CIS 22A – Lecture 17

Manish Goel

Pass-by-Reference

- Allows a function to work with the original argument from the function call, not a copy of the argument
- Allows the function to modify values stored in the calling environment
- Provides a way for the function to 'return' more than one value
- A reference variable is an alias for another variable
- Defined with an ampersand (`&`)

```
void getDimensions(int&, int&);
```

- Space between type and `&` is unimportant
- Must use `&` in both prototype and header
- Changes to a reference variable are made to the variable it refers to
- Use reference variables to implement passing parameters *by reference*

Overloading Functions

- Overloaded functions have the same name but different parameter lists

- Can be used to create functions that perform the same task but take different parameter types or different number of parameters

- Compiler will determine which version of function to call by argument and parameter lists

Using these overloaded functions,

```
void getDimensions(int);             // 1
void getDimensions(int, int);        // 2
void getDimensions(int, double);     // 3
void getDimensions(double, double);// 4
```

the compiler will use them as follows:

```
int length, width;
double base, height;
getDimensions(length);             // 1
getDimensions(length, width);      // 2
getDimensions(length, height);     // 3
getDimensions(height, base);       // 4
```

Arrays as Function Arguments

- To pass an array to a function, just use the array name:

```
showScores(tests);
```

- Array names in functions are like reference variables – changes made to array in a function are reflected in actual array in calling function

- To define a function that takes an array parameter, use empty `[]` for array argument:

```
void showScores(int []);
                // function prototype
void showScores(int tests[])
                // function header
```

- When passing an array to a function, it is common to pass array size so that function knows how many elements to process:

```
showScores(tests, ARRAY_SIZE);
```

- Array size must also be reflected in prototype, header:

```
void showScores(int [], int);
                // function prototype
void showScores(int tests[], int size)
                // function header
```

Passing File Stream Objects to Functions

- Always pass file stream objects by reference so both calling program and function act on same file
- Otherwise, file accessing code will need to be repeated in the function and it will be hard to maintain file operations in sync

Program 12-5

```
// This program demonstrates how file stream objects may
// be passed by reference to functions.
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

// Function prototypes
bool openFileIn(fstream &, string);
void showContents(fstream &);

int main()
{
   fstream dataFile;

   if (openFileIn(dataFile, "demofile.txt"))
   {
      cout << "File opened successfully.\n";
      cout << "Now reading data from the file.\n\n";
      showContents(dataFile);
      dataFile.close();
      cout << "\nDone.\n";
   }
   else
       cout << "File open error!" << endl;

   return 0;
}

//*********************************************************
// Definition of function openFileIn. Accepts a reference   *
// to an fstream object as an argument. The file is opened  *
// for input. The function returns true upon success, false *
// upon failure.                                            *
//*********************************************************

bool openFileIn(fstream &file, string name)
{
   file.open(name.c_str(), ios::in);
   if (file.fail())
      return false;
   else
      return true;
}

//*********************************************************
// Definition of function showContents. Accepts an fstream  *
// reference as its argument. Uses a loop to read each name *
// from the file and displays it on the screen.             *
//*********************************************************
```

Recursive Functions

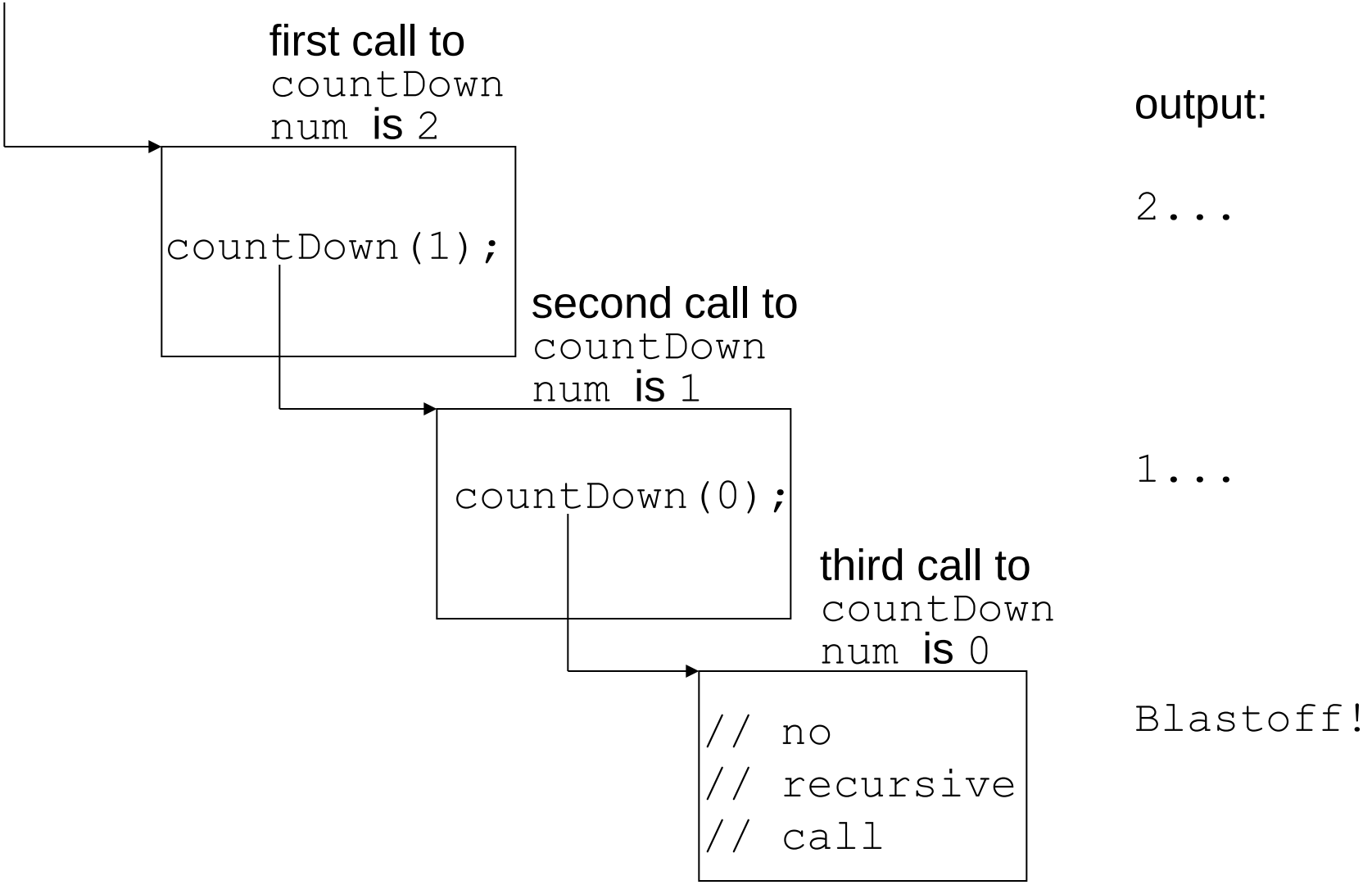
- A recursive function contains a call to itself:

```
void countDown(int num)
{
   if (num == 0)
      cout << "Blastoff!";
   else
   {
      cout << num << "...\n";
      countDown(num-1); // recursive
   }                    // call
}
```

If a program contains a line like `countDown(2);`

1. `countDown(2)` generates the output `2...`, then it calls `countDown(1)`
2. `countDown(1)` generates the output `1...`, then it calls `countDown(0)`
3. `countDown(0)` generates the output `Blastoff!`, then returns to `countDown(1)`
4. `countDown(1)` returns to `countDown(2)`
5. `countDown(2)` returns to the calling function

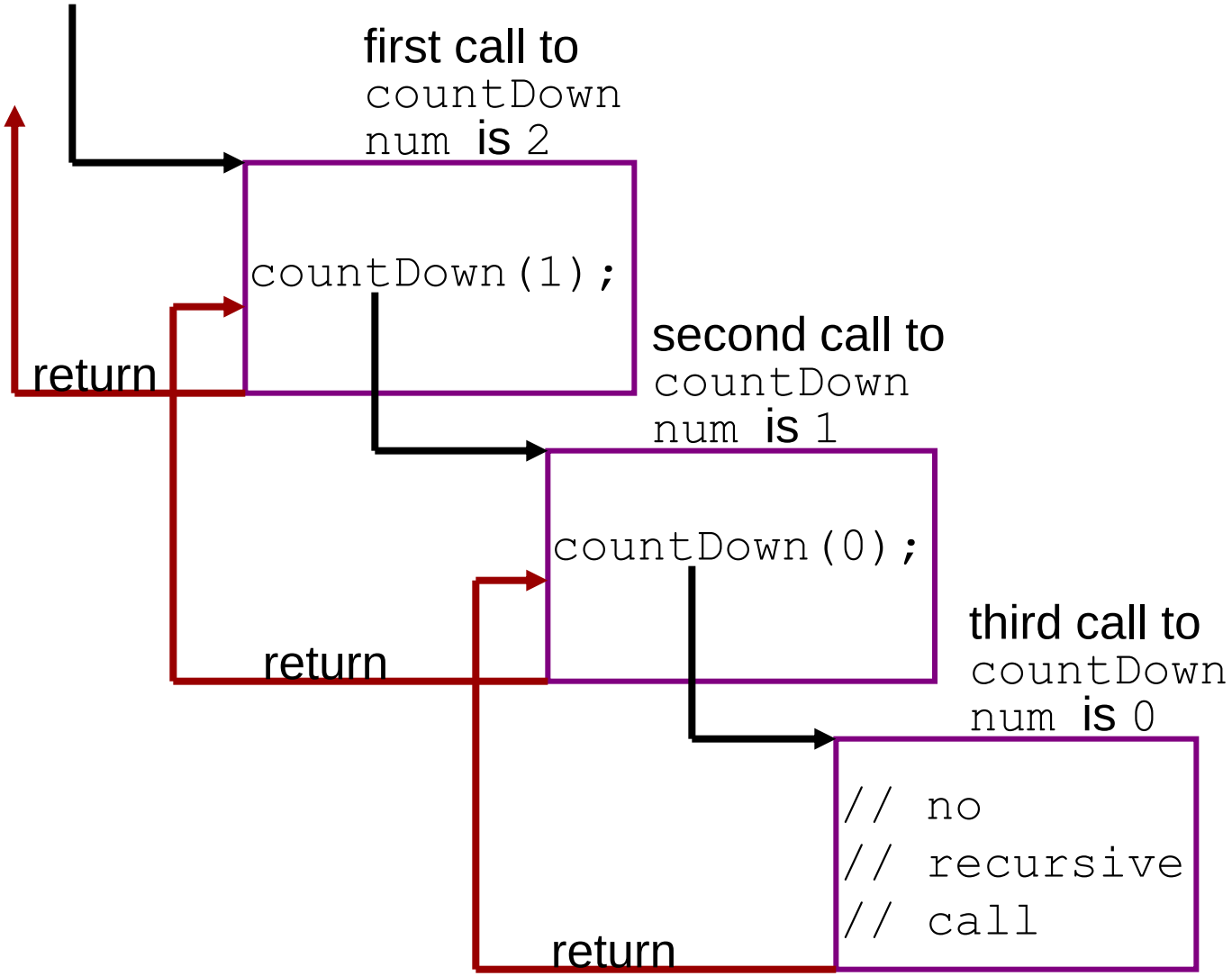
What Happens When Called?



Recursive Functions - Usage

- Recursive functions are used to reduce a complex problem to a simpler-to-solve problems.
- The simplest-to-solve problem is known as the base case
- Recursive calls stop when the base case is reached
- A recursive function must always include a test to determine if another recursive call should be made, or if the recursion should stop with this call
- Each time a recursive function is called, a new copy of the function runs, with new instances of parameters and local variables created
- As each copy finishes executing, it returns to the copy of the function that called it
- When the initial copy finishes executing, it returns to the part of the program that made the initial call to the function

What Happens When Called?



output:

```
2...

1...

Blastoff!
```

The Recursive Factorial Function

- The factorial function:
  `n! = n*(n-1)*(n-2)*...*3*2*1` if `n > 0`
  `n! = 1` if `n = 0`

- Can compute factorial of `n` if the factorial of `(n-1)` is known:
  `n! = n * (n-1)!`

- `n = 0` is the base case

```
int factorial (int num)
{
  if (num > 0)
     return num * factorial(num - 1);
 else
     return 1;
}
```

Program 19-3

```
1  // This program demonstrates a recursive function to
2  // calculate the factorial of a number.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototype
7  int factorial(int);
8
9  int main()
10 {
11    int number;
12
13    // Get a number from the user.
14    cout << "Enter an integer value and I will display\n";
15    cout << "its factorial: ";
16    cin >> number;
17
18    // Display the factorial of the number.
19    cout << "The factorial of " << number << " is ";
20    cout << factorial(number) << endl;
21    return 0;
22 }
23
24 //*************************************************************
25 // Definition of factorial. A recursive function to calculate *
26 // the factorial of the parameter n.                          *
27 //*************************************************************
28
29 int factorial(int n)
30 {
31    if (n == 0)
32       return 1;                    // Base case
33    else
34       return n * factorial(n - 1);  // Recursive case
35 }
```

Program Output with Example Input Shown in Bold

```
Enter an integer value and I will display
its factorial: 4 [Enter]
The factorial of 4 is 24
```

Introduction to the STL `vector`

- A data type defined in the Standard Template Library
- Can hold values of any type:

```
vector<int> scores;
```

- Automatically adds space as more is needed – no need to determine size at definition
- Can use `[]` to access elementsYou must `#include<vector>`

Declare an `int` vector: `vector<int> scores;`
Vector with initial size 30: `vector<int> scores(30);`
Vector all elements init to 0: `vector<int> scores(30, 0);`
Vector initialized to size and contents of another vector:

```
vector<int> finals(scores);
```

Vector Functions

- Use `push_back` member function to add element to a full array or to an array that had no defined size:

```
scores.push_back(75);
```

- Use `size` member function to determine size of a vector:

```
howbig = scores.size();
```

- Use `pop_back` member function to remove last element from vector:

```
scores.pop_back();
```

- To remove all contents of vector, use `clear` member function:

```
scores.clear();
```

- To determine if vector is empty, use `empty` member function:

```
while (!scores.empty()) ...
```

Other Useful Member Functions

Member Function	Description	Example
`at(elt)`	Returns the value of the element at position `elt` in the vector	`cout << vec1.at(i);`
`capacity()`	Returns the maximum number of elements a vector can store without allocating more memory	`maxelts = vec1.capacity();`
`reverse()`	Reverse the order of the elements in a vector	`vec1.reverse();`
`resize (elts,val)`	Add elements to a vector, optionally initializes them	`vec1.resize(5,0);`
`swap(vec2)`	Exchange the contents of two vectors	`vec1.swap(vec2);`