



Chapter Five: Functions II

Lecture Goals

- 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

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



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

Stepwise Refinement



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

Stepwise Refinement



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

Stepwise Refinement



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

(Darn!)

Stepwise Refinement



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

(Darn!)

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 < 1000  
@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 $< \$1000$ 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 \geq 100

 name = name of hundreds in part + " hundred"
 Remove hundreds from part

If part \geq 20

 Append tens_name(part) to name
 Remove tens from part.

Else if part \geq 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.
- 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.

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

ch05/intname.cpp

The Complete Program

```
/**
 * 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

```
/**
 * 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

```
/**
 * 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 "";
}
```

```
/**
 * 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;
    }
}
```

```
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.

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



*Whew!
That 'breaking'
word always
scares me*

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.

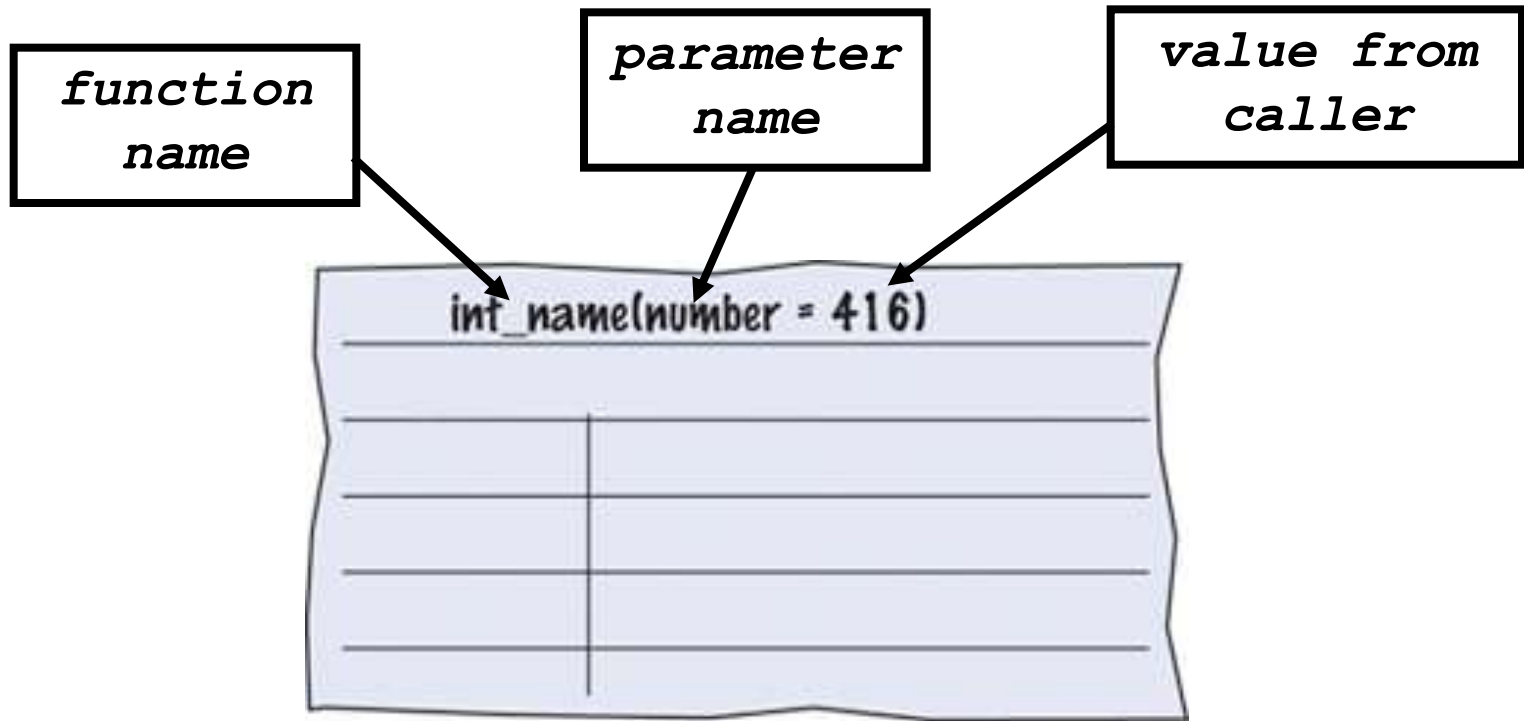
Tracing Functions

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.

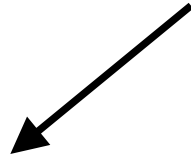
```
string int_name(int number)
{
    int part = number; // The part that still needs
                       // to be converted
    string name; // The return value, initially ""
```

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

int_name(number = 416)	
part	name
416	""

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.

<code>int_name(number = 416)</code>	
<code>part</code>	<code>name</code>
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 had 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	int
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 be the result:

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

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

(OK, only you 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



?



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);
    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.

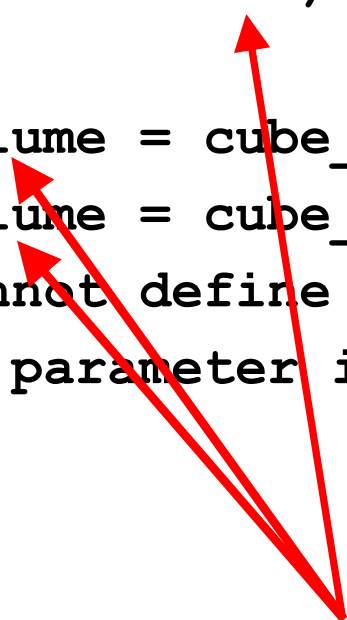
But there are other blocks.

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:

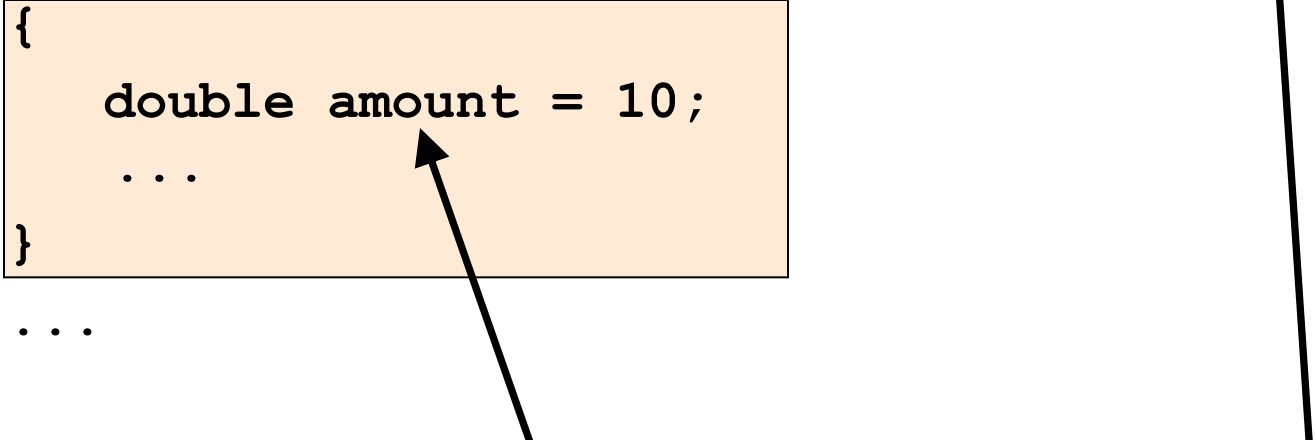
```
int test(double volume)           ERRORS!!!
{
    double volume = cube_volume(2);
    double volume = cube_volume(10);
// ERROR: cannot define another volume variable
// ERROR: or parameter in the same scope
...
}
```



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

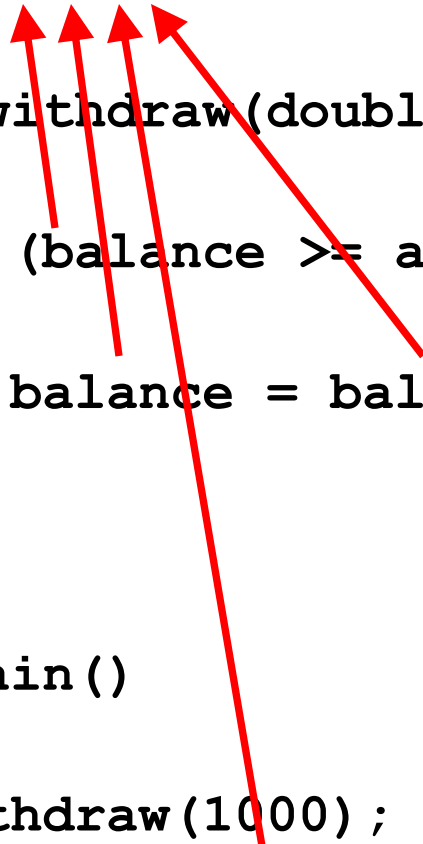
But in a banking program, how many functions should have direct access to a balance variable?

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 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

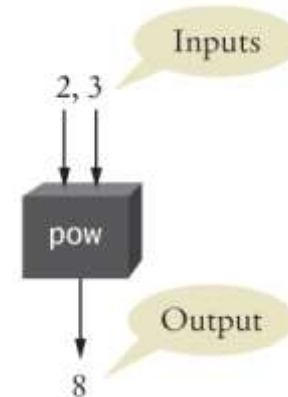
When multiple functions update global variables, the result can be *difficult* to predict.

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.

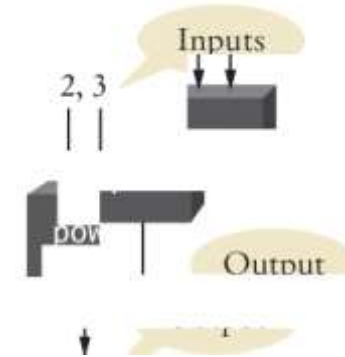
Programs with global variables are difficult to maintain and extend because you can no longer view each function as a “black box” that simply receives parameter values and returns a result or does something.



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?

You should *avoid* global variables in your programs!

Reference Parameters

- 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 double PENALTY = 10;
    if (balance >= amount)
    {
        balance = balance - amount;
    }
    else
    {
        balance = balance - PENALTY;
    }
}
```

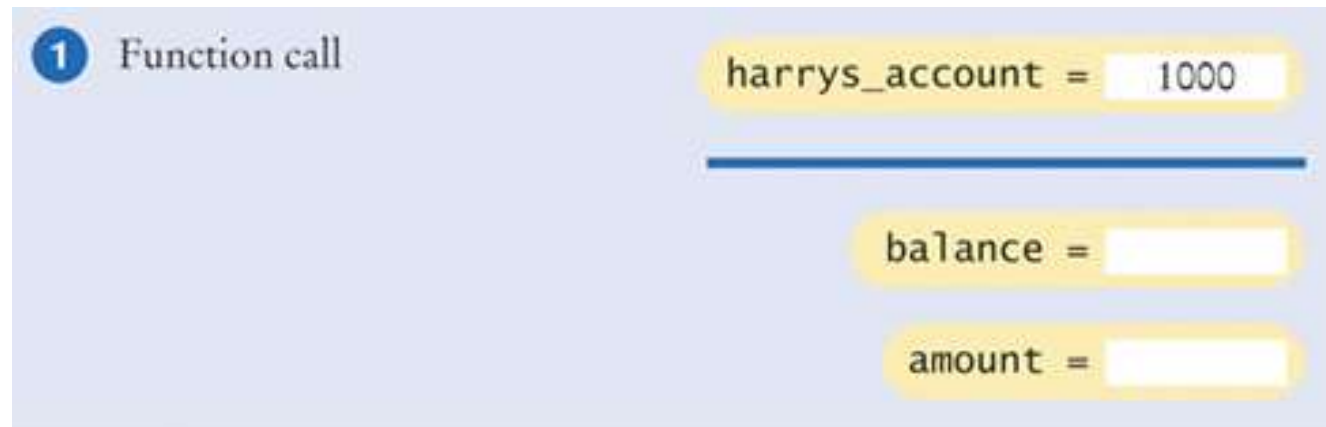
But this doesn't work.

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;
```

...

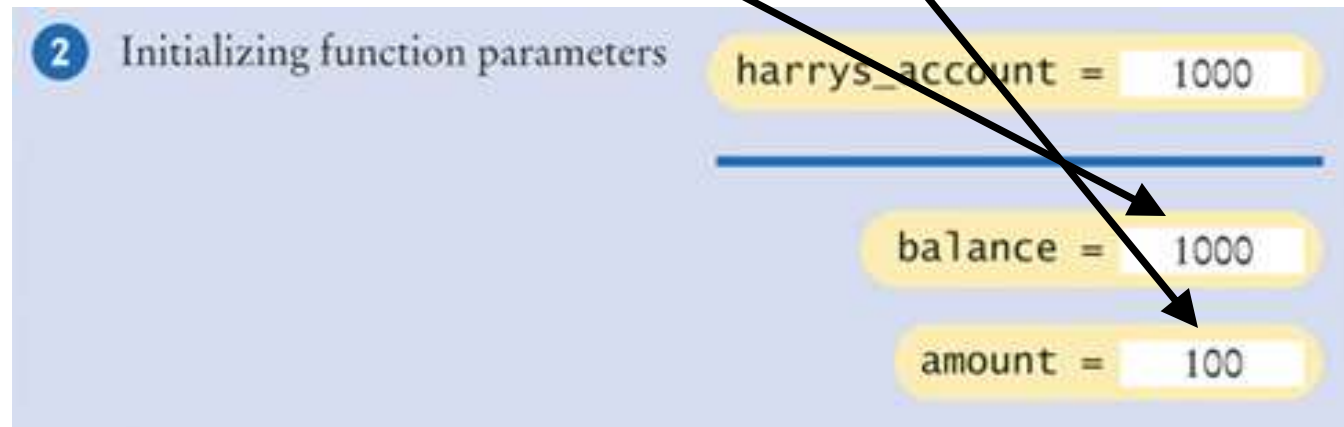
```
withdraw(harrys_account, 100);
```

...

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

```
{
```

```
    const int PENALTY = 10;
```

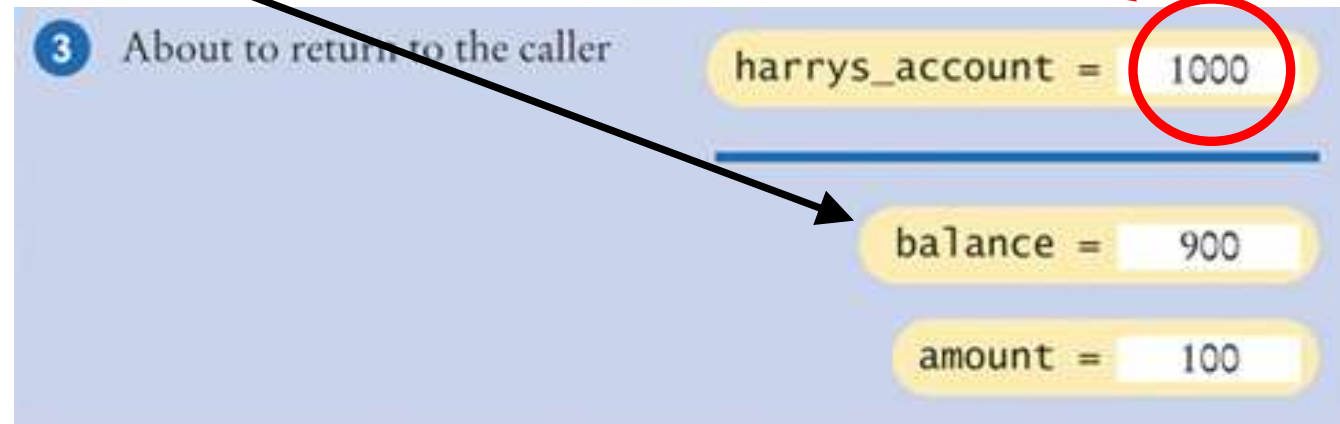


Reference Parameters

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

```
double harrys_account = 1000;  
...  
else  
{  
    balance = balance - amount;  
}
```

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

Here is correct code, using reference parameters:

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

Let's see this in action.

Reference Parameters

Now, using the function with reference parameters let's again call the function passing in 100 to be taken from `harrys_account`.

```
double harrys_account = 1000;  
    ↓  
withdraw(harrys_account, 100);
```

1 Function call

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

harrys_account = 1000

balance =

amount =

Reference Parameters

Notice that `balance` now refers to `harrys_account`.

```
double harrys_account = 1000;
```

...

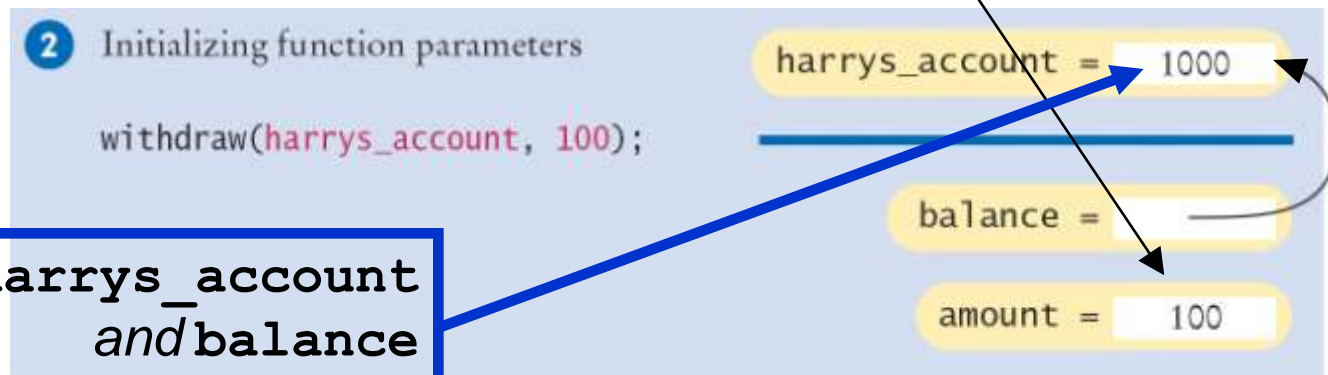
```
withdraw(harrys_account, 100);
```

...

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

```
{
```

```
    const int PENALTY = 10;
```



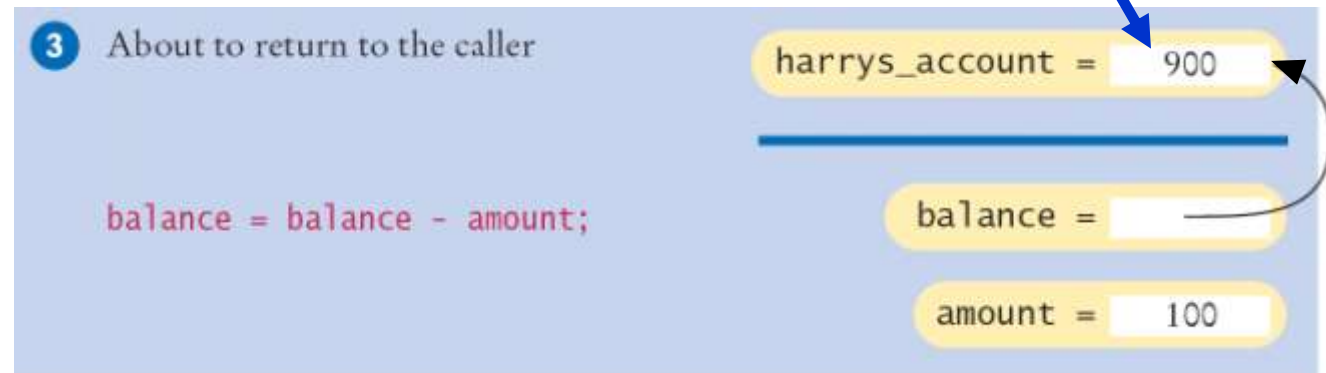
`harrys_account`
and `balance`
are both *names*
for *this memory*

Reference Parameters

The test is **false**, the variable `harrys_account` is updated when the reference parameter `balance` is assigned to.

```
double harrys_account = 1000;  
...  
else  
{  
    balance = balance - amount;  
}
```

recall that
`harrys_account`
and `balance`
are **both** *names*
for *this* memory



Reference Parameters

The function call has ended.

Local names in the function are gone and...

harrys_balance was correctly changed!

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

4 After function call

harrys_account = 900



```
/**
    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 double PENALTY = 10;
    if (balance >= amount)
    {
        balance = balance - amount;
    }
    else
    {
        balance = balance - PENALTY;
    }
}
```



```
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

The type `double&` is pronounced:

reference to double

or

double ref

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

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

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).

Constant References

It is not very efficient to have a value parameter that is a large object (such as a string value).

Copying the object into a parameter variable is less efficient than using a reference parameter.

With a reference parameter, only the location of the variable, not its value, needs to be transmitted to the function.

Constant References

You can instruct the compiler to give you the efficiency of a reference parameter and the meaning of a value parameter, by using a *constant reference*:

```
void shout(const string& str)
{
    cout << str << "!!! " << endl;
}
```

This is a bit more efficient than having `str` be a value parameter.

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.

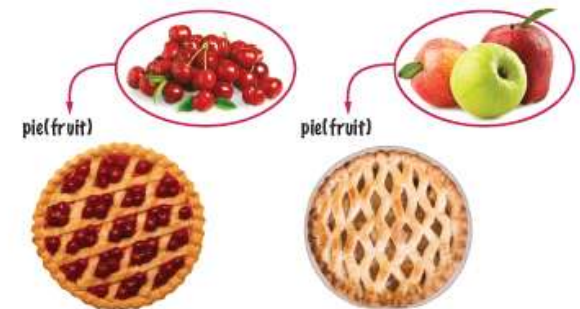
Be able to implement functions.



- 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 Functions II