Software Design and Analysis for Engineers

by

Dr. Lesley Shannon

Email: lshannon@ensc.sfu.ca

Course Website: http://www.ensc.sfu.ca/~lshannon/courses/ensc251

Simon Fraser University

Slide Set: 7

Date: October 5, 2015

What we’re learning in this Slide Set:

- Recursion

Textbook Chapters:

Relevant to this slide set:

- Chapter 14

Coming soon:

- Chapter 15 & 16

In CMPT 128, you were previously introduced to recursion.

Recursive Definitions define an object in terms of itself

- For example: The GNU Acronym stands for “GNU Not Unix”

This is also a common method of defining solutions to mathematical and computational problems:

- The Fibonacci sequence defines f(n) in terms of f(n-1) and f(n-2)

Recursive functions have their own problems, so use them wisely.

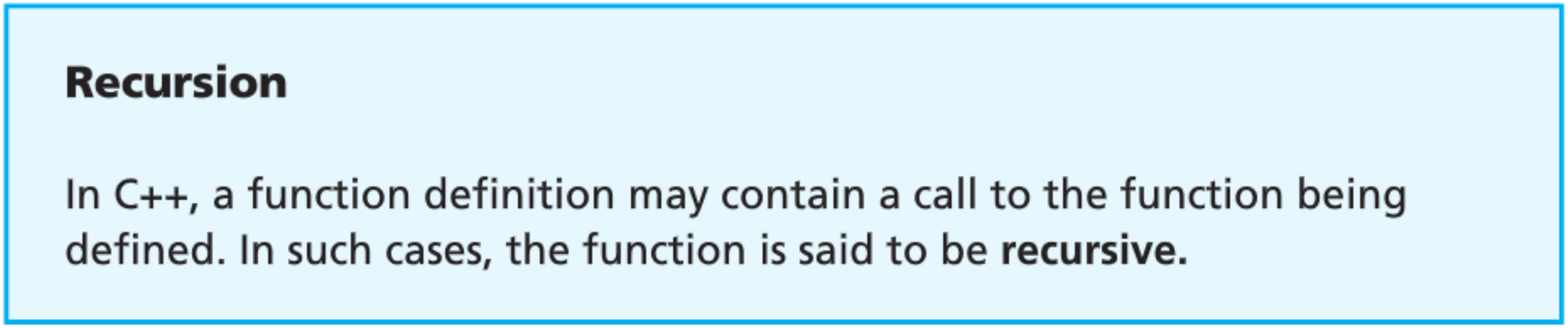
- Think: What do you need to worry about?

When you are designing a function to solve a task, one technique has you break the function down into a series of subtasks that must be completed to complete the function

- Sometimes one or more of these tasks is a recurrence of the original function on a smaller version/subset of the input data.

For example, if you wish to search an array for a particular value, you could subdivide into halves and then search these smaller arrays for the value

- You could then repeat this task of halving the array and searching the smaller versions until you found the value (or determined that it wasn't in the array)



Recursion

In C++, a function definition may contain a call to the function being defined. In such cases, the function is said to be **recursive.**

Essentially, you are decomposing the problem into a series of sub-problems and this can be a very useful technique for solving algorithmic problems.

- Think of our array search problem in Lab Assignment 1
  - We'll look at a solution at the end of this slide set

Here's an example from the text book where a recursive void function writes numbers to the screen vertically:

```
void write_vertical(int n);
//Precondition: n >= 0.
//Postcondition: The number n is written to the screen
//vertically with each digit on a separate line.
```

The simple case (base case) is if `n` is one digit (i.e. `n<10`), then simply write `n` to the screen.

What if `n` is comprised of more than one digit?

- First We need to print all the higher digits to the screen
- Then we need to print the last digit to the screen (i.e. `n<10`)

So the pseudocode for our solution is:

```
if (n < 10)
{
    cout << n << endl;
}
else //n is two or more digits long:
{
    write_vertical(the number n with the last digit removed);
    cout << the last digit of n << endl;
}
```

Recursive subtask

How do we calculate "the number n with the last digit removed?"

How do we calculate the last digit of n?

Hint: These are integers.

Let’s look at the pseudocode for an alternate version of:

```
void write_vertical(int n);
//Precondition: n >= 0.
//Postcondition: The number n is written to the screen
//vertically with each digit on a separate line.
```

Step 1: Output the first digit of n

Step 2: Output the number n with the first digit removed.

This is a valid decomposition that can be implemented recursively, but our other definition is easier. Why?

An important concept to keep in mind when choosing your decomposition is that eventually you want to have a subcase that is not a recursive call (like the base case in an inductive proof).

A successful definition of a recursive function requires at least one case that does not involve a recursive call as well as one or more cases that do require recursive calls.

So how do we code a recursive function and how does it actually execute at run time?

```
void write_vertical(int n);
//Precondition: n >= 0.
//Postcondition: The number n is written to the screen vertically
//with each digit on a separate line.

int main( )
{
    cout<< "write_vertical(3):" <<endl;
    write_vertical(3);
    cout<< "write_vertical(12):" <<endl;
    write_vertical(12);

    cout<< "write_vertical(123):" <<endl;
    write_vertical(123);

    return 0;
}

//uses iostream:
void write_vertical(int n)
{
    if (n < 10)
    {
        cout << n << endl;
    }
    else //n is two or more digits long:
    {
        write_vertical(n / 10);
        cout << (n % 10) << endl;
    }
}
```

Let’s look at what happens when we call: `write_vertical(3);`

What parts of the function get exercised?

```
void write_vertical(int n)
{
    if (n < 10)
    {
        cout << n << endl;
    }
    else //n is two or more digits long:
    {
        write_vertical(n / 10);
        cout << (n % 10) << endl;
    }
}
```

Let’s look at what happens when we call: `write_vertical(12);`

What parts of the function get exercised?

```
void write_vertical(int n)
{
    if (n < 10)
    {
        cout << n << endl;
    }
    else //n is two or more digits long:
    {
        write_vertical(n / 10);
        cout << (n % 10) << endl;
    }
}
```

To understand what's happening, let's trace : `write_vertical(123);`

Here's our function:

```
void write_vertical(int n)
{
    if (n < 10)
    {
        cout << n << endl;
    }
    else //n is two or more digits long:
    {
        write_vertical(n / 10);
        cout << (n % 10) << endl;
    }
}
```

What happens in the first call?

To understand what's happening, let's trace : `write_vertical(123);`

```
if (123 < 10)
{
    cout << 123 << endl;
}
else //n is two or more digits long:
{
    write_vertical(123 / 10);  <——
    cout << (123 % 10) << endl;
}
```

Computation will stop here until the recursive call returns.

Since 123 is greater than 10, the "else" case of the if statement gets exercised.

Once the new function call happens, execution of this current function call is halted.

To understand what's happening, let's trace : `write_vertical(123);`

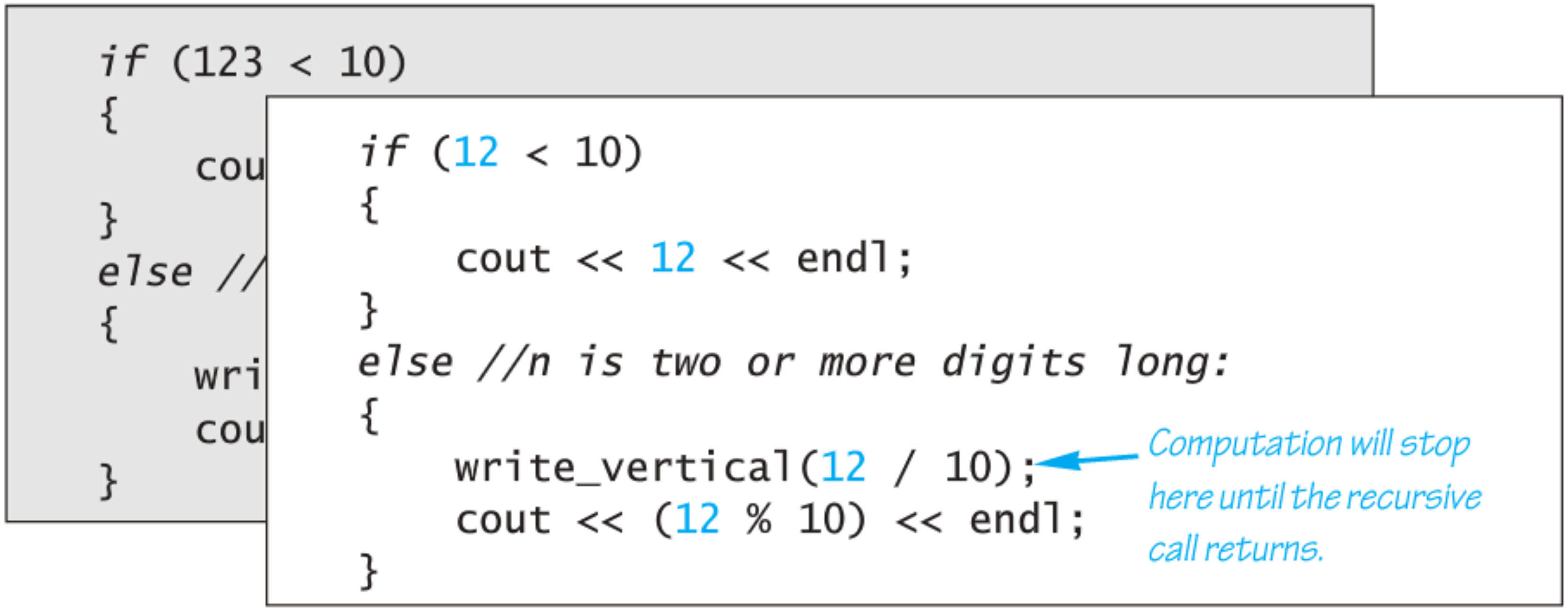
```
if (123 < 10)
{
    cout << 123 << endl;
}
else //n is two or more digits long:
{
    write_vertical(123 / 10);
    cout << (123 % 10) << endl;
}
```

Computation will stop here until the recursive call returns.

Since 123 is greater than 10, the "else" case of the if statement gets exercised.

Once the new function call `write_vertical(123 / 10);` (`write_vertical(n / 10);`) happens, execution of this current function call is halted.

To understand what's happening, let's trace : `write_vertical(123);`



```
if (123 < 10)
{
    cou
}
else //
{
    wri
    cou
}
```

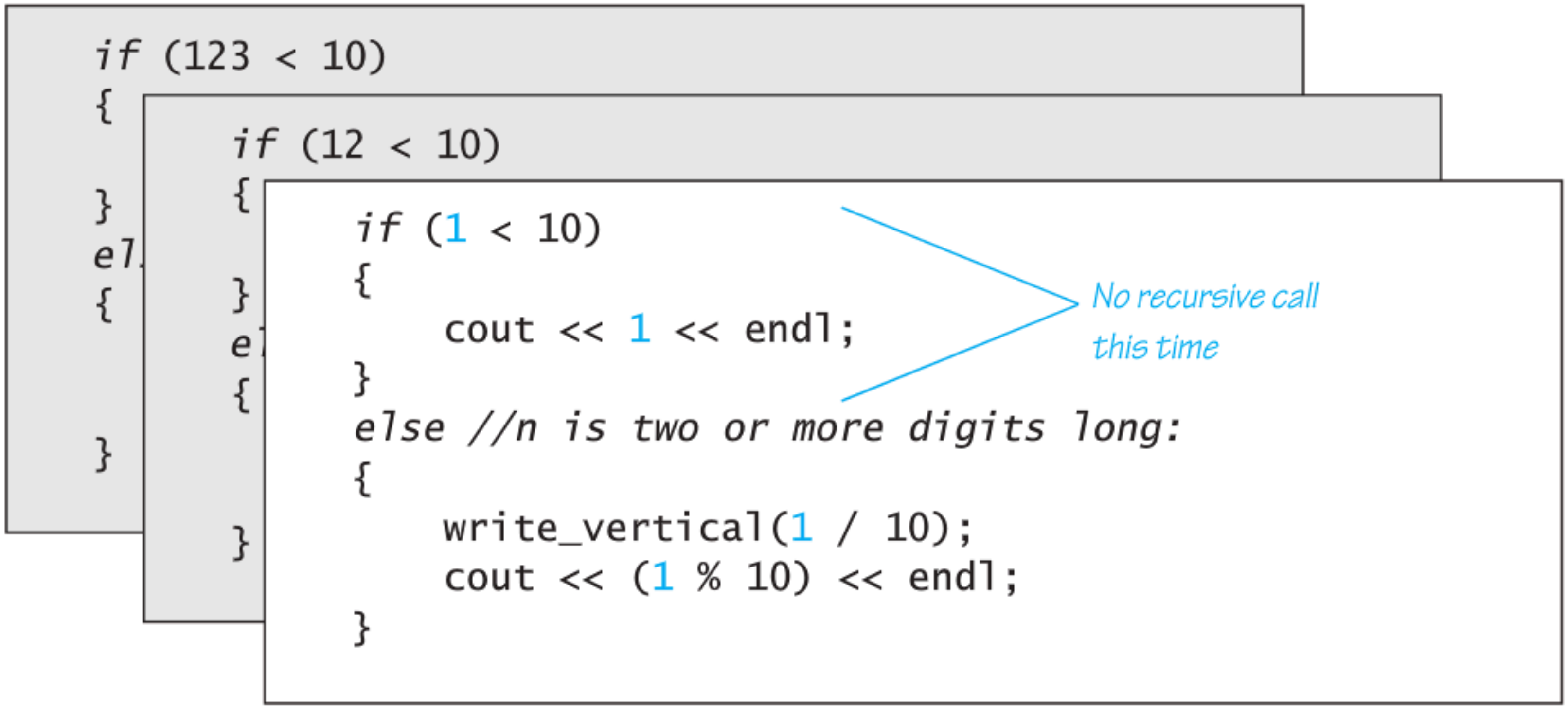
```
if (12 < 10)
{
    cout << 12 << endl;
}
else //n is two or more digits long:
{
    write_vertical(12 / 10);
    cout << (12 % 10) << endl;
}
```

Computation will stop here until the recursive call returns.

Since 123/10 is equivalent to 12 in integer mathematics,
`write_vertical(123 / 10);` becomes `write_vertical(12);`

Again, with this recursive call, the else case gets exercised (12 >10)

To understand what's happening, let's trace : `write_vertical(123);`



```
if (123 < 10)
{

}
el
{

}
```

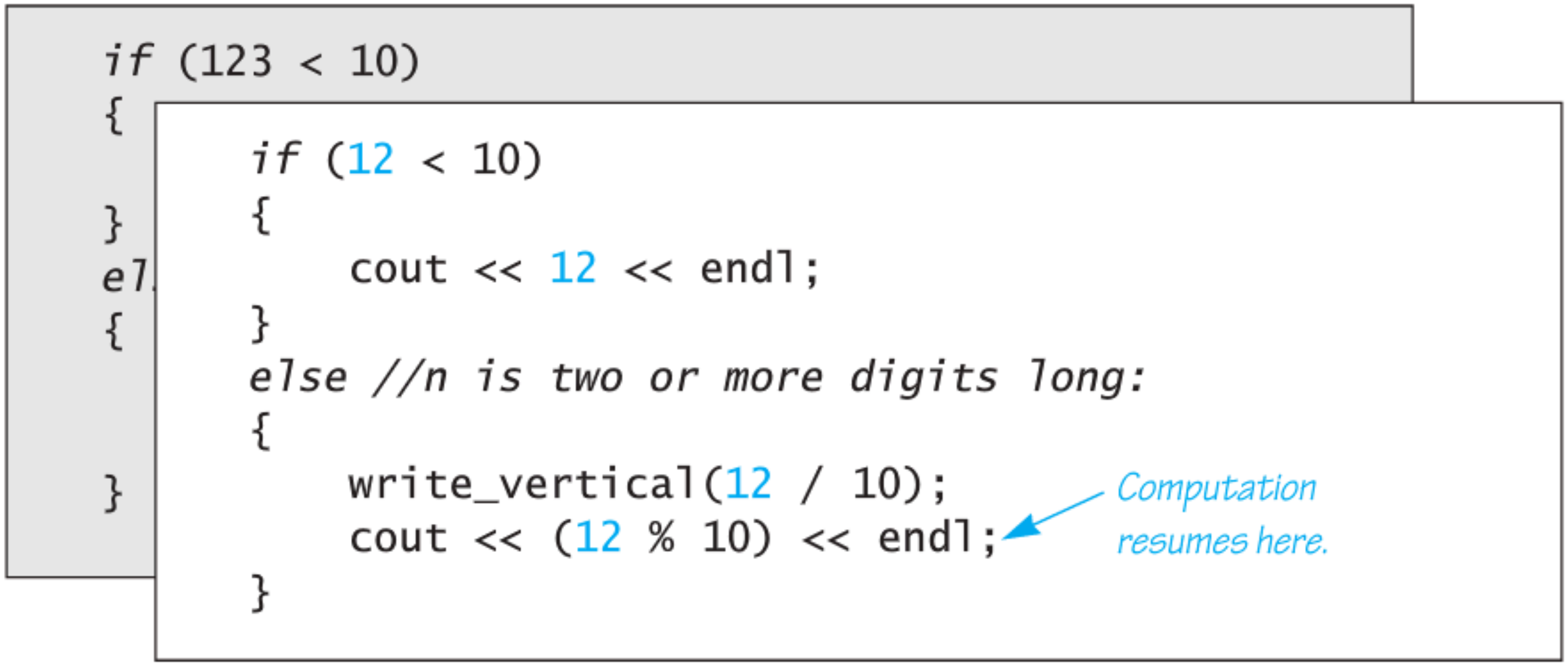
```
if (12 < 10)
{

}
e
{

}
```

```
if (1 < 10)
{
    cout << 1 << endl;
}
else //n is two or more digits long:
{
    write_vertical(1 / 10);
    cout << (1 % 10) << endl;
}
```

Since 12/10 is equivalent to 1 in integer mathematics,
`write_vertical(12/ 10);` becomes `write_vertical(1);`

Since 1< 10, the if case gets exercised, and '1' is output to the screen and the function call returns.

To understand what's happening, let's trace : `write_vertical(123);`



```
if (12 < 10)
{
    cout << 12 << endl;
}
else //n is two or more digits long:
{
    write_vertical(12 / 10);
    cout << (12 % 10) << endl;
}
```

The suspended computation of the previous function call now resumes.

The modulus of 12 % 10 is then printed to the screen after which this function call terminates.

To understand what's happening, let's trace : `write_vertical(123);`

```
if (123 < 10)
{
    cout << 123 << endl;
}
else //n is two or more digits long:
{
    write_vertical(123 / 10);
    cout << (123 % 10) << endl;
}
```

Computation resumes here.

The suspended computation of the original function call now resumes.

The modulus of 123 % 10 is then printed to the screen after which this function call terminates.

The result is the numbers 1,2, and 3 being printed sequentially to the screen with one digit per line.

Although the function `write_vertical()` uses recursion, it is evaluated the same as any other function call.

The difference is that with each recursive call, the input parameter to the function is altered.

The computer views recursion function calls the same as any other function call.

The arguments to the function are passed on the stack to the next function call (which just happens to be the same executable code as the current function call).

The key to using recursion to solve computational problems is to solve the subtask locally, and then pass the relevant data back to the previous function call.

Don’t forget that each function call (even recursive ones) have their own scope.

As such, you have to explicitly return information to the previous calling function if it is required to solve the task.

Although C++ does not restrict how recursive calls are used (similar to normal functions), the design of the function must ultimately terminate in a base case that does not depend on recursion.

As such, the general rules for defining/designing successful recursive functions are:

- Identify one or more cases in which the function accomplishes its task without any use of recursive calls. These cases are called the ***base cases*** (or ***stopping cases***).
- Identify one or more cases in which the function can accomplish its task by using recursive calls to accomplish one or more smaller versions of the task.

Generally, you are going to require an if-else statement to determine which of the cases is to be executed in the specific function call.

- Typically either the if or else case will then require a recursive call of the function.

Recursive functions that do not include base cases will result in an infinite chain of recursive calls.

This will eventually cause your program to crash as you will eventually overflow your stack.

Here's an example of infinite recursion:

```
void new_write_vertical(int n)
{
    new_write_vertical(n / 10);
    cout << (n % 10) << endl;
}
```

Note, this version of the function will compile; however, there is not case for termination.

Eventually, the function `new_write_vertical(0);` will be called infinitely.

To reach the base case(s), each iteration will generally reduce the value of the input parameter/size of the input set, to reduce the complexity of the problem to be solved.

As previously mentioned, most computer systems use a stack to keep track of:

- The return address after the completion of a function call, and
- To pass parameters between the scopes of the different functions.

Each of one of these blocks of data defines the local scope of a function's execution and is known as an activation frame (referring to a reference to the local function code and a copy of its parameters and storage for any output parameters.

Since stacks are Last In- First Out structures, the top of the stack will always store the context of the last function call.

- This allows you to embed as many functions as you want within functions ensuring that if all of these function calls were actually "in-lined" into your executable, the code execution order would still be the same.

A couple notes about the finite stack size of your computing program:

Obviously, the stack has to be finite in size because memory is finite in size.

While it is easy to have a recursive function overflow your stack (if you make to many function calls), this can also happen due to repeated interrupts (and interrupt service routines) that lead to you calling a function within a function within a function …

All this being said, the stack is quite large, so overflows are not easy to achieve.

If you are worried that someone may use a function with a large enough parameter to overflow the stack make this a test case.

- If it passes, you are fine.
- If it doesn’t, you can increase the size of your stack using a linker command.

Recursion versus iteration:

- Some languages don't allow recursion
- It is not absolutely necessary: you can achieve any task without using recursion if you want to by using loops.
  - This is generally referred to as an iterative version.

The next slide show you how to implement the `write_vertical()` without using recursion.

Why do we use recursion?

- Because the solutions are more elegant and often simpler to write

Iterative version:

versus…

```
//Uses iostream:
void write_vertical(int n)
{
    int tens_in_n = 1;
    int left_end_piece = n;
    while (left_end_piece> 9)
    {
        left_end_piece = left_end_piece/10;
        tens_in_n = tens_in_n * 10;
    }
    //tens_in_n is a power of ten that has the same number
    //of digits as n. For example, if n is 2345, then
    //tens_in_n is 1000.

    for (int power_of_10 = tens_in_n;
        power_of_10 > 0; power_of_10 = power_of_10/10)
    {
        cout << (n/power_of_10) <<endl;
        n = n % power_of_10;
    }
}
```

Recursive version:

```
void write_vertical(int n)
{
    if (n < 10)
    {
        cout << n << endl;
    }
    else //n is two or more digits long:
    {
        write_vertical(n / 10);
        cout << (n % 10) << endl;
    }
}
```

Costs of recursion:

Just because recursive functions look simpler doesn’t make them faster.

Generally they are slower than the iterative versions and use more storage than the equivalent iterative version.

- Iterative versions may require more static code to implement the singular function, however,
- They also require only one function call whereas the recursive version requires multiple function calls (using multiple stack frames).
  – Since the computer automatically does this for you, you don’t see this cost

As such, using recursion can make your job easier as a programmer, but you need to think about your application to determine if the cost of recursion is acceptable to runtime memory usage and performance:

- Think of one example for each situation.

As mentioned previously, recursive functions generally return values unlike our previous example.

They can return any type of value (like any other function) and use the same style as other recursive functions:

- One or more cases return a fixed value without using any recursive calls (the base case/stopping case(s))
- At least one case returns a value computed in terms of a recursive call of the same function using a smaller value/set of values.

The next example uses recursion to calculate `int y = power(2, 3);` ($2^3 = 8$)

The definition of this function is based on the equation $x^n = x^{n-1} * x$; and the fact that $x^0 = 1$;

What happens when:

- `power(2, 0)` ?
- `power(2, 1)` ?

```
//Program to demonstrate the recursive function power.
#include <iostream>
#include <cstdlib>
using namespace std;

int power(int x, int n);
//Precondition: n > = 0.
//Returns x to the power n.

int main( )
{
    for (int n = 0; n < 4; n++)
        cout << "3 to the power " << n
             << " is " << power(3, n) << endl;

    return 0;
}

//uses iostream and cstdlib:
int power(int x, int n)
{
    if (n < 0)
    {
        cout << "Illegal argument to power.\n";
        exit(1);
    }

    if (n > 0)
        return ( power(x, n - 1) * x);
    else // n == 0
        return (1);
}
```

What about our original

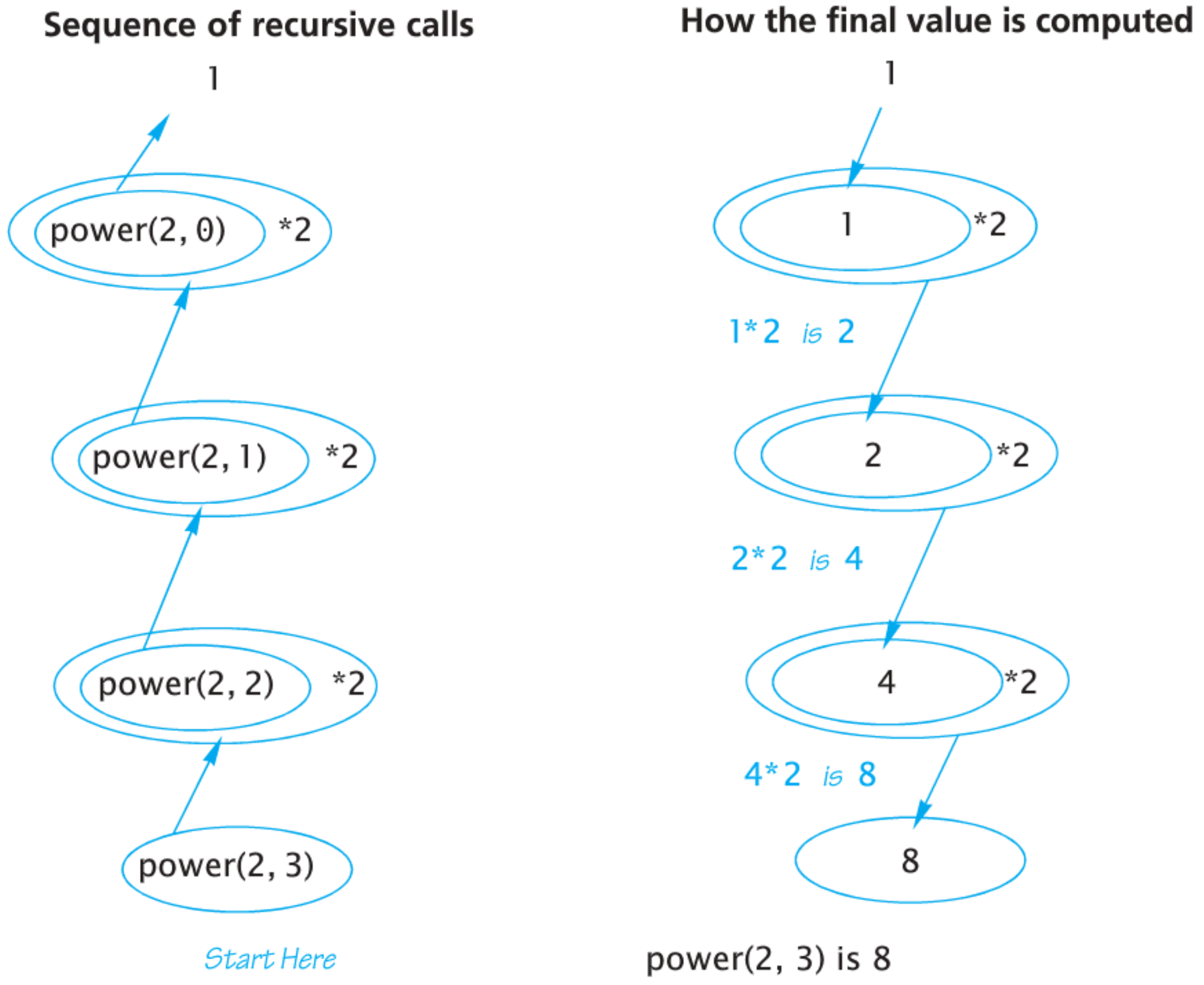
```
power(2, 3);?
```

```
//Program to demonstrate the recursive function power.
#include <iostream>
#include <cstdlib>
using namespace std;

int power(int x, int n);
//Precondition: n > = 0.
//Returns x to the power n.

int main( )
{
    for (int n = 0; n < 4; n++)
        cout << "3 to the power " << n
             << " is " << power(3, n) << endl;

    return 0;
}

//uses iostream and cstdlib:
int power(int x, int n)
{
    if (n < 0)
    {
        cout << "Illegal argument to power.\n";
        exit(1);
    }

    if (n > 0)
        return ( power(x, n - 1) * x);
    else // n == 0
        return (1);
}
```

What about our original `power(2, 3);`?

Recall:

```
power(2, 3) is power(2, 2) * 2
power(2, 2) is power(2, 1) * 2
power(2, 1) is power(2, 0) * 2
power(2, 0) is 1 (stopping case)
```



When designing recursive functions, things to keep in mind:

Other than doing a quick check that your stack won't overflow for the largest expected input values, you don't need to worry about the runtime operations of the stack and suspended computations (the computer does this for you).

Recursive functions need to satisfy the following criteria:

1. There is no possibility of infinite recursion.
2. Each base case/stopping case returns the correct value for that case.
3. For the recursive cases, if all recursive calls return the correct value, then the final value returned by the function is the correct value.

Note: If all recursive calls return the correct value, then the final value returned by the function will be correct.

- (What does this remind you of????)

Recursive functions need to satisfy the following criteria:

1. There is no possibility of infinite recursion.
2. Each base case/stopping case returns the correct value for that case.
3. For the recursive cases, if all recursive calls return the correct value, then the final value returned by the function is the correct value.

1. To ensure there is no infinite recursion, you need to make sure your recursive case(s) properly decompose your problem into a sub-problem that will eventually reach a base case

- For example: with the power function, we kept decreasing the integer exponent by one so that it would eventually reach zero)

Recursive functions need to satisfy the following criteria:

1. There is no possibility of infinite recursion.
2. Each base case/stopping case returns the correct value for that case.
3. For the recursive cases, if all recursive calls return the correct value, then the final value returned by the function is the correct value.

2. Think carefully about your problem: is there only one base case? Either way, make sure that the correct value is returned in all cases

- For example: with the power function, the only base case is when the exponent equals zero as the correct value for $x^0 = 1$;

Recursive functions need to satisfy the following criteria:

1. There is no possibility of infinite recursion.
2. Each base case/stopping case returns the correct value for that case.
3. For the recursive cases, if all recursive calls return the correct value, then the final value returned by the function is the correct value.

3. Depending on the nature of the computation, there may be one or more recursive cases. Make sure that your solution is valid for the ***entire set*** of possible input values.

- For example: with the power function, the only recursive case is when n>0; in that case, you want to return `power(x, n-1) * x`;

Think: How would you prove this to yourself?

Recursive functions need to satisfy the following criteria:

1. There is no possibility of infinite recursion.
2. Each base case/stopping case returns the correct value for that case.
3. For the recursive cases, if all recursive calls return the correct value, then the final value returned by the function is the correct value.

Final notes, the above conditions basically apply to recursive functions that do not return values (i.e. `void` return values), except:

1. There is no possibility of infinite recursion.
2. Each base case/stopping ***case performs the correct action*** for that case.
3. For the recursive cases, if all recursive calls ***perform their actions correctly***, then the final ***operation performed by the original function call is correct***.

Binary search algorithm using recursion.

This algorithm searches a ***sorted*** array for a specified value.

Instead of searching every single array entry for the value or searching the array in sequence until you have passed the point at which the value is stored, we will be using a traditional “binary search” algorithm.

Our algorithm will have two call-by-reference parameters:

-`found` – a boolean parameter that will be set to `true` if the value is found

-`location` – an integer value that will be set to the index of the value’s location if it is found.

So the pre and post conditions are:

Precondition: a[0] through a[final_index] are sorted in increasing order.

Postcondition: if key is not one of the values a[0] through a[final_index], then found == *false*; otherwise, a[location] == key and found == *true*.

Binary search algorithm using recursion.

A binary search of a sorted list assumes you can subdivide the problem into two parts and consider the two sub-problems independently.

```
found = false; //so far.
mid = approximate midpoint between 0 and final_index;
if (key == a[mid])
{
    found = true;
    location = mid;
}
else if (key < a[mid])
    search a[0] through a[mid - 1];
else if (key > a[mid])
    search a[mid + 1] through a[final_index];
```

If the value you are searching for (the ***key***) is in a sorted list, it will either be at the mid-point, in the first half of the list or the second half of the list.

- ***This first step has either solved the problem or reduced our search space by half***

Binary search algorithm using recursion.

The smaller search space can now be searched recursively using the same algorithm. However, we need to be able to adjust our search space window, so we need two additional parameters (`first` and `last`)

```
To search a[first] through a[last] do the following:
found = false; //so far.
mid = approximate midpoint between first and last;
if (key == a[mid])
{
    found = true;
    location = mid;
}
else if (key < a[mid])
    search a[first] through a[mid - 1];
else if (key > a[mid])
    search a[mid + 1] through a[last];
```

By adding the parameters `first` and `last` to our function call, we can initially set their values to `0` and `final_index` and then adjust their value with each recursive function call to shrink the search space.

Binary search algorithm using recursion.

What do we do if the number isn’t in the list?

Does this code have the necessary base case?

```
To search a[first] through a[last] do the following:
found = false; //so far.
mid = approximate midpoint between first and last;
if (key == a[mid])
{
    found = true;
    location = mid;
}
else if (key < a[mid])
    search a[first] through a[mid - 1];
else if (key > a[mid])
    search a[mid + 1] through a[last];
```

Final pseudocode: We specify the last base case by recognizing that if the value of `first` ever becomes larger than `last`, then there is nowhere left to search.

```
int a[Some_Size_Value];

Algorithm to search a[first] through a[last]

//Precondition:
//a[first]<= a[first + 1] <= a[first + 2] <= ... <= a[last]

To locate the value key:

if (first > last) //A stopping case
    found = false;
else
{
    mid = approximate midpoint between first and last;
    if (key == a[mid]) //A stopping case
    {
        found = true;
        location = mid;
    }
    else if key < a[mid] //A case with recursion
        search a[first] through a[mid - 1];
    else if key > a[mid] //A case with recursion
        search a[mid + 1] through a[last];
}
```

Actual code:

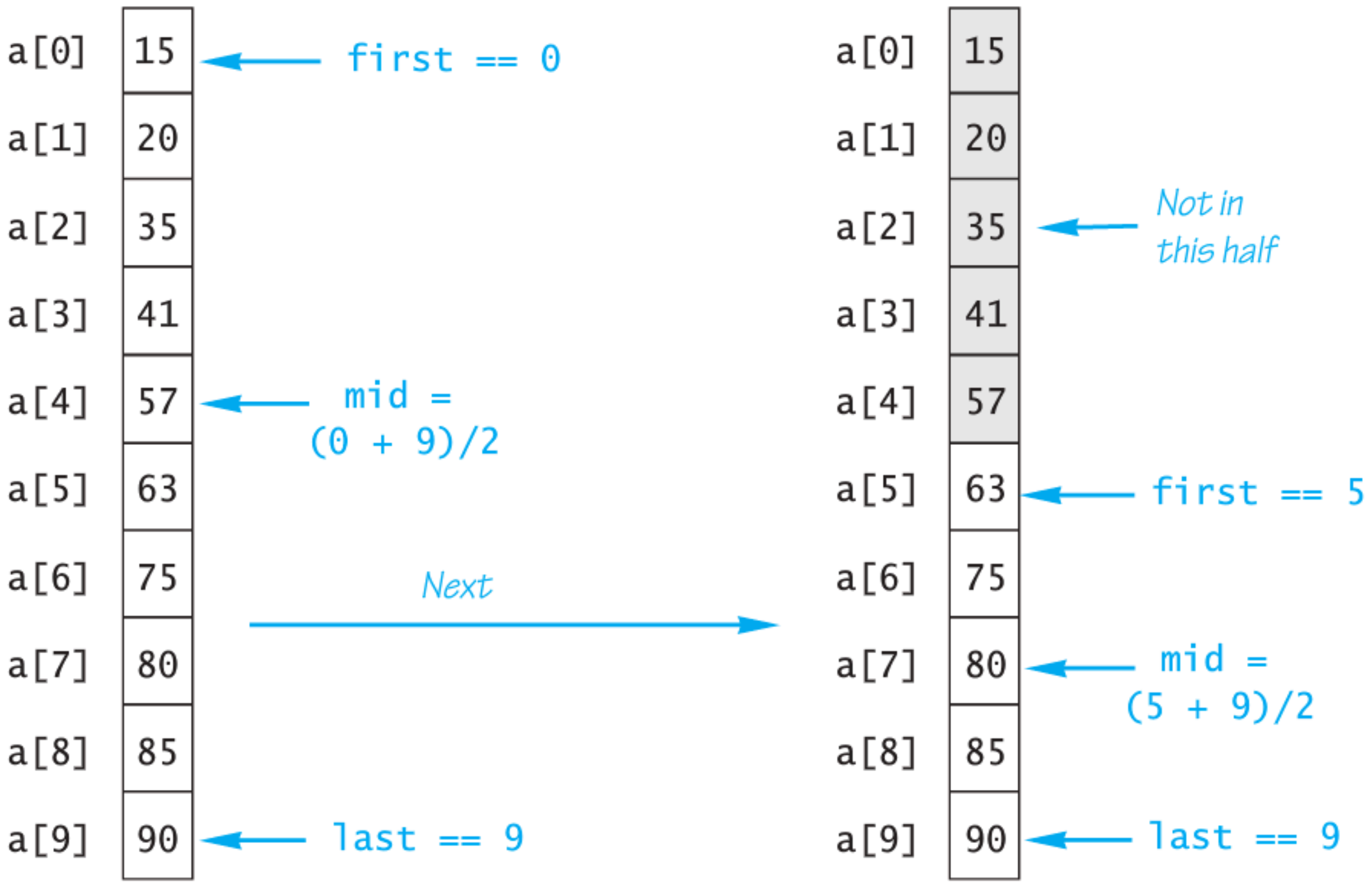
```
//Program to demonstrate the recursive function for binary search.
#include <iostream>
using namespace std;
const int ARRAY_SIZE = 10;


void search(const int a[], int first, int last,
    int key, bool& found, int& location);
//Precondition: a[first] through a[last] are sorted in increasing order.
//Postcondition: if key is not one of the values a[first] through a[last],
//then found == false; otherwise, a[location] == key and found == true.


int main( )
{
    int a[ARRAY_SIZE];
    constint final_index = ARRAY_SIZE - 1;
    int key, location;
    bool found;
    cout << "Enter number to be located: ";
    cin >> key;
    search(a, 0, final_index, key, found, location);

    if (found)
        cout << key << " is in index location "
             << location <<endl;
    else
        cout << key << " is not in the array." <<endl;

    return 0;
}
```

Actual

code:

```
void search(const int a[], int first, int last,
                                    int key, bool& found, int& location)
{
      int mid;
      if (first > last)
      {
            found = false;
      }
      else
      {
            mid = (first + last)/2;

            if (key == a[mid])
            {
                  found = true;
                  location = mid;
            }
            else if (key < a[mid])
            {
                  search(a, first, mid -1, key, found, location);
            }
            else if (key > a[mid])
            {
                  search(a, mid + 1, last, key, found, location);
            }
      }
}
```

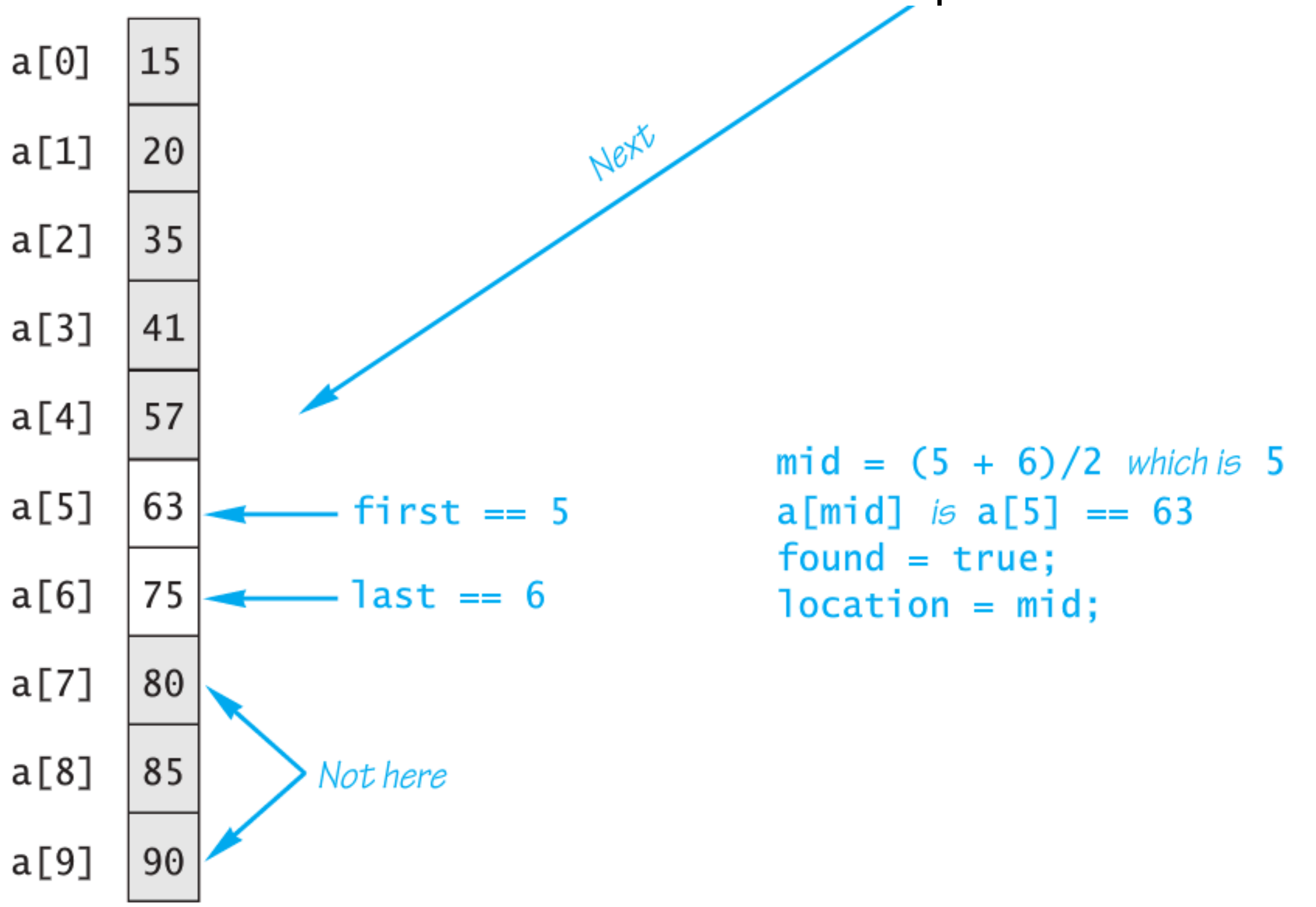
Assuming the key is 63, let's look at how the function executes:



Next…

Assuming the key is 63, let's look at how the function executes:

From the previous iteration



Binary search algorithm using recursion.

Checking our algorithm:

1. There is no infinite recursion: Since the search space is reduced during each recursive call, this function will terminate.
2. Each stopping case performs the correct action: There are two stopping cases either `key == a[mid]` or `first > last`.
    1. If `first > last`, there are no array elements between `a[first]` and `a[last]` so the key is not in the array. In this case, the function correctly returns `found` set to `false` and the function terminates.
    2. If `key == a[mid]` the algorithm sets `found` equal to `true`

Thus both stopping cases are correct.

Binary search algorithm using recursion.

Checking our algorithm:

3. For each recursive case, if all recursive functions perform the correct actions, the initial function call will perform the correct actions. There are two cases for recursive calls: when `key < a[mid]` and when `key > a[mid]`

    1. Case 1: `key < a[mid]`. Since the list is sorted, we know that if the if the key is in the list, it must be one of the elements in `a[first]` through `a[mid-1]`, hence the recursive call:

    ```
    search(a, first, mid - 1, key, found, location);
    ```

    2. Case 1: `key > a[mid]`. Since the list is sorted, we know that if the if the key is in the list, it must be one of the elements in`a[mid+1]` through `a[last]`, hence the recursive call:

    ```
    search(a, mid + 1, last, key, found, location);
    ```

Binary search algorithm using recursion.

Since this solution passes all three criteria, we can be satisfied that the recursive function definition is good.

Something else to keep in mind: We don’t have to use this function to search an entire search space. Based on its definition, we can use it to search any contiguous subspace as well.

Efficiency of Binary search algorithm (Hint: Intro to Complexity Analysis)

A binary search is generally much faster than trying to search all of the array elements in order because it initially reduces the search space by half, and then by an additional quarter, and then by an eighth (and so on and so on…).

For an array of 100 elements, the binary search will never need to compare more than 7 array elements to the key (100 -> 50 -> 25 -> 12 -> 6 -> 3 -> 1).

- Conversely if we simply searched in order, we could have to compare all 100 elements.

What's even better is the larger the array, the greater the savings will be.

For an array of 1000 elements, a binary search will have to compare 10 elements, where as a normal sequential search will require (on average) 500 elements (with a potential maximum of 1000 comparisons).

An iterative version of the Binary Search Algorithm:

Recall our original recursive function definition:

```
void search(const int a[], int first, int last,
    int key, bool& found, int& location);
//Precondition: a[first] through a[last] are sorted in increasing order.
//Postcondition: if key is not one of the values a[first] through a[last],
//then found == false; otherwise, a[location] == key and found == true.
```

Now look at the iterative function definition:

```
void search(const int a[], int low_end, int high_end,
int key, bool& found, int& location);
//Precondition: a[low_end] through a[high_end] are sorted in increasing
//order.
//Postcondition: If key is not one of the values a[low_end] through
//a[high_end], then found == false; otherwise, a[location] == key and
//found == true.
```

An iterative version of the Binary Search Algorithm (function implementation):

```
void search(const int a[], int low_end, int high_end,
    int key, bool& found, int& location)
{
    int first = low_end;
    int last = high_end;
    int mid;

    found = false; //so far
    while ( (first <= last) && !(found) )
    {
        mid = (first + last)/2;
        if (key == a[mid])
        {
            found = true;
            location = mid;
        }
        else if (key < a[mid])
        {
            last = mid -1;
        }
        else if (key > a[mid])
        {
            first = mid + 1;
        }
    }
}
```

Recursive member functions

Member functions can also be recursive (just like normal functions).

Remember the class BankAccount?

The following example overloads the member function named `update()`

- The first version has no arguments and posts one year of simple interest to the back account balance.
- The other (new) version of `update()` takes an integer argument of some number of years and updates the account by posting the interest for that many years. This version is recursive and has one parameter called `years`.

Recursive member functions

The algorithm for `update(years)` is:

```
If the number of years is 1, then //Base case
    Call the version of update with no arguments
Else if the number of years is greater than 1 //Recursive case
    Call update(years-1) to calculate the previous years'
    interest and then call the version of update with no
    arguments to calculate the interest accumulated for one
    additional year
```

This is based on the idea that the interest over x years is equal:

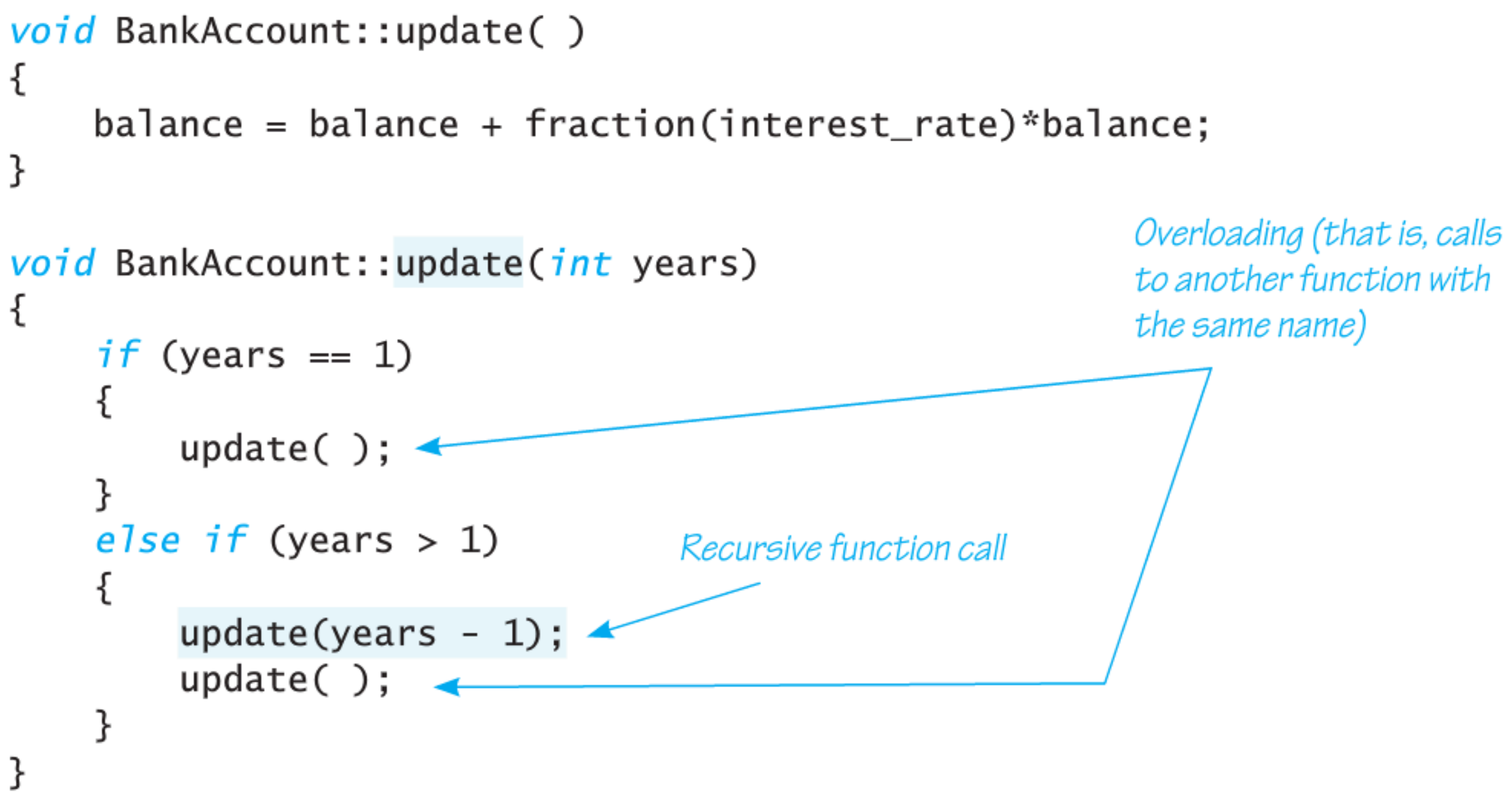
x years interest = (x-1) years interest + one years interest;

```cpp
class BankAccount
{
public:
    BankAccount(int dollars, int cents, double rate);
    //Initializes the account balance to $dollars.cents and
    //initializes the interest rate to rate percent.

    BankAccount(int dollars, double rate);
    //Initializes the account balance to $dollars.00 and
    //initializes the interest rate to rate percent.

    BankAccount( );
    //Initializes the account balance to $0.00 and
    //initializes the interest rate to 0.0%.

    void update( );
    //Postcondition: One year of simple interest
    //has been added to the account balance.

    void update(int years);
    //Postcondition: Interest for the number of years given has been added to the
    //account balance. Interest is compounded annually.

    double get_balance( );
    //Returns the current account balance.

    double get_rate( );
    //Returns the current account interest rate as a percentage.

    void output(ostream& outs);
    //Precondition: If outs is a file output stream, then outs has already
    //been connected to a file.
    //Postcondition: Balance & interest rate have been written to the stream outs.
private:
    double balance;
    double interest_rate;
    double fraction(double percent); //Converts a percentage to a fraction.
};
```

Two different functions with the same name

```
int main( )
{
    BankAccount your_account(100, 5);
    your_account.update(10);
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "If you deposit $100.00 at 5% interest, then\n"
         << "in ten years your account will be worth $"
         << your_account.get_balance( ) << endl;
    return 0;
}
```

All other member function definitions are the same as previously given.



```
void BankAccount::update( )
{
    balance = balance + fraction(interest_rate)*balance;
}

void BankAccount::update(int years)
{
    if (years == 1)
    {
        update( );
    }
    else if (years > 1)
    {
        update(years - 1);
        update( );
    }
}
```

Overloading (that is, calls to another function with the same name)
Recursive function call

Let’s check our three criteria.

All other member function definitions are the same as previously given.

```
void BankAccount::update( )
{
    balance = balance + fraction(interest_rate)*balance;
}

void BankAccount::update(int years)
{
    if (years == 1)
    {
        update( );
    }
    else if (years > 1)
    {
        update(years - 1);
        update( );
    }
}
```

Overloading (that is, calls to another function with the same name)

Recursive function call

1. There is no infinite recursion: Years is repeatedly reduced by 1 until it is equal to 1 (the stopping/base case).

All other member function definitions are the same as previously given.

```
void BankAccount::update( )
{
    balance = balance + fraction(interest_rate)*balance;
}

void BankAccount::update(int years)
{
    if (years == 1)
    {
        update( );
    }
    else if (years > 1)
    {
        update(years - 1);
        update( );
    }
}
```

Overloading (that is, calls to another function with the same name)

Recursive function call

2. Each stopping case performs the correct action for that case: In this case the base case calculates one year's interest (which is correct).

All other member function definitions are the same as previously given.

```
void BankAccount::update( )
{
    balance = balance + fraction(interest_rate)*balance;
}

void BankAccount::update(int years)
{
    if (years == 1)
    {
        update( );
    }
    else if (years > 1)
    {
        update(years - 1);
        update( );
    }
}
```

Overloading (that is, calls to another function with the same name)

Recursive function call

3. For each recursive case, the recursive call is performed correctly: when years>1, because the recursive call posts years-1 worth of interest and then adds in the additional year’s worth of interest using a verified function for calculating one year of interest.

All other member function definitions are the same as previously given.

```
void BankAccount::update( )
{
    balance = balance + fraction(interest_rate)*balance;
}

void BankAccount::update(int years)
{
    if (years == 1)
    {
        update( );
    }
    else if (years > 1)
    {
        update(years - 1);
        update( );
    }
}
```

Overloading (that is, calls to another function with the same name)

Recursive function call

Remember: Even though these overloaded functions appear to have the same name, the compiler recognizes them as different (based on the arity of their parameters).

We could have done the same version of update if the original update function (with no input parameters) had been called `post_one_year()`;.

```
void BankAccount::update(int years)
{
    if (years == 1)
    {
        post_one_year();
    }
    else if (years > 1)
    {
        update(years - 1);
        post_one_year();
    }
}
```

<u>Important Note:</u>

Although our example used recursion with overloading, they are unrelated.

- When you overload a function you are describing two different functions with the same name (but different types/arity of input parameters).
- Recursive functions must include a call to a function with the exact same definition (as opposed to one that happens to have the same name)

Do not confuse the two.

In closing, recursive solutions often require you to solve a more general problem than the given task

This is often necessary to enable proper recursive calls as the smaller problems may not be exactly the same as the given task.

Review Questions for Slide Set 7

- What is the definition of recursion?
- What are the benefits of designing a recursive solution to a function?
- When you create a recursive function, what do you need to worry about?
- How do you design a recursive function?
- How do you enable a recursive function to reach a base case?
- If you do not include a base case in your recursive function, what will happen?
- Why do you have to worry about stack overflows with recursion?

Review Questions for Slide Set 7

- Is there any task that needs to be solved recursively and cannot be solved iteratively? If no, why use recursion?
- Which is faster iterative or recursive solutions? Why?
- What are the three criteria that all recursive functions need to satisfy?
- How does the binary search algorithm work? What is the benefit of a binary search?
- Can you use recursive functions inside of class definitions?
- Are recursive functions and overloaded functions related?