Parameter Passing

3. The function computes the expression `side_length * side_length * side_length`, which has the value 8. That value is stored in the local variable `volume`.

```
[inside the function]
double volume = side_length * side_length * side_length;
```

3 About to return to the caller

`result1 =`

`side_length = 2`

`volume = 8`

Parameter Passing

4. The function returns. All of its variables are removed. The return value is transferred to the caller, that is, the function calling the `cube_volume` function.

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

4 After function call

`result1 =`

The function executed: `return volume;`
which gives the caller the value 8

Parameter Passing

4. The function returns. All of its variables are removed. The return value is transferred to the caller, that is, the function calling the `cube_volume` function.

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

the returned 8 is about to be stored

4 After function call

`result1 =`

The function is over.
`side_length` and `volume` are gone.

Parameter Passing

The caller stores this value in their local variable `result1`.

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

4 After function call

`result1 = 8`

Return Values (5.4)

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;
}
```

option: combine calculation & return stmt into one stmt

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;
   }
   else
   {
      return 0;
   }
}
```

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) aka Subroutines

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:

tells compiler/computer function will do something not return a value

```
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*. (aka Subroutine)

```
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");
```

to call a void function list its name and parameter(s) value(s) on a line by itself

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();
   if (n == 0) { return; }        // Return immediately
   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

EX:

Get coffee

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

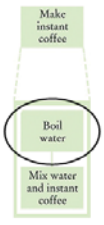
Stepwise Refinement

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

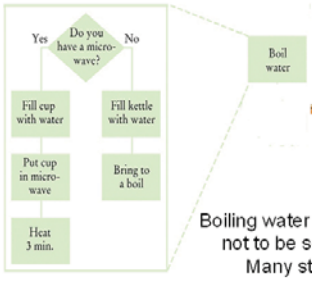
The make coffee sub-problem can be broken into:
if we have instant coffee, we can make that
but if not, we can brew coffee
(maybe these will have parts)

Stepwise Refinement



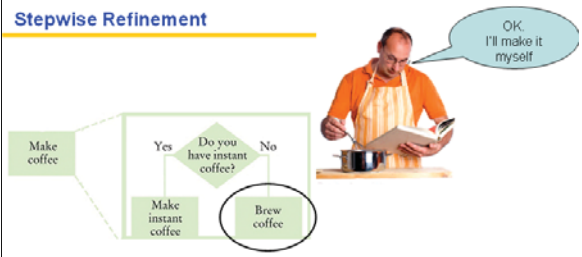
Making instant coffee breaks into:
1. Boil Water
2. Mix (stir if you wish)
(Do these have sub-problems?)

Stepwise Refinement



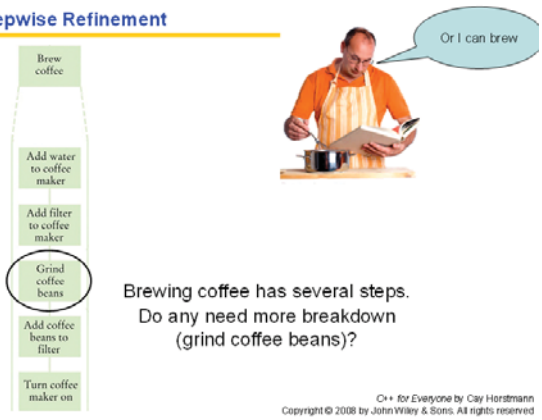
Boiling water appears
not to be so easy.
Many steps,
but none have sub-steps.

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



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

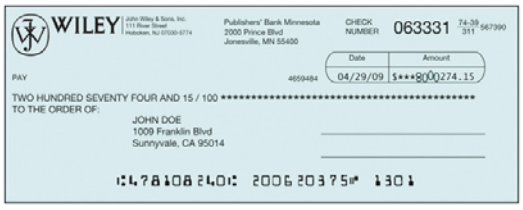
Stepwise Refinement

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 Example



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

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

Stepwise Refinement

We will write 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.

$0 to $999

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

Stopped

- 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.