

# Computer Science II — Lecture 2

## Strings, Vectors and Recursion

### 1 Overview of Lecture 2

The following topics will be covered quickly

- strings
- vectors as “smart arrays”
- Basic recursion

Mostly, these are assumed to be review. We will return to each of them during the course of the semester.

### 2 C++ Strings

#### 2.1 Example 1

Here is example output of a program that reads in a string and outputs a framed greeting.

```
Please enter your first name: Chuck
```

```
*****  
*                               *  
* Hello, Chuck! *  
*                               *  
*****
```

The program is on the next page.

```

-----
// ask for a person's name, and generate a framed greeting
#include <iostream>
#include <string>

int main()
{
    std::cout << "Please enter your first name: ";
    std::string name;
    std::cin >> name;

    // build the message that we intend to write
    const std::string greeting = "Hello, " + name + "!";

    // build the second and fourth lines of the output
    const std::string spaces( greeting.size(), ' ' );
    const std::string second = "* " + spaces + " *";

    // build the first and fifth lines of the output
    const std::string first(second.size(), '*');

    // write it all
    std::cout << std::endl;
    std::cout << first << std::endl;
    std::cout << second << std::endl;
    std::cout << "* " << greeting << " *" << std::endl;
    std::cout << second << std::endl;
    std::cout << first << std::endl;

    return 0;
}
-----

```

## 2.2 string objects

- A `string` is an object type defined in the standard library to contain a sequence of characters.
- The `string` type, like all types (including `int`, `double`, `char`, `float`), defines an interface, which includes construction (initialization), operations, functions (methods), and even other types(!).
- When an object is created, a special function is run called a “constructor”, whose job it is to initialize the object. The greeting example code exhibits three ways of constructing string objects:
  - By default to create an empty string

- With a specified number of instances of a single char
- From another string
- The notation

```
greeting.size()
```

is a call to a function `size` that is defined as a **member function** of the `string` class. There is an equivalent member function called `length`.

- Input to string objects through streams includes the following steps:
  1. The computer inputs and discards white-space characters, one at a time, until a non-white-space character is found.
  2. A sequence of non-white-space characters is input and stored in the string. This overwrites anything that was already in the string.
  3. Reading stops either at the end of the input or upon reaching the next white-space character (without reading it in).
- The (overloaded) operator `'+'` is defined on strings. It concatenates two strings to create a third string, without changing either of the original two strings.
- The assignment operation `'='` on strings overwrites the current contents of the string.

### 2.3 Exercise

1. Consider the following code fragment:

```
std::string a, b, c;
std::cin >> a >> b >> c;
```

and the input

```
all-cows   eat123
           grass. every good boy
deserves fudge!
```

What will be the values of strings `a`, `b` and `c` at the end of the code fragment?

2. Write a C++ program that reads in two strings, outputs the shorter string on one line of output, and then outputs the two strings concatenated together with a space between them on the second line of output.

### 2.4 C++ vs. Java

- Standard C++ library `std::string` objects behave like a combination of Java `String` and `StringBuffer` objects. If you aren't sure of how a `std::string` member function (or operator) will behave, check its semantics or try it on small examples (or both, which is preferable).

- Java objects must be created using `new`, as in

```
String name = new String("Chris");
```

This is not necessary in C++. The C++ (approximate) equivalent to this example is

```
std::string name("Chris");
```

On the other hand, there is a `new` operator in C++ and its behavior is somewhat similar to the `new` operation in Java. We will study it later in the semester.

## 2.5 More on strings

- Strings behave like arrays when using the subscript operator `[]`.
  - This gives access to the individual characters in the string.
  - Subscript 0 corresponds to the first character.
  - For example, given

```
std::string a = "Susan";
```

Then `a[0] == 'S'` and `a[1] == 'u'` and `a[4] == 'n'`.

- Strings define a special type, which is the type returned by the string functions `size()` and `length()`:

```
string::size_type
```

- The `::` notation means that `size_type` is defined within the scope of the `string` type.
- `string::size_type` is generally equivalent to `unsigned int`.
- You may have compiler warnings and potential compatibility problems if you compare an `int` variable to `a.size()`.

## 2.6 Example Problem: Writing a name along a diagonal

Here is a more interesting problem:

What is your first name? Peter

```
*****
*      *
* P    *
* e    *
*  t   *
*   e  *
*    r *
*      *
*****
```

## 2.7 Solution Approach #1

- Initial stuff: read the name, and output a blank line, as in an earlier example.
- Let's think about the main output: think of the output region as a grid of rows and columns:
  - How big is this region?
  - What gets output where?
- This leads to an implementation with two nested loops, and conditionals used to guide where characters should be printed.

## 2.8 Solution 1

---

```
// Program: diagonal_name
// Author: Chuck Stewart
// Purpose: A program that outputs an input name along a diagonal.

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

int main()
{
    cout << "What is your first name? ";
    string first;
    cin >> first;

    const string star_line( first.size()+4, '*' );
    const string blanks( first.size()+2, ' ' );
    const string empty_line = '*' + blanks + '*';

    cout << '\n' << star_line << '\n' << empty_line << '\n';

    // Output the interior of the framed greeting, one line at a time.
    for ( string::size_type i = 0; i<first.size(); ++ i )
    {
        // Job of loop body is to write a row of output containing *'s
        // in the first (0-th) and last columns, the i-th letter in
        // column i+2, and a blank everywhere else.

        for ( string::size_type j = 0; j < first.size()+4; ++ j )
        {
            if ( j == 0 || j == first.size()+3 )
                cout << '*';
            else if ( j == i+2 )
                cout << first[i];
            else
                cout << ' ';
        }
        cout << '\n';
    }

    cout << empty_line << '\n' << star_line << '\n';
    return 0;
}
```

---

## 2.9 Aside: Ending a Line of Output

- There are two common ways to end a line of output in a C++ program:

```
std::cout << '\n';
```

and

```
std::cout << std::endl;
```

What is the difference?

- C++ streams store their output in an *output buffer*. This buffer is not immediately written to a file or displayed on your screen. The reason is that the writing process is much slower than the other computations. It is much faster overall when output is buffered and then done “all at once” — in large chunks — when the buffer is full.
- Thus, just outputting the `'\n'` — the end-of-line character — just adds one more character to the buffer.
- Outputting `std::endl` has two effects:
  - Outputting the `'\n'` character, **and**
  - Causing the buffer to be “flushed” — actually output to the file or screen.
- Why should you care?
  - When your program crashes, the contents of the output buffer are lost and not actually output. As a result, when looking at your output it often appears that your program crashed much earlier than it actually did. Therefore, using `std::endl` helps greatly with debugging.
  - Using `std::endl` can substantially slow down a program. Therefore, when a program is fully debugged (and needs to run at a reasonable speed), `std::endl` should be replaced by `'\n'`.

## 2.10 Loop Invariants

- Definition: a *loop invariant* is a logical assertion that is true at the start of each iteration of a loop.
  - In `for` loops, the “start” is defined as coming after the initialization/increment and termination test, but before execution of the next loop iteration.
- An invariant is stated in a comment; it is not part of the actual code.
- It helps determine:
  - The conditions that may be assumed to be true at the start of each iteration.
  - What must be done in each iteration.
  - What must be done at the end of each iteration to restore the invariant.
- Analyzing the code relative to the stated invariant also helps explain the code and think about its correctness.

## 2.11 L-Values and R-Values

- Consider the simple code

```
string a = "Kim";  
string b = "Tom";  
a[0] = b[0];
```

String `a` is now "Tim".

- No big deal, right? Wrong!
- Let's look closely at the line:

```
a[0] = b[0];
```

and think about what happens.

- In particular, what is the difference between the use of `a[0]` on the left hand side of the assignment statement and `b[0]` on the right hand side?
- Syntactically, they look the same. But,
  - The expression `b[0]` gets the char value, 'T', from string location 0 in `b`. This is an *r-value*
  - The expression `a[0]` gets a reference to the memory location associated with string location 0 in `a`. This is an *l-value*.
  - The assignment operator stores the value in the referenced memory location.

The difference between an *r-value* and an *l-value* will be especially significant when we get to writing our own operators.

## 2.12 A Second Solution To Our Diagonal Name Problem

Here are ideas for a second solution:

- Think about what changes from one line to the next.
- Suppose we had a "blank line" string, containing only the beginning and ending asterisks and the intervening blanks.
- We could overwrite the appropriate blank character, output the string, and then restore the blank character.

The second solution will be posted on line.

### 2.13 Thinking about problem solving

- Often it is easier to start by working on simplified versions of the problem to get a “feel” for the core issues. In the example we considered, a good start would be to think about writing the name along the diagonal without any framing.
- We considered two solution approaches:
  - Thinking of the output as a two-dimensional grid and using logical operations to figure out what to output at each location.
  - Thinking of the output as a series of strings, one string per line, and then thinking about the differences between lines.
- There are often many ways to solve a programming problem. Sometimes you can think of several, while sometimes you struggle to come up with one.
- When you have finished a problem or when you are thinking about someone else’s solution, it is useful to think about the core ideas used. If you can abstract and understand these ideas, you can later apply them to other problems.

## 3 Standard Library (STL) Vectors

### 3.1 Problem: Grade Statistics

- Read an unknown number of grades.
- Compute:
  - Mean (average)
  - Standard deviation
  - Median (middle value)
- Accomplishing this requires the use of vectors.

### 3.2 Standard Deviation

- Definition: if  $a_0, a_1, a_2, \dots, a_{n-1}$  is a sequence of  $n$  values, and  $\mu$  is the average of these values, then the standard deviation is

$$\left[ \frac{\sum_{i=0}^{n-1} (a_i - \mu)^2}{n - 1} \right]^{\frac{1}{2}}$$

- Computing this equation requires two passes through the values:
  - Once to compute the average
  - A second time to compute the standard deviation
- Thus, we need a way to store the values.
- The only tool we have so far is arrays. But arrays are fixed in size and we don't know in advance how many values there will be.
- This illustrates one reason why we generally will use *standard library vectors* instead of arrays.

### 3.3 Vectors

- Standard library “container class” to hold sequences.
- A vector acts like a dynamically-sized, one-dimensional array.
- Capabilities:
  - Holds objects of any type
  - Starts empty unless otherwise specified
  - Any number of objects may be added to the end — there is no limit on size.
  - It can be treated like an ordinary array using the subscripting operator.
  - There is NO automatic checking of subscript bounds.

### 3.4 Computing the Standard Deviation

A program to compute the standard deviation is provided with these notes (`average_and_deviation.cpp`). We will use this to discuss the use of vectors.

- Header files:

```
#include <vector>
```

defines the vector class

```
#include <cmath>
```

includes the C standard math library prototypes and declarations so that we can use the `sqrt` function.

- The following creates an empty vector of integers:

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

Vectors are an example of a *templated container class*.

- The angle brackets `< >` are used to specify the type of object (the “template type”) that will be stored in the vector.
- `push_back` is a vector function to append a value to the end of the vector, increasing its size by one. This is an  $O(1)$  operation (on average).
  - There is NO corresponding `push_front` operation for vectors.
- `size` is a function defined by the vector type (the vector class) that returns the number of items stored in the vector.
- After vectors are initialized and filled in, they may be treated *just like arrays*.
  - In the line

```
sum += scores[i];
```

`scores[i]` is an “r-value”, accessing the value stored at location `i` of the vector.
  - We could also write statements like

```
scores[4] = 100;
```

to change a score. Here `scores[4]` is an “l-value”, providing the means of storing 100 at location 4 of the vector.
  - It is the job of the programmer to ensure that any subscript value `i` that is used is legal — at least 0 and strictly less than `scores.size()`.

### 3.5 Median

- Intuitively, a median value of a sequence is a value that is less than half of the values in the sequence, and greater than half of the values in the sequence.
- More technically, if  $a_0, a_1, a_2, \dots, a_{n-1}$  is a sequence of  $n$  values AND if the sequence is sorted such that  $a_0 \leq a_1 \leq a_2 \leq \dots \leq a_{n-1}$  then the median is

$$\begin{cases} a_{(n-1)/2} & \text{if } n \text{ is odd} \\ \frac{a_{n/2-1} + a_{n/2}}{2} & \text{if } n \text{ is even} \end{cases}$$

- Sorting is therefore the key to finding the median.

### 3.6 Standard Library Sort Function

- The standard library has a series of algorithms built to apply to container classes.
- The prototypes for these algorithms (actually the functions implementing these algorithms) are in header file `algorithm`.
- One of the most important of the algorithms is `sort`.
- It is accessed by providing the beginning and end of the container's interval to sort.
- As an example, the following code reads, sorts and outputs a vector of doubles

```
double x;
std::vector<double> a;
while ( std::cin >> x ) a.push_back(x);
std::sort( a.begin(), a.end() );
for ( unsigned int i=0; i<a.size(); ++i )
    std::cout << a[i] << '\n';
```

- `a.begin()` is an *iterator* referencing the first location in the vector, while `a.end()` is an *iterator* referencing one past the last location in the vector.
  - We will learn much more about iterators in the next few weeks.
  - Every container has iterators: strings have `begin()` and `end()` iterators defined on them.
- The ordering of values by `std::sort` is least to greatest (technically, non-decreasing). We will see ways to change this.

### 3.7 Computing the Median

A program, `median_grade.cpp`, that uses a vector to compute the median of a set of grades is provided with these notes and will be posted on-line.

### 3.8 Passing Vectors (and Strings) As Parameters

The following outlines rules for passing vectors as parameters. The same rules apply to passing strings.

- If you are passing a vector as a parameter to a function and you want to make a (permanent) change to the vector, then you should pass it **by reference**.
  - This is illustrated by the function `read_scores` in the program `median_grade`.
  - This is very different from the behavior of arrays as parameters.
- What if you don't want to make changes to the vector or don't want these changes to be permanent?
  - The answer we've learned so far is to pass by value.
  - The problem is that the entire vector is copied when this happens!
- The solution is to pass by **constant reference**: pass it by reference, but make it a constant so that it can not be changed.
  - This is illustrated by the functions `compute_avg_and_std_dev` and `compute_median` in the program `median_grade`.
- As a general rule, you should not pass a container object such as a vector or a string, by value because of the cost of copying. There are rare circumstances in which this rule may be violated, but not in CS II.

### 3.9 Initializing a Vector — The Use of Constructors

- Here is an example of several different ways to initialize a vector:

```
vector<int> a;                // #1
int n = 100;
vector<double> b( 100, 3.14 ); // #2
vector<int> c( n*n );        // #3
vector<double> d( b );       // #4
vector<int> e( b );          // #5
```

- Version #1 “constructs” an empty vector of integers. Values must be placed in the vector using `push_back`.
- Version #2 constructs a vector of 100 doubles, each entry storing the value 3.14. New entries can be created using `push_back`, but these will create entries 100, 101, 102, etc.
- Version #3 constructs a vector of 10,000 ints, but provides no initial values for these integers. Again, new entries can be created for the vector using `push_back`. These will create entries 10000, 10001, etc.
- Version #4 constructs a vector that is an exact copy of vector `b`.
- Version #5 is a compiler error because no constructor exists to create an int vector from a double vector. These are different types.

### 3.10 Exercises

1. After the above code constructing the three vectors, what will be output by the following statement?

```
cout << a.size() << endl
     << b.size() << endl
     << c.size() << endl;
```

2. Write code to construct a vector containing 100 doubles, each having the value 55.5.
3. Write code to construct a vector containing 1000 doubles, containing the values 0, 1,  $\sqrt{2}$ ,  $\sqrt{3}$ ,  $\sqrt{4}$ ,  $\sqrt{5}$ , etc. Write it two ways, one that uses `push_back` and one that does not use `push_back`.

### 3.11 Additional Example — Alphabetize Strings

Attached to the end of the notes is an additional example of using vectors, strings and the `sort` function to alphabetize last names. The source code will also be posted on the web. This is an important example for future homeworks.

## 4 Recursion: The Basics

### 4.1 Recursive Definitions of Factorials and Integer Exponentiation

- The factorial is defined for non-negative integers as

$$n! = \begin{cases} n \cdot (n-1)! & n > 0 \\ 1. & n == 0 \end{cases}$$

- Computing integer powers is defined as:

$$n^p = \begin{cases} n \cdot n^{p-1} & p > 0 \\ 1. & p == 0 \end{cases}$$

- These are both examples of *recursive definitions*.

### 4.2 Recursive C++ Functions

C++, like other modern programming languages, allows functions to call themselves. This gives a direct method of implementing recursive functions.

- Here's the implementation of factorial:

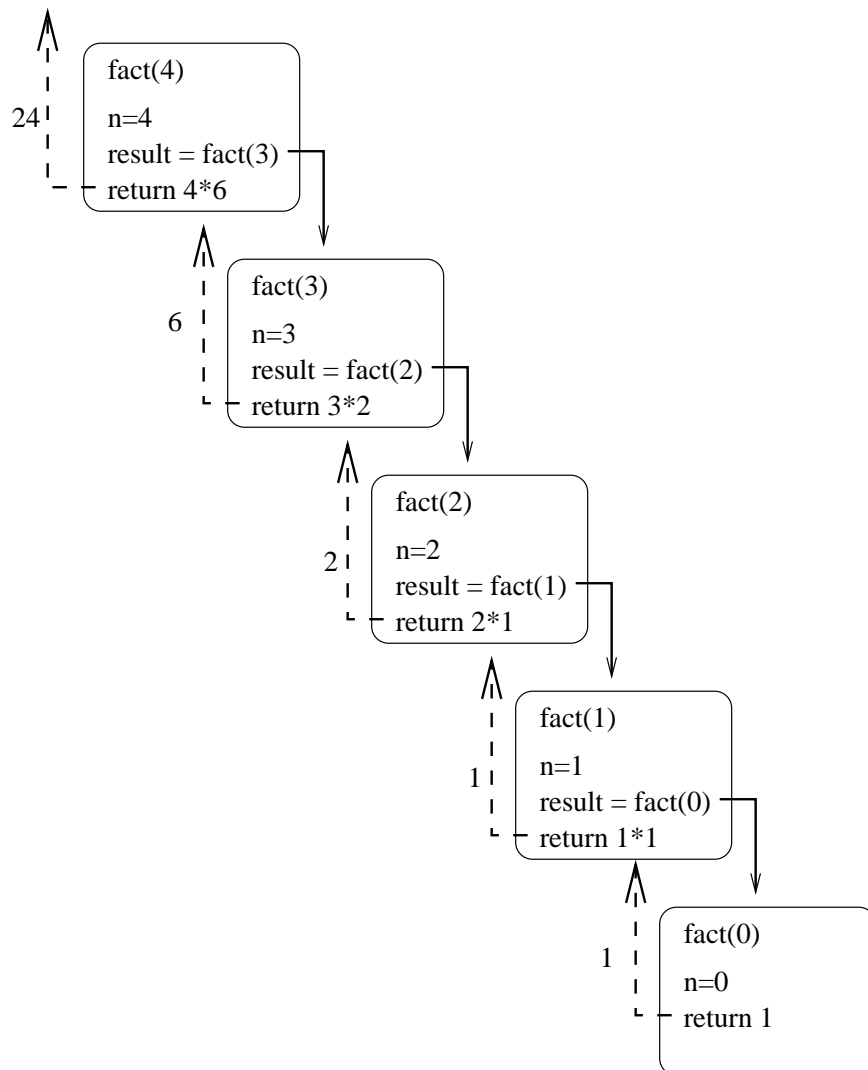
```
int fact( int n )
{
    if ( n == 0 )
        return 1;
    else
    {
        int result = fact( n-1 );
        return n * result;
    }
}
```

- Here's the implementation of exponentiation:

```
int intpow( int n, int p )
{
    if ( p == 0 )
        return 1;
    else
    {
        return n * intpow( n, p-1 );
    }
}
```

### 4.3 The Mechanism of Recursive Function Calls

- When it makes a recursive call (or any function call), a program creates an *activation record* to keep track of
  - Each newly-called function’s own **completely separate instances** of parameters and local variables.
  - The location in the calling function code to return to when the newly-called function is complete.
  - Which activation record to return to when the function is done.
- This is illustrated in the following diagram of the call `fact(4)`. Each box is an activation record, the solid lines indicate the function calls, and the dashed lines indicate the returns.



### 4.4 Iteration vs. Recursion

- Each of the above functions could also have been written using a for loop, i.e. *iteratively*.

- For example, here is an iterative version of factorial:

```
int ifact( int n )
{
    int result = 1;
    for ( int i=1; i<=n; ++i )
        result = result * i;
    return result;
}
```

- Iterative functions are generally faster than their corresponding recursive functions. Compiler optimizations sometimes (but not always!) can take care of this by automatically eliminating the recursion.
- Sometimes writing recursive functions is more natural than writing iterative functions, however. Most of our examples will be of this sort.

#### 4.5 Exercise

Write an iterative version of `intpow`.

#### 4.6 Rules for Writing Recursive Functions

Here is an outline of five steps I find useful in writing and debugging recursive functions:

1. Handle the base case(s) first, at the start of the function.
2. Define the problem solution in terms of smaller instances of the problem. This defines the necessary recursive calls. It is also the hardest part!
3. Figure out what work needs to be done before making the recursive call(s).
4. Figure out what work needs to be done after the recursive call(s) complete(s) to finish the computation.
5. Assume the recursive calls work correctly, but make sure they are progressing toward the base case(s)!

#### 4.7 Example: Printing the Contents of a Vector

The following example is important for thinking about the mechanisms of recursion.

- Here is a function for printing the contents of a vector. Actually, it is two functions: driver function, and a true recursive function.

```
void print_vec( vector<int>& v )
{
    print_vec( v, 0 );
}

void print_vec( vector<int>& v, unsigned int i )
```

```

{
    if ( i < v.size() )
    {
        cout << i << ": " << v[i] << endl;
        print_vec( v, i+1 );
    }
}

```

- **Exercise:** What will this print when called in the following code?

```

int main()
{
    vector<int> a;
    a.push_back( 3 ); a.push_back( 5 ); a.push_back( 11 );
    a.push_back( 17 );
    print_vec( a );
}

```

- Note: the idea of a “driver function” that just initializes a recursive function call is quite common.
- **Exercise:** How can you change the second `print_vec` function as little as possible to write a recursive function to print the contents of the vector in reverse order?

## 4.8 Binary Search

- Suppose you have a `vector<T> v`, sorted so that

$$v[0] \leq v[1] \leq v[2] \leq \dots$$

- Now suppose that you want to find if a particular value  $x$  is in the vector somewhere.
- How can you do this without looking at every value in the vector?
- The solution is a recursive algorithm called *binary search*, based on the idea of checking the middle item of the search interval within the vector and then looking either in the lower half or the upper half of the vector, depending on the result of the comparison:

```

-----

// Here is the recursive function. The "invariant" is that if x is
// in the vector then it must be located within the subscript range
// low to high. Therefore, when low and high are equal, their common
// value is the only possible place for x. Otherwise, the middle
// value is checked, and the search continues recursively in either
// the lower or upper half of the vector.

bool binsearch( const vector<double>& v, int low, int high, double x )
{
    if ( high == low ) return x == v[low];

    int mid = (low+high) / 2;
    if ( x <= v[mid] )
        return binsearch( v, low, mid, x );
    else
        return binsearch( v, mid+1, high, x );
}

// The driver function. It establishes the search range for the
// value of x based on the minimum and maximum subscripts in the
// vector.

bool binsearch( const vector<double>& v, double x )
{
    return binsearch( v, 0, v.size()-1, x );
}

-----

```

## 4.9 Exercises

1. Write a non-recursive version of binary search.
2. If we replaced the if-else structure inside the recursive binsearch function (above) with

```

    if ( x < v[mid] )
        return binsearch( v, low, mid-1, x );
    else
        return binsearch( v, mid, high, x );

```

would the function still work correctly?

## 5 Summary

- C++ strings from the standard library hold sequences of characters and have a sophisticated set of operations defined on them.
- Vectors, also from the standard library, can be thought of as smart, dynamically-sized arrays. Vectors should almost always be used instead of arrays, but as we will see vectors are defined internally in terms of arrays.
- Recursion is a way of defining a function and or a structure in terms of simpler instances of itself. While we have seen simple examples of recursion here, ones that are easily replaced by iterative, non-recursive functions, later in the semester when we return to recursion we will see much more sophisticated examples where recursion is not easily removed.