Pearson Problem Solving with C++ TENTH EDITION Savitch GLOBAL EDITION

Unit 2 (ch4, ch5, ch14)

Functions & Recursion

Prof. Chien-Nan (Jimmy) Liu
Dept. of Electronics & Electrical Engr.
Nat'l Yang Ming Chiao Tung Univ.

Tel: (03)5712121 ext:31211
E-mail: jimmyliu@nycu.edu.tw
http://mseda.ee.nctu.edu.tw/jimmyliu

Overview

2.1 Functionalize a Program

2.2 Passing Parameters into a Function

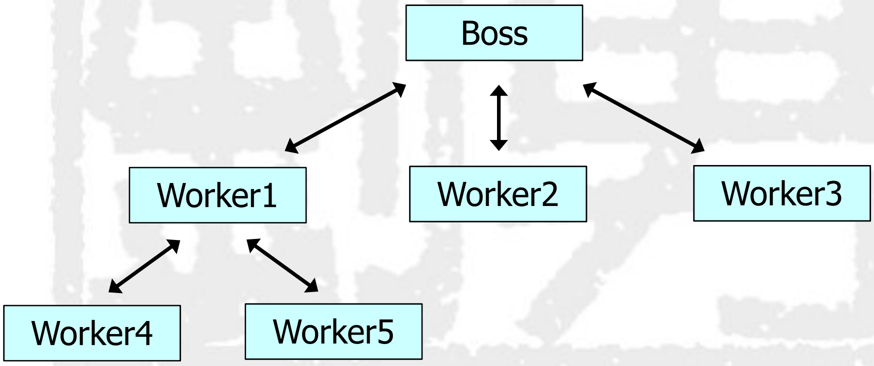
2.3 Overloading Function Names

2.4 Recursive Function Calls

2.5 Thinking Recursively

Top-Down Design

- Top Down Design (also called stepwise refinement)
  - Break the algorithm into subtasks
  - Break each subtask into smaller subtasks
  - Eventually the smaller subtasks are trivial to implement in the programming language



Benefits of Top-Down Design

- Motivations for "functionalizing" a program
  - Divide-and-conquer makes program development more manageable
  - Software reusability—using existing functions as building blocks to create new programs
  - Avoid repeating code in a program
    - Packaging code as a function allows the code to be executed from different locations in a program
- A programmer only needs to know what will be produced after arguments are put into the box
- Designing with the black box in mind allows us
  - Easily change or improve a function without forcing programmers changing what they have done

Function Call Syntax

- Function_name (*Argument_List*)
  - *Argument_List* is a comma separated list: (Argument_1, Argument_2, … , Argument_Last)
- Example:
  - side = sqrt(area);
  - cout << “2.5 to the power 3.0 is “ << pow(2.5, 3.0);
- The corresponding library must be “included” in a program to make the predefined functions available
- Ex: to include the math library containing sqrt():
  #include <cmath>

Example of Function Call

```
//DISPLAY 4.1 A Function Call
//Computes the size of a dog house that can
be purchased given the user's budget.
#include <iostream>
#include <cmath>
using namespace std;

int main( )
{
   const double COST_PER_SQ_FT = 10.50;
   double budget, area, length_side;

   cout << "Enter the amount budgeted for
            your dog house $";
   cin >> budget;

   area = budget/COST_PER_SQ_FT;

   length_side = sqrt(area);
```

```
   cout.setf(ios::fixed);
   cout.setf(ios::showpoint);
   cout.precision(2);
   cout << "For a price of $" << budget << endl
        << "I can build you a luxurious square dog
            house\n"
        << "that is " << length_side
        << " feet on each side.\n";

   return 0;
}
```

Programmer-Defined Functions

- Two components for a function definition
  - Function declaration (or function prototype)
    - Shows how the function is called
    - Must appear in the code BEFORE the function can be called
    - Syntax:
      Type_returned *Function_Name* (Parameter_List) ;
      //Comment describing what function does
  - Function definition (or function body)
    - Description for real actions in this function
    - Can appear before or after the function is called
    - Syntax:

```
Type_returned  Function_Name (Parameter_List)
{
        //code to make the function work
}
```

Must match to each other

Only header is required with an extra semicolon.

Function Declaration

- Function declaration require more information:
  - Tells the return type
  - Tells the name of the function
  - Tells how many arguments are needed
  - Tells the types of the arguments
  - Tells the formal parameter names
    - Formal parameters are like placeholders for the actual arguments used when the function is called
- Example:

```
double totalCost(int numberPar, double pricePar);
// Compute total cost including 5% sales tax on
// numberPar items at cost of pricePar each
```

Function Definition

- Provides the same information as the declaration
- Describes how the function does its task
- Example:

return type   function header

```
double totalCost (int numberPar, double pricePar)
{
    const double TAX_RATE = 0.05; //5% tax
    double subtotal;
    subtotal = pricePar * numberPar;
    return (subtotal + subtotal * TAX_RATE);
}
```

function body

Example for User-Defined Function

Function declaration

```
//DISPLAY 4.3 A Function Definition
#include <iostream>
using namespace std;

double totalCost(int numberPar, double pricePar);
//Computes the total cost, including 5% sales tax,
//on numberPar items at a cost of pricePar each.

int main( )
{
    double price, bill;
    int number;

    cout << "Enter the number of items purchased: ";
    cin >> number;
    cout << "Enter the price per item $";
    cin >> price;

    bill = totalCost(number, price);
```

Function call

```
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << number << " items at "
         << "$" << price << " each.\n"
         << "Final bill, including tax, is $" << bill
         << endl;

    return 0;
}
```

Function definition

```
double totalCost(int numberPar, double pricePar)
{
    const double TAX_RATE = 0.05; //5% sales tax
    double subtotal;

    subtotal = pricePar * numberPar;
    return (subtotal + subtotal*TAX_RATE);
}
```

The Function Call

- After the function is well designed (declaration and definition), we can use function call to use it
  - Tells the name of the function and lists the arguments
- Can be used in a statement where the returned value makes sense
  - Ex: double bill = totalCost(number, price);
- The values of the arguments are plugged into the formal parameters by call-by-value mechanism
  - The first argument is used for the first parameter, the second argument for the second parameter, and so forth
  - The value plugged into the formal parameter is used in the function body only
    - The "copy" of input values will not affect the original variables

Function Call Procedure

```
double totalCost(int numberPar, double pricePar)
{
   const double TAX_RATE = 0.05; //5% sales tax
   double subtotal;

   subtotal = pricePar * numberPar;
   return (subtotal + subtotal * TAX_RATE);
}
```

In totalCost function:
numberPar = 2
pricePar = 10.10

20.20 + 20.20*0.05 = 21.21

bill = return value
= 21.21

```
int main()
{
   double price, bill;
   int number;
   cout << "Enter the number of item purchased: ";
   cin >> number;    2
   cout << "Enter the price per item $";
   cin >> price;    10.10

   bill = totalCost(number, price);

   cout.setf(ios::fixed);
   cout.setf(ios::showpoint);
   cout.precision(2);
   cout << number << "item at"
       << "$" << price << "each .\n"
       << "Final bill, including tax, is $" << bill << endl;
   return 0;
}
```

Default Arguments

- If a function is frequently used with the same argument value for a particular parameter
  → specify that such a parameter has a default value
- When an argument is omitted in a function call, the compiler inserts the default value of that argument
- Default arguments must be the rightmost (trailing) arguments in a function's parameter list
- Default arguments must be specified with the first occurrence of the function name
  - Typically in the function prototype
- Default values can be any expression, including constants, global variables or function calls

Example for Default Arguments (1/2)

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

//function prototype that specifies default arguments
int boxVolume(int length = 1, int width = 1, int height = 1);

int main()
{
   //no arguments--use default values for all dimensions
   cout << "The default box volume is: " << boxVolume();

   //specify length; default width and height
   cout << "\n\nThe volume of a box with length 10,\n"
        << "width 1 and height 1 is: " << boxVolume(10);

   //specify length and width; default height
   cout << "\n\nThe volume of a box with length 10,\n"
        << "width 5 and height 1 is: " << boxVolume(10, 5);
```

Example for Default Arguments (2/2)

```
 //specify all arguments
   cout << "\n\nThe volume of a box with length 10,\n"
        << "width 5 and height 2 is: " << boxVolume(10, 5, 2) << endl;
} //end main

int boxVolume(int length, int width, int height)
{
   return length * width * height;
}
```

```
The default box volume is : 1

The volume of a box with length 10,
width 1 and height 1 is : 10

The volume of a box with length 10,
width 5 and height 1 is : 50

The volume of a box with length 10,
width 5 and height 2 is : 100
```


Overview

Local Variables

- Variables declared in a function:
  - Called "local variables" to that function
  - Cannot be used from outside the function
  - Have the function as their scope
- Variables declared in the main part of a program:
  - Are local to the main part of the program
  - Cannot be used from outside the main part
  - Have the main part as their scope
- Local variables cannot be used in another functions. Only their values can be passed
  - Passed-by-value mechanism

Global Variables

- Global variables
  - Available to more than one function as well as the main part of the program
  - Declared outside any function body (including main)
- Can be used when more than one function must use a common variable
  - Generally make programs more difficult to understand and maintain
- Ex:

```
const double PI = 3.14159;
double volume(double);
int main()
{…}
```

  - PI is available to main function and function volume

Code Example for Global Constant

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

const double PI = 3.14159;
double area(double radius);
double volume(double radius);

int main( )
{
    double radiusOfBoth, areaOfCircle, volumeOfSphere;

    cout << "Enter a radius to use for both a circle\n"
         << "and a sphere (in inches): ";
    cin >> radiusOfBoth;

    areaOfCircle = area(radiusOfBoth);
    volumeOfSphere = volume(radiusOfBoth);
    cout << "Radius = " << radiusOfBoth << " inches\n"
         << "Area of circle = " << areaOfCircle
         << " square inches\n"
         << "Volume of sphere = " << volumeOfSphere
         << " cubic inches\n";

    return 0;
}
```

```
double area(double radius)
{
    return (PI * pow(radius, 2));
}

double volume(double radius)
{
    return ((4.0/3.0) * PI * pow(radius, 3));
}
```

Sample Dialogue

```
Enter a radius to use for both a circle
and a sphere (in inches): 2
Radius = 2 inches
Area of circle = 12.5664 square inches
Volume of sphere = 33.5103 cubic inches
```


Formal Parameters are Local

- Formal parameters are actually local variables to the function definition
  - Just as if they were declared in the function header
  - Do NOT re-declare the formal parameters in the function body → double declaration
- Call-by-value mechanism gives the initial values
  - When a function is called, the formal parameters are initialized to the given values from the function call
  - The formal parameters can be altered later in the function, but has no impact to the original variables

Formal Parameters Used as Local

Sample Dialogue

```
Welcome to the offices of
Dewey, Cheatham, and Howe.
The law office with a heart.
Enter the hours and minutes of your consultation:
2 45
For 2 hours and 45 minutes, your bill is $1650.00
```

```
//DISPLAY 4.13: Law office billing program.
#include <iostream>
using namespace std;

const double RATE = 150.00;
double fee(int hoursWorked, int minutesWorked);

int main( )
{
    ......
    bill = fee(hours, minutes);
    ......
}
```

```
double fee(int hoursWorked, int minutesWorked)
{
    int quarterHours;

    minutesWorked = hoursWorked*60 +
                minutesWorked;
    quarterHours = minutesWorked/15;
    return (quarterHours*RATE);
}
```

2 45

2*60+45=165

165/15=11

11*150=1650

Call-by-Reference Parameters

- Call-by-value (default mechanism) means that the formal parameters receive values only
  - Changing the values of internal variables will not change the original data
- Call-by-reference parameters allow us to change the variable used in the function call
  - Pass the "address" for exchanging data between functions
  - Arguments for call-by-reference parameters must be variables, not constant numbers

Call-by-Reference Example

```
void getInput(double& fVariable)
 {
      using namespace std;
      cout << " Convert a Fahrenheit temperature to Celsius.\n"
            << " Enter a temperature in Fahrenheit: ";
      cin  >> fVariable;
}
```

- '**&**' symbol (ampersand) identifies *fVariable* as a call-by-reference parameter
  - Used in both declaration and definition !!
- Whatever is done to a formal parameter in the function, is actually done to the value at the given memory addr.
  - Work almost as if the argument variable substitutes the formal parameter, not the argument's value

Behavior of Call-by-Reference

Assume the two variables are assigned at the memory address 1010 and 1012

```
int main( )
{
   int firstNum, secondNum;

   getNumbers(firstNum, secondNum);
   swapValues(firstNum, secondNum);
   showResults(firstNum, secondNum);
   return 0;
}
void getNumbers(int& input1, int& input2)
{
   using namespace std;
   cout << "Enter two integers: ";
   cin >> input1
      >> input2;
}
```

No return value is required because the variables have been updated inside function

the variable at memory location 1010

the variable at memory location 1012

Write result to memory location 1010

Write result to memory location 1012

Call by Reference v.s. Value

- Call-by-reference
  - The function definition:
    void f(int& ref_par);
  - The function call:
    f(age);

- Call-by-value
  - The function definition:
    void f(int var_par);
  - The function call:
    f(age);

The same address with two different names

Two different variables with the same value

Memory

Name	Location	Contents
age	**1001**	**34**
initial	1002	A
hours	1003	23.5
	1004	

Choosing Parameter Types

- Call-by-value and call-by-reference parameters can be mixed in the same function
- Ex: void goodStuff(int& par1, int par2, double& par3);
  - par1 and par3 are call-by-reference parameters
    - Changes in par1 and par3 change the argument variable
  - par2 is a call-by-value parameter 沒有&
    - Changes in par2 do not change the argument variable
- How do you decide whether a call-by-reference or call-by-value formal parameter is needed?
  - Does the function need to change the value of the variable used as an argument?
  - Yes? Use a call-by-reference formal parameter
  - No? Use a call-by-value formal parameter

Comparing Argument Mechanism

call-by-value parameter:
not changed after function

call-by-reference parameter:
value updated after function

```
//DISPLAY 5.6 Comparing Argument Mechanisms
#include <iostream>

void doStuff(int par1Value, int& par2Ref);
//par1Value is a call-by-value formal parameter and
//par2Ref is a call-by-reference formal parameter.

int main( )
{
    using namespace std;
    int n1, n2;

    n1 = 1;
    n2 = 2;
    doStuff(n1, n2);
    cout << "n1 after function call = " << n1 << endl;
    cout << "n2 after function call = " << n2 << endl;
    return 0;
}
```

```
void doStuff(int par1Value, int& par2Ref)
{
    using namespace std;
    par1Value = 111;
    cout << "par1Value in function call = "
         << par1Value << endl;
    par2Ref = 222;
    cout << "par2Ref in function call = "
         << par2Ref << endl;
}
```

Sample Dialogue

```
par1Value in function call = 111
par2Ref in function call = 222
n1 after function call = 1
n2 after function call = 222
```

not affect
original value

Overview

2.1 Functionalize a Program

2.2 Passing Parameters into a Function

2.3 Overloading Function Names

2.4 Recursive Function Calls

2.5 Thinking Recursively

Overloading Function Names

- C++ allows more than one definition for the same function name
  - Very convenient when the "same" function is needed for different numbers or types of arguments
- If there are more than one definition using the same function name, how to choose correct one?
  - Clear rules are required to make decisions automatically
- Requirements for overloaded functions
  - Must have different numbers of formal parameters AND / OR
  - Must have at least one different type of parameter
  - Must return a value of the same type

Overloading Examples

- double ave(double n1, double n2)
  {
      return ((n1 + n2) / 2);
  }
- double ave(double n1, double n2, double n3)
  {
      return ((n1 + n2 + n3) / 3);
  }

3個 parameter

- Compiler checks the number and types of arguments in the function call to decide which function to use

  cout << ave(10, 20, 30);

  uses the second definition because of the argument numbers

Example of Overloading (1/2)

```
//DISPLAY 4.18 Overloading a Function Name
//Determines whether a round pizza or a rectangular
pizza is the best buy.
#include <iostream>

double unitPrice(int diameter, double price);
//Returns the price per square inch of a round pizza.
//The formal parameter named diameter is the diameter
of the pizza
//in inches. The formal parameter named price is the
price of the pizza.

double unitPrice(int length, int width, double
price);
//Returns the price per square inch of a rectangular
pizza
//with dimensions length by width inches.
//The formal parameter price is the price of the pizza.
```

```
int main( )
{
    using namespace std;
    int diameter, length, width;
    double priceRound, unitPriceRound,
          priceRectangular, unitPriceRectangular;

    cout << "Welcome to the Pizza Consumers
              Union.\n";
    cout << "Enter the diameter in inches"
          << " of a round pizza: ";

    cin >> diameter;
    cout << "Enter the price of a round pizza: $";

    cin >> priceRound;
    cout << "Enter length and width in inches\n"
          << "of a rectangular pizza: ";

    cin >> length >> width;
    cout << "Enter the price of a rectangular pizza: $";

    cin >> priceRectangular;
```


Example of Overloading (2/2)

```
unitPriceRectangular =
        unitPrice(length, width, priceRectangular);
unitPriceRound = unitPrice(diameter, priceRound);

cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
cout << endl
     << "Round pizza: Diameter = "
     << diameter << " inches\n"
     << "Price = $" << priceRound
     << " Per square inch = $" << unitPriceRound
     << endl
     << "Rectangular pizza: Length = "
     << length << " inches\n"
     << "Rectangular pizza: Width = "
     << width << " inches\n"
     << "Price = $" << priceRectangular
     << " Per square inch = $ “
     << unitPriceRectangular
     << endl;
```

```
 if (unitPriceRound < unitPriceRectangular)
      cout << "The round one is the better buy.\n";
 else
      cout << "The rectangular one is the better buy.\n";
      cout << "Buon Appetito!\n";
 return 0;
}
double unitPrice(int diameter, double price)
{
   const double PI = 3.14159;
   double radius, area;

   radius = diameter/static_cast<double>(2);
   area = PI * radius * radius;
   return (price/area);
}
double unitPrice(int length, int width, double price)
{
   double area = length * width;
   return (price/area);
}
```

Type Conversion Problem

- Given the definition

```
double mpg(double miles, double gallons)
{
    return (miles / gallons);
}
```

  What will happen if mpg is called in this way?

```
cout << mpg(45, 2) << " miles per gallon";
```

  The arguments are converted to type double (45.0 and 2.0)

- Given another mpg definition in the same program

```
int mpg(int goals, int misses)  // the Measure of Perfect Goals
{
    return (goals – misses);
}
```

  What happens if mpg is called this way now?

```
cout << mpg(45, 2) << " miles per gallon";
```

  Compiler chooses this function because the parameter types match

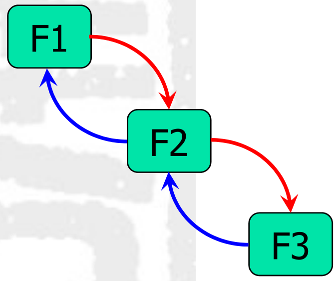
- Do not use the same function name for unrelated functions!!

Overview

Functions Calling Functions

- A function body may contain a call to another function
  - Ex: void order(int& n1, int& n2)

```
{
    if (n1 > n2)
        swapValues(n1, n2);
}
```



    - *swapValues* is another function to make n1 and n2 in ascending order
  - Return to the upper level only, not the top level
- The called function must be declared before it is used
  - Functions cannot be defined in the body of another function → often put all function declarations at top

Recursive Function Call

- A recursive function is a function that calls itself, either directly, or indirectly (through another function).
- The function only knows how to solve the simplest case(s), or so-called base case(s)
  - If the function is called with a base case, the function simply returns a result
- The recursive function divides a complex problem into
  - What it can do (base case) → return the result
  - What it cannot do → resemble the original problem, but be a slightly simpler or smaller version
  - The function calls a new copy of itself (recursion step) to solve the smaller problem
- Eventually base case gets solved
  - Return the result to solve the problem at upper level

Example: Factorial Function

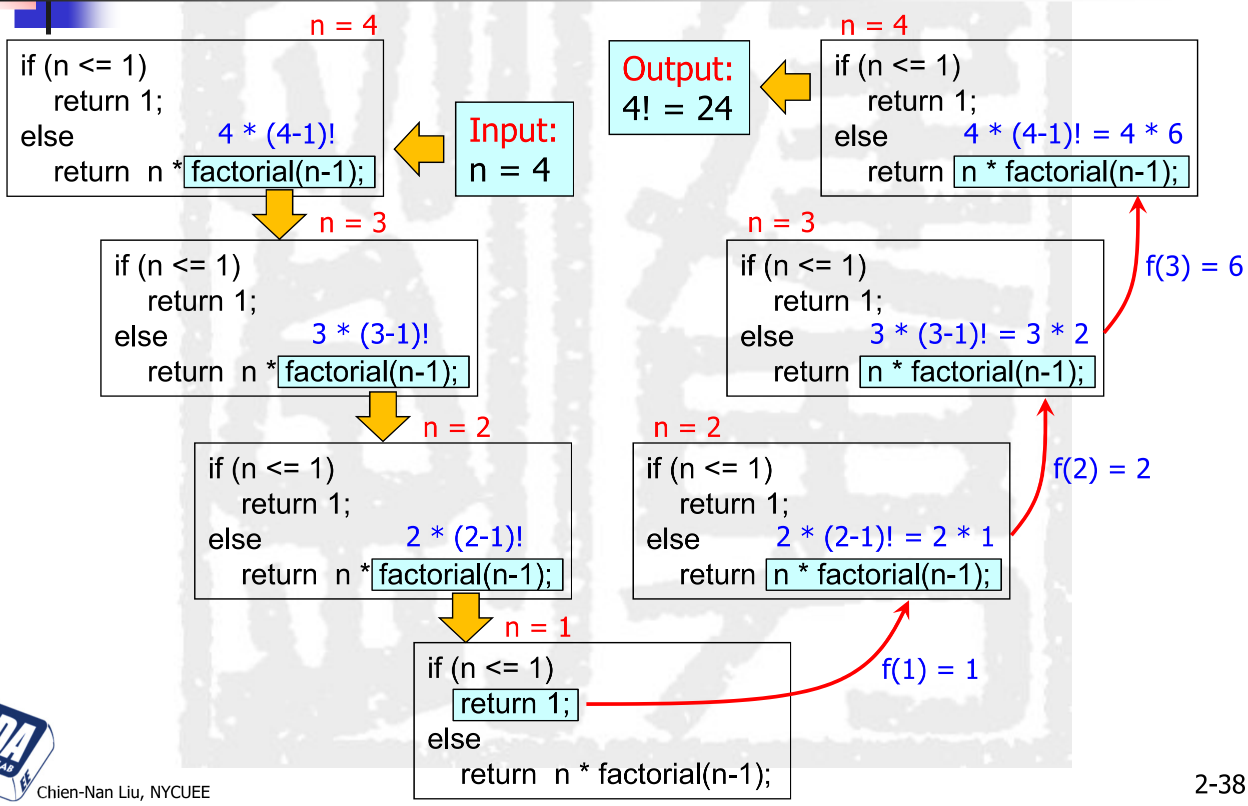
- n! = n x <u>(n-1) x (n-2) x … x 2 x 1</u>
  = n x (n-1)!
- 1! = 1, 0! = 1 → base case
- This function can be solved iteratively or recursively

```
// Iterative version
int factorial(int n)
{
    int product = 1;
    while (n > 0)
    {
        product = n * product;
        n--;
    }
    return product;
}
```

```
// Resursive version
int factorial(int n)
{
    if (n <= 1)   // base case
        return 1;
    else          // recursive step
        return  n * factorial(n-1);
}
```

n x (n-1)!

Recursive Execution of factorial(4)



Case Study: Vertical Numbers

- Problem Definition:
  - void writeVertical(int n);
    //Precondition: n >= 0
    //Postcondition: n is written to the screen vertically
    // with each digit on a separate line

Ex: 103 → 1
0
3

- 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

Case Study: Vertical Numbers (cont.)

- The writeVertical algorithm:

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

- 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

Code for Vertical Number

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

void writeVertical(int n);

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

    cout << "writeVertical(12):" << endl;
    writeVertical(12);

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

     return 0;
}
```

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

Sample Dialogue

```
writeVertical(3):
3
writeVertical(12):
1
2
writeVertical(123):
1
2
3
```


Tracing a Recursive Call

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

Calls writeVertical(12)

resume

Output 3

Function call ends

Tracing writeVertical(12)

- writeVertical(12)

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

Calls writeVertical(1)

resume

Output 2

Function call ends

Tracing writeVertical(1)

- writeVertical(1)

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

Simplest case is now true

Output 1

Function call ends

A Closer Look at Recursion

- writeVertical 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 – a smaller version
  - Resuming the stopped execution
- Eventually one of the recursive calls must not depend on another recursive call
  - 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 often run out of resources and the program will terminate abnormally
- Ex: Function writeVertical, without the base case

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

will eventually call newWriteVertical(0), which will call newWriteVertical(0), which will call newWriteVertical(0), which will call newWriteVertical(0), which will call newWriteVertical(0), which will call newWriteVertical(0), which will call newWriteVertical(0), …

Stacks for Recursion

堆疊

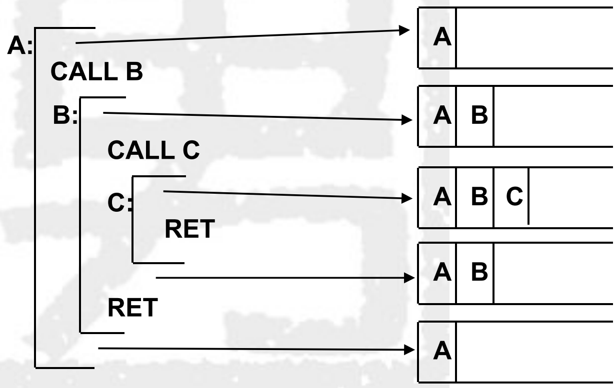
- Computers use a structure called a ***stack*** to keep track of recursion
  - A stack is a last-in/first-out (**LIFO**) memory structure
  - The last item placed is the first that can be removed
- Stack memory structure analogous to a stack of paper
  - To place information on the stack, write it on a piece of paper and place it on top of the stack
  - To ***add*** 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

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 **new arguments** are plugged into parameters
    - Execution of the recursive call begins with **new parameters**

Stacks and Ending Recursive Calls

- When a recursive function call is able to complete its computation with no recursive calls (base case)
  - Start to "return" back to its predecessor
  - Computer retrieves the "top sheet of paper" from the stack
  - Resumes computation based on the saved information on the sheet and discard that paper
  - When that computation ends, the next sheet of paper on the stack is retrieved (return again)
  - The process continues until no sheets remain in the stack



Stack Overflow

- The computer does not use paper
  - Portions of memory are used
  - The contents of these portions of memory is called an *activation frame*
- 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
  - The result is a **stack overflow**
  - A stack overflow causes abnormal termination of the program

一般 loop

Recursion v.s. Iteration

- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls
- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can have infinite loops
  - Make sure the termination condition eventually occurs
- Balance
  - Choice between performance (iteration) and good software engineering (recursion)
  - Avoid using recursion in performance situations
    - Recursive calls take time and consume additional memory

Code Comparison (writeVertical)

Recursion:

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

Iterative:

```
void writeVertical(int n)
{
   int tensInN = 1;
   int leftEndPiece = n;
   while (leftEndPiece > 9)
   {
      leftEndPiece = leftEndPiece/10;
      tensInN = tensInN*10;
   }
   //tensInN is a power of ten that has the same
   //number of digits as n. For example, if n is 2345,
   //thentensInN is 1000.

   for (int powerOf10 = tensInN;
         powerOf10 > 0; powerOf10 = powerOf10/10)
   {
      cout << (n/powerOf10) << endl;
      n = n%powerOf10;
   }
}
```

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
  - The computation result is based on the returned value from the 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)

即 base case

Another Example: Power (X^y)

- $X^n = X * \underline{X * X * \ldots * X}$
  $= X * X^{(n-1)}$
- $X^0 = 1 \rightarrow$ base case
- This function can be solved iteratively or recursively

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
```

```
// Iterative version
int power(int x, int y)
{
    int product = 1;
    for (int i=1; i<=y; i++)
        product = x * product;

    return product;
}
```

```
// Resursive version
int power(int x, int y)
{
    if (y < 1)     // base case
        return 1;
    else           // recursive step
        return  x * power(x, y-1);
}
```

X * X(n-1)

Tracing power(2,1)

- int power(2, 1)

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

Call to power(2,0)

resume

answer = 1

return 1*2 = 2

Function Ends

Tracing power(2,0)

- int power(2, 0)

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

Function call ends

1 is returned

Overview

Thinking Recursively

- When designing a recursive function, you do not need to trace out the entire sequence of calls
- Ex: If the function returns a value
  - Check that there is no infinite recursion: Eventually a stopping case is reached
  - Check the returned value is correct at each stopping case
  - Check the recursive equation. If all recursive calls return the correct value, then the entire case performs correctly

Reviewing the power Function

- There is no infinite recursion ⇒有 終止條件
  - Notice that the second argument is decreased at each call
  - Eventually, the second argument must reach 0 (stop case)
- Each stopping case returns the correct value
  - power(x, 0) should return x_0 = 1 which it does
- All recursive calls return the correct value so the final value returned is correct
  - If n > 1, recursion is used. So power(x,n-1) must return x_{n-1} so power(x, n) can return x_{n-1} * n = x_n which it does

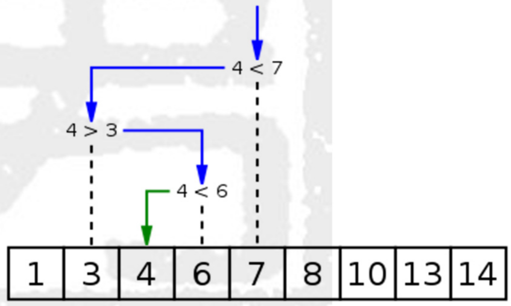
```
int power(int x, int n)
{
   …
   if (n > 0)
      return ( power(x, n-1) * x);
   else
      return (1);
}
```


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 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
  - Because the array is sorted, we know a[0] <= a[1] <= a[2] <= … <= a[final_index]
  - If the item is in the list, return where it is in the list
- Instead of searching the array sequentially, we divide the array into "larger" portion and "smaller" portion
  - Extremely fast by skipping the search for unmatched portion
    → skip about half of elements
  - Improve searching efficiency significantly for large array
    → from O(N) to O(logN)



Binary Search Algorithm Design

- Look at the item in the middle !! (middle == key)
  - 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 (middle > key → smaller portion)
  - If it is less than the number we are looking for, look in the second half of the list (middle < key → larger portion)
- Searching each of the shorter lists is a smaller version of the task we are working on

  → A recursive approach is natural
  - Need additional parameters to specify the subrange to search instead of searching the whole array
  - Add parameters *first* and *last* to indicate the first and last indices of the subrange

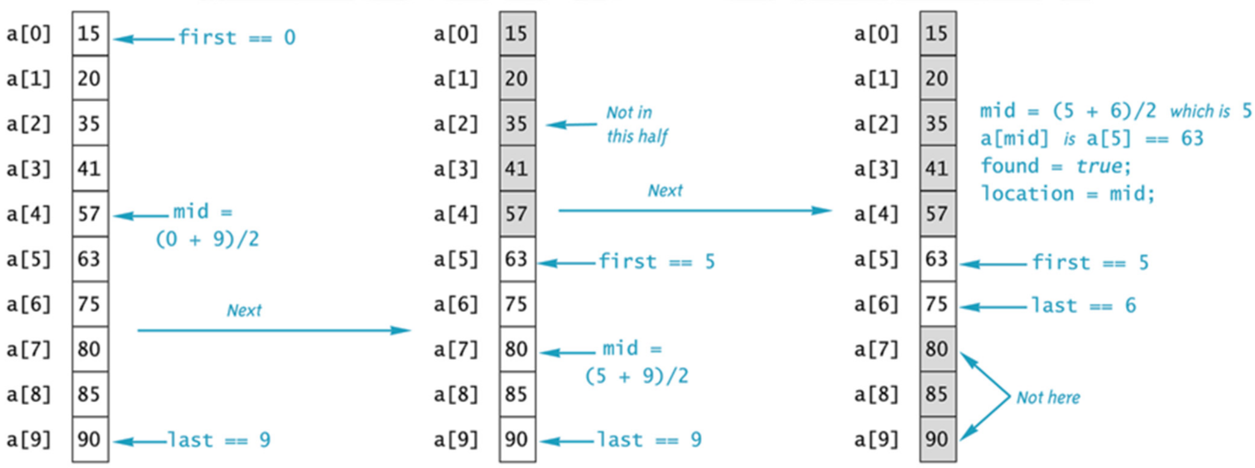
Binary Search Algorithm Design (cont.)

- Be sure to consider the stopping case:

```
if (first > last)   // violate the order, so it's stopping case
    found = false;
else
{
    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]
}
```


Execution of the Function search

- Given key = 63



Example 14-7

Code for Recursive Binary Search

```
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);
        }
    }
}
```


Code for Iterative Binary Search

- Binary search can be implemented with iteration, too
  - May run faster on some systems without extra overhead

```
void search(const int a[], int lowEnd, int highEnd,
                    int key, bool& found, int& location)
{
    int first = lowEnd;
    int last = highEnd;
    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;
        }
    }
}
```