

# Unit 4. Advanced Topics

4.0 Multi-Dimensional Arrays .....	309
4.1 Allocating Memory .....	324
4.2 String Class.....	337
4.3 Command Line.....	346
4.4 Instrumentation .....	356
Unit 4 Practice Test .....	361
Unit 4 Project : Sudoku .....	363

## 4.0 Multi-Dimensional Arrays

Sam had so much fun dabbling with ASCII-art that he thought he would try his hand at computer graphics. The easiest way to get started is to load an image from memory and display it on the screen. This seems challenging, however; memory (including the type of data stored in an array) is one-dimensional but images are two-dimensional. How can he store two-dimensional data in an array? How can he convert the one-dimensional data in a file into this array? While trying to figure this out, Sue introduces him to multi-dimensional arrays.

### Objectives

By the end of this class, you will be able to:

- Declare a multi-dimensional array.
- Pass a multi-dimensional array to a function as a parameter.
- Read multi-dimensional data from a file and put it in an array.

### Prerequisites

Before reading this section, please make sure you are able to:

- Declare an array to solve a problem (Chapter 3.0).
- Write a loop to traverse an array (Chapter 3.0).
- Pass an array to a function (Chapter 3.0)
- Write the code to read data from a file (Chapter 2.6).
- Write the code to write data to a file (Chapter 2.6).

### Overview

Often we work with data that is inherently multi-dimensional. A few common examples include pictures (row and column), coordinates (latitude, longitude, and altitude), and position on a grid (x and y). The challenge arises when we need to store the multi-dimensional data in a memory store that is inherently one dimensional.

Consider the following code to put the numbers 0-15 on the screen:

```
for (int index = 0; index < 16; index++)
    cout << index << '\t';
cout << endl;
```

Observe how the numbers are one-dimensional (just an index). However, we would like to put the numbers in a nice two-dimensional grid that is  $4 \times 4$ . How do we do this? The first step is we need some way to detect when we are on the 4<sup>th</sup> column. When we are on this column, we display a newline character rather than a white space to properly align the columns.

	column 0	column 1	column 2	column 3	
0	1	2	3		row 0
4	5	6	7		row 1
8	9	10	11		row 2
12	13	14	15		row 3

Are there any patterns in the numbers? Can we find any way to derive the row or column based on the index? The first thing to realize is that the column numbers seem to increase by one as the index increases by one. This occurs until we get to the end of the row. When that happens, the column number seems to reset.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
column	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

This pattern should be familiar. As we divide the index (index) by the number of columns (numCol), the remainder appears to be the column (column) value.

```
column = index % numCol;
```

The row value appears to be an entirely different equation. We increment the row value only after we increment four index values:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
row	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3

This pattern is also familiar. We can derive row by performing integer division on index by numCol:

```
row = index / numCol;
```

Based on these observations, we can re-write our loop to display the first 16 whole numbers:

```
for (int index = 0; index < 16; index++)
    cout << index << (index % 4 == 3 ? '\n' : '\t');
```

In other words, when the column value (index % 4) is equal to the fourth column ( == 3) then display a newline character ('\n') rather than the tab character ('\t'). We can re-write this more generally as:

```

/*****
 * DISPLAY NUMBERS
 * Display the first 'number' whole numbers
 * neatly divided into a grid of numCol columns
 *****/
void displayNumbers(int number, int numCol)
{
    for (int index = 0; index < number; index++)
        cout << index << (index % numCol == numCol - 1 ? '\n' : '\t');
}

```

After converting an inherently one-dimensional value (index) into a two-dimensional pair (row & column), how do we convert two-dimensional values back into an index? To accomplish this, we need to recall the things we learned when going the other way:

- An increase in the `index` value yields an increase in the `column` value. To turn this around, we could also say that an increase in the `column` value yields an increase in the `index` value.
- The `row` value changes one fourth ( $1 / \text{numCol}$ ) as often as the `index` value. To turn this around, a change in the `row` value yields a jump in the `index` value by four (`numCol`).

We can combine both these principles in a single equation:

```
index = row * 4 + column;
```

Check this equation for correctness:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
row	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3
column	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

If we add one to the `row`, then the `index` jumps by four. If we add one to the `column`, the `index` jumps by one. Thus we have the ability to convert two-dimensional coordinates (`row` & `column`) into a one dimensional value (`index`). The general form of this equation (worth memorizing) is:

```
index = row * numCol + column;
```

Why would we ever want to do this? Consider the scenario when we want to put the multiplication tables for the values 0 through 3 in an array. This can be accomplished with:

```
{
    int grid[4 * 4];                                // the area of the array is the width
                                                    // times the height.
    for (int row = 0; row < 4; row++)                // rows first, 0...3
        for (int col = 0; col < 4; col++)            // columns next, also 0...3
            grid[row * 4 + col] = row * col;         // convert to row,col to index for the []
                                                    // the right-side is the product
}
```

In memory, the resulting `grid` array appears as the following:

	row 0				row 1				row 2				row 3			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
grid	0	0	0	0	0	1	2	3	0	2	4	6	0	3	6	9

In both of these cases (converting 2-dimensional to 1 and converting 1-dimensional to 2), the same piece of information is needed: the number of columns (`numCol`) in the data. This should make sense. If you have 32 items in a data-set, is the grid  $1 \times 32$  or  $2 \times 16$  or  $4 \times 8$  or  $8 \times 4$  or  $16 \times 2$  or  $32 \times 1$ ? Each of these possibilities is equally likely. One must know either the number of columns or the number of rows to make the conversion.

# Syntax

As you may have noticed, multi-dimensional arrays are quite commonly needed to solve programming problems. Similarly, the conversion from index to coordinates and back is tedious and overly complicated. Fortunately, there is an easier way:

Declaring an array	Referencing an array	Passing as a parameter
<p>Syntax:</p> <pre>&lt;Type&gt; &lt;name&gt;[size][size]</pre> <p>Example:</p> <pre>int data[200][15];</pre> <p>A few details:</p> <ul style="list-style-type: none"><li>• Any data-type can be used.</li><li>• The size must be a natural number {1, 2, etc.} and not a variable.</li></ul>	<p>Syntax:</p> <pre>&lt;name&gt;[index][index]</pre> <p>Example:</p> <pre>cout &lt;&lt; data[i][j];</pre> <p>A few details:</p> <ul style="list-style-type: none"><li>• The index starts with 0 and must be within the valid range.</li></ul>	<p>Syntax:</p> <pre>(&lt;Type&gt; &lt;name&gt;[][size])</pre> <p>Example:</p> <pre>void func(int data[][15])</pre> <p>A few details:</p> <ul style="list-style-type: none"><li>• You must specify the base-type.</li><li>• No size is passed in the square brackets [].</li></ul>

## Declaring an array

Multi-dimensional arrays are declared by specifying the base-type and the size of each dimension. The basic syntax is:

```
<base-type> <variable>[<number of rows>][<number of columns>;
```

A grid of integers that is  $3 \times 4$  can be declared as:

```
int grid[4][3];
```

We can also initialize a multi-dimensional array at declaration time. The best way to think of the initialization syntax is “an array of arrays.” Consider the following example:

```
{  
    int grid[4][3] =  
    {  
        { 8, 12, -5 }, // row 0  
        { 421, 4, 153 }, // row 1  
        { -15, 20, 91 }, // row 2  
        { 4, -15, 182 }, // row 3  
    };  
}
```



### Sue's Tips

Notice how the horizontal dimension comes *second* in multi-dimensional arrays. In Geometry, we learned to specify coordinates as (X, Y) where the horizontal dimension comes first. Multi-dimensional arrays are the opposite! Rather than trying to re-learn (Y, X) (which just doesn't feel right, does it?), it is more convenient to use (Row, Column) as our array dimensions.

Storing a digital image is a slightly more complex example. Each pixel consists of three values (red, green, and blue) with 256 possible values in each (char). The pixels themselves are arrayed in a two-dimensional image ( $4,000 \times 3,000$ ). The resulting declaration is:

```
char image[3][3000][4000];
```

In this example, each element is a char (eight bits in a byte so there are  $2^8$  possible values). The first dimension ([3]) is for the three channels (red, green, and blue). The next is the horizontal size of the image (4,000). The final dimension is the vertical dimension (3,000). The total size of the image is:

```
int size = sizeof(char) * sizeof(3) * sizeof(3000) * sizeof(4000);
```

This is 36,000,000 bytes of data (34.33 megabytes). A twelve mega-pixel image is rather large!

## Referencing an array

When referencing an array, it is important to specify each of the dimensions. Again, we use the vertical dimension first so we use (Row, Column) variables rather than (X, Y). Back to our  $3 \times 4$  grid example:

```
{
    int grid[4][3] =
    {
        { 8, 12, -5 }, // row 0
        { 421, 4, 153 }, // row 1
        { -15, 20, 91 }, // row 2
        { 4, -15, 182 }, // row 3
    };

    int row; // vertical dimension
    int col; // horizontal dimension

    cout << "Specify the coordinates (X, Y) "; // people think in terms of X,Y
    cin >> col >> row;

    assert(row >= 0 && row < 4); // a loop would be a better tool here
    assert(col >= 0 && col < 3); // always check before indexing into
                                // an array

    cout << grid[row][col] << endl;
}
```

Working with more than two-dimensions is the same. Back to our image example consisting of a two-dimensional grid of pixels ( $4,000 \times 3,000$ ) where each pixel has three values. If the user wishes to find the value of the top-left pixel, then the following code would be required:

```
cout << "red:  " << image[0][0][0] << endl
     << "green: " << image[1][0][0] << endl
     << "blue:  " << image[2][0][0] << endl;
```

## Passing as a parameter

Passing arrays as parameters works much the same for multi-dimensional arrays as they do for their single-dimensional brethren. There is one important exception, however. Recall from earlier in the semester (Chapter 3.0) that arrays are just pointers to the first item in the list. Only having this pointer, the callee does not know the length of the buffer. For this reason, it is important to pass the size of the array as a parameter.

There is another important component to understanding multi-dimensional array parameters. Recall that, for a given 32 slots in memory, there may be many possible ways to convert it into a two-dimensional grid {  $(32 \times 1)$ ,  $(16 \times 2)$ ,  $(8 \times 4)$ ,  $(4 \times 8)$ ,  $(2 \times 16)$ , or  $(1 \times 32)$  }. The only way to know which conversion is correct is to know the number of columns (typically called the numCol variable). This information is essential to performing the conversion.

When using the double-square-bracket notation for multi-dimensional arrays (array[3][4] instead of array[3 \* numCol + 4]), the compiler needs to know the numCol value. The same is true when passing multi-dimensional

arrays as parameters. In this case, we specify the size of all the dimensions except the left-most dimension. Back to our  $3 \times 4$  example, a prototype might be:

```
void displayGrid(int array[][3]);    // column size must be present
```

Back to our image example, the following code will fill the image with data from a file.

```

/*****
 * READ IMAGE
 * Read the image data from a file
 *****/
bool readImage(unsigned char image[][3000][4000],    // specify all dimensions but first
               const char fileName[])                // also need the filename as const
{
    // open stream
    ifstream fin(fileName);
    if (fin.fail())                                // never forget error checking
        return false;                             // return and report

    bool success = true;                            // our return value

    // read the grid of data
    for (int row = 0; row < 3000; row++)             // rows are always first
        for (int col = 0; col < 4000; col++)         // then columns
            for (int color = 0; color < 3; color++)   // three color dimensions: r, g, b
            {
                int input;                            // data in the file is a number so
                fin >> input;                          // we read it as an integer
                if ( input < 0 || input >= 256 ||        // before storing it as a
                    fin.fail())                        // char (a small integer). Make
                    success = false;                   // sure it is valid!
                image[color][row][col] = input;
            }

    // paranoia!
    if (fin.fail())                                  // report if anything bad happened
        success = false;

    // make like a tree
    fin.close();                                     // never forget to close the file
    return success;
}

/*****
 * MAIN
 * Simple driver for readImage
 *****/
int main()
{
    unsigned char image[3][3000][4000];             // 12 megapixel image
    if (!readImage(image, "image.bmp"))               // .bmp images are just arrays
        return 1;                                    // of pixels! Note that they
    else                                              // are binary files, not text
        return 0;                                    // files so this will not quite
                                                    // work the way you expect...
}

```

One quick disclaimer about the above example... Images are stored not as text files (which can be opened and read in emacs) but as binary files (such as a.out. Try opening it in emacs to see what I mean). To make this above example work, we will need to create 36 million integers ( $3 \times 3,000 \times 4,000$ ), each of which with a value between 0 and 255. That might take a bit of patience.

## Example 4.0 – Array of Strings

Demo

This example will demonstrate how to create, pass to a function, and manipulate an array of strings. This is a multi-dimensional array of characters.

Solution

Since strings are arrays, to have an array of strings we will need a two dimensional array. Note that the first dimension must be the number of strings and the second the size of each.

```

/*****
 * PROMPT NAMES
 * Prompt the user for his or her name
 *****/
void promptNames(char names[][256])    // the column dimension must be the
{                                     //   buffer size
    // prompt for name (first, middle, last)
    cout << "What is your first name? ";
    cin  >> names[0];                  // passing one instance of the array
    cout << "What is your middle name? "; //   of names to the function CIN
    cin  >> names[1];                  // Note that the data type is
    cout << "What is your last name? ";  //   a pointer to a character,
    cin  >> names[2];                  //   what CIN expects
}

/*****
 * MAIN
 * Just a silly demo program
 *****/
int main()
{
    char names[3][256];                // arrays of strings are multi-
                                     //   dimensional arrays of chars

    // fill the array
    promptNames(names);                // pass the entire array of strings

    // first name:
    cout << names[0] << endl;          // this is an array of characters

    // middle initial
    cout << names[1][0] << endl;       // first letter of second string

    // loop through the names for output
    for (int i = 0; i < 3; i++)
        cout << names[i] << endl;

    return 0;
}

```

Challenge

As a challenge, extend the names array to include an individual's title. Thus the promptNames() function will consider the fourth row in the names array to be the title. You will also need to modify main() so the output can be displayed.

See Also

The complete solution is available at [4-0-arrayOfStrings.cpp](#) or:

/home/cs124/examples/4-0-arrayOfStrings.cpp





## Example 4.0 – Array of Integers

### Demo

This example will create a  $4 \times 4$  array of integers. This will be done both the old-fashioned way of using a single-dimensional array as well as the new double-bracket notation. In both cases, the arrays will be filled with multiplication tables ( $\text{row} * \text{col}$ ).

The 16 items in a  $4 \times 4$  multiplication table represented as a single-dimensional array are:

	row 0				row 1				row 2				row 3			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
grid	0	0	0	0	0	1	2	3	0	2	4	6	0	3	6	9

To write a function to fill this array, two parameters are needed: the number of rows (`numRow`) and the number of columns (`numCol`).

```
void fillArray1D(int grid[], int numCol, int numRows)
{
    for (int row = 0; row < numRows; row++)
        for (int col = 0; col < numCol; col++)
            grid[row * numCol + col] = row * col;
}
```

To do the same thing as a multi-dimensional array, the data representation is:

	grid[0][]				grid[1][]				grid[2][]				grid[3][]			
	0,0	0,1	0,2	0,3	1,0	1,1	1,2	1,3	2,0	2,1	2,2	2,3	3,0	3,1	3,2	3,3
grid	0	0	0	0	0	1	2	3	0	2	4	6	0	3	6	9

To work with multi-dimensional arrays, the compiler has to know the number of rows in the array. This means that, unlike with the single-dimensional version, we can only pass the `numRow` parameter. The `numCol` must be an integer literal specified in the parameter.

```
void fillArray2D(int grid[][4], int numRows)
{
    for (int row = 0; row < numRows; row++)
        for (int col = 0; col < 4; col++)
            grid[row][col] = row * col;
}
```

### Challenge

As a challenge, change the program to display a  $5 \times 6$  table. What needs to change in the calling function? What needs to change in the two fill functions?

### See Also

The complete solution is available at [4-0-arrayOfIntegers.cpp](#) or:

`/home/cs124/examples/4-0-arrayOfIntegers.cpp`



## Example 4.0 – Convert Place To Points

Demo

Recall that there are two main uses for arrays: either they are a “bucket of variables” useful for storing lists of items, or they are tables useful for table-lookup scenarios. This example will demonstrate the table-lookup use for arrays.

Problem

Sam can make varsity on the track team if he gets 12 points. There are 10 races and points are awarded according to his placing:

Place	Points
1	5
2	3
3	2
4	1

Solution

We can create a data-driven program to compute how many points Sam will get during the season. If he gets 12 points and his varsity jacket, possibly Sue will want to go on another date with him!

```
{
    int points = 0;                                // initial points for the season
    int breakdown[4][2] =
    {
        {1, 5},                                     // 1st place gets 5 points
        {2, 3},                                     // 2nd place gets 3...
        {3, 2},
        {4, 1}
    };

    // Loop through the 10 races in the season
    for (int cRace = 0; cRace < 10; cRace++)          // “cRace” for “count Race”
    {
        // get the place for a given race
        int place;
        cout << "what was your place? ";
        cin >> place;

        // add the points to the total
        for (int cPlace = 0; cPlace < 4; cPlace++) // Loop through all the places
            if (breakdown[cPlace][0] == place)      // if place in the table matches
                points += breakdown[cPlace][1];      // assign the points
    }

    cout << points << endl;
}
```

Observe how the first column is directly related to the row (`breakdown[row][0] == row + 1`). This means we technically do not need to have a multi-dimensional array for this problem.

Challenge

As a challenge, adapt this solution to the points awarded to the finishers at the Tour de France:

20	17	15	13	12	10	9	8	7	6	5	4	3	2	1
----	----	----	----	----	----	---	---	---	---	---	---	---	---	---

In other words, the first finisher wins 20 points, the second 17, and so on.

See Also

The complete solution is available at [4-0-convertPlaceToPointers.cpp](#) or:

`/home/cs124/examples/4-0-convertPlaceToPoints.cpp`



## Example 4.0 – Read Scores

Demo

This example will demonstrate how to fill a multi-dimensional array of numbers from a file, how to display the contents of the array, and how to process data from the array.

Problem

Write a program to read assignment scores from 10 students, each student completing 5 assignments. The program will then display the average score for each student and for each assignment. If there were three students, the file containing the scores might be:

```
92 87 100 84 95
71 79 85 62 81
95 100 100 92 99
```

Solution

The function to read five scores for numStudents individuals is the following:

```
bool readData(int grades[][5], int numStudents, const char * fileName)
{
    ifstream fin(fileName);
    if (fin.fail())
        return false;

    // read the data from the file, one row (student) at a time
    for (int iStudent = 0; iStudent < numStudents; iStudent++)
    {
        // read all the data for a given student: 5 assignments
        for (int iAssign = 0; iAssign < 5; iAssign++)
            fin >> grades[iStudent][iAssign];

        if (fin.fail())
        {
            fin.close();
            return false;
        }
    }

    fin.close();
    return true;
}
```

Observe how two loops are required: the outer loop `iStudent` to go through all the students in the list. The inner loop `iAssign` reads all the scores for a given student.

Challenge

As a challenge, modify the above program and the associated data file to contain 6 scores for each student. What needs to change? Can you create a `#define` to make changes like this easier?

See Also

The complete solution is available at [4-0-readScores.cpp](#) or:

```
/home/cs124/examples/4-0-readScores.cpp
```



## Example 4.0 – Pascal’s Triangle

Pascal’s triangle is a triangular array of numbers where each value is the sum of the two numbers “above” it:

```
      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 1 5 10 10 5 1
```

Consider the number 6 in the second from bottom row. It is the sum of the 3 and the 3 from the preceding row. For a graphical representation of this relationship, please see [this animation](#).

We will implement Pascal’s triangle by turning the triangle on its side:

1	1	1	1	1	1	1	1	1	1	1
1	2	3	4	5	6	7	8	9	10	11
1	3	6	10	15	21	28	36	45	55	66
1	4	10	20	35	56	84	120	165	220	286
1	5	15	35	70	126	210	330	495	715	1001
1	6	21	56	126	252	462	792	1287	2002	3003
1	7	28	84	210	462	924	1716	3003	5005	8008
1	8	36	120	330	792	1716	3432	6435	11440	19448
1	9	45	165	495	1287	3003	6435	12870	24310	43758
1	10	55	220	715	2002	5005	11440	24310	48620	92378
1	11	66	286	1001	3003	8008	19448	43758	92378	184756

One the first row, the values are 1, 1, 1, etc. From here, the first item on each new row is also the value 1. Every other item is the sum of the previous row and the previous column.

```
void fill(int grid[][SIZE])
{
    // 1. fill the first row
    for (int column = 0; column < SIZE; column++)
        grid[0][column] = 1;

    for (int row = 1; row < SIZE; row++)
    {
        // 2. The first item on a new row is 1
        grid[row][0] = 1;

        // 3. Every other item is the sum of the item above and to the left
        for (int column = 1; column < SIZE; column++)
            grid[row][column] = grid[row - 1][column] + grid[row][column - 1];
    }
}
```

The complete solution is available at [4-0-pascalsTriangle.cpp](#) or:

/home/cs124/examples/4-0-pascalsTriangle.cpp



### Problem 1

What is returned if the input is 82?

```
char convert(int input)
{
    char letters[] = "ABCDF";
    int minRange[] =
        {90, 80, 70, 60, 0};

    for (int i = 0; i < 5; i++)
        if (minRange[i] <= input)
            return letters[i];

    return 'F';
}
```

Answer:

---

*Please see page 218 for a hint.*

### Problem 2

What is the output of the following code?

```
int num(int n, float * a)
{
    int s = 0;

    for (int i = 0; i < n; i++)
        s += (a[i] >= 80.0);

    return s;
}

int main()
{
    cout << num(5, {71.3, 84.7, 63.9, 99.8, 70})
        << endl;

    return 0;
}
```

Answer:

---

*Please see page 223 for a hint.*

### Problem 3

What is the output of the following code?

```
{
    char a[8] = "Rexburg";
    bool b[8] =
        {true, false, true, true,
         false, true, true, false};

    for (int i = 0; i < 8; i++)
        if (b[i])
            cout << a[i];

    cout << endl;
}
```

Answer:

---

*Please see page 218 for a hint.*

### Problem 4

What is the syntax error?

```
{
    char letter = 'a';

    switch (letter)
    {
        case 'a':
            cout << "A\n";
        case true:
            cout << "B!\n";
            break;
            break;
        case 1:
            cout << "C!\n";
            break;
    }
}
```

Answer:

---

*Please see page 287 for a hint.*

### Problem 5

Declare a variable to represent a Sudoku board:

---

*Please see page 49 for a hint.*

### Problem 6

What is wrong with each of the following array declarations?

```
int x = 6;  
int array[x][x];
```

```
const float x = 1;  
int array[x]
```

```
int array[][]
```

```
int array[6 * 5 + 2][4 / 2];
```

*Please see page 215 for a hint.*

### Problem 7

What is the output of the following code fragment?

```
{  
    int array[2][2] =  
        { {3, 4}, {1, 2} };  
  
    cout << array[1][0];  
}
```

Answer:

*Please see page 313 for a hint.*

### Problem 8

Consider an array that is declared with the following code:

```
int array[7][21];
```

Write a prototype of a function that will accept this array as a parameter.

Answer:

*Please see page 313 for a hint.*

## Assignment 4.0

Write a function to read a Tic-Tac-Toe board into an array. The file format is:

```
x o .  
.:.  
.x.
```

The character 'x' means that the 'x' player has taken that square. The character '.' means that the square is currently unclaimed. There is one space between each symbol.

**Note:** You will need to store the results in a 2D array. The function should take the filename as a parameter.

Write a function to display a Tic-Tac-Toe board on the screen. Given the above board, the output is:

```
x | o |  
---+---  
| | |  
---+---  
| x |
```

Write a function to write the board to a file. The file format is the same as with the read function.

## Example

The user input is underlined.

```
Enter source filename: board.txt  
x | o |  
---+---  
| | |  
---+---  
| x |  
Enter destination filename: board2.txt  
File written
```

## Assignment

The test bed is available at:

```
testBed cs124/assign40 assignment40.cpp
```

Don't forget to submit your assignment with the name "Assignment 40" in the header.

*Please see page 234 for a hint.*



## 4.1 Allocating Memory

After Sam finished his image program involving a multi-dimensional array, something was bothering him. While the program worked with the current 12 megapixel camera he owns, it will not work with images of any other size. This struck him as short-sighted; an image program should be able to work with any size image, even image sizes not known at compile time. In an effort to work around this glaring shortcoming, he discovers memory allocation.

### Objectives

By the end of this class, you will be able to:

- Allocate memory with the `new` operator.
- Free allocated memory with the `delete` operator.
- Allocate and free single and multi-dimensional arrays.

### Prerequisites

Before reading this section, please make sure you are able to:

- Declare a pointer variable (Chapter 3.3).
- Get the data out of a pointer (Chapter 3.3).
- Pass a pointer to a function (Chapter 3.3).

### Overview

Dynamic memory allocation is the process of a program reserving an amount of memory that is known at runtime rather than at compile time. In other words, the program is able to reserve as much memory as the user requires, even when the programmer has no idea how much memory that will be.

This is best explained by an analogy. Imagine a developer wishing to purchase an acre of land. He goes to City Hall to acquire two things: a deed and the address of the land. The deed is a guarantee the land will not be developed by anyone else, and the address is a pointer to the land so he knows where to find it. If there is no land available, then he will have to deal with the setback and make other plans. If land is available, the developer will walk out with a valid address. With this address and deed, the developer goes off to do something useful and productive with the newly acquired acre. Of course, the acre is not clean. The previous inhabitant of the land left some landscaping and structures on the land which will need to be removed and leveled before any building occurs. The developer retains ownership of the land until his business is completed. At this time, he returns to City Hall and returns the deed and forgets the address of the land.

This process is exactly what happens when working with memory allocation. The program (developer) asks the operating system (City Hall) for a range of memory (acre of land). If the request is greater than the amount of available memory, the request returns a failure condition which the program will need to handle. Otherwise, a pointer to the memory (address) will be given to the program as well as a guarantee that no other program will be given the same memory (deed). This memory is filled with random 1's and 0's from the previous occupant (landscaping and structures on the land) which will need to be initialized (leveled). When the program is finished with the memory, it should be returned to the operating system (returns the deed to City Hall) and the pointer should be set to `NULL` (forgets the address).

There are three parts to this process:

- `NULL`: The empty address indicating a pointer is invalid
- `new`: The operator used to request memory from the operating system
- `delete`: The operator used to tell the operating system the memory is no longer needed

## NULL Pointer

Up until this point, all the pointers we have used in our programs pointed to an existing location in memory. This location was always a local variable, meaning we could always assume that the pointer is referring to a valid location in memory. However, there often arises the occasion when the pointer refers to nothing. This situation requires us to mark the pointer so we can tell by inspection whether the address is valid.

To illustrate this point, remember our assignment (Assignment 3.2) where we computed the student's grade. The important thing about this assignment is that we are not to factor in the assignments the student has yet to complete. We marked these assignments with a -1 score. In essence, the -1 is a special token indicating the score is invalid or not yet completed. Thus, by inspection, we can tell if a score is invalid:

```
if (scores[i] == -1)
    cout << "No score for assignment " << i << endl;
```

The `NULL` pointer is essentially the same thing: an indication that a given pointer refers to no location in memory. We can check the validity of a pointer with a “NULL-check:”

```
if (p == NULL)
    cout << "The pointer does not refer to a valid location in memory\n";
```

## Definition of NULL

The first thing to realize about `NULL` is that it is an address. While we have created pointers to characters and pointers to integers, `NULL` is a pointer to `void`. This means we can assign `NULL` to any pointer without error:

```
{
    int    * pGrade   = NULL;           // we can assign NULL to any
    float  * pAccount = NULL;           // type of pointer without
    char   * name     = NULL;           // casting
}
```

The second thing to realize about `NULL` is that the numeric address is zero. Thus, the definition of `NULL` is:

```
#define NULL (void *)0x00000000
```

As you can well imagine, choosing zero as the `NULL` address was done on purpose. All valid memory locations are guaranteed to be not zero (the operating system owns that location: the first instruction to be executed when a computer boots). Also, zero is the only `false` value so `NULL` is the only `false` address. This makes doing a “NULL-check” easy:

```
if (p) // same as "if (p != NULL)"
    cout << "The address of p is valid!\n";
```

## Using NULL

One use of the NULL address is to indicate that a pointer is not valid. This can be done when there is nothing to point to. Consider the following function displaying the highest ‘A’ in a list of numeric grades. While commonly there is at least one ‘A’ in a list of student grades, it is not always the case.

```
/******  
 * DISPLAY HIGHEST A  
 * Given a list of numeric grades  
 * for a class, display the highest  
 * A if one exists  
 *****/  
void displayHighestA(const int grades[], // we won't be changing this so  
                    int num)           // the array is a const  
{  
    const int * pHighestA = NULL;       // Initially no 'A's were found  
  
    // find the highest A  
    for (int count = 0; count < num; count++) // loop through all the grades  
        if (grades[count] >= 90)           // only A's please  
        {  
            if (pHighestA == NULL)         // if none were found, then any  
                pHighestA = &(grades[count]); // 'A' is the highest  
            else if (*pHighestA < grades[count]) // otherwise, only the highest if it  
                pHighestA = &(grades[count]); // is better than any other  
        }  
  
    // output the highest A  
    if (pHighestA) // classic NULL check: only display the  
        cout << *pHighestA << endl; // 'A' if one was found  
    else // otherwise (pHighestA == NULL),  
        cout << "There was not an A\n"; // none was found  
}
```

## NULL check

Probably the most common use of NULL is to do a “NULL-check.” Because we expect our program to be set up correctly and pointers to always have valid addresses, it is common to add an assert just before dereferencing a pointer to make sure we won’t crash.

```
{  
    char * pLetter = NULL; // first set the pointer to NULL  
                           // to indicate it is uninitialized  
  
    if (isalpha(value[0]))  
        pLetter = value + 0; // observe how pLetter is set in both  
    else // conditions of the IF statement  
        pLetter = value + 1;  
  
    assert(pLetter != NULL); // like any assert, this should never  
    cout << "Letter: " // fire unless the programmer made  
        << *pLetter // a mistake. It is better to fire  
        << endl; // than to crash, of course!  
    pLetter = NULL; // indicate we are done by setting the  
} // pointer back to NULL
```

If we habitually set our pointers to NULL and then assert just before they are dereferenced, we can catch a ton of bugs. These bugs are also easy to fix, of course; much easier than a random crash!

## Allocation with New

When we declare a local variable (also known as a stack variable), the compiler takes care of memory management. The compiler makes sure that there is memory reserved for the variable as soon as the variable falls into scope. The compiler also makes sure the memory is freed as soon as the variable falls out of scope. While this is very convenient, it can also be very limiting: the compiler needs to know the size of the block of memory to be reserved and how long it will be needed. Memory allocation relaxes both of these constraints.

We request new memory from the operating system with the `new` operator. The syntax is:

```
<pointer variable> = new <data-type>;
```

If, for example, a `double` is to be allocated, it is accomplished with:

```
{
    double * p;           // "p" is a pointer to a double
    p = new double;       // allocating a double returns a pointer to a double
}
```

We can also initialize a block of memory at allocation time. The syntax is very similar:

```
<pointer variable> = new <data-type> ( <initialization value> );
```

If, for example, you wish to allocate a character and initialize it with the letter 'A', then:

```
{
    char * p = new char('A');
}
```

This does three things: it reserves a byte of memory (`sizeof(char) == 1`), it initializes that value to 65 (`'A' == 65`), and it sends that address to the variable `p`.

## Memory allocation failure

We cannot generally assume that a memory allocation is successful. In other words, it might be the case that there is no more memory to be had. Our code needs to be able to detect this condition and gracefully handle the error.

When a new request fails, the resulting pointer is `NULL`. However, we need to tell `new` that we wish to be notified of a failure in terms of the `NULL` pointer. This is done with the `nothrow` parameter:

```
{
    int * p = new (nothrow) int;           // notice the nothrow parameter
    if (p == NULL)                         // failure comes in the form of a
        cout << "Memory allocation failure!\n"; // NULL pointer
}
```

Every memory allocation should be accompanied by a `NULL` check; never assume an allocation succeeded.

### Sam's Corner

There are two ways the `new` operator reports errors: returning a `NULL` pointer or throwing an exception. Since exception handling is a CS 165 topic, we will use the `NULL` check this semester.



## Allocating arrays

We allocate arrays much the same way we allocate individual data-types. The difference, of course, is that we need to tell new how many instances of the data-type are needed. The syntax is:

```
<array variable> = new <data-type> [ <size> ];
```

Note that, unlike with array local variables, the size parameter does not need to be a constant or a literal. In other words, since new is essentially a function call, the compiler does not need to know how much data will be allocated; it can be determined at run-time. Consider the following code:

```
{
    // get the size of the text
    int size;
    do
    {
        cout << "How long is your name? ";
        cin >> size;
    }
    while (size <= 0);

    // allocate the memory
    char * text = new(nothrow) char [size + 1];
    if (!text)
        cout << "No memory! This is bad!\n";

    // prompt for the name
    cout << "What is your name? ";
    cin.getline(text, size + 1);

    // memory size variables are integers
    // continue prompting until the
    // user gives us a positive
    // size
    // allocate one more for \0
    // same as "if (text != NULL)"
    // should return because we will
    // crash in a minute...
    // treat "text" like any other string
}
```

## Freeing with Delete

Once we are finished with a given block of memory, it is important to return it to the operating system so another program (or part of our own program!) can use it. This is accomplished with the delete operator. Note that we don't need to do this with traditional local variables because, once the variable falls out of scope, the compiler frees it for us. However, with memory allocation, the programmer (not the compiler!) indicates when the memory is no longer needed.

The syntax for the delete operator is:

```
delete <pointer variable>;
delete [] <array pointer variable>;
```

Consider the following example to allocate an integer and a string:

```
{
    int * p = new int;
    char * text = new char[256];

    delete p;
    delete [] text;
}
```



### Sue's Tips

To make sure we don't try to use newly freed memory, always assign the pointer to NULL after delete.

## Example 4.1 – AllocateValue

Demo

In the past, we used pointers to refer to data that was declared elsewhere. In the following example, the pointer is referring to memory we newly allocated: a `float`. We will allocate space for a variable, fill the variable with a value, and free the memory when completed.

Solution

There are four parts to this process: creating a pointer variable so we can remember the memory location that was allocated, allocate the memory with `new`, use the memory location using the dereference operator `*`, and freeing the memory with `delete` when finished.

```
/* *****  
 * EXAMPLE  
 * This is a bit contrived so I can't think  
 * of a better name  
 * ***** */  
void example()  
{  
    // At first, the pointer refers to nothing. We use the NULL pointer  
    // to signify the address is invalid or uninitialized  
    float * pNumber = NULL;  
  
    // now we will allocate 4 bytes for a float.  
    pNumber = new(nothrow) float;  
    if (!pNumber)  
        return;  
  
    // at this point (no pun intended), we can use it like any other pointer  
    assert(pNumber);  
    *pNumber = 3.14159;  
  
    // Regular variables get recycled after they fall out of scopes. Not true  
    // with allocated data. We need to free it with delete  
    delete pNumber;  
    pNumber = NULL;  
}
```

Challenge

As a challenge, try to break this example into three functions: one function to allocate the memory returning a pointer to a float, one to change the value taking a pointer to a float as a parameter, and a final function to free the data.

See Also

The complete solution is available at [4-1-allocateValue.cpp](#) or:

`/home/cs124/examples/4-1-allocateValue.cpp`



Unit 4

## Example 4.1 – Allocate Array

Demo

This example will demonstrate how to allocate an array of integers. Unlike with traditional arrays, we will be able to prompt the user for the number of items in the array.

Problem

Write a program to prompt the user for the number of items in a list and the values for the list.

```
How many items? 3
Please enter 3 values
# 1: 100
# 2: 200
# 3: 400
A display of the list:
100
200
400
```

Solution

First, we will write a function to allocate the list given, as a parameter, the number of items.

```
int * allocate(int numItems)
{
    assert(numItems > 0);           // better be a positive number!
    // Allocate the necessary memory
    int *p = new(nothrow) int[numItems];    // all the work is done here.

    // if p == NULL, we failed to allocate
    if (!p)
        cout << "Unable to allocate " << numItems * sizeof(int) << " bytes\n";
    return p;
}
```

This function is called by main(), which also calls a function to fill and display the list.

```
int main()
{
    int numItems = getNumItems();
    assert(numItems > 0);

    // allocate the memory
    int * list = allocate(numItems); // allocated arrays go in pointer variables
    if (list == NULL)
        return 1;

    // do something with it
    fillList(list, numItems);        // always pass the size with the array
    displayList(list, numItems);

    // make like a tree
    delete [] list;                  // never forget to release the memory
    list = NULL;                     // you can say I am a bit paranoid
    return 0;
}
```


See Also

The complete solution is available at [4-1-allocateArray.cpp](#) or:

```
/home/cs124/examples/4-1-allocateArray.cpp
```



## Example 4.1 – Expanding Array

Demo	This example will demonstrate how to grow an array to accomidate an unlimited amount of data. This will be accomplished by detecting when the array is full, allocating a new buffer of twice the size as the first, copying the original data to the new buffer, and freeing the original buffer.
Problem	<p>Write a program to read all the data in a file into a single string, then report how much data was read.</p> <pre> Filename: <b>4-1-expandingArray.cpp</b> reallocating from 4 to 8 reallocating from 8 to 16 reallocating from 16 to 32 reallocating from 32 to 64 reallocating from 64 to 128 reallocating from 128 to 256 reallocating from 256 to 512 reallocating from 512 to 1024 reallocating from 1024 to 2048 reallocating from 2048 to 4096 Total size: 2965 </pre>
Solution	<p>Most of the work is done in the reallocate function. It will double the size of the current buffer.</p> <pre> char * reallocate(char * bufferOld, int &amp;size) {     cout &lt;&lt; "reallocating from " &lt;&lt; size &lt;&lt; " to " &lt;&lt; size * 2 &lt;&lt; endl;      // allocate the new buffer     char *bufferNew = new(nothrow) char[size * 2];     if (NULL == bufferNew)     {         cout &lt;&lt; "Unable to allocate a buffer of size " &lt;&lt; size &lt;&lt; endl;         size /= 2;                // reset the size         return bufferOld;     }      // copy the data into the new buffer     int i;     for (i = 0; bufferOld[i]; i++)                // use index because it is easier         bufferNew[i] = bufferOld[i];              //   than two pointers     bufferNew[i] = '\0';                          // don't forget the NULL      // delete the old buffer     delete [] bufferOld;      // return the new buffer     return bufferNew; } </pre>
Challenge	It may seem a bit wasteful to double the size of the buffer with every reallocation. Would it be better to increase the size by 50%, by 200%, or by a fixed amount (say 100 characters)? Modify the above code to accommodate these different strategies and find out which has the smallest amount of wasted space and the smallest number of reallocations.
See Also	<p>The complete solution is available at <a href="#">4-1-expandingArray.cpp</a> or:</p> <pre>/home/cs124/examples/4-1-expandingArray.cpp</pre> 



## Example 4.1 – Allocate Images

### Demo

Our final example will demonstrate how to allocate the space necessary to display a digital picture. In this example, the user will provide the size of the image; it is not known at compile time. There is one important side-effect from this: we cannot use the multi-dimensional array notation with allocated memory because the compiler must know the size of the array to use that notation. Since the size is not known until run-time, this approach is impossible.

The code to allocate the image is the following:

```
/* *****
 * ALLOCATE
 * Grab the memory, returning NULL if
 * anything went wrong
 * ***** */
char * allocate(int numRows, int numCol)
{
    assert(numRow > 0 && numCol > 0);

    // we allocate a 1-dimensional array and do the
    // two dimensional math ourselves
    char * image = new(nothrow) char[numRow * numCol];
    if (!image)
    {
        cout << "Unable to allocate "
              << numRows * numCol * sizeof(char)
              << " bytes for a "
              << numCol << " x " << numRows
              << " image\n";
        return NULL;
    }
    return image;
}
```

### Solution

Observe how we must work with 1-dimensional arrays even though the image is 2-dimensional. Therefore we must do the transformations ourselves:

```
/* *****
 * DISPLAY
 * Display the image. This is ASCII-art
 * so it is not exactly "High resolution"
 * ***** */
void display(const char * image, int numRows, int numCol)
{
    // paranoia
    assert(image);
    assert(numRow > 0 && numCol > 0);

    // display the grid
    for (int row = 0; row < numRows; row++)          // two dimensional loop, first
    {                                                  // the rows, then
        for (int col = 0; col < numCol; col++)        // the columns
            cout << image[row * numCol + col];        // do the [] math ourselves
        cout << endl;
    }
}
```

### See Also

The complete solution is available at:

```
/home/cs124/examples/4-1-allocateImages.cpp
```

### Problem 1

What is the output of the following code?

```
{  
    float * p = NULL;  
  
    cout << sizeof(p) << endl;  
}
```

Answer:

---

*Please see page 258 for a hint.*

### Problem 2

What is the output of the following code?

```
{  
    int a[40];  
  
    cout << sizeof(a[42]) << endl;  
}
```

Answer:

---

*Please see page 215 for a hint.*

### Problem 3

How much memory does each of the following variables require?

<code>char text[2]</code>	
<code>char text[] = "Software";</code>	
<code>int nums[2];</code>	
<code>bool values[8];</code>	

*Please see page 215 for a hint.*

### Problem 4

Write the code to declare a pointer to an integer variable and allocate it.

Answer:

*Please see page 327 for a hint.*

### Problem 5

How do you indicate that you no longer need memory that was previously allocated? Write the code to free the memory pointed to by the variable p.

Answer:

*Please see page 328 for a hint.*

### Problem 6

What statement is missing in the following code?

```
{  
    float * pNum = new float;  
    delete pNum;  
    <statement belongs here>  
}
```

Answer:

*Please see page 325 for a hint.*

### Problem 7

How much memory is allocated with each of the following?

<code>p = new double;</code>	
<code>p = new char[8];</code>	
<code>p = new int[6];</code>	
<code>p = new char(65);</code>	

*Please see page 328 for a hint.*

### Problem 8

Write the code to prompt the user for a number of `float` grades, then allocate an array just big enough to store the array.

*Please see page 330 for a hint.*

## Assignment 4.1

Write a program to:

- Prompt the user for the number of characters in a string
- Allocate a string of sufficient length (one more than # of characters!)
- Prompt the user for the string using `getline`
- Display the string back to the user
- Don't forget to release the memory and check for allocation failures!

Note that since the first `cin` will leave the stream pointer on the newline character, you will need to use `cin.ignore()` before `getline()` to properly fetch the section input.

### Examples

Three examples... The user input is underlined.

#### Example 1

```
Number of characters: 13
Enter Text: NoSpacesHere!
Text: NoSpacesHere!
```

#### Example 2

```
Number of characters: 45
Enter Text: This is a ton of characters. How long is it?
Text: This is a ton of characters. How long is it?
```

#### Example 3 (allocation failure)

```
Number of characters: -10
Allocation failure!
```

### Assignment

The test bed is available at:

```
testBed cs124/assign41 assignment41.cpp
```

Don't forget to submit your assignment with the name "Assignment 41" in the header.

*Please see page 330 for a hint.*

## 4.2 String Class

Sue has just finished the Mad Lib® assignment and found working with strings to be tedious and problematic. There were so many ways to forget the NULL character! She decides to spend a few minutes and create her own string type so she never has to make that mistake again. As she sits in the lab drafting a solution, Sam walks in and peers over her shoulder. “You know,” he says “there is already a tool that does all those things...”

### Objectives

By the end of this class, you will be able to:

- Use the String Class to solve a host of text manipulation problems.
- Understand which string operations are expensive and which are not.

### Prerequisites

Before reading this section, please make sure you are able to:

- Understand the role the NULL character plays in string definitions (Chapter 3.2).
- Write a loop to traverse a string (Chapter 3.2).

## Overview

One of the principles of Object Oriented programming (the topic of CS 165) is encapsulation, the process of hiding unimportant implementation details from the user of a tool so he can focus on how the tool can be used to solve his problem. To date, we have used two tools exemplifying this property: input file streams (`cin` and `fin`) and output file streams (`cout` and `fout`). We learned how to use these tools to solve programming problems, but never got into the details of how they work. Another similarly powerful tool is the String Class.

The String Class is a collection of tools defined in a library allowing us to easily manipulate text. With the String Class, the programmer does not need to worry about buffer sizes (we were using 256 up to this point), NULL characters, or using loops to copy text. In fact, most of the most common operations work as though they are operating on a simple data type (such as an integer), allowing the programmer to forget he is even working with arrays. Consider the following simple example:

```
#include <string>                                // use the string library

/*****
 * DEMO: simple string-class demo
 *****/
void demo()
{
    string lastName;                             // the data-type is "string," no []s
    cout << "What is your last name? ";          //      as were needed with c-strings
    cin  >> lastName;                             // cin works the way you expect

    string fullName = "Mr. " + lastName;          // the + operator appends

    cout << "Hello " << fullName << endl;        // cout works the way you expect
}
```

# Syntax

The syntax of the String Class is designed to be as streamlined and intuitive as possible. There are several components: the string library, declaring a string, interfacing with streams, and performing common text manipulation operations.

## String library

Unlike c-strings (otherwise known as “the array type for strings”), the String Class is not built into the C++ language. These tools are provided in the string library as part of the standard namespace. Therefore, it is necessary to include the following at the top of your programs:

```
#include <string>
```

If this line is omitted, the following compiler message will appear:

```
example.cpp: In function “int main()”:  
example.cpp:4: error: “string” was not declared in this scope
```

Note that our compiler actually includes the string library as part of iostream. This, technically, is not part of the iostream library design. We should never rely on this quirk of our current compiler library and always include the string library when using the String Class.

## Declaring a string

With a c-string and all other built-in data-types in C++, variables are not initialized when they are declared. This is not true with the String Class. The act of declaring a string variable also initializes it to the empty string.

```
{  
    string text1;           // initialized to the empty string  
    cout << text1;         // displays the empty string: nothing  
  
    char text2[256];        // not initialized  
    cout << text2;         // random data will be displayed  
}
```

Observe how we do not specify the size of a string when we declare it. This is because the authors of the String Class did not want you to have to worry about such trivial details. Initially, the size is zero. However, as more data is added to the string, it will grow. We can always ask our string variable for its current capacity:

```
{  
    string textEmpty;  
    string textFull = "Introduction to Software Development";  
  
    cout << textEmpty.capacity() << endl;           // 0: the buffer is currently empty  
    cout << textFull.capacity() << endl;           // 64: the first power of 2 greater  
}
```

### Sam's Corner

The String Class buffer is dynamically allocated. This allows the buffer to grow as the need arises. It also means that we need to use the capacity() function to find the size rather than sizeof(). When the string variable falls out of scope, its storage capacity is automatically freed unless it has been allocated with new.



## Stream interfaces

We were able to use `cin` and `cout` with all the built-in data-types (`int`, `float`, `bool`, `char`, `double`, etc.), but not with arrays (but we are able to use them with individual elements of arrays). The one exception to this is c-strings; `cin` and `cout` treat pointers-to-characters as c-strings. When we include the string library, `cin` and `cout` are also able to work with string variables:

```
{
    string text;

    cin  >> text;           // works the same as a c-string
    cout << text << endl;   // both cin and cout work
}
```



### Sue's Tips

When working with c-strings, we had to be careful to not put more data in the buffer than there was room. This was problematic with c-strings, unfortunately, because there was no way to tell `cin` how big the string is:

```
char text[10];
cin >> text;    // BUG! The user can enter more than 9 characters
```

With the String Class, the buffer size grows to accommodate the user input. This means that it is impossible for the user to specify more input than there is space in the buffer; the buffer will simply grow until it is big enough.

```
string text;
cin >> text;    // Safe! text can accommodate any amount
                // of user input
```

With c-strings, we can use `getline` to fetch an entire line of text. We can also use `getline` with the String Class, but the syntax is quite odd:

```
{
    // first the c-string syntax
    char text1[256];           // don't forget the buffer
    cin.getline(text1, 256);   // the buffer size is a required parameter

    // now the String Class
    string text2;              // no buffer needed here
    getline(cin, text2);       // note how cin is the parameter!
}
```

This syntax is, unfortunately, something we will just have to remember.

### Sam's Corner



The reason for the String Class' strange `getline` syntax is a bit subtle. All the functionality associated with `cin` and `cout` are in the `iostream` library. If the `getline` method associated with `cin` took a string as a parameter, then the `iostream` library would need to know about the String Class. The `iostream` library must be completely ignorant of the String Class; otherwise everyone would be required to include the string library when they do any console I/O. This would make the coupling between the libraries tighter than necessary.

The string library extends the definition of `cin` and `cout`. We will learn more about how this is done in CS 165 where we learn to make our own custom data-types work with `cin` and `cout`.



## Text manipulation

Great pains was taken to make text manipulation with `string` objects as easy and convenient as possible. We can append with the plus operator:

```
{
    string prefix = "Mr. ";
    string postfix = "Smith";

    string name = prefix + postfix;    // concatenation with the + operator. There is a
}                                     //   FOR loop hidden here. See Sue's tip below
```

We can copy strings with the equals operator:

```
{
    string text = "CS 124";
    string copy;

    copy = text;                      // copy with the = operator. There is a FOR loop
}                                     //   hidden here. See Sue's tip below.
```

We can compare with the double-equals operator: Note that `>`, `>=`, `<`, `<=` and `!=` work as you would expect.

```
{
    string text1;
    string text2;

    cin >> text1 >> text2;

    if (text1 == text2)                // same with the other comparison operators
        cout << "Same!\n";           //   such as == != > >= < <=
    else                               //   There is a FOR loop hidden here!
        cout << "Different!\n";       //   See Sue's tip below for a hint.
}
```

Finally, we can retrieve individual members of a string with the square-brackets operator:

```
{
    string text = "CS 124";

    for (int i = 0; i < 6; i++)
        cout << text[i] << endl;    // access data with the [] operator;
}                                     //   this is very fast! No FOR loops here
```

For more functionality associated with the `String` Class, please see:

[http://en.cppreference.com/w/cpp/string/basic\\_string](http://en.cppreference.com/w/cpp/string/basic_string)



### Sue's Tips

Be careful of hidden performance costs when working with the `String` Class. Seemingly innocent operations (like `=` or `==`) must have a `FOR` loop in them. Ask yourself “if this were done with `c-strings`, would a `LOOP` be required?”

Please see the following example for a test of the performance of the `append` operator in the `String` Class at [4-2-stringPerformance.cpp](#) or:

```
/home/cs124/examples/4-2-stringPerformance.cpp
```

## Example 4.2 – String Demo

Demo

This example will demonstrate how to declare a string, accept user input into a string, append text onto the end of a string, copy a string, and display the output on the screen.

Solution

All these common string operations will be demonstrated in a single function:

```
#include <iostream>
#include <string>          // don't forget this library
using namespace std;

/*****
 * MAIN: Simple demo of the string class.
 *****/
int main()
{
    string firstName;    // no []s required. The string takes care of the buffer
    string lastName;

    // cin and cout work as you would expect with the string class
    cout << "What is your name (first last): ";
    cin >> firstName >> lastName;

    // Append ", " after last name so "Young" becomes "Young, ".
    // To do this, we will use the += operator.
    lastName += ", ";

    // Create a new string containing the first and last name so
    // "Brigham" "Young"
    // becomes
    // "Young, Brigham".
    // To do this we will use the + operator to combine
    // two strings and the = operator to assign the results to a new string.
    string fullName = lastName + firstName;

    // display the results of our nifty creation
    cout << fullName << endl;

    return 0;
}
```

Challenge

As a challenge, change the above program to prompt the user for his middle name. Append this new string into fullName so we get the expected output.

See Also

The complete solution is available at [4-2-stringDemo.cpp](#) or:

/home/cs124/examples/4-2-stringDemo.cpp



## Using string objects as file name variables

One might be tempted to use the string class for a file name. This makes the `getFilename()` function more intuitive and elegant:

```
/* *****  
 * GET FILENAME  
 * Prompt the user for a filename and return it  
 * ***** */  
string getFilename()                // no pass-by-pointer parameter!  
{  
    string fileName;                // local variable  
    cout << "Please provide a filename: ";  
    cin >> fileName;  
    return fileName;                // we can return a local variable  
}                                   // without fear it will be destroyed
```

From this point, how do we use the string `fileName`? Consider the following code:

```
{  
    string fileName = getFilename();    // this works as you might expect  
    ifstream fin(fileName);            // ERROR! Wrong parameter!  
}
```


The error is:

```
example.cpp:215: error: no matching function for call to "std::basic_ofstream<char,  
std::char_traits<char> >::open(std::string&)" /usr/lib/gcc/x86_64-redhat-  
linux/4.4.5/../../../../include/c++/4.4.5/fstream
```

This means that the file name parameter needs to be a c-string, not a string class. To do this, we need to convert our string variable in to a c-string variable. This is done with the `c_str()` function:

```
{  
    string fileName = getFilename();  
    ifstream fin(fileName.c_str());    // c_str() returns a pointer to a char  
}
```

## Example 4.2 – String Properties

Demo	This example will demonstrate how to manipulate a string class object in a similar way to how we did this with c-strings. This will include using pointers as well as using the array notation to traverse a string.
Problem	<p>Write a program to prompt the user for some text, display the number of characters, the number of spaces, and the contents of the string backwards.</p> <pre>Please enter some text: <u>Software Development</u> Number of characters: 20 Number of spaces: 1 Text backwards: tnempoleveD erawtfoS</pre>
Solution	<p>We can get the length of a string with:</p> <pre>// find the number of characters cout &lt;&lt; "\tNumber of characters: "     &lt;&lt; text.length()     &lt;&lt; endl;</pre> <p>We can traverse the string using a pointer notation by getting a pointer to the start of the string with the <code>c_str()</code> method. This will return a constant pointer to a character.</p> <pre>// find the number of spaces int numSpaces = 0; for (const char *p = text.c_str(); *p; p++)     if (*p == ' ')         numSpaces++;</pre> <p>Finally, we can traverse the string using the array index notation:</p> <pre>// display the string backwards. cout &lt;&lt; "\tText backwards: "; for (int i = text.length() - 1; i &gt;= 0; i--)     cout &lt;&lt; text[i]; cout &lt;&lt; endl;</pre>
Challenge	<p>As a challenge, try to change the case of all the characters in the string. This means converting uppercase characters to lowercase, and vice-versa. As you may recall, we did this earlier (please see p. 249).</p> <p>As another challenge, try to count the number of digits in the string. You may need to use <code>isdigit()</code> from the <code>cctype</code> library to accomplish this.</p>
See Also	<p>The complete solution is available at <a href="#">4-2-stringProperties.cpp</a> or:</p> <pre>/home/cs124/examples/4-2-stringProperties.cpp</pre> 

## Problem 1

Declare three string class variables. Prompt the user for the values and display them:

```
Please specify a name: Sam
Please specify another name: Sue
Please specify yet another name: Sid
The names are: "Sam" "Sue," and "Sid"
```

Answer:

*Please see page 338 for a hint.*

## Problem 2

From the code written for Problem 1, create a new string called `allNames`. This string will be created in the following way: first name, ", ", second name, ", and ", and third name. Display the new string.

```
Please specify a name: Sam
Please specify another name: Sue
Please specify yet another name: Sid
The names are: "Sam, Sue, and Sid"
```

Answer:

*Please see page 341 for a hint.*

## Problem 3

From the code written for Problem 2, append the strings in alphabetical order to `allName`. You will need to use the `>` operator to compare strings, which works much like it does with integers.

```
Please specify a name: Sam
Please specify another name: Sue
Please specify yet another name: Sid
The sorted names are: "Sam, Sid, and Sue"
```

Answer:

*Please see page 340 for a hint.*

## Assignment 4.2

Consider the child's song "Dem Bones" found at ([http://en.wikipedia.org/wiki/Dem\\_Bones](http://en.wikipedia.org/wiki/Dem_Bones) or <http://www.youtube.com/>). Sue would like to write a program to display the first eight verses of the song. However, realizing that the song is highly repetitive, she would like to write a function to help her with the task.

Please write a function to generate the Dem Bones song. This function takes an array of strings as input and returns a single string that constitutes the entire song as output:

```
string generateSong(string list[], int num);
```

Consider the case where `num == 4` and `list` has the following items: `toe`, `foot`, `knee`, and `hip`. This will generate the following string:

```
toe bone connected to the foot bone  
foot bone connected to the knee bone  
knee bone connected to the hip bone
```

## Directions

For this problem, the stub function `generateSong()` as well as `main()` is written for you. Your job is to implement the function `generateSong()`. The file is located at:

```
/home/cs124/assignments/assign42.cpp
```

Please start with the above file because **most of the program is written for you!**

## Assignment

The test bed is available at:

```
testBed cs124/assign42 assignment42.cpp
```

Don't forget to submit your assignment with the name "Assignment 42" in the header.

*Please see page 341 for a hint.*

## 4.3 Command Line

Sue has just finished writing a program for her mother to convert columns of Euros to Dollars. In order to make this program as convenient and user-friendly as possible, she chooses to allow the input to be specified by command line parameters.

### Objectives

By the end of this chapter, you will be able to:

- Write a program to accept passed parameters.
- Understand jagged arrays.

### Prerequisites

Before reading this chapter, please make sure you are able to:

- Create a function in C++ (Chapter 1.4).
- Pass data into a function using both pass-by-value and pass-by-reference (Chapter 1.4).
- Declare a pointer variable (Chapter 3.3).
- Pass a pointer to a function (Chapter 3.3).
- Declare a multi-dimensional array (Chapter 4.0).

## Overview

Most operating systems (Windows, Unix variants such as Linux and OS X, and others) enable you to pass parameters to your program when the program starts. In other words, it is possible for the user to send data to the program before the program is run. Most of the Linux commands you have been using all semester take command line parameters:

Command	Example	Parameters
List the contents of a directory	<code>ls *.cpp</code>	<code>*.cpp</code>
Submit a homework assignment	<code>submit project4.cpp</code>	<code>project4.cpp</code>
Change to a new directory	<code>cd submittedHomework</code>	<code>submittedHomework</code>
Copy a file from one location to another	<code>cp /home/cs124/template.cpp assignment43.cpp</code>	<code>/home/cs124/template.cpp</code> <code>assignment43.cpp</code>

In each of these cases, the programmers configured their programs to accept command line parameters. The purpose of this chapter is to learn how to do this for our programs.

# Syntax

To configure your program to accept command line parameters, it is necessary to define `main()` a little differently than we have done this semester up to this point.

```
int main(int argc, char ** argv)
{
    ...
}
```

Observe the two cryptic parameter names. In the past we left the parameter list of `main()` empty. When this list is empty, we are ignoring any command line parameters the user may send to us. However, when we specify the `argc` and `argv` parameters, we can access the user-sent data from within our program.

## argc

The first parameter is the number of arguments or parameters the user typed. For example, if the program name is `a.out` and the user typed...

```
a.out one two three
```

... then `argc` would equal 4 because four arguments were typed: `a.out`, `one`, `two`, `three`. Just like with any passed parameter, the user can name this anything he wants. The convention is to use the name `argc`, but you may want to name it something more meaningful like `countArgument`.

## argv

The second parameter is the list of arguments. In all cases, we get an array of c-strings. Back to our example above where the user typed...

```
a.out one two three
```


... then `argv[0]` equals `"a.out"` and `argv[1]` equals `"one"`. Again, note that the first parameter is always the name of the program. As with `argc`, `argv` is not the best name. It means “argument vector” or “list of unknown arguments.” Possibly a better name would be `listArguments`.

Note the unusual `**` notation for the syntax of `argv`. This means that we have a pointer to a pointer. Possibly a better definition would be:

```
int main(int argc, char * argv[])
{
    ...
}
```

This might better illustrate what is going on. Here, we have an array of c-strings. How many items are in the c-string? This is where `argc` comes into play.



Example 4.3 – Reflect the Command Line	
Demo	This example will demonstrate how to accept data passed from the user through the command line.
Problem	<p>Write a program to display on the screen the name of the program, how many parameters were passed through the command line, and the contents of each parameter.</p> <pre> /home/cs124/examples&gt; <u>a.out one two three</u> The name of the program is: a.out There are 3 parameters     Parameter 1: one     Parameter 2: two     Parameter 3: three </pre>
Solution	<p>The first thing to look for is how <code>main()</code> has two parameters. Of course we can call these anything we want, but <code>argc</code> and <code>argv</code> are the standard names. Next, observe how <code>argv[0]</code> is the name of the program. Finally, we access each parameter with <code>argv[iArg]</code>.</p> <pre> #include &lt;iostream&gt; using namespace std;  /*****  * MAIN: Reflect back to the user what he  * typed on the command prompt  *****/ int main(int argc, char ** argv)    // again, MAIN really takes two parameters {     // name of the program     cout &lt;&lt; "The name of the program is: "          &lt;&lt; argv[0]                // the first item in the list is always          &lt;&lt; endl;                  // the command the user typed      // number of parameters     cout &lt;&lt; "There are "          &lt;&lt; argc - 1               // don't forget to subtract one due to          &lt;&lt; " parameters\n";       // the first being the program name      // show each parameter     for (int iArg = 1; iArg &lt; argc; iArg++) // standard command line loop         cout &lt;&lt; "\tParameter" &lt;&lt; iArg               &lt;&lt; ": " &lt;&lt; argv[iArg] &lt;&lt; endl; // each argv[i] is a c-string      return 0; } </pre>
Challenge	As a challenge, try to rename <code>a.out</code> (using the <code>mv</code> command) and run it again. How do you suppose the program knows that it was renamed after compilation?
See Also	<p>The complete solution is available at <a href="#">4-3-commandLine.cpp</a> or:</p> <pre>/home/cs124/examples/4-3-commandLine.cpp</pre> 

### Example 4.3 – Get Filename

Demo

This example will demonstrate one of the most common uses for command line parameters: fetching the filename from the user. This will be done using the string class.

Problem

Write a program to prompt the user for a filename. This can be provided either through a command line parameter or, if none was specified, through a traditional prompt.

```
/home/cs124/examples> a.out fileName.txt
The filename is "fileName.txt"
```

```
/home/cs124/examples> a.out
Please enter the filename: fileName.txt
The filename is "fileName.txt"
```

Solution

There are three parts to this program. First, the program will determine if an erroneous number of parameters were specified. If this proves to be the case, the program exists with a suitable error message. Next, the program prompts the user for a filename if there are no parameters specified on the command line. Finally, if there is a parameter specified, it is copied into the `fileName` variable.

```
int main(int argc, char ** argv)
{
    // ensure the correct number of parameters was specified
    if (argc > 2) // one for the name of the program, one for the filename
    {
        cout << "Unexpected number of parameters.\nUsage:\n";
        cout << "\t" << argv[0] << " [filename]\n";
        return 1;
    }

    // parse the command line
    string fileName;
    if (argc == 1) // only the program name was specified
        fileName = getFilename();
    else
        fileName = argv[1];

    // display the results
    cout << "The filename is \"" << fileName << "\"\n";

    return 0;
}
```

Challenge

As a challenge, can you do this without the `string` class? Hint: use `strcpy()` from the `cstring` library. Another challenge is to modify Project 3 to accept a filename as a command line parameter. Can you do this for Project 4 as well?

See Also

The complete solution is available at [4-3-getFilename.cpp](#) or:

```
/home/cs124/examples/4-3-getFilename.cpp
```

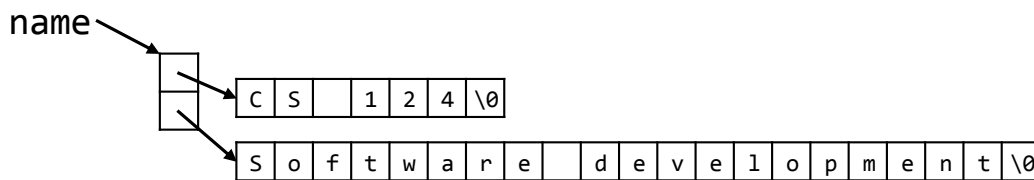


## Jagged Arrays

Multi-dimensional arrays can be thought of as arrays of arrays where each row is the same length and each column is the same length. In other words, multi-dimensional arrays are square. There is another form of “arrays of arrays” where each row is not the same length. We call these “jagged arrays.” Consider the following code:

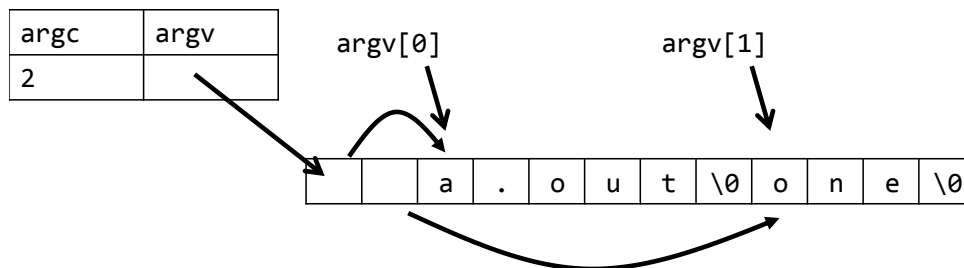
```
{
    char * name[] =
    {
        "CS 124",
        "Software development"
    };
}
```

The first string is a different size than the second. Thus we have:



In this example, we have declared an array of pointers to characters (`char *`). Each one of these pointers is pointing to a string literal. Observe how each string is of a different size. Also, the two strings do not need to be next to each other in memory as multi-dimensional arrays do.

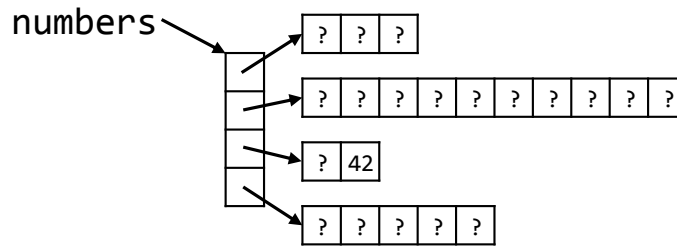
It turns out that `argv` works much like a pointer to dynamically allocated memory. The only difference is that the operating system typically puts each command line parameter immediately after the array of pointers in memory:



### Example 4.3 – Jagged Array of Numbers

Demo

This example will demonstrate how to create a jagged array of integers. In this case, each row will be allocated dynamically. There will be four rows, each containing a different number of items.



Solution

There are five steps to setting up an jagged array of numbers:

1. Allocate the array of pointers. This must happen before the rows are allocated.
2. Allocate each row individually. This will require separate new statements.
3. Use the array. In this case, the value '42' will be assigned to one cell.
4. Free the individual rows.
5. Free the array of pointers. This must happen after the individual rows are freed.

Note that if the number of rows is known at compile time then step 1 and 5 can be skipped.

```
{
    int ** numbers;                // pointer to a pointer to an integer

    // 1. allocate the array of arrays.
    numbers = new int *[4];        // an array of pointers to integers

    // 2. allocate each individual array
    numbers[0] = new int[3];       // an array of integers
    numbers[1] = new int[10];
    numbers[2] = new int[2];
    numbers[3] = new int[5];

    // 3. assign a value
    numbers[2][1] = 42;           // access is the same as with standard
                                // multi-dimensional array

    // 4. free each array
    for (int i = 0; i < 4; i++)    // we must free each individual array or we
        delete [] numbers[i];     // will forget about them and have a leak

    // 5. free the array of arrays
    delete [] numbers;           // finally free the original array
}
```

Challenge

As a challenge, change the code to work with arrays of floats. Make it work with five rows instead of four (making sure to free each row when finished). Finally, try to fill each item in the array with a number.

See Also

The complete solution is available at [4-3-jaggedArrayNumber.cpp](#) or:

/home/cs124/examples/4-3-jaggedArrayNumber.cpp



### Example 4.3 – Jagged Array of Strings

Demo

This example will demonstrate how to create a jagged array of strings. This allows us to use exactly the amount of memory required to represent the user's data.

Problem

Write a program to prompt the user for his name and store his name in a jagged array.

```
How man letters are there in your first, middle, and last name?  
4 5 20  
Please enter your first name: John  
Please enter your middle name: Jacob  
Please enter your last name: Jingleheimer Schmidt  
Your name is "John Jacob Jingleheimer Schmidt"
```

Solution

To allocate the jagged array, we need to know the size of each row (sizeFirst, sizeMiddle, sizeLast).

```
char ** names;  
names = new char *[3]; // three names  
names[0] = new char[sizeFirst + 1];  
names[1] = new char[sizeMiddle + 1];  
names[2] = new char[sizeLast + 1];
```

From here, it behaves like any other multi-dimensional array.

```
// fill the jagged array  
cout << "Please enter your first name: ";  
cin.getline(names[0], sizeFirst + 1);  
cout << "Please enter you middle name: ";  
cin.getline(names[1], sizeMiddle + 1);  
cout << "Please enter your last name: ";  
cin.getline(names[2], sizeLast + 1);  
  
// display the results  
cout << "Your name is \"  
    << names[0] << ' '  
    << names[1] << ' '  
    << names[2] << "\"\n";
```

Finally, it is important to free the rows and the pointers in the correct order.

```
delete [] names[0];  
delete [] names[1];  
delete [] names[2];  
delete [] names;
```

Challenge

As a challenge, can you add another row to correspond to the user's title ("Dr.")? Again, don't forget to allocate the row and free the row.

See Also

The complete solution is available at [4-3-jaggedArrayString.cpp](#) or:

```
/home/cs124/examples/4-3-jaggedArrayString.cpp
```



### Problem 1

Given the following program:

```
int main(int argc, char ** argv)
{
    cout << argv[0] << endl;
    return 0;
}
```

What is the output if the user runs the following command:

```
%a.out one two three
```

Answer:

---

*Please see page 348 for a hint.*

### Problem 2

Given the following program:

```
int main(int argc, char ** argv)
{
    <code goes here>
}
```

Write the code that will display the value “three”:

```
%a.out one two three
three
```

Answer:

---

*Please see page 348 for a hint.*

### Problem 3

Given the following program:

```
int main(int argc, char ** argv)
{
    cout << argc << endl;

    return 0;
}
```

What is the output if the user runs the following command:

```
%a.out one two
```

Answer:

---

*Please see page 348 for a hint.*

### Problem 4

Given the following program:

```
int main(int argc, char ** argv)
{
    cout << atoi(argv[1])
          + atoi(argv[2]);
    return 0;
}
```

What is the output if the user types in:

**a.out 4 5 6**

Answer:

*Please see page 348 for a hint.*

### Problem 5

Describe, in English, the functionality of the following program:

```
int main(int argc, char ** argv)
{
    while (--argc)
        cout << argv[argc] << endl;
    return 0;
}
```

Answer:

*Please see page 347 for a hint.*

### Problem 6

Write a program to compute the absolute value given numbers specified on the command line:

```
%a.out -4.2 96.2 -3.90
The absolute value of -4.2 is 4.2
The absolute value of 96.2 is 96.2
The absolute value of -3.90 is 3.9
```

Answer:

*Please see page 347 for a hint.*

## Assignment 4.3

Write a program to convert feet to meters. The conversion from feet to meters is:

1 foot = 0.3048 meters

The input will be numbers passed on the command line. You will want to use the library function `atof` to convert a string into a float. For example, consider the following code:

```
#include <cstdlib>    // the library for atof()
#include <iostream>
using namespace std;

int main()
{
    char text[] = "3.14159";    // a c-string
    float pi;                  // the float where the answer will go
    pi = atof(text);           // atof() translates the c-string into a float
    cout << pi << endl;        // this better be 3.14159

    return 0;
}
```

Pay special attention to the `#include <cstdlib>` code; you will need that for this assignment.

## Example

Consider the output of a program called `a.out`:

```
a.out 1 2 3
1.0 feet is 0.3 meters
2.0 feet is 0.6 meters
3.0 feet is 0.9 meters
```

## Assignment

The test bed is available at:

```
testBed cs124/assign43 assignment43.cpp
```

Don't forget to submit your assignment with the name "Assignment 43" in the header.

*Please see page 348 for a hint.*



## 4.4 Instrumentation

Sue has just finished her first draft of a function to solve a Sudoku puzzle. It seems fast, but she is unsure how fast it really is. How do you measure performance on a machine that can do billions of instructions per second? To get to the bottom of this, she introduces counters in key parts of her program to measure how many times certain operations are performed.

### Objectives

By the end of this class, you will be able to:

- Instrument a function to determine its performance characteristics.

### Prerequisites

Before reading this section, please make sure you are able to:

- Search for a value in an array (Chapter 3.1).
- Look up a value in an array (Chapter 3.1).
- Declare an array to solve a problem (Chapter 3.0).
- Write a loop to traverse an array (Chapter 3.0).
- Predict the output of a code fragment containing an array (Chapter 3.0).

### Overview

One of the most important characteristics of a sorting or search algorithm is how fast it accomplishes its task. There are several ways we can measure speed: the number of steps it takes, the number of times an expensive operation is performed, or the elapsed time.

To measure the number of steps the operation takes, we need to add a counter to the code. This counter will increment with every step, resulting in one metric of the speed of the algorithm. We call the process of adding a counter “**instrumenting**.” Instrumenting is the process of adding code to an existing function which does not change the functionality of the task being measured. Instead, the instrumenting code measures how the task was performed. For example, if consider the following code determining if two strings are equal:

```

/*****
 * IS EQUAL
 * Simple function to tell if two strings are equal
 *****/
bool isEqual(const char * text1, const char * text2)
{
    while (*text1 == *text2 && *text2)
    {
        text1++;
        text2++;
    }

    return *text1 == *text2;
}

```

This same code can be instrumented by adding a counter keeping track of the number of characters compared:

```
/* *****
 * IS EQUAL
 * Same function as above except it will keep track
 * of how many characters are looked at to determine
 * if the two strings are equal. This function is
 * instrumented
 * ***** */
bool isEqual(const char * text1, const char * text2)
{
    int numCompares = 1; // keeps track of the number of compares

    while (*text1 == *text2 && *text2)
    {
        text1++;
        text2++;
        numCompares++; // with every compare (in the while loop), add one!
    }

    cout << "It took " << numCompares << " compares\n";
    return *text1 == *text2;
}
```

Write a program to manage your personal finances for a month. This program will ask you for your budget income and expenditures, then for how much you actually made and spent. The program will then display a report of whether you are on target to meet your financial goals



### Sue's Tips

It is a good idea to put your instrumentation code in `#ifdefs` so you can easily remove it before sending the code to the customer. The most convenient way to do that is with the following construct:

```
#ifdef DEBUG
#define Debug(x) x
#else
#define Debug(x)
#endif
```

Observe how everything in the parentheses is included in the code if `DEBUG` is defined, while it is removed when it is not. With this code, we would say something like:

```
Debug(numCompare++);
```

Again, if `DEBUG` were defined, the `numCompare++;` would appear in the code. If it were not defined, then an empty semi-colon would appear instead.

## Example 4.4 – Instrument the Bubble Sort

### Demo

This example will demonstrate how to instrument a complex function: a sort algorithm. This algorithm will order the items in an array according to a given criterion. In this case, the criterion is to reorder the numbers in array to ascending order. The question is: how efficient is this algorithm?

Possibly the simplest algorithm to order a list of values is the **Bubble Sort**. This algorithm will first find the largest item and put it at the end of the list. Then it will find the second largest and put it in the second-to-last spot (`iSpot`) and so on.

We will instrument this function with the `numCompare` variable. Initially set to zero, we will increment `numCompare` each time a pair of numbers (`array[iCheck] > array[iCheck + 1]`) is compared. For convenience, we will return this value to the caller.

### Solution

```
/* *****  
 * BUBBLE SORT  
 * Instrumented version of the Bubble Sort. We will return the number  
 * of times elements in the array were compared.  
 * ***** */  
int bubbleSort(int array[], int numElements)  
{  
    // number of comparisons is initially zero  
    int numCompare = 0;  
  
    // did we switch two values on the last time through the outer loop?  
    bool switched = true;  
  
    // for each spot in the array, find the item that goes there with iCheck  
    for (int iSpot = numElements - 1; iSpot >= 1 && switched; iSpot--)  
        for (int iCheck = 0, switched = false; iCheck <= iSpot - 1; iCheck++)  
        {  
            numCompare++;           // each time we are going to compare, add one  
  
            if (array[iCheck] > array[iCheck + 1])  
            {  
                int temp = array[iCheck];    // swap 2 items if out of order  
                array[iCheck] = array[iCheck + 1];  
                array[iCheck + 1] = temp;  
                switched = true;             // a swap happened, do outer loop again  
            }  
        }  
  
    return numCompare;  
}
```

It is important to note that instrumenting should not alter the functionality of the program. Removing the instrumentation code should leave the algorithm unchanged.

### See Also

The complete solution is available at [4-4-bubbleSort.cpp](#) or:

```
/home/cs124/examples/4-4-bubbleSort.cpp
```



## Example 4.4 – Instrument Fibonacci

Demo

This example will demonstrate how to instrument two functions computing the Fibonacci sequence. It will not only demonstrate how to add counters at performance-critical locations in the code, but will demonstrate how to surround the instrumentation code with `#ifdefs` enabling the programmer to conveniently remove the instrumentation code or quickly add it back.

Problem

As you may recall, the Fibonacci sequence is defined as the following:

$$F(n) := \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

Write a program to compute the  $n^{\text{th}}$  Fibonacci number.

```
How many Fibonacci numbers shall we identify? 10
Array method: 55
Recursive method: 55
```

In debug mode (compiled with the `-DDEBUG` switch) , the program will display the cost:

```
How many Fibonacci numbers shall we identify? 10
Number of iterations for the array method: 9
Number of iterations for the recursive method: 177
```

Solution

In order to not influence the normal operation of the Fibonacci functions, the following code was added to the top of the program:

```
#ifdef DEBUG          // all the code that will only execute if DEBUG is defined
int countIterations = 0;
#define increment()    countIterations++
#define reset()        countIterations = 0
#define getIterations() countIterations

#else // !DEBUG
#define increment()
#define reset()
#define getIterations() 0
#endif
```

Thus in ship mode (when `DEBUG` is not defined), the “functions” `increment()`, `reset()`, and `getIterations()` are defined as nothing. In debug mode, these manipulate the global variable `countIterations`. Normally global variables are dangerous. However, since this variable is only visible when `DEBUG` is defined, its damage potential is contained. The last thing to do is to carefully put `reset()` before we start counting and `getIterations()` when we are done counting.

```
reset();
int valueArray = computeFibonacciArray(num);
int costArray  = getIterations();
```

This, coupled with `increment()` when counting constitutes all the instrumentation code in the program.

See Also

The complete solution is available at [4-4-fibonacci.cpp](#) or:

```
/home/cs124/examples/4-4-fibonacci.cpp
```



## Assignment 4.4

Our assignment this week is to determine the relative speed of a linear search versus a binary search using instrumentation. To do this, start with the code at:

```
/home/cs124/assignments/assign44.cpp
```

Here, you will need to modify the functions `binary()` and `linear()` to count the number of comparisons that are performed to find the target number.

Next, you will need to modify `computeAverageLinear()` and `computeAverageBinary()` to determine the number of compares on average it takes to find each element in the array.

### Example

Consider the file `numbers.txt` that has the following values:

```
1 4 10 36 47 92 100 110 125 136 142 143 150 160 167
```

For the above example, the list is:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	4	10	36	47	92	100	110	125	136	142	143	150	160	167

When we run the program on the above list, the program will compute how long it takes to find an element in the list using the linear search method and using the binary search method. If I call `linear()` (a function performing a linear search from left to right) with the search parameter set to 4, then I will find it with two comparisons. This means `linear(list, num, 4) == 2` because the function `linear()` will return the number of comparisons, `list` contains the list of numbers, and `num` is the number of items in `list`. Thus `linear(list, num, 47) == 5`. To find the average cost of the linear search, the equation will be:

$$\begin{aligned} & (\text{linear}(\text{list}, \text{num}, 1) + \text{linear}(\text{list}, \text{num}, 4) + \dots + \text{linear}(\text{list}, \text{num}, 167)) / \text{num} == \\ & \quad (1 + 2 + \dots + 15) / 15 == \\ & \quad 120 / 15 == \\ & \quad 8.0 \end{aligned}$$

To compute the average cost of the binary search, the equation will be:

$$(\text{binary}(\text{list}, \text{num}, 1) + \text{binary}(\text{list}, \text{num}, 4) + \dots + \text{binary}(\text{list}, \text{num}, 167)) / \text{num} == 3.3$$

The user input is underlined.

```
Enter filename of list: numbers.txt
Linear search:      8.0
Binary search:     3.3
```

### Assignment

The test bed is available at:

```
testBed cs124/assign44 assignment44.cpp
```

Don't forget to submit your assignment with the name "Assignment 44" in the header.

*Please see page 358 for a hint.*

## Unit 4 Practice Test

### Practice 4.2

Sam has read that many programmers get paid by the number of lines of code they generate. Wanting to maximize the size of his paycheck, he has decided to write a program to count the number of lines in a given file.

### Problem

Write a program to perform the following tasks:

1. Attempt to gather the filename from the command line
2. If no filename is present, prompt the user for the filename
3. Loop through the file, counting the number of lines
4. Display the results using the correct tense for empty, 1, and more than one.

### Example

The user may specify the filename from the command line. Assume the file “example1.txt” has 2 lines:

```
a.out example1.txt
example1.txt has 2 lines.
```

If no file is specified on the command line, then the program will prompt the user for a file. Assume the file “example2.txt” has 1 line of text:

```
a.out
Please enter the file name: example2.txt
example2.txt has 1 line.
```

Note how “line” is singular. Finally, if the file is empty or non-existent, then the following message will appear:

```
a.out missing.txt
missing.txt is empty.
```

### Grading

The test bed is:

```
testBed cs124/practice42 practice42.cpp
```

The sample solution is:

```
/home/cs124/tests/practice42.cpp
```

*Continued on the next page*

## Grading for Test4

Sample grading criteria:

	Exceptional 100%	Good 90%	Acceptable 70%	Developing 50%	Missing 0%
Modularization 10%	Perfect cohesion and coupling	No bugs exist in calling functions, though modularization is not perfect	Redundant data passed between functions	A bug exists passing parameters or calling a function	Only one function used
Command line 10%	Input logic "elegant"	Able to access the filename from the command line and from a prompt	One bug	An attempt was made to accept input from both the command line and from prompts.	No attempt was made to access the command line parameters
File 20%	"Good" and perfect error detection	Able to fetch all the text from a file	One bug	Elements of the solution exist	No attempt was made to read from the file
Problem Solving 30%	The most elegant and efficient solution was found	Zero test bed errors	Correct on at least one test case	Elements of the solution are present	No attempt was made to count all the words <u>or</u> program failed to compile
Handle tense on output 10%	The most elegant and efficient solution was found	All three cases handled	One bug	Logic exists to attempt to detect need for different tenses	No attempt was made to handle tense
Programming Style 20%	Well commented, meaningful variable names, effective use of blank lines	Zero style checker errors	One or two minor style checker errors	Code is readable, but serious style infractions	No evidence of the principles of "elements of style" in the program

# Unit 4 Project : Sudoku

Write a program to allow the user to play Sudoku. For details on the rules of the game, see:

<http://en.wikipedia.org/wiki/Sudoku>

The program will prompt the user for the filename of the game he or she is currently working on and display the board on the screen. The user will then be allowed to interact with the game by selecting which square he or she wishes to change. While the program will not solve the game for the user, it will ensure that the user has not selected an invalid number. If the user types 's' in the prompt, then the program will show the user the possible valid numbers for a given square. When the user is finished, then the program will save the board to a given filename and exit.

This project will be done in three phases:

- Project 11 : Design the Sudoku program
- Project 12 : Allow the user to interact with the Sudoku game
- Project 13 : Enforce the rules of the Sudoku game

## Interface Design

Consider a game saved as `myGame.txt`:

```
7 2 3 0 0 1 5 9
6 0 0 3 0 2 0 0 8
8 0 0 0 1 0 0 0 2
0 7 0 6 5 4 0 2 0
0 0 4 2 0 7 3 0 0
0 5 0 9 3 1 0 4 0
5 0 0 0 7 0 0 0 3
4 0 0 1 0 3 0 0 6
9 3 2 0 0 0 7 1 4
```

Note that '0' corresponds to an unknown value. The following is an example run of the program. Please see the following program for an example of how this works:

```
/home/cs124/projects/prj13.out
```

### Program starts

An example of input is underlined.

```
Where is your board located? myGame.txt
```

With the filename specified, the program will display a menu of options:

```
Options:
? Show these instructions
D Display the board
E Edit one square
S Show the possible values for a square
Q Save and quit
```

After this, the board as read from the file will be displayed:



	A	B	C	D	E	F	G	H	I
1	7	2	3				1	5	9
2	6			3	2				8
3	8				1				2
-----+-----+-----									
4		7		6	5	4			2
5			4	2		7	3		
6		5		9	3	1			4
-----+-----+-----									
7	5				7				3
8	4			1		3			6
9	9	3	2				7	1	4

Here, the user will be prompted for a command (the main prompt).

```
> Z
```

Please note that you will need a newline, a carat ('>'), and a space before the prompt.

The next action of the program will depend on the user's command. If the user types an invalid command, then the following message will be displayed:

```
ERROR: Invalid command
```

## Show Instructions

If the user types '?', then the menu of options will be displayed again. These are the same instructions that are displayed when the program is first run.

## Display the Board

If the user types 'd', then the board will be redrawn. This is the same as the drawing of the board when the program is first run.

## Save and Quit

If the user types 'q', then he or she will be prompted for the filename:

```
What file would you like to write your board to: newGame.txt
```

The program will display the following message:

```
Board written successfully
```

Then the program will terminate.

## Edit One Square

If the user types 'e', then the program will prompt him for the coordinates and the value for the square to be edited:

```
What are the coordinates of the square: A1
```

If the value is invalid or if the square is already filled, then the program will display one of the following message and return to the main prompt:

```
ERROR: Square 'zz' is invalid
ERROR: Square 'A1' is filled
```

With a valid coordinate, then the program next prompts the user for the value:

```
What is the value at 'A1': 9
```

If the user types a value that is outside the accepted range ( $1 \leq \text{value} \leq 9$ ) or does not follow the rules of Sudoku, then a message appears and the program returns to the main prompt:

```
ERROR: Value '9' in square 'A1' is invalid
```

### Show Possible Values

If the user types 's', then the program will prompt him for the coordinates and display the possible values:

```
What are the coordinates of the square: A1
```

The same parsing logic applies here as for the Edit One Square case. Once the user has selected a valid coordinate, then the program will display the possible values:

```
The possible values for 'A1' are: 1, 5, 8, 9
```

After the message appears, the program returns to the main prompt.

# Project 11

---

The first part of the project is the design document. This consists of three parts:

1. Create a structure chart describing the entire Sudoku program.
2. Write the pseudocode for the function `computeValues()`. This function will take as parameters coordinates (row and column) for a given square on the board and calculate the possible values for the square. To do this, `computeValues()` must be aware of all the rules of Sudoku. Make sure to include both the logic for the rules of the game (only one of each number on a row, column, and inside square), but also to display the values.
3. Write the pseudocode for the function `interact()`. This function will prompt the user for his option (ex: 'D' for "Display the board" or 'S' for "Show the possible values for a square") and call the appropriate function to carry out the user's request. Note that the program will continue to play the game until the user has indicated he is finished (with the 'Q' option).

On campus students are required to attach [this rubric](#) to your design document. Please self-grade.

## Project 12

The second part of the Sudoku Program project (the first part being the design document due earlier) is to write the code necessary to make the Sudoku appear on the screen:

```
Where is your board located? prj4.txt
Options:
? Show these instructions
D Display the board
E Edit one square
S Show the possible values for a square
Q Save and Quit

  A B C D E F G H I
1  7 2 3 |   | 1 5 9
2  6   | 3  2 |   8
3  8   |   1 |   2
  ----+----+----
4   7  | 6 5 4 |  2
5     4 | 2  7 |  3
6   5  | 9 3 1 |  4
  ----+----+----
7  5   |   7 |   3
8  4   |  1  3 |   6
9  9 3 2 |   | 7 1 4

> E
What are the coordinates of the square: e5
What is the value at 'E5': 8

> q
What file would you like to write your board to: deleteMe.txt
```

Perhaps the easiest way to do this is in a four-step process:

1. Create the framework for the program using stub functions based on the structure chart from your design document.
2. Write each function. Test them individually before "hooking them up" to the rest of the program.
3. Verify your solution with testBed:

```
testBed cs124/project12 project12.cpp
```

4. Submit it with "Project 12, Sudoku" in the program header.

An executable version of the project is available at:

```
/home/cs124/projects/prj12.out
```

## Project 13

The final part of the Sudoku Program project is to enforce the rules of Sudoku. This means that there can be only one instance of a given number on a row, column, or inside square.

```
Where is your board located? prj4.txt
Options:
? Show these instructions
D Display the board
E Edit one square
S Show the possible values for a square
Q Save and Quit

  A B C D E F G H I
1  7 2 3|   | 1 5 9
2  6   |3  2|   8
3  8   |  1 |   2
  ----+----+----
4    7 |6 5 4|  2
5     4|2  7|3
6    5 |9 3 1|  4
  ----+----+----
7  5   |  7 |   3
8  4   |1  3|   6
9  9 3 2|   |7 1 4

> s
What are the coordinates of the square: b2
The possible values for 'B2' are: 1, 4, 9

> e
What are the coordinates of the square: b2
What is the value at 'B2': 2
ERROR: Value '2' in square 'B2' is invalid

> q
What file would you like to write your board to: deleteMe.txt
```

Perhaps the easiest way to do this is in a four-step process:

1. Start with the code written in Project 12.
2. Fix any necessary bugs.
3. Verify your solution with testBed:

```
testBed cs124/project13 project13.cpp
```

4. Submit it with "Project 13, Sudoku" in the program header.

An executable version of the project is available at:

```
/home/cs124/projects/prj13.out
```