Chapter Five: Functions I

Slides by Evan Gallagher & Nikolay Kirov

C++ for Everyone by Cay Horstmann

Chapter Goals

- To be able to implement functions
- To become familiar with the concept of parameter passing
- To appreciate the importance of function comments

Call a Function to Get Something Done

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

What Is a Function? Why Functions?

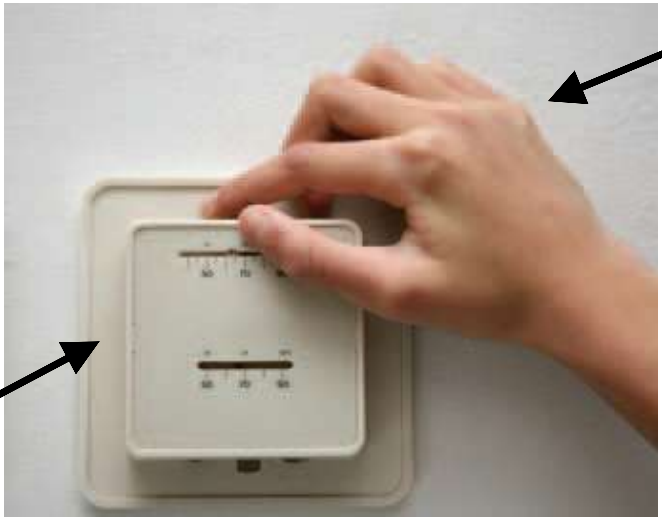
A function is a sequence of instructions with a name.

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

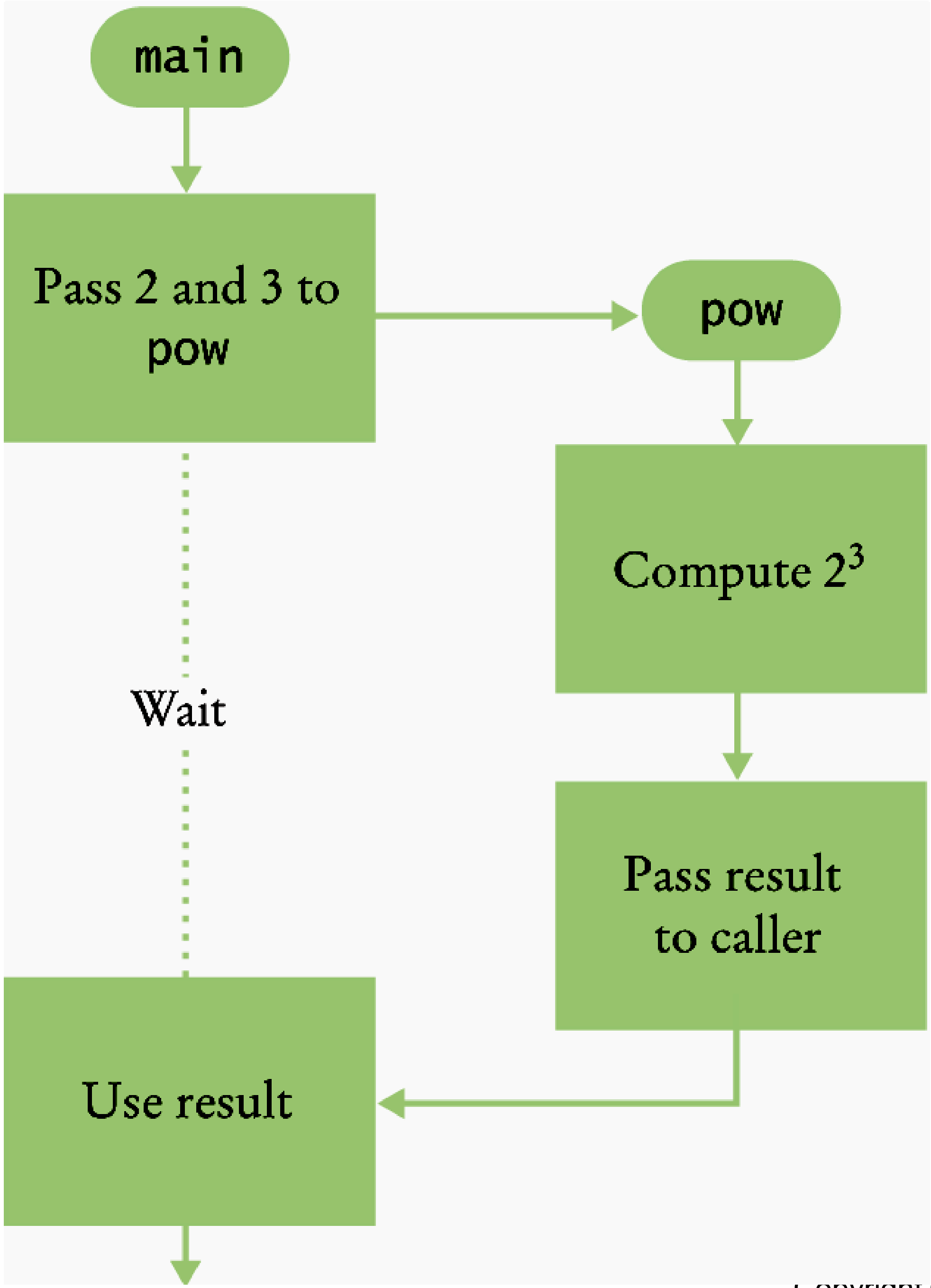
```
int main()
{
   double z = pow(2, 3);
   ...
}
```

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.



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 (**`cin >>`**), these values are called *parameter values*.

The "output" that the **`pow`** function computes is called the return value (not output using **`<<`**).

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 **<<** 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 << does *not* communicate with the calling programmer

The Return Statement Does Not Display (Good!)

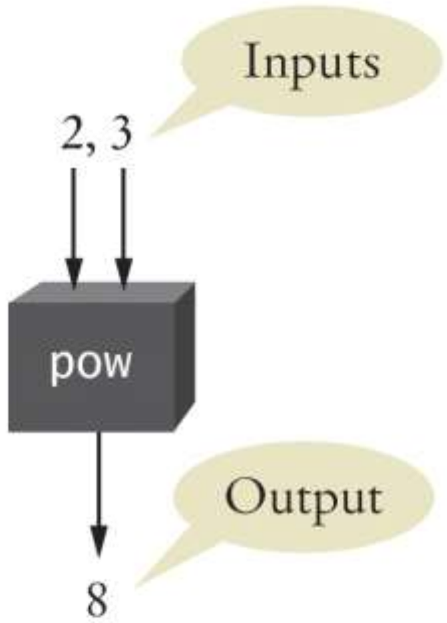
```
int main()
{
   double z = pow(2, 3);

   // display result of calculation
   // stored in variable z
   cout << z << endl;

   // 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?)

`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

Here's the side length you need to calculate the volume for me

```
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

I'll do the calculating right now.

```
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;
}
```

Implementing Functions

Here it is.
I calculated it,
just for you.

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

Implementing Functions

```
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 Testing Program

ch05/cube.cpp

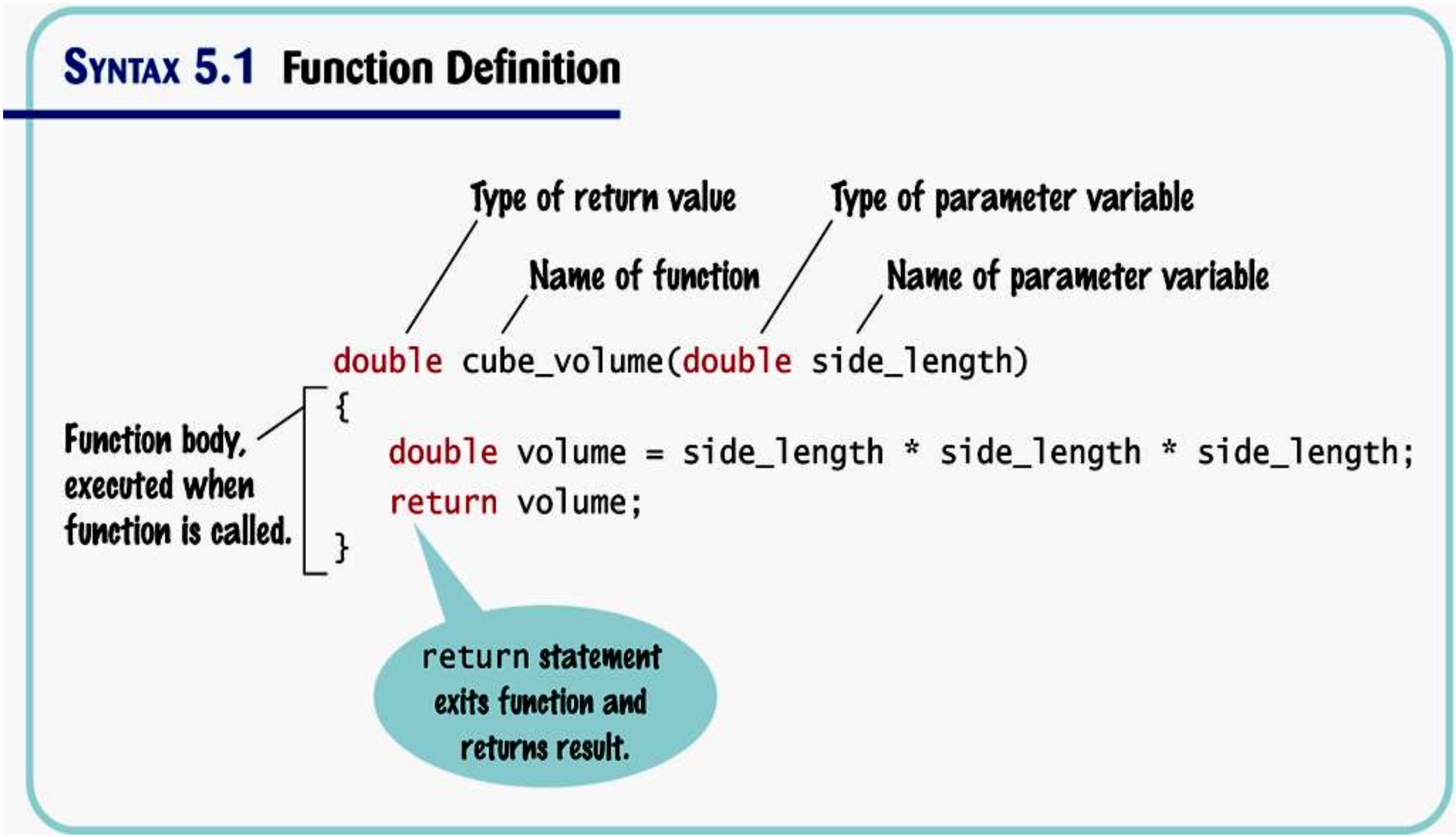
```
#include <iostream>
using namespace std;

/**
   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

ch05/cube.cpp

```
int main()
{
   double result1 = cube_volume(2);
   double result2 = cube_volume(10);
   cout << "A cube with side length 2 has volume "
      << result1 << endl;
   cout << "A cube with side length 10 has volume "
      << result2 << endl;

   return 0;
}
```

Implementing Functions



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

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);
. . .
```

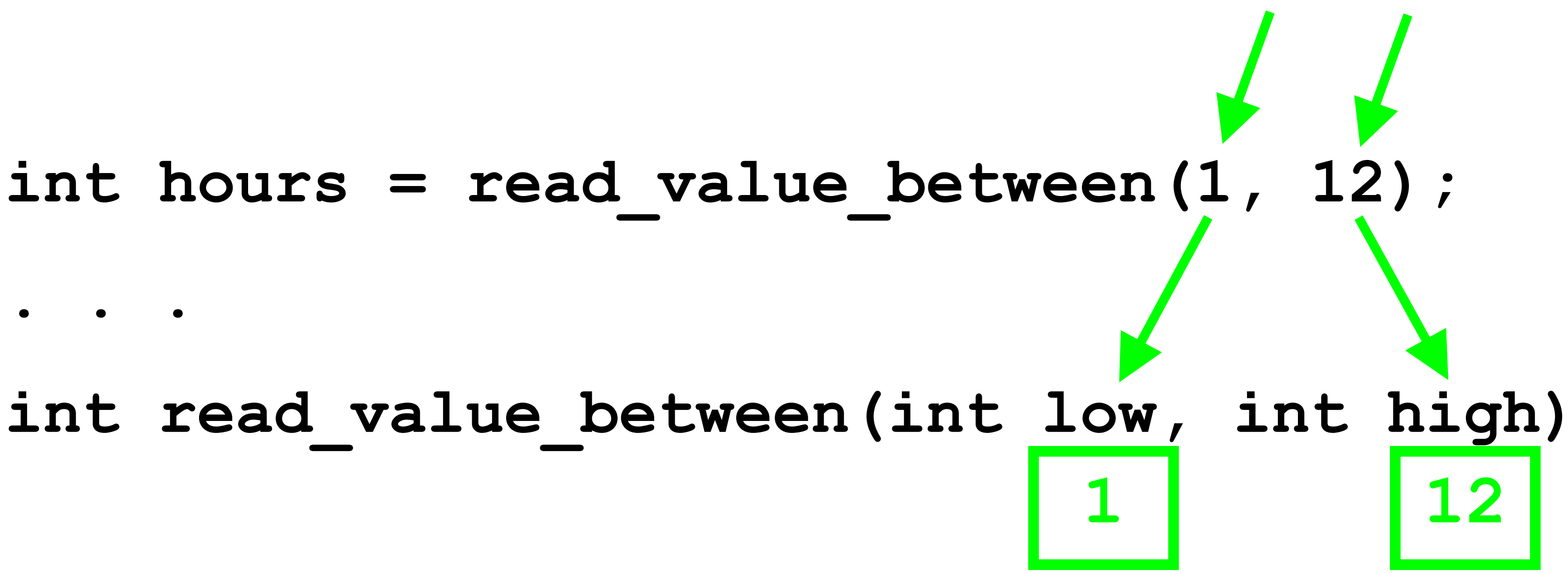
Parameter Passing

When a function is called,
a *parameter variable* is created for each value passed in.
(Another commonly used term is *formal parameter*.)
(Parameters that take values are also known as *value parameters.*)

```
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.



Parameter Passing

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


int read_value_between(int low, int high)
{
   int input;
   do
   {
      cout << "Enter a value between "
         << low << " and " << high << ": ";
      cin >> 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;
}
```

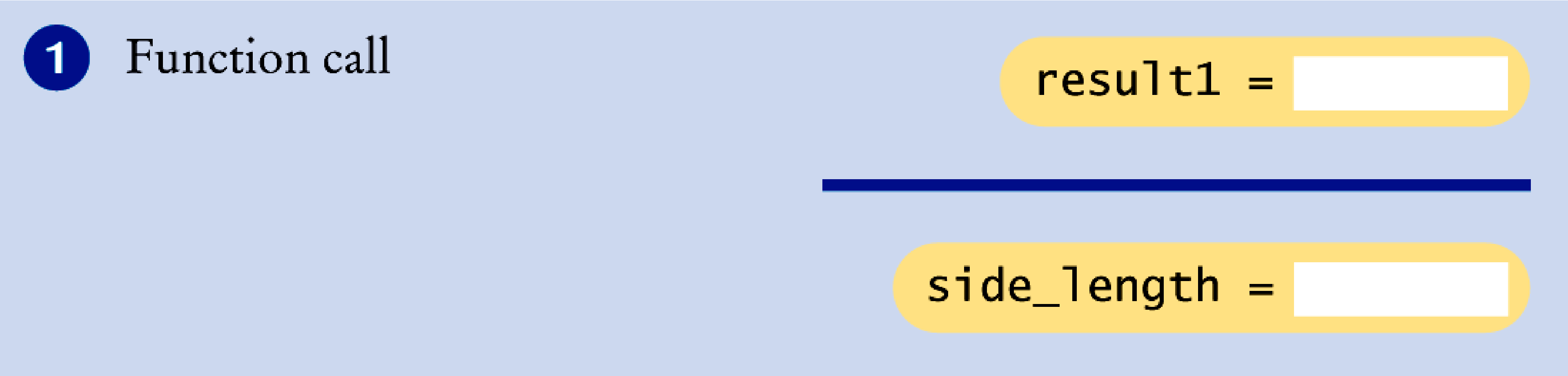
We'll keep up with their variables and parameters:

```
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.

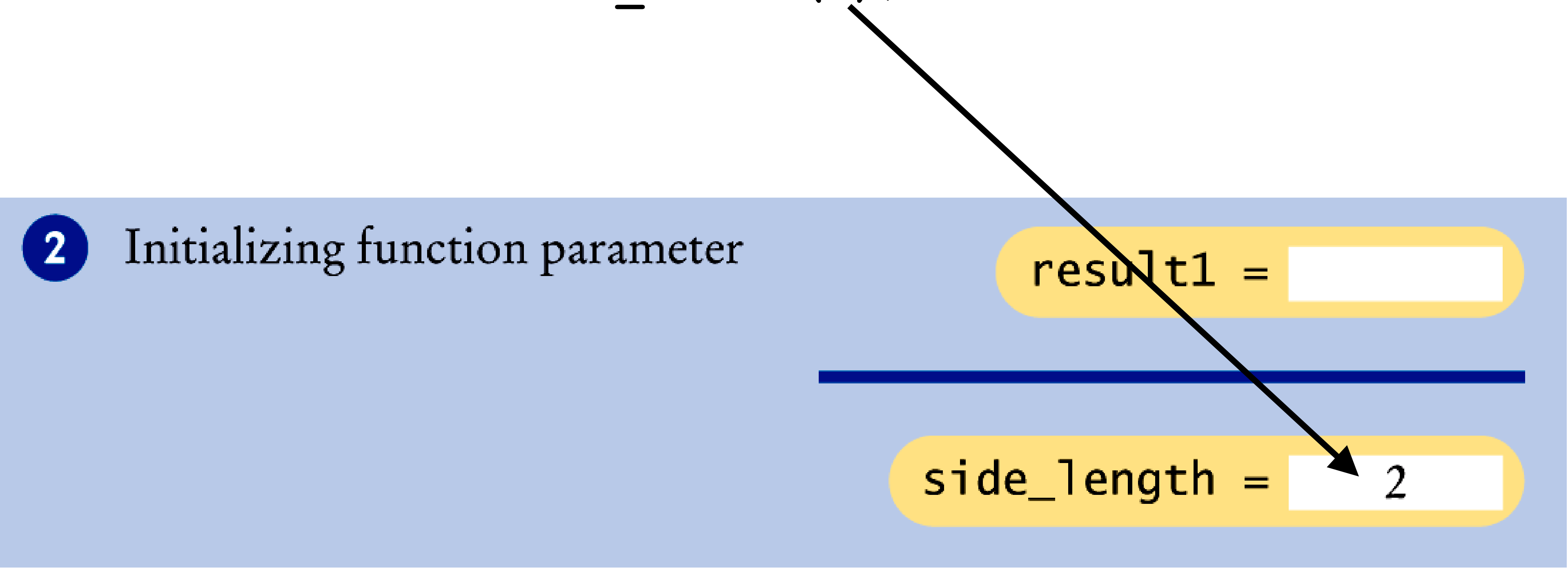
```
double result1 = cube_volume(2);
```



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.

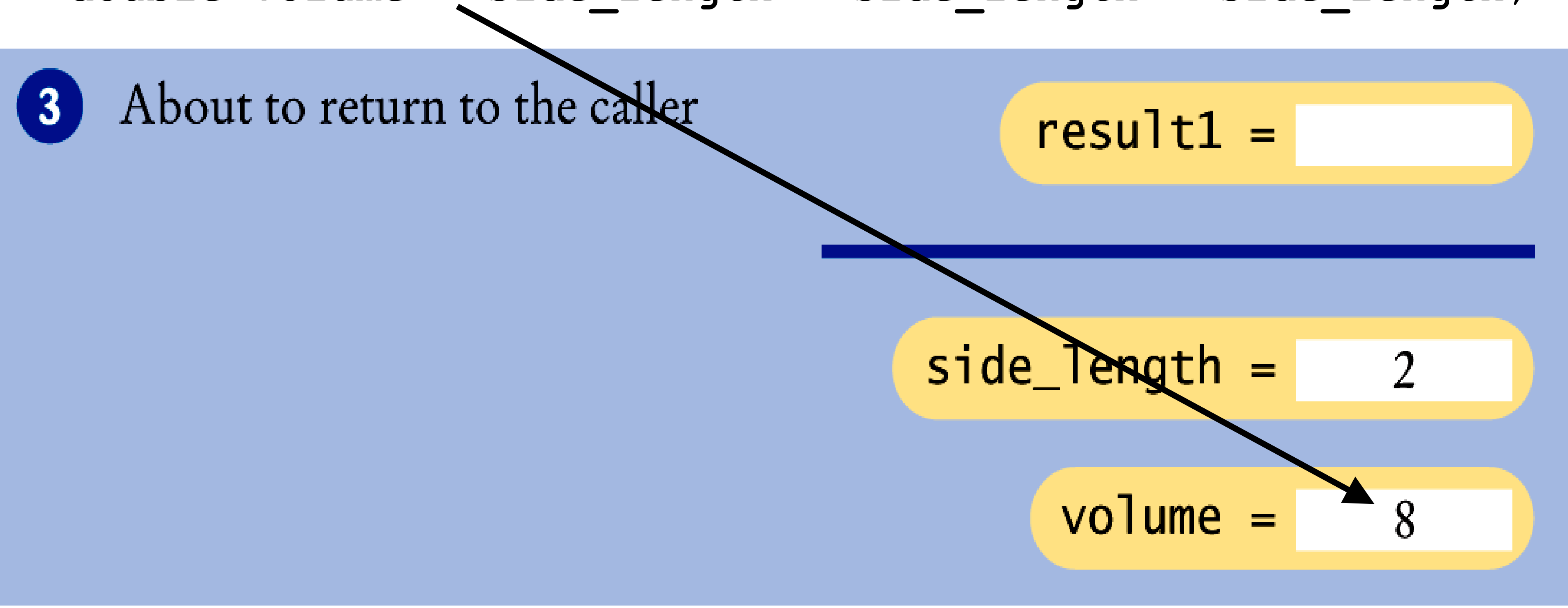
```
double result1 = cube_volume(2);
```



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

// you are here

Return Values

Also,

The **`return`** statement

– terminates a function call

– immediately

```
// you are here

return
// now you are here
```

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(string str)
```

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(string str)
```

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

`void` functions are used to

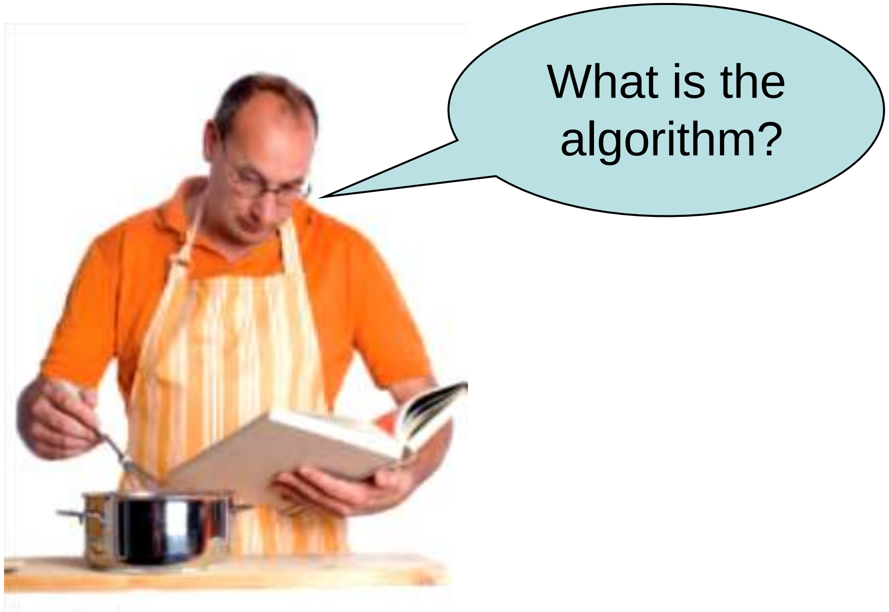
- simply do a sequence of instructions
  - They do not return a value to the caller.

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, surrounded with a ! to the left and right.
- Print another line containing the - character n + 2 times.

Functions Without Return Values – The `void` Type

```
void box_string(string str)
{
   int n = str.length();
   for (int i = 0; i < n + 2; i++){ cout << "-"; }
   cout << endl;
   cout << "!" << str << "!" << endl;
   for (int i = 0; i < n + 2; i++) { cout << "-"; }
   cout << endl;
}
```

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("Hello");
```

but not this kind:

```
result = box_string("Hello");
  // 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(string str)
{
   int n = str.length();
   int n = str.length();
   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
{
   cout << "Enter a value between 0 and 23:";
   cin >> hours;
} while (hours < 0 || hours > 23);

int minutes;
do
{
   cout << "Enter a value between 0 and 59: ";
   cin >> 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
{
   cout << "Enter a value between 0 and 23:";
   cin >> hours;
} while (hours < 0 || hours > 23);
```

```
int minutes;
do
{
   cout << "Enter a value between 0 and 59: ";
   cin >> minutes;
} while (minutes < 0 || minutes > 59);
```

Designing Functions – Turn Repeated Code into Functions

The names of the variables are different.

```
int hours;
do
{
   cout << "Enter a value between 0 and 23:";
   cin >> hours;
} while (hours < 0 || hours > 23);

int minutes;
do
{
   cout << "Enter a value between 0 and 59: ";
   cin >> minutes;
} while (minutes < 0 || minutes > 59);
```

Designing Functions – Turn Repeated Code into Functions

But there is common behavior.

```
int hours;
do
{
   cout << "Enter a value between _ and __:";
   cin >> hours;
} while (hours < _ || hours > __);
int minutes;
do
{
   cout << "Enter a value between _ and __: ";
   cin >> 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
    {
        cout << "Enter a value between "
            << "0 and " << high << ": ";
        cin >> 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
    {
        cout << "Enter a value between "
            << "0 and " << high << ": ";
        cin >> 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
{
    cout << "Enter a value between 1 and 12:";
    cin >> month;
} while (month < 1 || month > 12);

int minutes;
do
{
    cout << "Enter a value between 0 and 59: ";
    cin >> 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
{
   cout << "Enter a value between 1 and 12:";
   cin >> month;
} while (month < 1 || month > 12);

int minutes;
do
{
   cout << "Enter a value between 0 and 59: ";
   cin >> minutes;
} while (minutes < 0 || minutes > 59);
```

Designing Functions – Turn Repeated Code into Functions

But the names of the variables are different.

```
int month;
do
{
   cout << "Enter a value between 1 and 12:";
   cin >> month;
} while (month < 1 || month > 12);

int minutes;
do
{
   cout << "Enter a value between 0 and 59: ";
   cin >> minutes;
} while (minutes < 0 || minutes > 59);
```

Designing Functions – Turn Repeated Code into Functions

Notice the common behavior?

```
int month;
do
{
   cout << "Enter a value between _ and __:";
   cin >> month;
} while (month < _ || month > __);
```

```
int minutes;
do
{
   cout << "Enter a value between _ and __: ";
   cin >> 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
    {
        cout << "Enter a value between "
            << low << " and " << high << ": ";
        cin >> 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
   {
      cout << "Enter a value between "
         << low << " and " << high << ": ";
      cin >> 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!

End Functions I