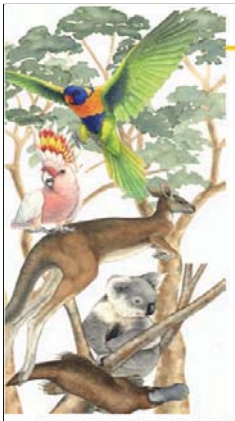


Chapter Five: Functions

Slides prepared by Evan Gallagher, New York University

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

What Is a Function? Why Functions? (5.1)

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);
   ...
}
```

8

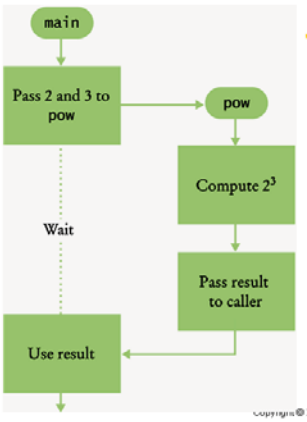
By using the expression: `pow(2, 3)`
`main` *calls* the `pow` function, asking it to compute 2^3.

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);
   ...
}
```

the inputs to a function are called parameters or parameter values

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 <<).

↖ the output of the function (8 in this case)

An Output Statement Does Not Return a Value

cout return

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

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

Implementing Functions (5.2)

EX: 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 value

```
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 value

`double` `cube_volume(double side_length)`

Implementing Functions

Steps

When writing this function, you need to:

1. Pick a good, descriptive name for the function
2. 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.
3. Specify the type of the return value
4. Now write the *body* of the function:

   the code to do the cubing

Implementing Functions

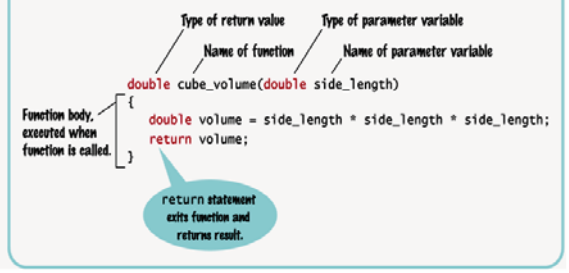
The code the function names must be in a block:

```
double cube_volume(double side_length)
{
   // the code that does the cubing   (1)
   // a return statement will give    (2)
   // the caller the calculated value
}
```

minimum of two statements

Implementing Functions

Syntax 5.1 Function Definition



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

the function comes before the "main" program

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

/**
   Computes the volume of a cube.          ← state what function will do
   @param side_length = the side length of the cube   ↖ list parameters
   @return the volume                      ↖ list what will be returned
*/                                         } Standard comments
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;
}
```

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.

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 1 and 12:";
   cin >> hours;
} while (hours < 1 || hours > 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

The values for the range are different.

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

The names of the variables are different.

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

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

```
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

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

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

Note how the code has become much easier to understand.

And we are not rewriting code

– code reuse!

Calling Functions

Consider the order of activities when a function is called.

Parameter Passing (5.3)

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.

```
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
   {
      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);
```

1 Function call

`result1 =`

`side_length =`

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);
```

2 Initializing function parameter

`result1 =`

`side_length = 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;
```

3 About to return to the caller

`result1 =`

`side_length = 2`

`volume = 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);
```

4 After function call

`result1 =`

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

4 After function call

`result1 =`

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);
```

4 After function call

`result1 = 8`

Return Values (5.4) Stopped

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

Return Values

Also,

The `return` statement
- terminates a function call
- immediately

// you are here