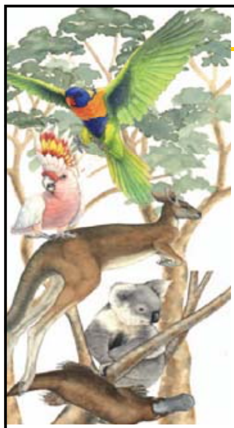


Chapter Five: Functions

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

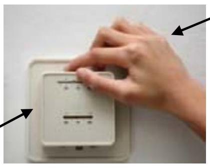
What Is a Function? Why Functions? (5.1)

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.



Calling a Function

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

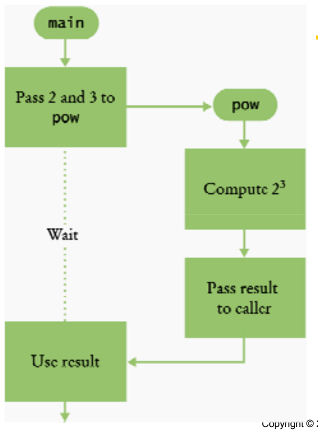
By using the expression: **`pow(2, 3)`**
`main` *calls* the **`pow`** function, asking it to compute **2^3**.

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 (**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

- 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:

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

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

Now write the *body* of the function:

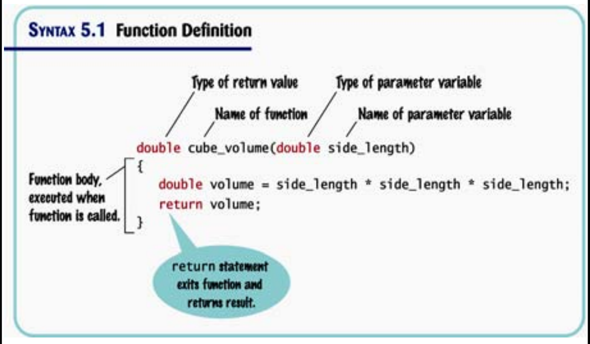
the code to do the cubing

Implementing Functions

The code the function names must be in a block:

```
double cube_volume(double side_length)
{

   // the code that does the cubing

   // a return statement will give
   // the caller the calculated value

}
```

Implementing Functions


Syntax 5.1 Function Definition
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.

```
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;
}
```

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;
}
```

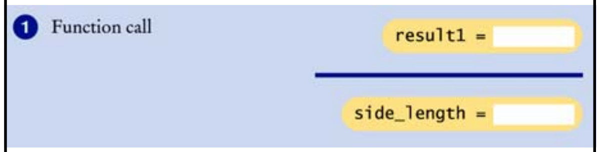
We'll keep up with their variables and parameters:

```
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.

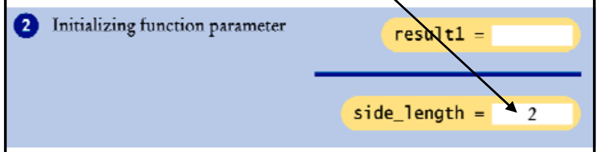
```
double result1 = cube_volume(2);
```



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.

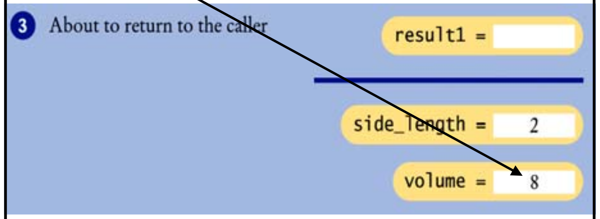
```
double result1 = cube_volume(2);
```



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);
```

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

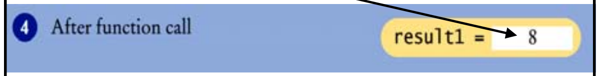
4 After function call
result1 =

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);
```



Return Values (5.4)

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 (5.5)

EX: 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.

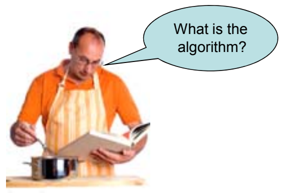
`void` functions are used to
simply do a sequence of instructions
– They do not return a value to the caller.

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;
}
```

Functions Without Return Values – The `void` Type

- Note that the previous function doesn't compute a 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
   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;
}
```

Stepwise Refinement (5.6)

- 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)

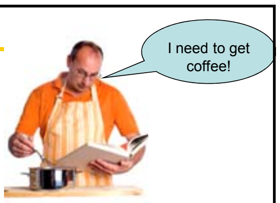
(and for then those steps that can be further broken, we'll break them)

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

… and so on…

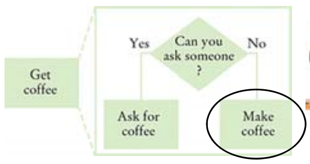
until the sub-problems are small enough to be just a few steps

Stepwise Refinement



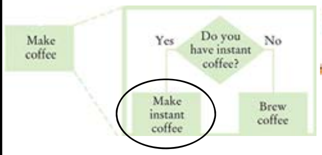
This is the whole problem: this is like **`main`**.

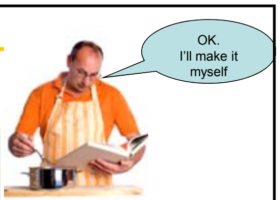
Stepwise Refinement



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)

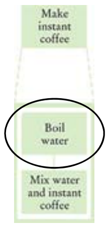
Stepwise Refinement

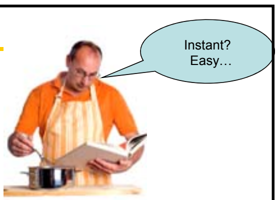




The make coffee sub-problem can be broken into:
if we have instant coffee, we can make that
but if not, we can brew coffee
(maybe these will have parts)

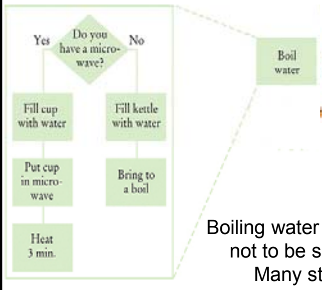
Stepwise Refinement

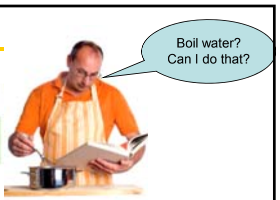




Making instant coffee breaks into:
1. Boil Water
2. Mix (stir if you wish)
(Do these have sub-problems?)

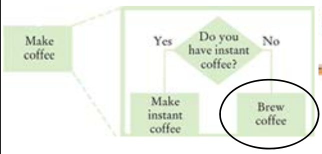
Stepwise Refinement

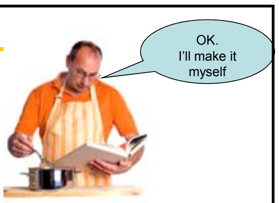




Boiling water appears not to be so easy.
Many steps,
but none have sub-steps.

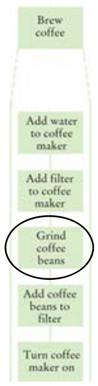
Stepwise Refinement

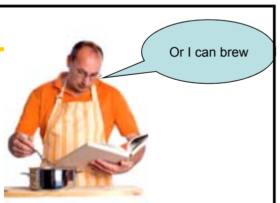




Going back to the branch between
instant or brew, we need to think about brewing.
Can we break that into parts?

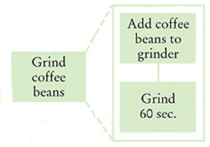
Stepwise Refinement

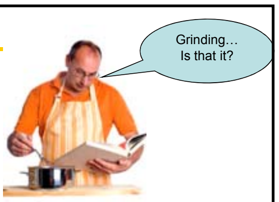

Brew coffee
Add water to coffee maker
Add filter to coffee maker
Grind coffee beans
Add coffee beans to filter
Turn coffee maker on



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

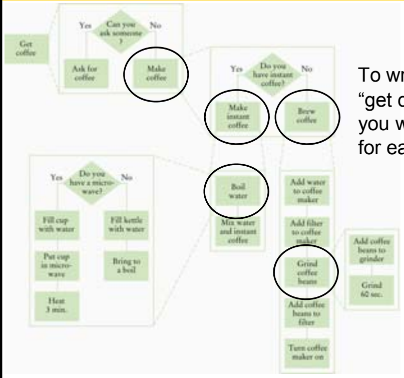
Stepwise Refinement





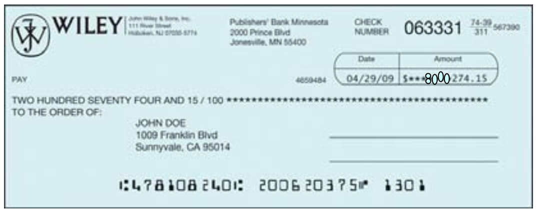
Grinding is a two step process
with no sub-sub-steps.

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

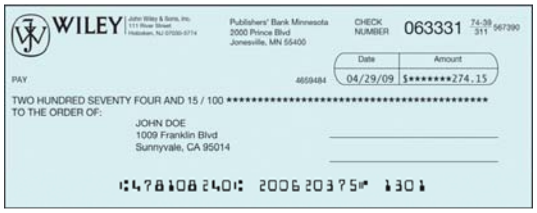
Date	Amount
04/29/09	$***8000274.15

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

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

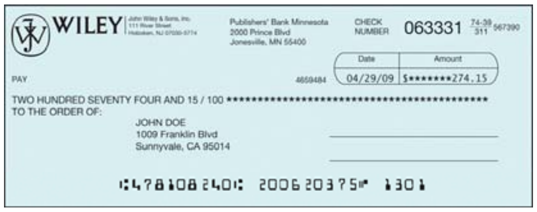
Date	Amount
04/29/09	$*******274.15

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

PAY 4659484

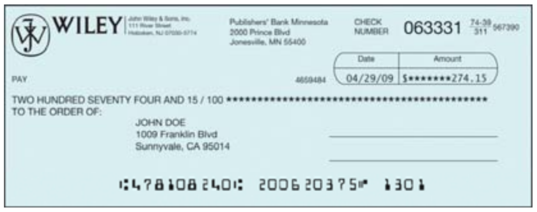
Date	Amount
04/29/09	$*******274.15

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

JOHN DOE
1009 Franklin Blvd
Sunnyvale, CA 95014

EX: Write a program to take an amount and produce the text.

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

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

JOHN DOE
1009 Franklin Blvd
Sunnyvale, CA 95014

We will 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 the name of number (e.g., "two hundred seventy four")
*/
string 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 the name of digit (“one” ... “nine”)
*/
string 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 the name of the number (“ten” ...
  “nineteen”)
*/
string 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 the name of the tens part of the number ("twenty"..."ninety")
*/
string 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)
name = "" (The name of the number starts as the empty string)
If part >= 100
{
    name = name of hundreds in part + " hundred"
    Remove hundreds from part
}

If part >= 20
{
    Append tens_name(part) to name
    Remove tens from part}
    Else if part >= 10
{
    Append teen_name(part) to name
    part = 0
}
If (part > 0)
{
    Append digit_name(part) to name
}
```

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)
{
   name = name + " " + tens_name(part);
   part = part % 10;
}
else if (part >= 10)
{
   name = name + " " + teen_name(part);
   part = 0;
}
if (part > 0)
{
   name = name + " " + 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)
{
    name = digit_name(part / 100) + " hundred";
    part = part % 100;
}
```

Stepwise Refinement – Pseudocode to C++

Now for the complete program.

ch04/sentinel.cpp

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

The Complete Program

ch04/sentinel.cpp

```
/**
   Turns a digit into its English name.
   @param digit = an integer between 1 and 9
   @return the name of digit ("one" ... "nine")
*/
string digit_name(int digit)
{
   if (digit == 1) return "one";
   if (digit == 2) return "two";
   if (digit == 3) return "three";
   if (digit == 4) return "four";
   if (digit == 5) return "five";
   if (digit == 6) return "six";
   if (digit == 7) return "seven";
   if (digit == 8) return "eight";
   if (digit == 9) return "nine";
   return "";
}
```

The Complete Program

ch04/sentinel.cpp

```
/**
   Turns a number between 10 and 19 into its English name.
   @param number = an integer between 10 and 19
   @return the name of the given number ("ten" ... "nineteen")
*/
string teens_name(int number)
{
   if (number == 10) return "ten";
   if (number == 11) return "eleven";
   if (number == 12) return "twelve";
   if (number == 13) return "thirteen";
   if (number == 14) return "fourteen";
   if (number == 15) return "fifteen";
   if (number == 16) return "sixteen";
   if (number == 17) return "seventeen";
   if (number == 18) return "eighteen";
   if (number == 19) return "nineteen";
   return "";
}
```

The Complete Program

ch04/sentinel.cpp

```
/**
   Gives the name of the tens part of a number between 20 and 99.
   @param number = an integer between 20 and 99
   @return the name of the tens part of the number ("twenty" ...
   "ninety")
*/
string tens_name(int number)
{
   if (number >= 90) return "ninety";
   if (number >= 80) return "eighty";
   if (number >= 70) return "seventy";
   if (number >= 60) return "sixty";
   if (number >= 50) return "fifty";
   if (number >= 40) return "forty";
   if (number >= 30) return "thirty";
   if (number >= 20) return "twenty";
   return "";
}
```

The Complete Program

ch04/sentinel.cpp

```
/**
   Turns a number into its English name.
   @param number = a positive integer < 1,000
   @return the name of the number (e.g. "two hundred seventy four")
*/
string int_name(int number)
{
   int part = number; // The part that still needs to be converted
   string name; // The return value

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

The Complete Program

ch04/sentinel.cpp

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

   if (part > 0)
   {
      name = name + " " + digit_name(part);
   }

   return name;
}

int main()
{
   cout << "Please enter a positive integer: ";
   int input;
   cin >> input;
   cout << int_name(input) << endl;
   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.

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.

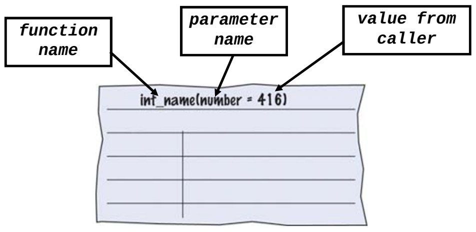
Tracing Functions

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:

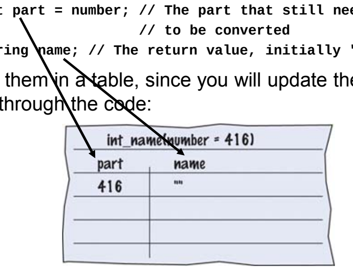


Tracing Functions

Then write the names and values of the function variables.

```
string int_name(int number)
{
   int part = number; // The part that still needs
                      // to be converted
   string name; // The return 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)
   {
      name = digit_name(part / 100) + " hundred";

      part = part % 100;
   }
```

Tracing Functions

`part / 100` is 4.

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

      part = part % 100;
   }
```

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

Tracing Functions

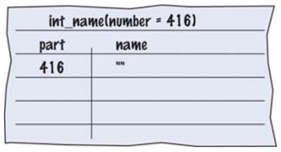
```
if (part >= 100)
   {
      name = digit_name(part / 100) + " hundred";

      part = part % 100;
   }
```

`part % 100` is 16.

Tracing Functions

`name` has changed to
`name + " " + digit_name(part / 100) + "hundred"`
which is the string "four hundred",

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



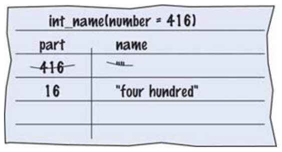
int_name(number = 416)	
part	name
416	""

Tracing Functions

`name` has changed to
`name + " " + 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.



int_name(number = 416)	
part	name
~~416~~	~~""~~
16	"four hundred"

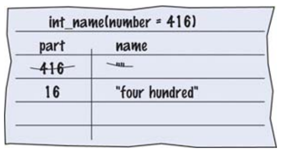
Tracing Functions

If **`digit_name`**'s parameter been complicated, you would have started *another* sheet of paper to trace that function call.

Your work table will probably be covered with sheets of paper (or envelopes) by the time you are done tracing!

Tracing Functions

Let's continue…
Here is the status of the parameters and variables now:



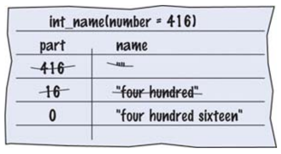
int_name(number = 416)	
part	name
~~416~~	~~""~~
16	"four hundred"

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) {
   name = name + " " + 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) {
   name = name + " " + teens_name(part);
   part = 0;
}

if (part > 0)
{
   name = name + " " + 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.

```
/**
Turns a digit into its English name.
@param digit = an integer between 1 and 9
@return the name of digit (“one” ... “nine”)
*/
string digit_name(int digit)
{
    return "mumble";
}
/**
Gives the name of the tens part of a number between 20 and 99.
@param number = an integer between 20 and 99
@return the tens name of the number (“twenty” ... “ninety”)
*/
string tens_name(int number)
{
    return "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 *eveyone* 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...

Variable Scope (5.7)

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 use the same name in different functions?

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

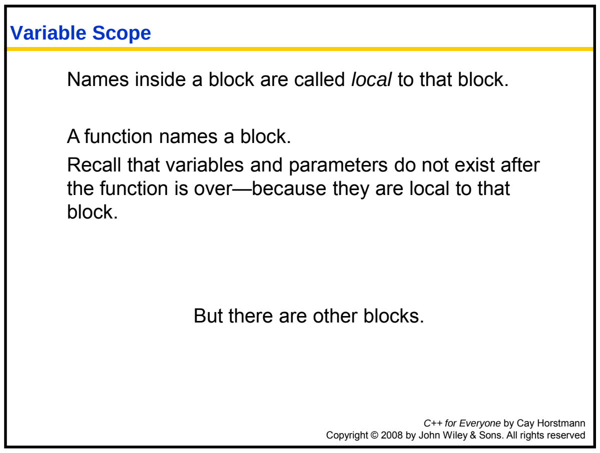
```
double cube_volume(double side_len)
{
  double volume = side_len * side_len * side_len;
  return volume;
}
int main()
{
  double volume = cube_volume(2);
  cout << volume << endl;
  return 0;
}
```

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.

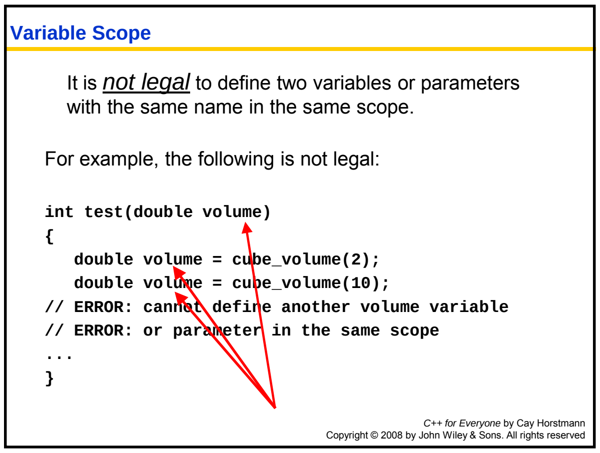


Variable Scope

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.

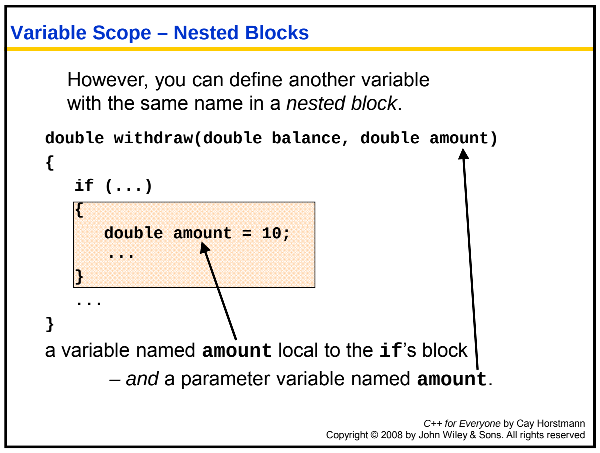


Variable Scope

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:

```
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
...
}
```

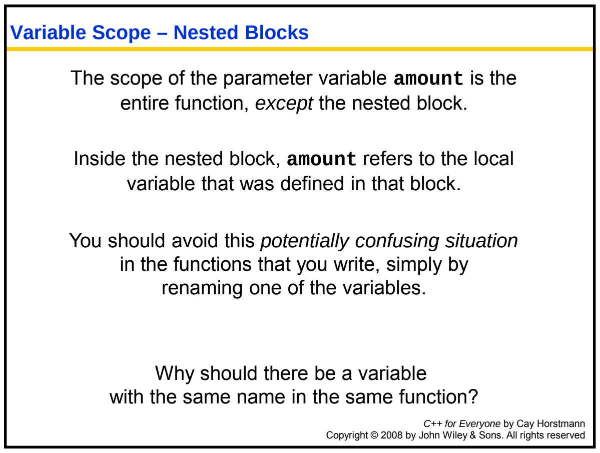


Variable Scope – Nested Blocks

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`**.



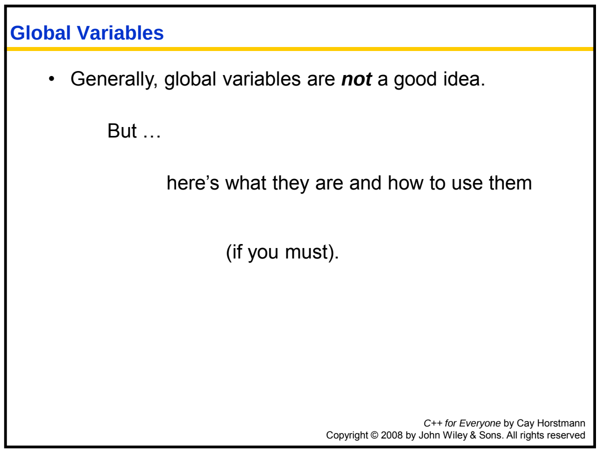
Variable Scope – Nested Blocks

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?



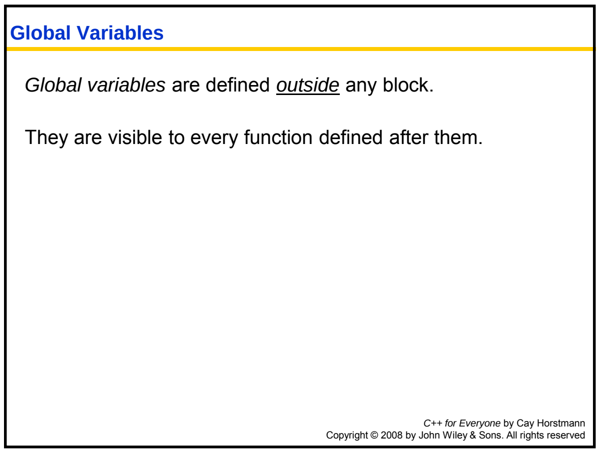
Global Variables

- 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

In some cases, this is a good thing:

The **`<iostream>`** header defines these global variables:

```
cin
cout
```

This is good because there should only be one of each of these and every function who needs them should have direct access to 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);
   cout << balance << endl;
   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 – written by group of programmers.

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

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.

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

Global Variables – Just Say “No”

You should *avoid* global variables in your programs!

Reference Parameters (5.8)

- Suppose you would like a function to get the user’s last name and ID number.
- The variables for this data are in your scope.
- But you want the function to change them for you.

- If you want to write a function that changes the value of a parameter, you must use a *reference parameter.*

Reference Parameters

To understand the need for a different kind of parameter, you must first understand why the parameters you now know do not work.

Reference Parameters

Consider a function that simulates withdrawing a given amount of money from a bank account, provided that sufficient funds are available.

If the amount of money is insufficient, a $10 penalty is deducted instead.

The function would be used as follows:

```
double harrys_account = 1000;
withdraw(harrys_account, 100);
     // Now harrys_account is 900
withdraw(harrys_account, 1000);
     // Insufficient funds.
     // Now harrys_account is 890
```

Reference Parameters

Here is a first attempt:

```
void withdraw(double balance, double amount)
{
   const int PENALTY = 10;
   if (balance >= amount)
   {
      balance = balance - amount;
   }
   else
   {
      balance = balance - PENALTY;
   }
}
```

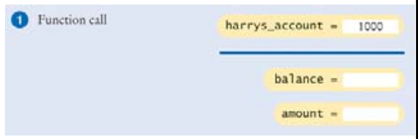
But this doesn’t work.

Reference Parameters

What is actually happening?

Let’s call the function passing in 100 to be taken from **harrys_account**.

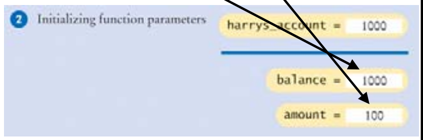
```
double harrys_account = 1000;

withdraw(harrys_account, 100);
```



Reference Parameters

The local variables, consts, and value parameters are initialized.

```
double harrys_account = 1000;
…
withdraw(harrys_account, 100);
…
void withdraw(double balance, double amount)
{
   const int PENALTY = 10;
```



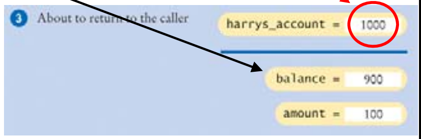
Reference Parameters

The test is **false**, the *LOCAL* variable **balance** is updated

```
double harrys_account = 1000;
…
   else
   {
      balance = balance - PENALTY;
   }
```

NOTHING happens to **harrys_balance** because it is a separate variable (in a different scope)

!



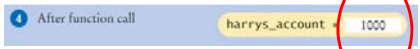
Reference Parameters

The function call has ended.
Local names in the function are gone and…

NOTHING happened to **harrys_balance**.

```
withdraw(harrys_account, 100);
```

!



Reference Parameters

A reference parameter refers to a variable that is supplied in a function call.

"refers" means that during the execution of the function, the reference parameter name is another name for the caller's variable.

This "referring" is how a function can change non-local variables**:**

changes to its local parameters' names cause changes to the callers variables they "refer" to.

Reference Parameters

To indicate a reference parameter, you place an **&** after the type name.

```
void withdraw(double& balance, double amount)
```

To indicate a value parameter, you do *not* place an **&** after the type name.

Reference Parameters

The type **double&** is pronouned:

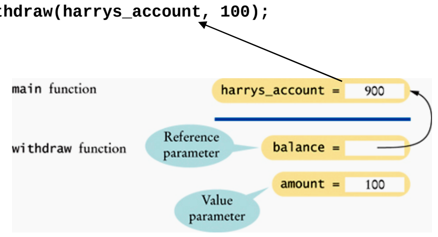
reference to double
or
double ref

(The type **double** is, of course, pronounced: *double)*

Reference Parameters

The parameter name **balance** “refers” to the caller’s variable named **harrys_account** so that changing **balance** changes **harrys_account**.

```
withdraw(harrys_account, 100);
```



Reference Parameters

A reference parameter must always be called with a variable.

It would be an error to supply a number:

```
withdraw(1000, 500);
       // Error: reference parameter must be a variable
```

The reason is clear—the function modifies the reference parameter, but it is impossible to change the value of a number.

Reference Parameters

```
/**
   Withdraws the amount from the given balance, or withdraws
   a penalty if the balance is insufficient.
   @param balance = the balance from which to make the
   withdrawal
   @param amount the amount to withdraw
*/
void withdraw(double& balance, double amount)
{
    const int PENALTY = 10;
    if (balance >= amount)
    {
        balance = balance - amount;
    }
    else
    {
        balance = balance - PENALTY;
    }
}
```

Reference Parameters

```
int main()
{
   double harrys_account = 1000;
   double sallys_account = 500;
   withdraw(harrys_account, 100);
       // Now harrys_account is 900
   withdraw(harrys_account, 1000); // Insufficient funds
       // Now harrys_account is 890
   withdraw(sallys_account, 150);
   cout << "Harry's account: " << harrys_account << endl;
   cout << "Sally's account: " << sallys_account << endl;

   return 0;
}
```

Reference Parameters

For the same reason, you cannot supply an expression:

```
withdraw(harrys_account + 150, 500);
// Error: reference parameter must be a variable
```

Prefer Return Values to Reference Parameters

Some programmers use reference parameters as a mechanism for setting the result of a function.

For example:

```
void cube_volume(double side_length, double& volume)
{
    volume = side_length * side_length * side_length;
}
```

However, this function is less convenient than our previous **cube_volume** function.

Prefer Return Values to Reference Parameters

```
void cube_volume(double side_length, double& volume)
{
      volume = side_length * side_length * side_length;
}
```

This function cannot be used in expressions such as:

`cout << cube_volume(2)`

Prefer Return Values to Reference Parameters

Another consideration is that the `return` statement can return only one value.

If caller wants more than two values, then the only way to do this is with reference parameters (one for each wanted value).

CHAPTER SUMMARY

1. A function is a named sequence of instructions.
2. Parameter values are supplied when a function is called.
   The return value is the result that the function computes.
3. When defining a function, you provide a name for the function, a name and type for each parameter, and a type for the result.
4. Function comments explain the purpose of the function, the meaning of the parameters and return value, as well as any special requirements.
5. Parameter variables hold the parameter values supplied in the function call.
6. The `return` statement terminates a function call and yields the function result.

CHAPTER SUMMARY continued

7. Turn computations that can be reused into functions.
8. Use a return type of `void` to indicate that a function does not return a value.
9. Use the process of stepwise refinement to decompose complex tasks into simpler ones.
10. The scope of a variable is the part of the program in which it is visible.
11. A local variable is defined inside a function.
    A global variable is defined outside a function.
12. A reference parameter refers to a variable that is supplied in a function call.