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"

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?

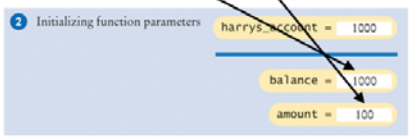
Let's call the function passing in 100 to be taken from `harrys_account`.

```
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;
```

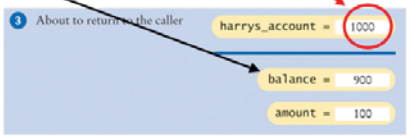


Reference Parameters

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)



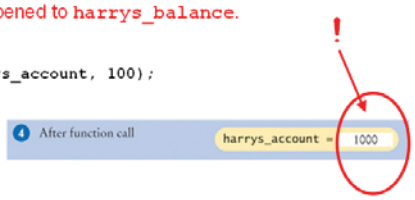
Reference Parameters

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.

like two names for same variable location/storage

Reference Parameters

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

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

use the & in the function name line

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

Reference Parameters

The type `double&` is pronounced:

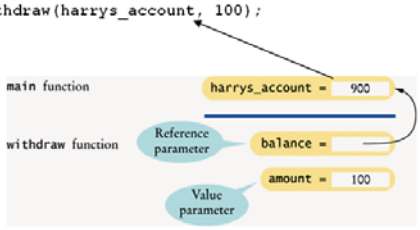
reference to double
or
double ref

(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:

must send a variable to a reference parameter

```
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;
}
```

will output: Harry's account: 890
Sally's account: 350

Reference Parameters

For the same reason, you cannot supply an expression:

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

this slide should be after 2 slides back

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.

(two or more values)

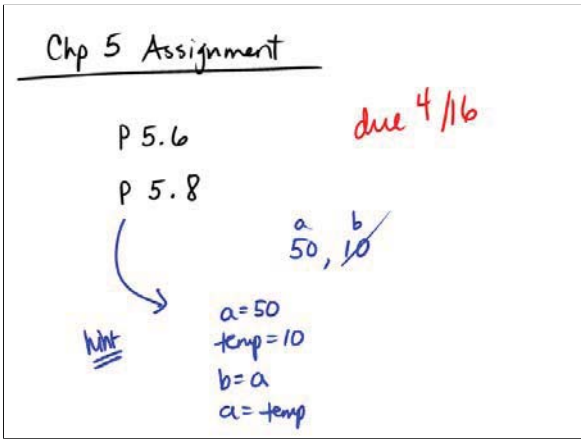
If caller wants more than ~~two~~ values, then the only way to do this is with reference parameters (one for each wanted value).

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.


Chp 5 Assignment
P 5.6
P 5.8
due 4/16
a b
50, 10
hint
a=50
temp=10
b=a
a=temp