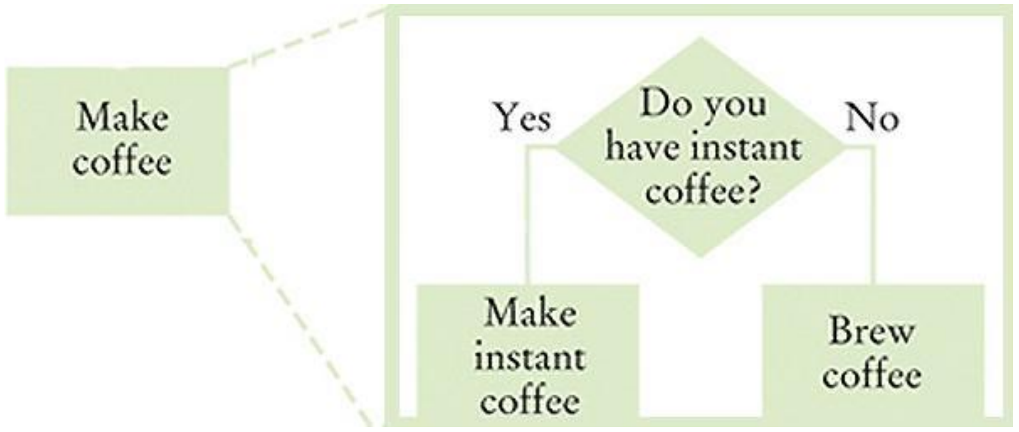
Topic 7

1. Functions as black boxes
2. Implementing functions
3. Parameter passing
4. Return values
5. Functions without return values
6. Reusable functions
7. <u>Stepwise refinement</u>
8. Variable scope and globals
9. Reference parameters
10. Recursive functions

Stepwise Refinement

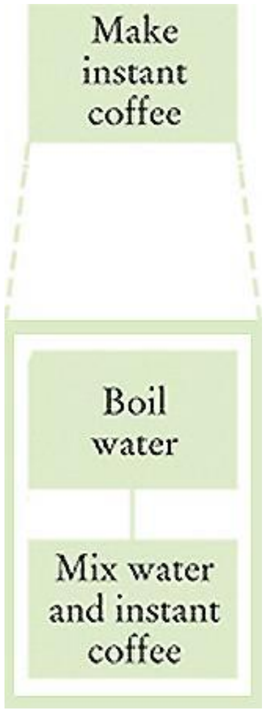
- One of the most powerful strategies for problem solving is the process of *stepwise refinement.*
- To solve a difficult task, break it down into simpler tasks.
- Then keep breaking down the simpler tasks into even simpler ones, until you are left with tasks that you know how to solve.

Stepwise Refinement Example: Coffee Making



The “make coffee” 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 #2

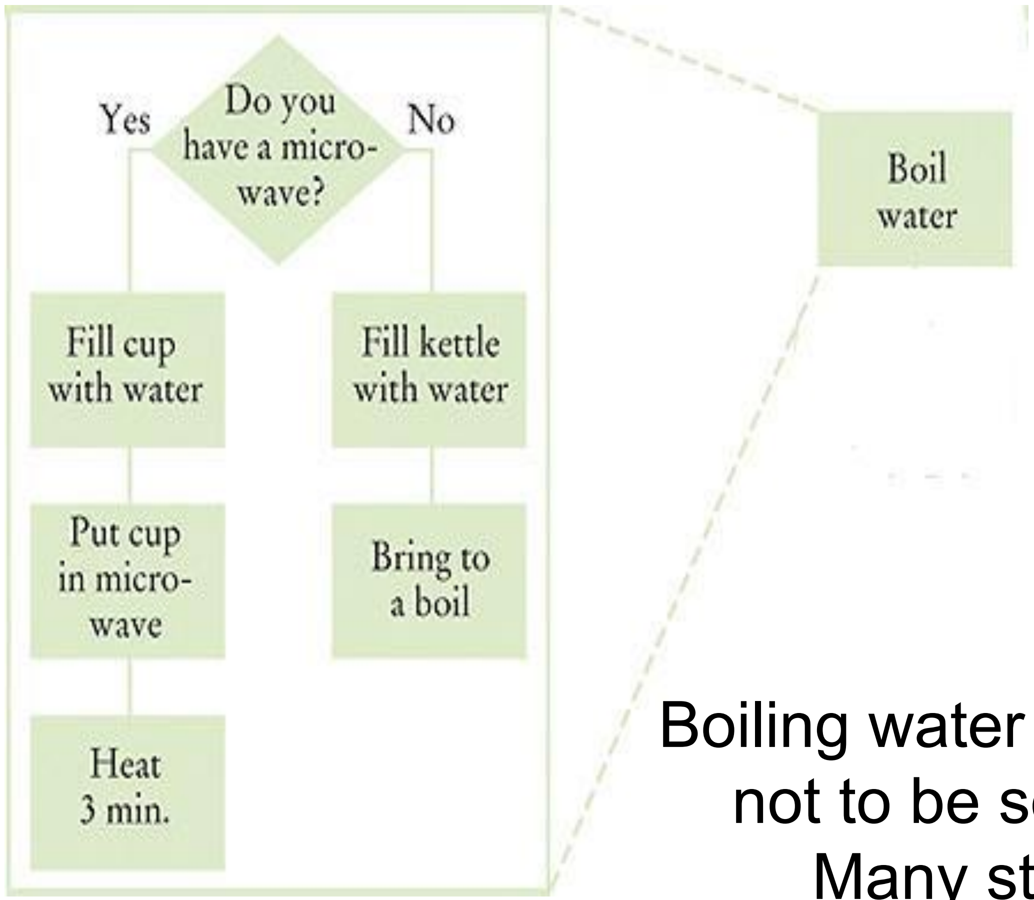


Making instant coffee breaks into:

1. Boil Water
2. Mix (stir if you wish)

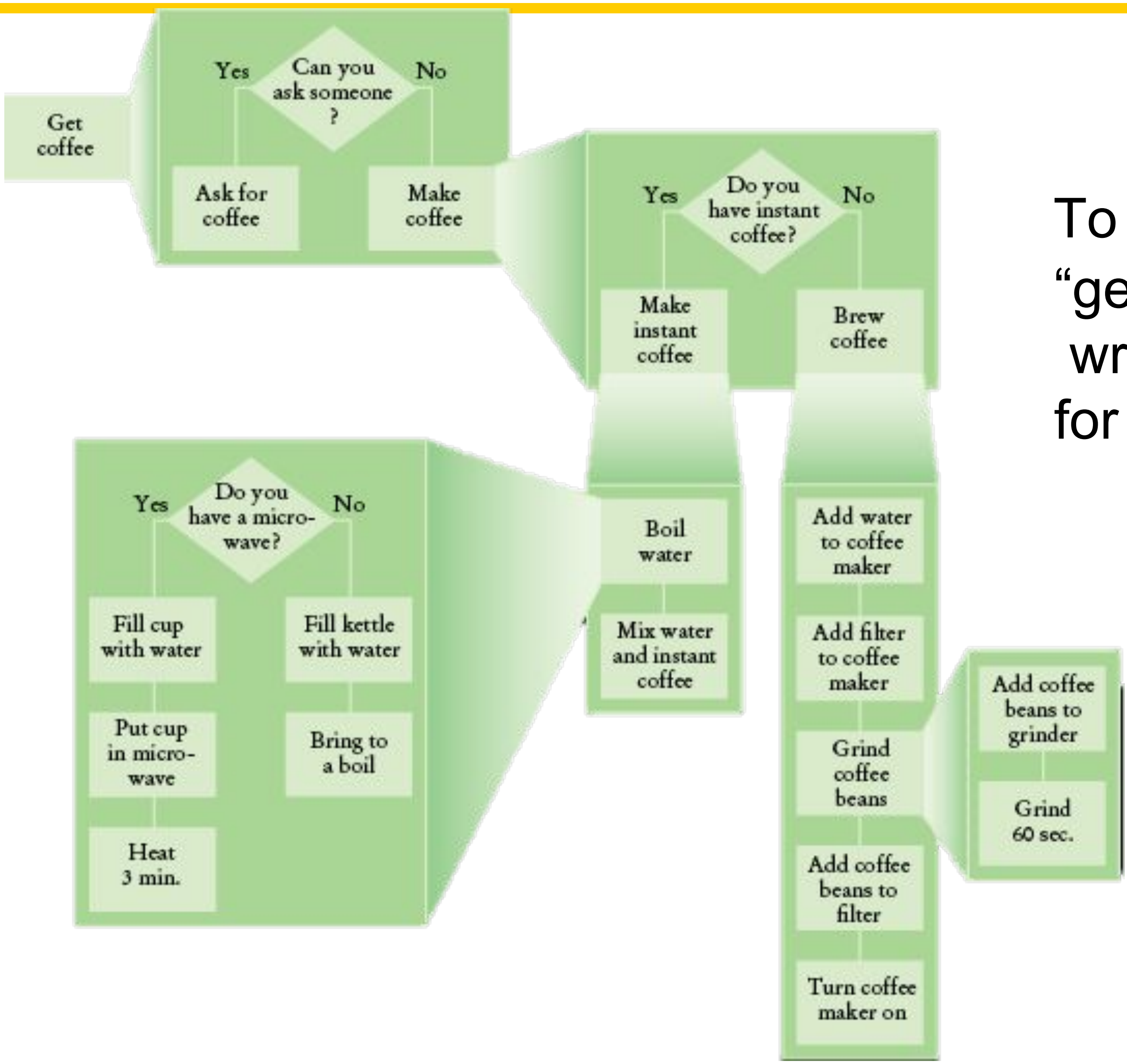
(Do these have sub-problems?)

Stepwise Refinement #3



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

Stepwise Refinement – The Complete Process Shown



To write the
“get coffee” program,
write functions
for each sub-problem.

Stepwise Refinement Example #2: Paycheck Printing

We will write a program to take a dollar amount as a `int` input and produce the text equivalent of the $$ amount, to print the English amount line on a check.

Sub Problem #1: `int` to text 0…9

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

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, turn that task into a function:

```
/**
   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: Another Function for Teens

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: Add a Function for Tens

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: Hundreds

- 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 – Analyzing 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 – 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++ (part 2)

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;
}
```

The Complete Code for the Check Printer (part 1)

```
// ch05/intname.cpp
#include <iostream>
#include <string>
using namespace std;
/**
   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 Code for the Check Printer (part 2)

```
/**
   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 Code for the Check Printer (part 3)

```
/**
   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 Code for the Check Printer (part 4)

```
/**
   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 Code for the Check Printer (part 5)

```
   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.
  - So it's tempting to write long functions to minimize their number and the overhead

- BUT as a rule of thumb, a function that is too long to fit on a single screen should be broken up.
  - into other functions
  - *Long functions are hard to understand and to debug*

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.

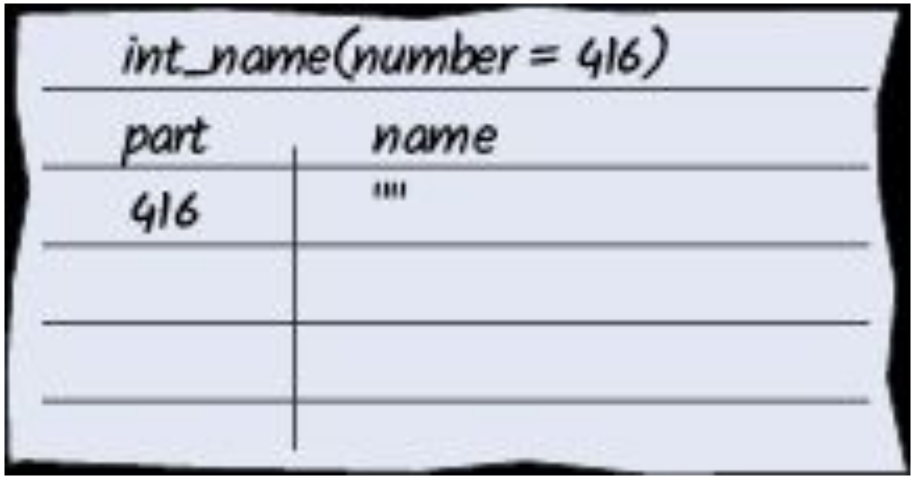
You should trace each of your functions separately.

Tracing Functions: Example with `int_name` function

Here is the call: `... int_name(416) ...`

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

Take an index card and write the name of the function and the names and values of the parameter variables, plus a table to show variable values at each step:



int_name(number = 416)

part	name
416	""

Tracing Functions: Midway Through

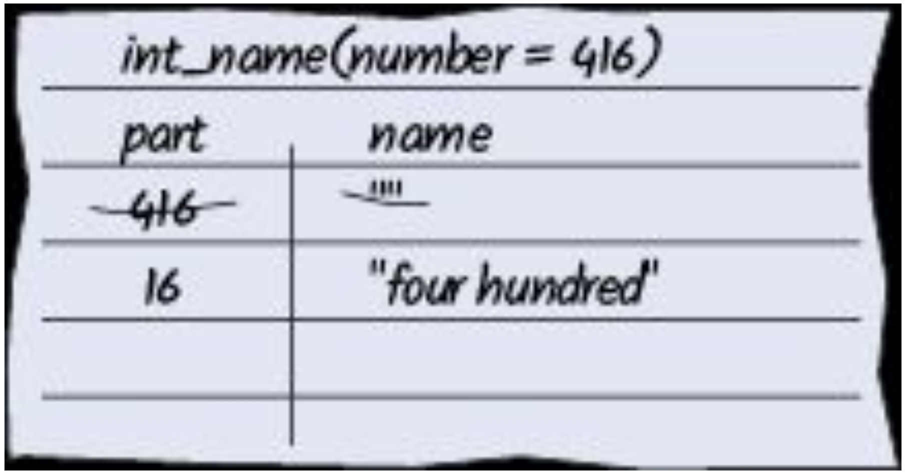
`name` has changed to

`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: Add a new Card for Each Function Called

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: An Error Found

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, don’t try to implement and test all functions at once.

- Temporarily implement the functions yet to be written as trivial “*stubs*”
  - A stub is a function that returns a simple value that is sufficient for testing another function.
  - It might also write a debug message on the screen to help you see the order of execution.

Stub Examples

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";
}
```

Stub Execution

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 shows that the basic logic of the **`int_name`** function is working correctly.

Now that you have tested **`int_name`**, you would "unstubify" another stub function, then another...