UCLA PIC 10 B

Problem Solving using C++ Programming

- **Instructor:** Ivo Dinov, Asst. Prof. in Mathematics, Neurology, Statistics
- **Teaching Assistant:**, Suzanne Nezzar, Mathematics

University of California, Los Angeles, Summer 2001

http://www.math.ucla.edu/~dinov/10b.1.011/

Chapter 12

Recursive Functions for Tasks

Problem:

Suppose we want to compute N-factorial

$$N! = 1*2*3* \ldots *(N-2)*(N-1)*N$$

$$N! \sim \sqrt{2*\Pi*N} * (N/e)^N, \text{ where } e \sim 2.718\ldots$$

See Online Class Notes for Week 2 for a JavaScript Computation

Inductive (recursive) computation is possible:

$N! = N * (N-1)!$; $(N-1)! = (N-1) * (N-2)!$; etc.

12 Recursive Functions for Tasks

12 Recursive Functions for Tasks

- In mathematics we do not allow circular definitions, where a thing is defined in terms of itself, or where a thing is defined in terms of another thing which in turn is defined in terms of the first thing.
- A kind of circularity that *is* allowed (even encouraged) in mathematics and in Computer Science is definitions that refer to smaller versions of the same problem. Such definitions are called recursive definitions.
- In Computer Science, (and in C++) a function that contains a call to itself is called a recursive function.
- The function is said to be recursive.
- This chapter is about how recursion may be correctly carried out.

12.1 Recursive Functions for Tasks

- A basic design technique in problem solving is divide and conquer, which means to divide the problem into subtasks.
- If at least one of the subtasks is smaller than the original problem, you can use a recursive function to solve the problem.
- With a bit of experience, this is a quick way to design an algorithm to solve such a problem.

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.

Case Study: Vertical Numbers (1 of 10)

- We design a recursive `void` function that writes numbers to the screen with digits written vertically. 1984 would be written as:

```
1
9
8
4
```

Problem Definition:

The prototype (and comments) is:

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


Case Study: Vertical Numbers (2 of 10)

- The case where the number is one digit long is simple: Just print it.

Simple Case: If `n<10`, then write the number `n` to the screen.

- Suppose the number `n` is `1234`.
- Suppose further that *we can already write numbers vertically.* Then one way to divide the problem into tasks is:

1) write the first 3 numbers:

```
1
2
3
```

2) write the 4th.

```
4
```

- Subtask 1 is smaller than the original problem, which suggests the algorithm in the next slide.

Case Study: Vertical Numbers (3 of 10)

- Pseudocode:

```
if (n < 10)
   cout << n << endl;
else                          //    The recursive subtask.
{
   write_vertical(the number n with the last digit removed);
   cout << the last digit of n << endl;
}
```

- To convert this to C++ we need to convert the pseudocode code expressions:

```
the number n with the last digit removed
```

- and

```
the last digit of n
```


Case Study: Vertical Numbers (4 of 10)

- We use % and / to carry out the conversions:
  - `n / 10 // the number n with the last digit removed`
- and
  - `n % 10 // the last digit of n`

We chose this decomposition into these two subtasks for two reasons:

1) we can easily compute the argument for the recursive subtask
2) the second task is *not involve a recursive call.*

- A successful recursive function must always include:
  - At least one case that is *not* recursive.
  - One or more other cases that involve at least one recursive call.

Display 12.1 A Recursive Output Function (1 of 2)

```
//      Program to demonstrate the recursive function write_vertical.

#include <iostream>
using namespace std;

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;
}
```


Display 12.1 A Recursive Output Function (2 of 2)

```
//    Program to demonstrate the recursive function write_vertical.

//    uses iostream:
void write_vertical(int n)
{
   if (n < 10)
   {    cout << n << endl;   }

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

Case Study: Vertical Numbers (5 of 10)

TRACING A RECURSIVE CALL

Let's see what happens when we make the following call:

write_vertical(123);

The initial call is exactly like any other function call. The argument is "plugged in for" the parameter n and the function body is executed:

```
if (123 < 10)
  cout << 123 << endl;
else // n has 2 or more digits
{
  write_vertical(123/10);   <-- Computation will stop here
  cout << (123 % 10 ) << endl;   until the recursive call
}                                returns.
```


Case Study: Vertical Numbers (6 of 10)

- **The expression, 123 < 10 is false, so the if-else statement's else clause is executed:**

  write_vertical(n / 10);

  With n = 123, the argument is 12.

  We are making a recursive call to write 12 vertically.

```
if (123 < 10)
{
    cout
}
else //n
{
    writ
    cout
}
```

```
if (12 < 10)
{
    cout << 12 << endl;
}
else //n is two or more digits long:
{
    write_vertical(12/10);   <-- Computation will stop here until the recursive call returns.
    cout << (12%10) << endl;
}
```


Case Study: Vertical Numbers (7 of 10)

- **Actions here are similar: n has the value 12, and there is a recursive call with argument 1, which is substituted for the parameter.**
- **The if control, n < 10, is *true* so the recursive case does not occur.**

```
if (123 < 10)
{
}
els
{
}
```

```
if (12 < 10)
{
}
els
{
}
```

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

No recursive call this time

Case Study: Vertical Numbers (8 of 10)

- **When write_vertical(1) runs the statement**

  cout << 1 << endl;

  is encountered and the output is 1. The call to write_vertical(1); ends.
- **Then the suspended computation for write_vertical(12) resumes.**
- **The computation state is:**

```
if (123 < 10)
{
}
els
{
}
```

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


Case Study: Vertical Numbers (9 of 10)

- **write_vertical(12) resumes, executing a statement**

  cout << 12%10 << endl;

  The output is 2. The call to write_vertical(12); ends.
- **Then the suspended computation for write_vertical(123) resumes.**

```
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.
}
```


Case Study: Vertical Numbers (10 of 10)

- **When the last suspended computation resumes, we encounter a statement:**

  cout << 123%10 << endl;

  which outputs 3 followed by a newline.
- **Collecting together the output, we find we have**
  1. **Done by the deepest recursion, a non-recursive case**
  2. **Done by the next deepest recursion, on unwinding**
  3. **Done by the first recursion, on unwinding the recursion.**

A Closer Look at Recursion(1 of 2)

- Our definition of write_vertical uses recursion
- The computer under control of the C++ runtime system keeps track of the recursions.
- HOW?
- In the initial call the computer plugs in the argument and starts the function.
- If a recursive call is encountered, computation is suspended, because the results of the recursion are needed to continue the computation.
- When the recursive call is completed, then the suspended computation continues. We stated these rules before:
- One or more cases must occur in which the function accomplishes its task without recursive calls. These are called the base cases or stopping cases.
- One or more cases occur in which the function accomplishes its task by recursive calls to carry out one or more smaller version of its task.

A Closer Look at Recursion(2 of 2)

- A common way to stop recursion is to have the recursive function test a positive numeric quantity that is decreased on each recursion, and to provide a stopping case for some small value.
- In the example write_vertical, the parameter value is the quantity mentioned above, and the "small value" is 10.

General Form of a Recursive Function Definition

The general outline of a successful recursive function definition is:

- One or more of the cases include one or more recursive calls to the function being defined. These recursive calls should solve "smaller" versions of the task performed by the function being defined.
- One or more of the cases that include no recursive calls. These cases without any recursive calls are called the base cases or stopping cases.

Pitfall: Infinite Recursion (1 of 2)

- Any recursion MUST ultimately reach one of the base, or stopping cases.
- The alternative is an infinite recursion, that is, a recursion that never ends, except in frustration ☺.
- Each recursive call causes suspension and saving of the computation that makes the recursive call.
- Saving the computations requires machine resources, which are quickly consumed.
- If you are using an operating system that is protected against application misbehavior (such as Unix, Linux, Windows NT or Windows 2000) your program will crash with antisocial consequences limited to temporary system slowing for other users.
- Otherwise your operating system is likely to crash, or worse, damage some system component.
- The moral is: Avoid infinite recursions.

Pitfall: Infinite Recursion (2 of 2)

- Examples of infinite recursion:

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

- This compiles and runs, but a call to the function will never return.
- This incorrect code has a certain reasonableness to it: It outputs a vertical list of successive digits, then prints the last digit.
- However, the last statement is never reached.
- There is no way for the code body to avoid executing the recursive call.
- The successive calls "bottom out" with calls with argument 0:

```
new_write_vertical(0);
```

- The output will be a vertical sequence of 0s.

Stacks for Recursion (1 of 2)

- The successive suspended computations are saved in a structure called a *stack*.
- A stack is structure like a stack of pieces of paper with information on each piece of paper. (Assume an unlimited supply of paper.)
- We have two operations: writing on a piece of paper then "pushing" it onto the stack, and "popping" a piece of paper off the top of the stack (and reading it, of course). ONLY the top is accessible.
- With these restrictions the stack is a last-in-first-out (LIFO) data structure.
- In a recursion, we save the suspended computations in the order of the recursive calls.
- Suspended computations are reactivated in reverse order of suspension.
- The computations are saved in order and the last one saved is needed first.

Stacks for Recursion (2 of 2)

- The stack structure saves the suspended computations in exactly the order needed for recursions and other nested function calls.
- When a function is called, the system creates a record that is placed on the system execution stack. These records are called activation frames, or sometimes stack frames.
- What is in an activation frame?
- Activation frames hold information necessary to run the function.
- The activation record contains memory for the function's local variables and parameters (which are initialized with the current arguments).
- The activation frame does not contain and does not need a complete copy of the function. There is only one copy of the function code.
- C++ and the operating system need to save other information as part of the activation frame. We won't treat these details here. Wait for the operating systems course. Better, read an operating systems book.

Pitfall: Stack Overflow

There is a limit to all computer resources.

- The system execution stack is built in memory which, excepting only CPU time, is the scarcest resource in the computer.
- Each function call uses a chunk of the stack we called a stack frame or activation frame.
- Using all the memory provided for a stack is an error called stack overflow.
- If you get an error message that says "stack overflow" it likely means some function calls have used all the stack.
- A common cause of stack overflow is infinite recursion.

Recursion vs. Iteration

- Recursion is not necessary.
- The nonrecursive version of a recursive function typically uses a loop (or loops) to replace the recursion. (Compare Displays 12.1 and 12.2.)
- Recursion can always be replaced by iteration. And conversely, iteration can always be replaced by recursion.
- In fact, recursion is almost always converted to iteration by the compiler before execution.
- Recursive implementations will almost always run slower and use more memory than iterative versions (because of the stuff that needs to be saved before recursions).
- Why then, should we ever use recursion?
- Ease of understanding code and ease of implementation are the primary reasons for using recursion.
- Some algorithms are easier to understand in the recursive version. The difference between the recursive and iterative forms of an algorithm can be dramatic. See Section 12.3, Binary Search.

Display 12.2 Iterative Version of the Function in Display 12.1 (1 of 2)

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

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;
}
```


Display 12.2 Iterative Version of the Function write_vertical

```
//      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 Function for Values

General form for a Recursive Function that Returns a Value

- Recursive functions we have seen so far have been void functions.
- Recursive functions can return other types than void.
- Successful value returning recursive functions must have:
  - One or more cases where the computed value is returned in terms of calls to the same function (using recursive calls). As in void recursive functions, the arguments for the recursive call must represent a "smaller" task.
  - One or more cases in which the value is computed without recursive calls. As with void recursive functions, these are base or stopping cases.
- Display 12.3 illustrates the method.

Display 12.3 The Recursive Function Power (1 of 2)

```
//    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;
}
```

Display 12.3 The Recursive Function Power (2 of 2)

```
//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         return (1);
}
```


Programming Example:
Another Powers Function (1 of 2)

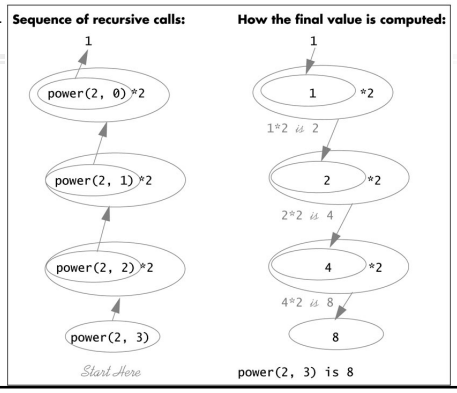
- We have seen the **pow** function that computes powers. The function takes two double arguments and returns a double. Its prototype is:
  `double pow( double, double);`
- The new function is similar in behavior, but takes int arguments and returns an int, and is called power. See Display 12.3 for the code.
- The definition of power is based on
  - $x^n = x^{n-1} * x, \; n > 1$ // This is the recursive case
  - $x^0 = 1.$ // This is the base case.
- The recursive case is translated to C++ as:

```
power(x, n-1) * x
```


Programming Example:
Another Powers Function(2 of 2)

- Examples: Let x be 2, and trace some actions:
  `int y = power(2, 0);` // This is the base case; it should assign 1 to y.
  `y = power(2,1);` // This should recur once and assign 2 to y.
- In the second example the return statement should be
  `return( power(x, n-1) * x);`
  with x having the value 2 and n the value 1.
- In the second example, we said that the return statement should be
  `return( power(x, n-1) * x);`
  with x having the value 2 and n the value 1.
  `return( power(2, 0) * 2);`
- This evaluates to 2.
- Display 12.4 (next) provides the detail of the recursive calls for
  `power(2, 3);`

Display 12.4 Evaluating the Recursive Function Call power(2,3)



Thinking Recursively
Recursive Design Techniques (1 of 3)

- The power of recursive programming comes from the ability to ignore details of the stack and suspended computations, the ability to let the computer take care of the bookkeeping details.
- In designing a recursive function we do need to trace any sequence of recursive calls. For value returning recursive functions it is necessary to confirm that:

1. There is no infinite recursion. All possible chains of recursive calls must lead to some stopping case.
2. Each stopping case yields a correct value for that case.
3. For each recursion:
   if every recursive call returns the correct value for that case, *then* the final value returned by the function is the correct value.

Recursive Design Techniques (2 of 3)

- Let's illustrate the verification technique for the power function of Display 12.3:

1. There is no infinite recursion:

The recursion stops because each time a recursion occurs, the value of argument corresponding to n is decreased by 1, guaranteeing that we reach the one stopping case, power(x, 0).

2. Each stopping case returns the correct value.

There is only one stopping place, power(x, 0), which returns 1, the correct value provided x != 0.

Recursive Design Techniques (3 of 3)

3. **For all recursive cases: *if* each recursive call returns the correct value, *then* the final value returned by the function is the correct value.**

When `n>1, power(x, n)` returns `power(x, n-1) * x`

Remember, *If* power(x, n-1) returns the correct value, *then* power(x, n) returns the correct value.

If power(x, n-1) returns the correct value, namely x^{n-1} then power(x,n) returns the correct value.

Suppose that power(x, n-1) returns this value. We know that power(x,n) returns powers(x, n-1) * x, so power(x,n) must return $x^{n-1} * x$ which is x^n.

Case Study: Binary Search -- An Example of Recursive Thinking (1 of 9)

- Here we develop a recursive function to search an array to find a specific value (the target of the search). An application might be searching an array of invalid credit card numbers.
- The index values are integers 0 through final_index. Our algorithm requires that the array be sorted, which means:
  `a[0] <= a[1] <= a[2] <= . . . <= a[final_index]`
- In searching we also may want to know *where* in the array the target item is. (The index of the bad credit card number may be an index into another array of information about the person.)
- PROBLEM DEFINITION:
  Prototype: `void search(int a[], int first, int last, int key, bool& found, int& location);`
  Pre: `a[0], …, a[final_index]` is sorted in increasing order.
  Post: `if key` is not one of the values `a[0]` through `a[final_index]`,
  `found == false`
  `else`
  `a[location] == key and found == true.`

Binary Search (2 of 9)

ALGORITHM DESIGN

- Suppose there are many numbers, and that each number is on a separate page of a book.
- We might open the book in the middle and see if we were lucky -- is the number there? Otherwise, we select the half of the book the number is in and repeat until the half is only one page.
- We now know whether the number is in the book.

Binary Search (3 of 9)

PSEUDO CODE

```
found = false; // so far
mid = some approximate midpoint between 0 and
   final_index;
if (key == a[mid])
{ found = true;
  locate = mid;
}
else if (key < a[mid])   search a[0] through a[mid - 1];
// translate into a recursive call

else if (key>a[mid])
       search a[mid + 1] through a[final_index];
 // translate into a recursive call
```


Binary Search (4 of 9)

PSEUDO CODE (continued)

Looks good so far, but we find that we need to search successively smaller pieces (half, quarter, and so on) of our array, so we need the two extra parameters mentioned in the problem definition, `first` and `last`, to specify the successively smaller pieces.

```
found = false; // so far
mid = some approximate midpoint between 0 and final_index;
if (key == a[mid])
{ found = true;
  locate = mid;
}
else if (key<a[mid])  search a[first] through a[mid-1];
       // translate into a recursive call

else if (key > a[mid]) search a[mid + 1] through a[last];
       // translate into a recursive call
```


Binary Search (5 of 9) Pseudo-code

```
int a[some_size];
Algorithm search a[first] through a[last]
//   Precondition: array a is sorted ascending
To locate the value key:
found = false; // so far
if  (first > last)      found = false;      // a stopping case
else
{   mid = some approximate midpoint between 0 and final_index;
    if (key == a[mid])
    {   found = true;
        locate = mid;
    }
    else if (key < a[mid])    search a[first] through a[mid - 1];
        // translate into a recursive call
    else if (key > a[mid]) search a[mid + 1] through a[last];
        // translate into a recursive call
```

Display 12.6 Recursive Function for Binary Search (1 of 2)

```
#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];
    const int final_index = ARRAY_SIZE - 1;
    for (int i = 0; i < 10; i++)     a[i] = 2*i;
    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;
}
```


Display 12.6 Recursive version of Binary Search (2 of 2)

```
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);
    }
}
```


Binary Search (6 of 9)

CHECKING THE RECURSION

1. There is no infinite recursion. On each recursive call, the value of first is increased or the value of last is decreased (by half the distance between them). The stopping conditions are 1) first > last (search fails) or an instance of the key is found. If the key is not found, the first > last condition is guaranteed to be reached.
2. Each stopping case performs the correct action for that case:

   There are two stopping cases. We will see that both are correct:

   i) first > last

   In this case, there are no elements between a[first] and a[last], so the key is not in the segment. Here, found is correctly set to false.

   ii) a[mid] == key

   Here, the algorithm correctly sets found to true, and location to mid.

Binary Search (7 of 9)

EFFICIENCY

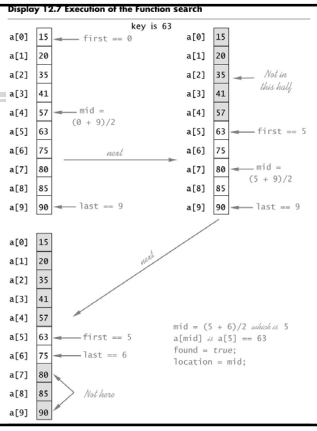
A measure of efficiency of search algorithms is the number of comparisons that are necessary to carry out a search as a function of the number of items being searched.

A failing search usually requires the most comparisons so we use that to compare the efficiency of the linear search and the binary search.

The binary search eliminates half the remaining elements from consideration at each comparison whereas the linear search requires comparison to the key of every element in the array.

754

Display 12.7 Execution of the Function Search



Binary Search (8 of 9)

EFFICIENCY (continued)

The binary search algorithm is much faster than linear search (examining each element, one at a time). We gain more when the array being searched is larger. A failing search uses the most comparisons.

Array Size	Comparisons necessary for a failing Linear Search	Binary Search
10	10	3
100	100	7
1000	1000	10

Binary Search (9 of 9)

- An iterative version of search is given in Display 12.8.
- The iterative version was derived from the recursive version.
- The local variables first and last mirror the roles of the recursive parameters first and last.
- It frequently makes sense to develop the recursive algorithm even when the intention is to convert to an iterative algorithm.

Display 12.8 Iterative Version of Binary Search

```
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.
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;
    }
}
```


A Programming Example
A Recursive Member Function(1 of 3)

- A class member function can be recursive in exactly the same way that ordinary functions are recursive.
- Display 12.9 modifies class BankAccount from Display 6.6 by overloading member function `update` by writing two versions:
  - A no parameter version that posts one year of simple interest to the bank account balance.
  - An int parameter version that updates the account by posting interest for the argument number of years.
- Second version algorithm:
  - If number of years is 1, then // Stopping case
    call the no parameter version of update
  - If the number of years > 1, then // Recursive case.
    Recursively call update to post number - 1 years of interest
    call the no parameter version of update to post one year's interest.

A Programming Example
A Recursive Member Function(2 of 3)

- To see that this algorithm is correct, we check the three points from the earlier section, "Recursive Design Techniques."

  1. There is no infinite recursion:
     Each recursive call is made with an argument that is decreased by 1, which will eventually reach 1, a stopping case.
  2. Each stopping case performs the correct action for that case.
     The one stopping case is for posting 1 year of interest. We checked the correctness of this function in Chapter 6.
  3. For each recursive case, *if* all recursive calls perform correctly, *then* the case performs correctly.
     The recursive case i.e., with argument years > 1, works correctly: If the recursive call correctly posts years - 1 worth of interest, then all that is needed to post an additional year's interest is to call the zero parameter version of update. We can conclude that the recursive case is correct.

A Programming Example
A Recursive Member Function(3 of 3)

- The two functions named `update` are different functions.
- The compiler distinguishes functions in a call by looking at the name first then at the argument list.
- If the argument list has the same number of arguments and the sequence of types is the same, then the function that matches best is called.
- For cases where the match is not exact, C++ has rules for what constitutes a "best match". You will see the rules in later courses.

Display 12.9 A Recursive Member Function update(years) (1 of 4)

```
//    The class BankAccount here is modified from the class in Display 6.6.
#include <iostream>
class BankAccount
{
public:
   BankAccount(int dollars, int cents, double rate);
   //    Sets account balance to $dollars.cents and the interest rate to rate percent.
   BankAccount(int dollars, double rate);
   //    Sets the account balance to $dollars.00 and the interest rate to rate percent.
   BankAccount( );
   //    Sets the account balance to $0.00 and  the interest rate to 0.0%.
   void update( );
   //    Postcondition: A year of simple interest is 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 percent.
   void output(std::ostream& outs);
   //    Pre:   If outs is a file output stream, then outs has been connected  to a file.
   //    Post: Account balance and interest rate have been written to the stream out.
```

Display 12.9 A Recursive Member Function update(years) (2 of 4)

```
//  The class BankAccount here is modified from  Display 6.6.
private:
   double balance;
   double interest_rate;
   double fraction(double percent);
   //Converts a percent to a fraction. For example, fraction(50.3) returns 0.503.
};
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;
}

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


Display 12.9 A Recursive Member Function update(years) (3 of 4)

```
void BankAccount::update(int years)
{  if (years == 1)   update( );
   else if (years > 1)
   {  update(years - 1);
      update( );
   }
}
BankAccount::BankAccount(int dollars, int cents, double rate)
{  balance = dollars + 0.01*cents;
   interest_rate = rate;
}
BankAccount::BankAccount(int dollars, double rate)
{  balance = dollars;
   interest_rate = rate;
}

BankAccount::BankAccount()
{  balance = 0;
   interest_rate = 0.0;
}
```


Display 12.9 A Recursive Member Function update(years) (4 of 4)

```
double BankAccount::fraction(double percent)
{     return (percent/100.0);    }

double BankAccount::get_balance( )
{  return balance;    }

double BankAccount::get_rate( )
{     return interest_rate;   }
//    Uses iostream:

void BankAccount::output(ostream& outs)
{  outs.setf(ios::fixed);
   outs.setf(ios::showpoint);
   outs.precision(2);
   outs << "Account balance $" << balance << endl;
   outs << "Interest rate " << interest_rate << "%" << endl;
}
```


Recursion and overloading

Do not confuse overloading and recursion. When you overload a function, you are giving two different functions the same name (but different argument lists so that C++ can distinguish them). If the definition of one of the functions calls the other, that is not recursion.

In a recursive function definition, the definition contains a call to exactly the same function, not to a function that has the same name but a different argument list (that C++ can distinguish).

It is not too serious to confuse these ideas as both are legal. However it is important that you get the terminology straight so you can communicate clearly with other programmers, and so you understand the underlying processes.

CHAPTER SUMMARY

- If a problem can be reduced to smaller instances of the same problem, then a recursive solution is likely to be easy to find and implement.
- A recursive algorithm for a function definition normally contains two kinds of cases: one or more cases that include at least one recursive call and one or more stopping cases in which the problem is solved without any recursive calls.
- When writing a recursive function definition, always check to see that the function will not produce infinite recursion.
- When you define a recursive function, use the three criteria given in the subsection "Recursive Design Techniques" to check that the function is correct.
- When you design a recursive function to solve a task it is often necessary to solve a more general problem than the given task. This may be required to allow for the proper recursive calls, since the smaller problems may not be exactly the same problem as the given task. For example, in the binary search problem, the task was to search an entire array, but the recursive solution is an algorithm to search any portion of the array (either all of it or a part of it).