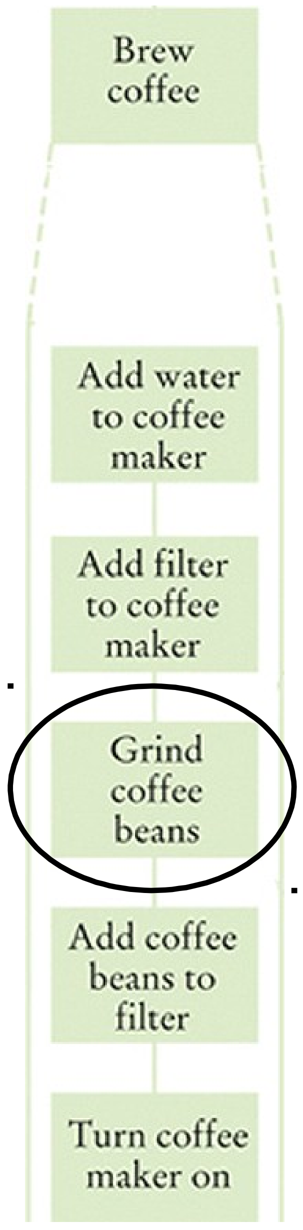
Boiling water appears
not to be so easy.
Many steps,
but none have sub-steps.

Stepwise Refinement



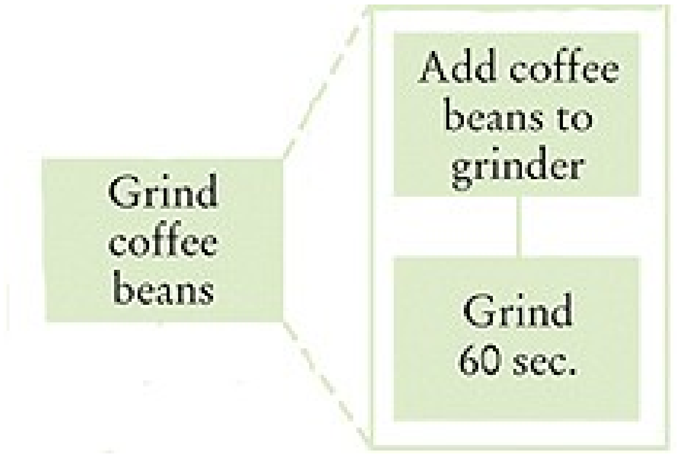
Going back to the branch between instant or brew, we need to think about brewing. Can we break that into parts?

Stepwise Refinement


Brew coffee
Add water to coffee maker
Add filter to coffee maker
Grind coffee beans
Add coffee beans to filter
Turn coffee maker on

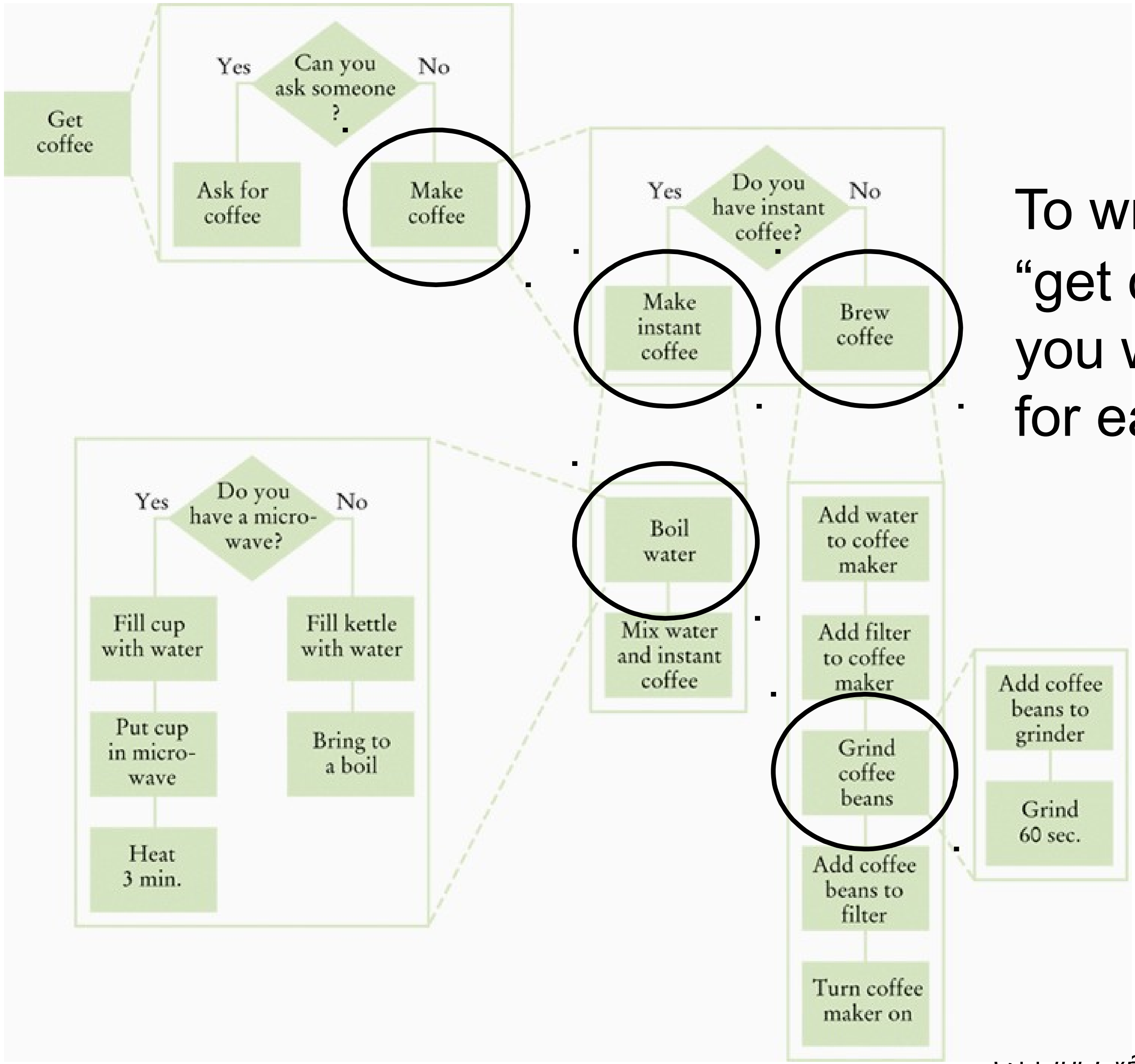
Brewing coffee has several steps.
Do any need more breakdown
(grind coffee beans)?

Stepwise Refinement



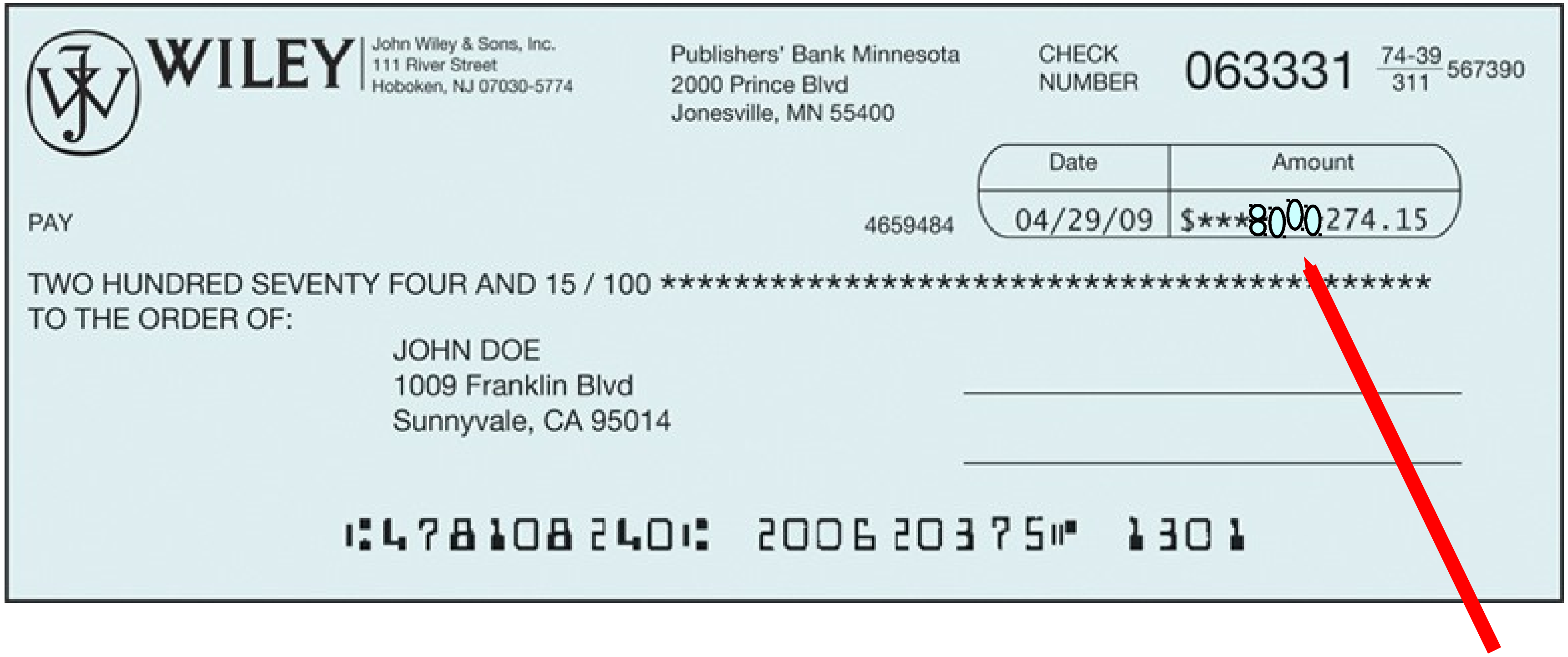
Grinding is a two step process with no sub-sub-steps.

Stepwise Refinement – The Complete Process Shown



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

Stepwise Refinement



WILEY | John Wiley & Sons, Inc.
111 River Street
Hoboken, NJ 07030-5774

Publishers' Bank Minnesota
2000 Prince Blvd
Jonesville, MN 55400

CHECK NUMBER 063331 74-39 / 311 567390

PAY 4659484

Date	Amount
04/29/09	$***8000274.15

TWO HUNDRED SEVENTY FOUR AND 15 / 100 ***

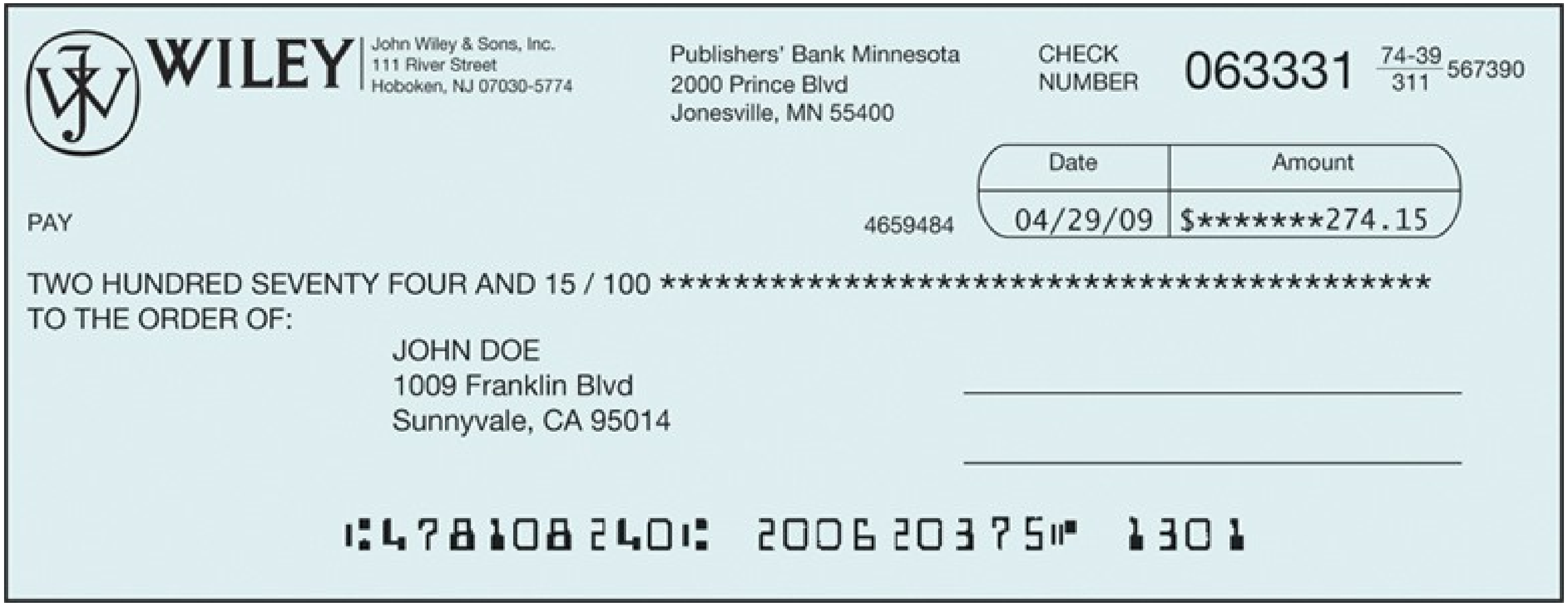
TO THE ORDER OF:

JOHN DOE
1009 Franklin Blvd
Sunnyvale, CA 95014

⑆478108240⑆ 2006 20375⑈ 1301

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

Stepwise Refinement



WILEY John Wiley & Sons, Inc.
111 River Street
Hoboken, NJ 07030-5774

Publishers' Bank Minnesota
2000 Prince Blvd
Jonesville, MN 55400

CHECK NUMBER 063331 74-39/311 567390

Date	Amount
04/29/09	$*******274.15

PAY 4659484

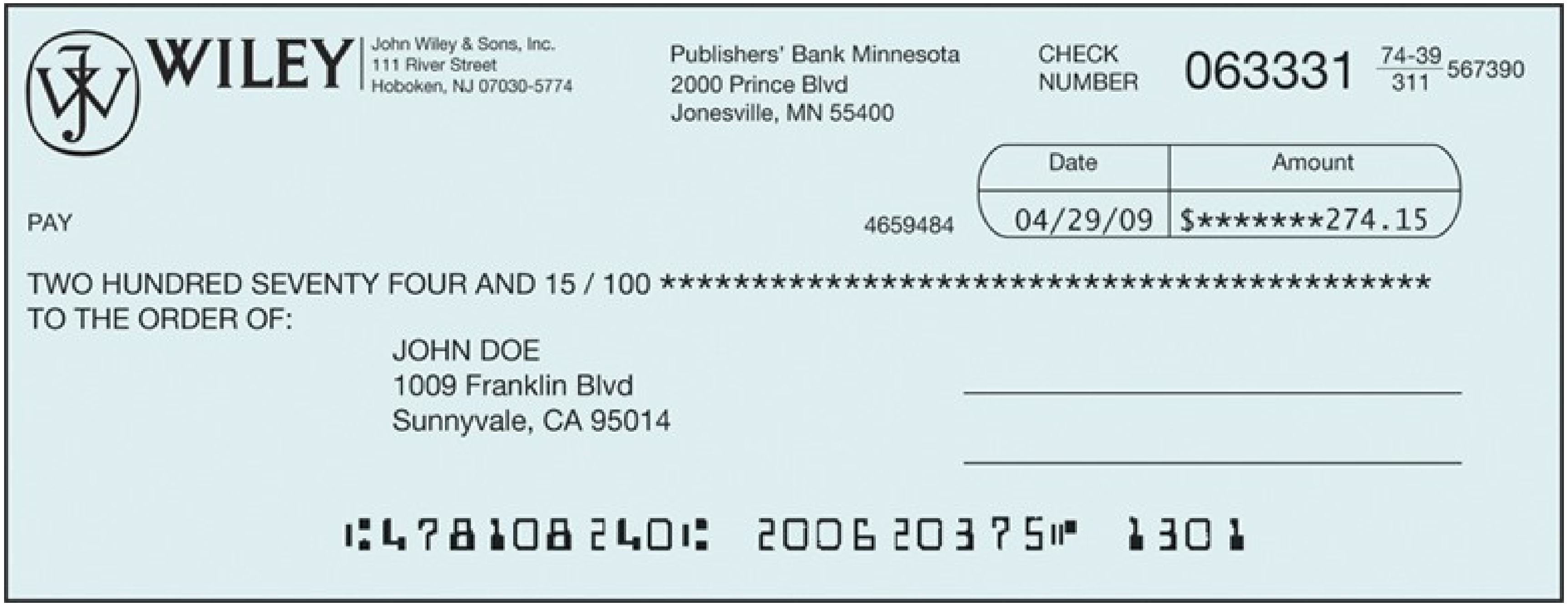
TWO HUNDRED SEVENTY FOUR AND 15 / 100 ***
TO THE ORDER OF:

JOHN DOE
1009 Franklin Blvd
Sunnyvale, CA 95014

⑆4781082 40⑆ 2006 20375⑈ 1301

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



WILEY | John Wiley & Sons, Inc.
111 River Street
Hoboken, NJ 07030-5774

Publishers' Bank Minnesota
2000 Prince Blvd
Jonesville, MN 55400

CHECK NUMBER 063331 74-39/311 567390

Date	Amount
04/29/09	$*******274.15

PAY 4659484

TWO HUNDRED SEVENTY FOUR AND 15 / 100 **
TO THE ORDER OF:

JOHN DOE
1009 Franklin Blvd
Sunnyvale, CA 95014

⑆478108240⑆ 2006203750⑈ 1301

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.

```
/**
 * Prints the English text for a number.
 *
 * @param number a positive number < 1,000
 */
void print_num(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

```
/**
 * Prints the English text for a digit.
 *
 * @param digit a number between 1 and 9
 * @return no return
*/
void print_digit(int digit)
```

Stepwise Refinement

Numbers between 10 and 19 are special cases.

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

```
/**
 * Prints the English text for a number
 * between 10 and 19.
 *
 * @param number an integer between 10 and 19
 */
void print_teen(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:

```
/**
 * Prints the English text for the tens digit
 * of a number between 20 and 99.
 *
 * @param number an integer between 20 and 99
 */
void print_tens(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 `print_tens`, and we are done.***
  - ***Otherwise, we print the tens with `print_tens` and the ones with `print_digit`.***

- 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

```
remaining = number (part that still needs to be converted)

If remaining >= 100
    Print hundreds digit + "hundred"
    Remove hundreds from remaining

If remaining >= 20
    Print tens digit
    Remove tens from remaining
Else if part >= 10
    Print teen number
    remaining = 0

If remaining > 0
    Print digit
```

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 "hundreds digit" and the like.***
  - ***Spaces—it would print output 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 (remaining >= 20) {
    print_tens(remaining);
    remaining = remaining % 10;
} else if (remaining >= 10) {
    print_teen(remaining);
    remaining = 0;
}

if (remaining > 0) {
    print_digit(remaining);
}
```

Stepwise Refinement – Pseudocode to C

Finally, the case of numbers between 100 and 999.
Because `remaining < 1000`, `remaining / 100` is a single digit,
and we print it by calling `print_digit`.
Then we print “hundred”:

```
if (remaining >= 100) {
    print_digit(remaining / 100);
    printf("hundred");
    remaining = remaining % 100;
}
```

Stepwise Refinement – Pseudocode to C

Now for the other functions.

The Complete Program

```
void print_digit(int digit)
{
    if (digit == 1) {
        printf("one");
    } else if (digit == 2) {
        printf("two");
    } else if (digit == 3) {
        printf("three");
    } … {
    } else if (digit == 9) {
        printf("nine");
    }
}
```

The Complete Program

```
void print_teen(int number)
{
    if (number == 10) {
        printf("ten");
    } else if (number == 11) {
        printf("eleven");
    } else if (number == 12) {
        printf("twelve");
    } ... {
    } else if (number == 19) {
        printf("nineteen");
    }
}
```

The Complete Program

```
void print_tens(int number)
{
    if (number >= 90) {
        printf("ninety");
    } else if (number >= 80) {
        printf("eighty");
    } else if (number >= 70) {
        printf("seventy");
    } ... {
    } else if (number >= 20) {
        printf("twenty");
    }
}
```

The Complete Program

```
int main()
{
   int number;
   printf("Please enter a positive integer: ");
   scanf("%d", &number);
   print_num(number);
   printf("\n");
   return EXIT_SUCCESS;
}
```

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.

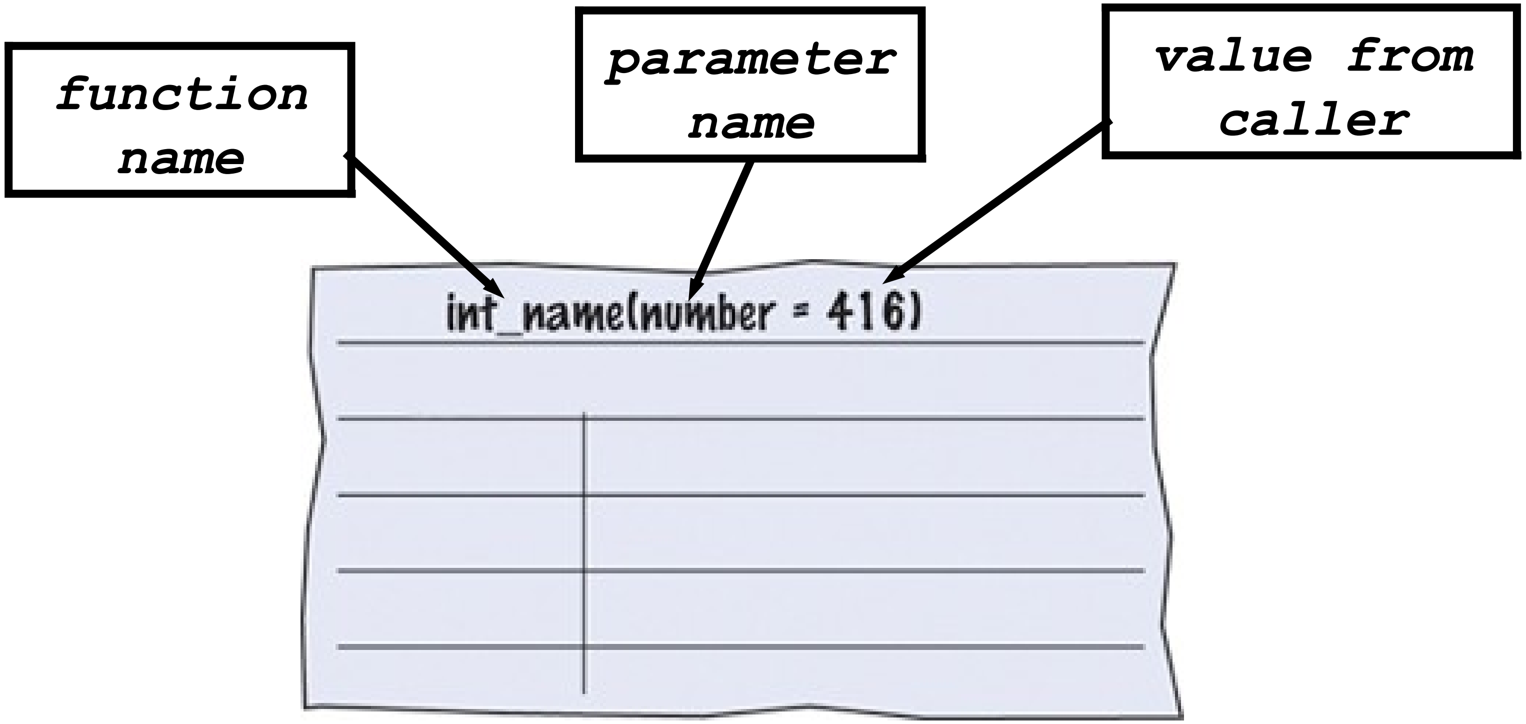
Tracing Functions

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:

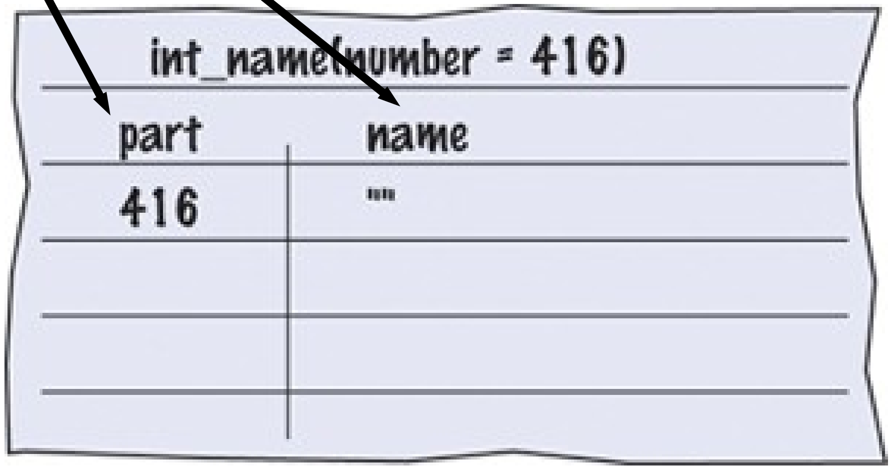


Tracing Functions

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:



int_name(number = 416)

part	name
416	""

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.

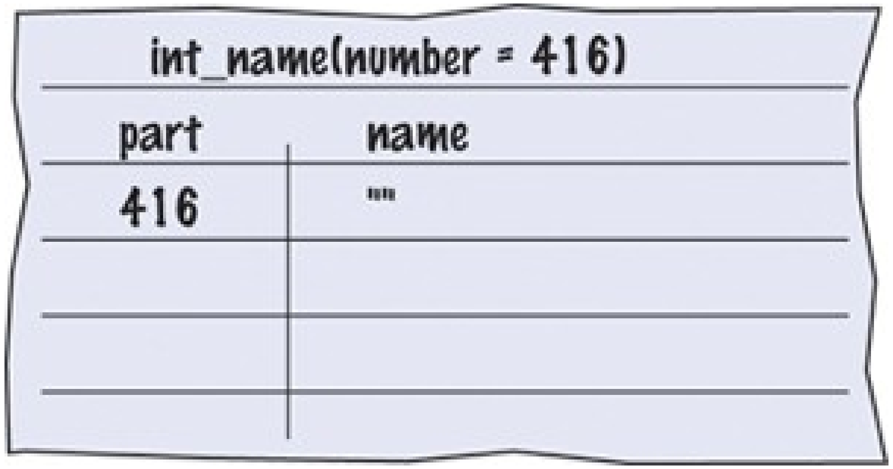
Tracing Functions

Output has changed to

`digit_name(part / 100) + "hundred“`

which is the string "four hundred",

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



int_name(number = 416)	
part	name
416	""

Tracing Functions

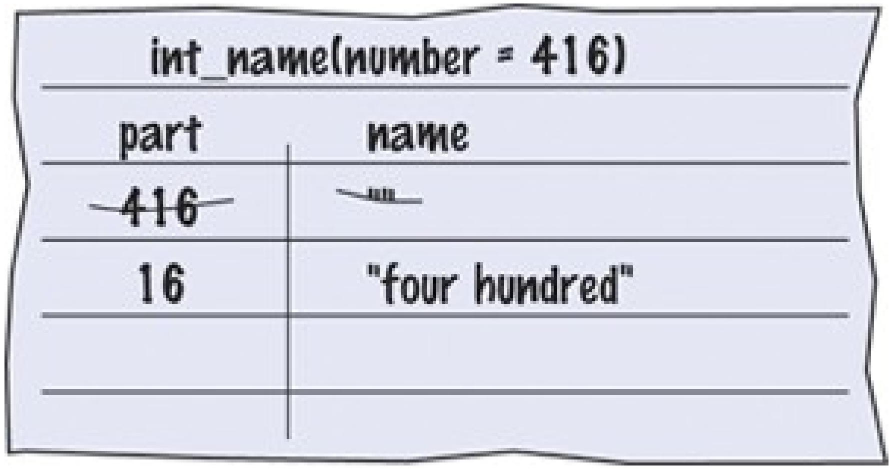
Output has changed to

`digit_name(part / 100) + " hundred“`

which is the string "four hundred",

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

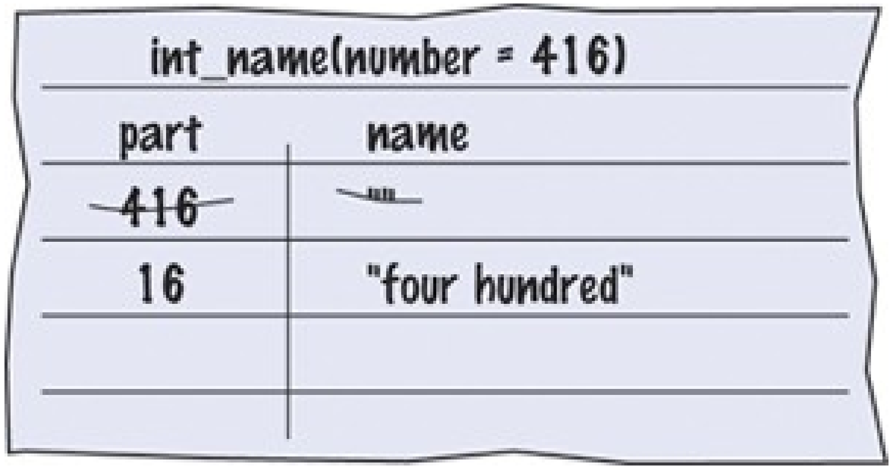
Cross out the old values and write the new ones.



Tracing Functions

Let’s continue…

Here is the status of the parameters and variables now:



int_name(number = 416)

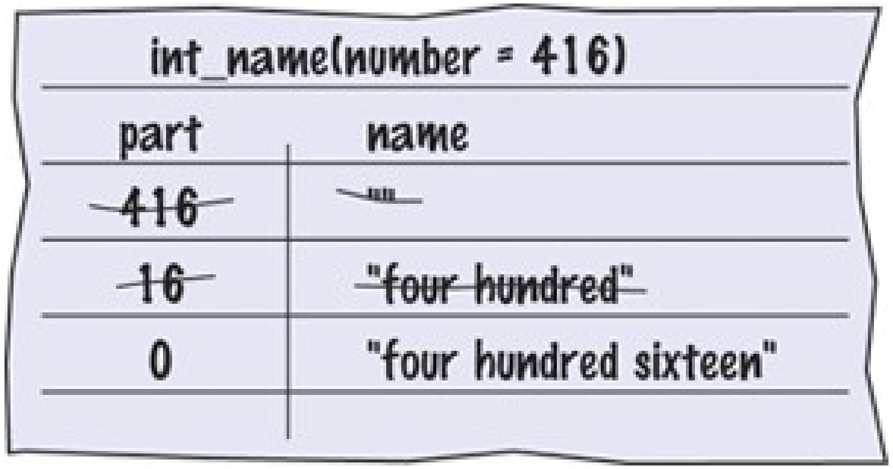
part	name
~~416~~	~~""~~
16	"four hundred"

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	name
~~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.

```
void print_digit(int digit)
{
   printf("mumble");
}


void print_tens(int number)
{
   printf("mumblety");
}
```

Stubs

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

```
Please enter a positive integer: 274
mumblehundredmumbletymumble
```

which indicates that the basic logic of the **`print_num`** function is working correctly.

Now that you have tested **`print_num`**, you would "unstubify" another stub function, then another...

Variable Scope

?

Variable Scope

Which main?

Variable Scope

You can only have *one* **`main`** function
but you can have as many variables and parameters
spread amongst as many functions as you need.

Can you have the same name in different functions?

Variable Scope

The **`railway_avenue`** *and* **`main_street`** *variables in the* **`oklahoma_city`** *function*

The **`south_street`** ***and*** **`main_street`** *variables in the* **`panama_city`** *function*

The **`n_putnam_street`** *and* **`main_street`** *variables in the* **`new_york_city`** *function*

Variable Scope

A variable or parameter that is defined within a function is visible from the point at which it is defined until the end of the block named by the function.

This area is called the *scope* of the variable.

Variable Scope

The scope of a variable is the part of the program in which it is *visible*.

Variable Scope

The scope of a variable is the part of the program in which it is *visible*.

Because scopes do not overlap, a name in one scope cannot conflict with any name in another scope.

Variable Scope

The scope of a variable is the part of the program in which it is *visible*.

Because scopes do not overlap,
a name in one scope cannot
conflict with any name in another scope.

A name in one scope is “invisible”
in another scope

Variable Scope

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

int main()
{
   double volume = cube_volume(2);
   printf("%f\n", volume);
   return 0;
}
```

Each **`volume`** variable is defined in a separate function, so there is not a problem with this code.

Variable Scope

Because of scope, when you are writing a function you can focus on choosing variable and parameter names that make sense for your function.

You do not have to worry that your names will be used elsewhere.

Variable Scope

Names inside a block are called *local* to that block.

A function names a block.

Recall that variables and parameters do not exist after the function is over—because they are local to that block.

But there are other blocks.

Variable Scope

It is *not legal* to define two variables or parameters with the same name in the same scope.

For example, the following is not legal:

ERRORS!!!

```
int test(double volume)
{
   double volume = cube_volume(2);
   double volume = cube_volume(10);
// ERROR: cannot define another volume variable
// ERROR: or parameter in the same scope
...
}
```

Variable Scope – Nested Blocks

However, you can define another variable with the same name in a *nested block*.

```
double withdraw(double balance, double amount)
{
   if (...)
   {
      double amount = 10;
      ...
   }
   ...
}
```

a variable named **`amount`** local to the **`if`**'s block

– *and* a parameter variable named **`amount`**.

Variable Scope – Nested Blocks

The scope of the parameter variable **`amount`** is the entire function, *except* the nested block.

Inside the nested block, **`amount`** refers to the local variable that was defined in that block.

You should avoid this *potentially confusing situation* in the functions that you write, simply by renaming one of the variables.

Why should there be a variable with the same name in the same function?

Global Variables

- Generally, global variables are ***not*** a good idea.

But …

here's what they are and how to use them

(if you must).

Global Variables

Global variables are defined *outside* any block.

They are visible to every function defined after them.

Global Variables

But in a banking program, how many functions should have direct access to a balance variable?

Global Variables

```
int balance = 10000; // A global variable

void withdraw(double amount)
{
    if (balance >= amount)
    {
        balance = balance - amount;
    }
}

int main()
{
    withdraw(1000);
    printf("%d\n", balance);
    return 0;
}
```

Global Variables

In the previous program there is only one function that updates the **`balance`** variable.

But there could be many, many, many functions that might need to update **`balance`** each written by any one of a huge number of programmers in a large company.

Then we would have a problem.

Global Variables

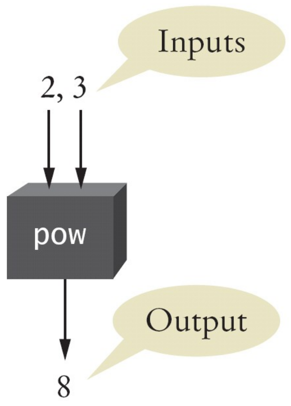
When multiple functions update global variables, the result can be *difficult* to predict.

Particularly in larger programs that are developed by multiple programmers, it is very important that the effect of each function be clear and easy to understand.

Global Variables – Breaking Open the Black Box

When functions modify global variables, it becomes more difficult to understand the effect of function calls.

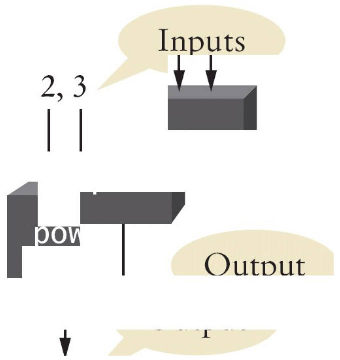
Programs with global variables are difficult to maintain and extend because you can no longer view each function as a “black box” that simply receives parameter values and returns a result or does something.



Global Variables – Breaking Open the Black Box

When functions modify global variables, it becomes more difficult to understand the effect of function calls.

Programs with global variables are difficult to maintain and extend because you can no longer view each function as a "black box" that simply receives parameter values and returns a result or does something.



And what good is a broken black box?

Global Variables – Just Say "No"

You should *avoid* global variables in your programs!

Example: Game of Chance

- One of the most popular games of chance is a dice game known as “craps.” The rules of the game are simple.
  - *A player rolls two dice. Each die has six faces. These faces contain 1, 2, 3, 4, 5, and 6 spots. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, the player wins. If the sum is 2, 3, or 12 on the first throw (called “craps”), the player loses (i.e., the “house” wins). If the sum is 4, 5, 6, 8, 9, or 10 on the first throw, then that sum becomes the player’s “point.” To win, you must continue rolling the dice until you “make your point.” The player loses by rolling a 7 before making the point.*

Example

- We will write a function *rollDice* to throw the two dice and take their sum.

  (we did this for one die in the previous lecture)

```
int rollDice()
{
   int die1, die2, workSum;

   die1 = 1 + (rand()%6);
   die2 = 1 + (rand()%6);
   workSum = die1 + die2;

   printf("Player rolled %d + %d = %d\n",
        die1, die2, workSum);
   return workSum;
}
```

Complete Program

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

enum Status {CONTINUE, WON, LOST};

int rollDice(); //function prototype

int main()
{
    int sum;
    int myPoint;
    enum Status gameStatus;

    srand(time(NULL));
    sum = rollDice();
```

Complete Program

```
switch(sum){
     case 7:
     case 11:
          gameStatus = WON;
          break;
     case 2:
     case 3:
     case 12:
          gameStatus = LOST;
          break;
     default:
          gameStatus = CONTINUE;
          myPoint = sum;
          printf("Point is %d\n", myPoint);
          break;
}
```

Complete Program

```
while(CONTINUE == gameStatus)
{
    sum = rollDice();
    if (sum == myPoint)
          gameStatus = WON;
    else{
          if (7 == sum){
                gameStatus = LOST;
          }
    }
}
if (WON == gameStatus){
    printf("Player wins\n");
}
else{
    printf("Player loses\n");
}
} //end of main
```

Enumeration constants

- The player may win or lose on the first roll, or may win or lose on any subsequent roll.
- Variable gameStatus, defined to be of a new type—enum Status—stores the current status.
- Line 8 creates a programmer-defined type called an enumeration.
- An enumeration, introduced by the keyword enum, is a set of integer constants represented by identifiers.
- Enumeration constants are sometimes called symbolic constants.
- Values in an enum start with 0 and are incremented by 1.

CHAPTER SUMMARY

Understand the concepts of functions, arguments, and return values.

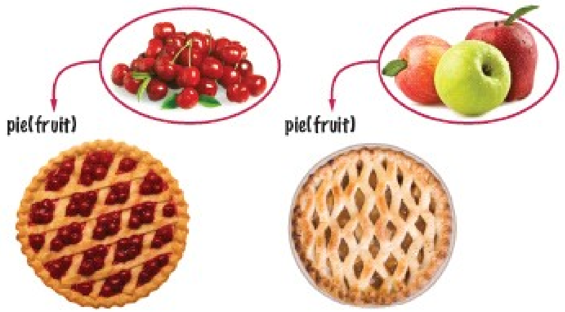
- A function is a named sequence of instructions.
- Arguments are supplied when a function is called. The return value is the result that the function computes.

Be able to implement functions.

- When defining a function, you provide a name for the function, a variable for each argument, and a type for the result.
- Function comments explain the purpose of the function, the meaning of the parameter variables and return value, as well as any special requirements.

Describe the process of parameter passing.

- Parameter variables hold the argument values supplied in the function call.



CHAPTER SUMMARY

Describe the process of returning a value from a function.

- The return statement terminates a function call and yields the function result.

Design and implement functions without return values.

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

Develop functions that can be reused for multiple problems.

- Eliminate replicated code or pseudocode by defining a function.
- Design your functions to be reusable. Supply parameter variables for the values that can vary when the function is reused.

Apply the design principle of stepwise refinement.

- Use the process of stepwise refinement to decompose complex tasks into simpler ones.
- When you discover that you need a function, write a description of the parameter variables and return values.
- A function may require simpler functions to carry out its work.

CHAPTER SUMMARY

Determine the scope of variables in a program.

- The scope of a variable is the part of the program in which it is visible.
- A variable in a nested block shadows a variable with the same name in an outer block.
- A local variable is defined inside a function. A global variable is defined outside a function.
- Avoid global variables in your programs.

Describe how reference parameters work.

- Modifying a value parameter has no effect on the caller.
- A reference parameter refers to a variable that is supplied in a function call.
- Modifying a reference parameter updates the variable that was supplied in the call.

End Chapter Five