Chapter 14

Recursion
Overview

14.1 Recursive Functions for Tasks
14.2 Recursive Functions for Values
14.3 Thinking Recursively
14.1

Recursive Functions for Tasks
Recursive Functions for Tasks

- A recursive function contains a call to itself
- When breaking a task into subtasks, it may be that the subtask is a smaller example of the same task
  - Searching an array could be divided into searching the first and second halves of the array
  - Searching each half is a smaller version of searching the whole array
  - Tasks like this can be solved with recursive functions
Case Study: Vertical Numbers

Problem Definition:

- `void write_vertical( int n );`
  //Precondition: n >= 0
  //Postcondition: n is written to the screen vertically
  // with each digit on a separate line
Case Study: Vertical Numbers

Algorithm design:

- **Simplest case:**
  If $n$ is one digit long, write the number

- **Typical case:**
  1) Output all but the last digit vertically
  2) Write the last digit

- Step 1 is a smaller version of the original task
- Step 2 is the simplest case
The write_vertical algorithm:

```cpp
if (n < 10)
{
    cout << n << endl;
}
else // n is two or more digits long
{
    write_vertical(n with the last digit removed);
    cout << the last digit of n << endl;
}
```
Case Study: Vertical Numbers (cont.)

- Translating the pseudocode into C++
  - \( n / 10 \) returns \( n \) with the last digit removed
    - \( 124 / 10 = 12 \)
  - \( n \% 10 \) returns the last digit of \( n \)
    - \( 124 \% 10 = 4 \)

- Removing the first digit would be just as valid for defining a recursive solution
  - It would be more difficult to translate into C++
A Recursive Output Function  (part 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;
}

// 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;
    }
}
A Recursive Output Function (part 2 of 2)

Sample Dialogue

write_vertical(3):
  3
write_vertical(12):
  1
  2
write_vertical(123):
  1
  2
  3
Tracing a Recursive Call

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

Calls `write_vertical(12)`

resume

Function call ends

Output 3
Tracing write_vertical(12)

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

Calls write_vertical(1)
resume
Function call ends
Output 2
Tracing write_vertical(1)

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

Simplest case is now true

Function call ends

Output 1
A Closer Look at Recursion

- **write_vertical uses recursion**
  - Used no new keywords or anything "new"
  - It simply called itself with a different argument

- **Recursive calls are tracked by**
  - Temporarily stopping execution at the recursive call
    - The result of the call is needed before proceeding
  - Saving information to continue execution later
  - Evaluating the recursive call
  - Resuming the stopped execution
How Recursion Ends

- Eventually one of the recursive calls must not depend on another recursive call

- Recursive functions are defined as:
  - One or more cases where the task is accomplished by using recursive calls to do a smaller version of the task
  - One or more cases where the task is accomplished without the use of any recursive calls

  These are called **base cases** or stopping cases
"Infinite" Recursion

A function that never reaches a base case, in theory, will run forever
- In practice, the computer will run out of resources and the program will terminate abnormally
Example: Infinite Recursion

Function write_vertical, without the base case

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

will eventually call write_vertical(0), which will call write_vertical(0), which will call write_vertical(0), which will call write_vertical(0), which will call write_vertical(0), which will call write_vertical(0), which will call write_vertical(0), ...

Stacks for Recursion

- Computers use a structure called a stack to keep track of recursion

  - A stack is a memory structure analogous to a stack of paper
    - To place information on the stack, write it on a piece of paper and place it on top of the stack
    - To place more information on the stack, use a clean sheet of paper, write the information, and place it on the top of the stack
    - To retrieve information, only the top sheet of paper can be read, and thrown away when it is no longer needed
Last-in - First-out

- A stack is a \textit{last-in-first-out} memory structure
  - The last item placed is the first that can be removed

- Whenever a function is called, the computer uses a "clean sheet of paper"
  - The function definition is copied to the paper
  - The arguments are plugged in for the parameters
  - The computer starts to execute the function body
Stacks and The Recursive Call

- When execution of a function definition reaches a recursive call
  - Execution stops
  - Information is saved on a "clean sheet of paper" to enable resumption of execution later
  - This sheet of paper is placed on top of the stack
  - A new sheet is used for the recursive call
    - A new function definition is written, and arguments are plugged into parameters
    - Execution of the recursive call begins
The Stack and Ending Recursive Calls

When a recursive function call is able to complete its computation with no recursive calls

- The computer retrieves the top "sheet of paper" from the stack and resumes computation based on the information on the sheet
- When that computation ends, that sheet of paper is discarded and the next sheet of paper on the stack is retrieved so that processing can resume
- The process continues until no sheets remain in the stack
Activation Frames

- The computer does not use paper
- Portions of memory are used
  - The contents of these portions of memory is called an activation frame
  - The activation frame does not actually contain a copy of the function definition, but references a single copy of the function
Because each recursive call causes an activation frame to be placed on the stack

- infinite recursion can force the stack to grow beyond its limits to accommodate all the activation frames required
- The result is a stack overflow
- A stack overflow causes abnormal termination of the program
Recursion versus Iteration

- Any task that can be accomplished using recursion can also be done without recursion
  - A nonrecursive version of a function typically contains a loop or loops
  - A non-recursive version of a function is usually called an iterative-version
  - A recursive version of a function
    - Usually runs slower
    - Uses more storage
    - May use code that is easier to write and understand
DISPLAY 14.2  Iterative Version of the Function in Display 14.1

```cpp
1   //Uses iostream:
2   void write_vertical(int n)
3   {
4       int tens_in_n = 1;
5       int left_end_piece = n;
6       while (left_end_piece > 9)
7           {
8               left_end_piece = left_end_piece/10;
9               tens_in_n = tens_in_n*10;
10          }
11     //tens_in_n is a power of ten that has the same number
12     //of digits as n. For example, if n is 2345, then
13     //tens_in_n is 1000.
14
15     for (int power_of_10 = tens_in_n;
16         power_of_10 > 0; power_of_10 = power_of_10/10)
17         {
18             cout << (n/power_of_10) << endl;
19             n = n % power_of_10;
20        }
21   }
```
14.2

Recursive Functions for Values
Recursive Functions for Values

- **Recursive functions can also return values**
- The technique to design a recursive function that returns a value is basically the same as what you have already seen
  - One or more cases in which the value returned is computed in terms of calls to the same function with (usually) smaller arguments
  - One or more cases in which the value returned is computed without any recursive calls (*base case*)
To define a new power function that returns an int, such that

\[ \text{int } y = \text{power}(2,3); \]

places 23 in y

- Use this definition:
  \[ x^n = x^{n-1} \times x \]

- Translating the right side to C++ gives:
  \[ \text{power}(x, n-1) \times x \]

- The base case: \( n = 0 \) and power should return 1
The Recursive Function `power`

```cpp
// 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);
}
```

**Sample Dialogue**

- 3 to the power 0 is 1
- 3 to the power 1 is 3
- 3 to the power 2 is 9
- 3 to the power 3 is 27
Tracing power(2,1)

```c
int power(2, 1)
{
    ...
    if (n > 0)
        return ( power(2, 1-1) * 2);
    else
        return (1);
}
```

Call to power(2,0)

resume

return 2

Function Ends
Tracing power(2,0)

```c
int power(2, 0)
{
    ...
    if (n > 0)
        return ( power(2, 0-1) * 2);
    else
        return (1);
}
```

Function call ends

1 is returned
Tracing power(2, 3)

- Power(2, 3) results in more recursive calls:
  - 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)
Evaluating the Recursive Function Call $\text{power}(2, 3)$

<table>
<thead>
<tr>
<th>Sequence of recursive calls</th>
<th>How the final value is computed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1$</td>
<td>$1$</td>
</tr>
<tr>
<td>$\text{power}(2, 0) \times 2$</td>
<td>$1$</td>
</tr>
<tr>
<td>$\text{power}(2, 1) \times 2$</td>
<td>$1 \times 2 = 2$</td>
</tr>
<tr>
<td>$\text{power}(2, 2) \times 2$</td>
<td>$2 \times 2 = 4$</td>
</tr>
<tr>
<td>$\text{power}(2, 3)$</td>
<td>$4 \times 2 = 8$</td>
</tr>
</tbody>
</table>

$\text{Start Here}$

power$(2, 3)$ is $8$
14.3

Thinking Recursively
Thinking Recursively

When designing a recursive function, you do not need to trace out the entire sequence of calls.

- If the function returns a value
  - Check that there is no infinite recursion: Eventually a stopping case is reached
  - Check that each stopping case returns the correct value
  - For cases involving recursion: if all recursive calls return the correct value, then the final value returned is the correct value
Reviewing the power function

- There is no infinite recursion
  - Notice that the second argument is decreased at each call. Eventually, the second argument must reach 0, the stopping case.

```c
int power(int x, int n)
{
    ...
    if (n > 0)
        return ( power(x, n-1) * x);
    else
        return (1);
}
```
Review of power (cont.)

Each stopping case returns the correct value
- power(x, 0) should return \( x^0 = 1 \) which it does

```c
int power(int x, int n)
{
    ...
    if (n > 0)
        return ( power(x, n-1) * x);
    else
        return (1);
}
```
All recursive calls return the correct value so the final value returned is correct

- If \( n > 1 \), recursion is used. So \( \text{power}(x, n-1) \) must return \( x^{n-1} \) so \( \text{power}(x, n) \) can return \( x^{n-1} \times n = x^n \) which it does

```c
int power(int x, int n)
{
    ...
    if (n > 0)
        return ( power(x, n-1) * x);  
    else
        return (1);  
}
```
Recursive void-functions

The same basic criteria apply to checking the correctness of a recursive void-function:

- Check that there is no infinite recursion
- Check that each stopping case performs the correct action for that case
- Check that for each recursive case: if all recursive calls perform their actions correctly, then the entire case performs correctly
Case Study: Binary Search

A binary search can be used to search a sorted array to determine if it contains a specified value

- The array indexes will be 0 through final_index
- Because the array is sorted, we know
  \[ a[0] \leq a[1] \leq a[2] \leq \ldots \leq a[\text{final\_index}] \]
- If the item is in the list, we want to know where it is in the list
Binary Search: Problem Definition

- The function will use two call-by-reference parameters to return the outcome of the search
  - One, called found, will be type bool. If the value is found, found will be set to true. If the value is found, the parameter, location, will be set to the index of the value

- A call-by-value parameter is used to pass the value to find
  - This parameter is named key
Binary Search

Problem Definition (cont.)

Pre and Postconditions for the function:

//precondition: a[0] through a[final_index] are sorted in increasing order

//postcondition: if key is not in a[0] - a[final_index]  
  //    found = = false;
otherwise
  //    found = = true
Our algorithm is basically:
- Look at the item in the middle
  - If it is the number we are looking for, we are done
  - If it is greater than the number we are looking for, look in the first half of the list
  - If it is less than the number we are looking for, look in the second half of the list
Here is a first attempt at our algorithm:

```java
found = false;
mid = approx. 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];
```
Since searching each of the shorter lists is a smaller version of the task we are working on, a recursive approach is natural.

- We must refine the recursive calls
  - Because we will be searching subranges of the array, we need additional parameters to specify the subrange to search
  - We will add parameters first and last to indicate the first and last indices of the subrange
Here is our first refinement:
found = false;
mid = approx. 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];
We must ensure that our algorithm ends
- If key is found in the array, there is no recursive call and the process terminates
- What if key is not found in the array?
  - At each recursive call, either the value of first is increased or the value of last is decreased
  - If first ever becomes larger than last, we know that there are no more indices to check and key is not in the array
Pseudocode for Binary Search

```c
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];
}
```
Execution of the Function search

key is 63

<table>
<thead>
<tr>
<th>a[0] 15</th>
<th>a[0] 15</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;first == 0&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;mid = (0 + 9)/2&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;last == 9&quot;</td>
<td></td>
</tr>
</tbody>
</table>

Not in this half

Mid = (5 + 9)/2 which is 5
a[mid] is a[5] == 63
found = true;
location = mid;

Not here

Next
Binary Search Writing the Code

Function search implements the algorithm:

- Function search interface:
  
  ```
  void search(const int a[], int first, int last, 
  int key, bool& found, int& location);
  //precondition: a[0] through a[final_index] are 
  // sorted in increasing order
  //postcondition: if key is not in a[0] - a[final_index] 
  // found = = false; otherwise
  // found = = true
  ```
Recursive Function for Binary Search (part 1 of 2)

// 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];
    const int final_index = ARRAY_SIZE - 1;

    // This portion of the program contains some code to fill and sort
    // the array a. The exact details are irrelevant to this example.
    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;
}
Recursive Function for Binary Search (part 2 of 2)

```c
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
Checking the Recursion

There is no infinite recursion
- On each recursive call, the value of first is increased or the value of last is decreased. Eventually, if nothing else stops the recursion, the stopping case of first > last will be called
Each stopping case performs the correct action
- If first > last, there are no elements between \(a[first]\) and \(a[last]\) so key is not in this segment and it is correct to set found to false
- If \(k = a[mid]\), the algorithm correctly sets found to true and location equal to mid
- Therefore both stopping cases are correct
Binary Search
Checking the Recursion (cont.)

For each case that involves recursion, if all recursive calls perform their actions correctly, then the entire case performs correctly.

Since the array is sorted...

- If key < a[mid], key is in one of elements a[first] through a[mid-1] if it is in the array. No other elements must be searched... the recursive call is correct.
- If key > a[mid], key is in one of elements a[mid+1] through a[last] if it is in the array. No other elements must be searched... the recursive call is correct.
Binary Search Efficiency

The binary search is extremely fast compared to an algorithm that checks each item in order:

- The binary search eliminates about half the elements between a[first] and a[last] from consideration at each recursive call.
- For an array of 100 items, the binary search never compares more than seven elements to the key.
- For an array of 100 items, a simple serial search will average 50 comparisons and may do as many as 100!
Binary Search
An Iterative Version

- The iterative version of the binary search may be run faster on some systems

- The algorithm for the iterative version was created by mirroring the recursive function
  - Even if you plan an iterative function, it may be helpful to start with the recursive approach
Iterative Version of Binary Search

Function Declaration

```c
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.
```

Function Definition

```c
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;
        }
    }
}
```
Program Example: A Recursive Member Function

- A member function of a class can be recursive
- The update function from class BankAccount of Display 6.6 is will be overloaded to create a recursive version
  - Update adds interest for a specified number of years to an account balance
DISPLAY 14.9  A Recursive Member Function (part 1 of 2)

```cpp
// Program to demonstrate the recursive member function update(years).
#include <iostream>
using namespace std;

// Class for a bank account:
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.
};

int main()
{
  BankAccount your_account(100, 5);
  your_account.update(10);
  cout.setf(ios::fixed);
  (continued)
```
DISPLAY 14.9 A Recursive Member Function (part 2 of 2)

```cpp
42    cout.setf(ios::showpoint);
43    cout.precision(2);
44    cout << "If you deposit $100.00 at 5% interest, then\n"  
45        << "in ten years your account will be worth $"  
46        << your_account.get_balance() << endl;
47    return 0;
48  
49  void BankAccount::update()
50  {
51    balance = balance + fraction(interest_rate)*balance;
52  }
53  
54  void BankAccount::update(int years)
55  {
56    if (years == 1)
57        {  
58          update();
59        }
60    else if (years > 1)
61        {
62          update(years - 1);
63          update();
64        }
65  }
66 

<Definitions of the other member functions are given in Display 10.5 and Display 10.6, but you need not read those definitions in order to understand this example.>

Sample Dialogue

If you deposit $100.00 at 5% interest, then in ten years your account will be worth $162.89
update uses one parameter, years, and the following algorithm:

- If the number of years is 1, then (// stopping case) call the other function named update

- If the number of years is greater than 1, then
  - Make a recursive call to post years – 1 worth of interest then call the other update function for one year's interest
Function update
Checking the Recursion

- There is no infinite recursion
  - Each recursive call reduces the number of years by one until years is 1, a stopping case

- Each stopping case performs the correct action
  - One stopping case is years == 1. It calls the other update function which was previously checked for correctness
For the cases that involve recursion, if all recursive calls perform correctly, the entire case performs correctly:

- When years > 1, the recursive call correctly posts years − 1 worth of interest, then calls the other update function to post one additional year's interest. The other update function correctly posts interest for one year. Then the entire action for years > 1 will be correct.
Overloaded Functions

There is no confusion (for the computer) between the two versions of update

- When the recursive version of update calls the version of update with no parameters, that is not a recursive call
- Only calls to the version of update with the exact same function declaration are recursive calls