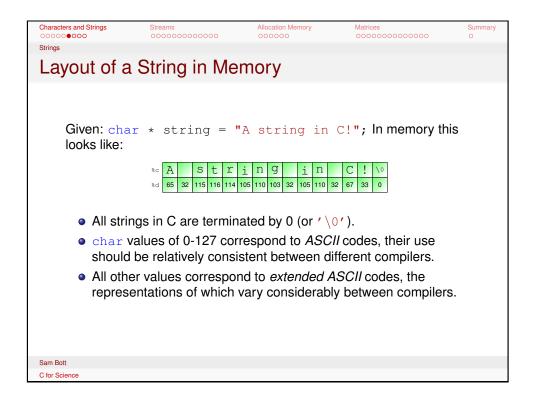
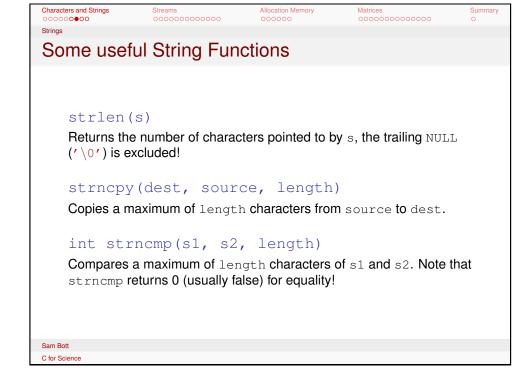


```
Characters and Strings
                                        Allocation Memory
                                                         Matrices
                                                                              Summary
000000000
Demo of char
     1 #include <stdio.h>
     3 int main()
     4 {
           char * name = "Grace";
          printf(name); /* not recommended, but allowed*/
          printf("\nname = %s\n", name);
          printf("name[0] = c = dn, name[0], name[0]);
    10 }
    Gives the following output:
    Grace
    name
                = Grace
    name[0] = G = 71
Sam Bott
C for Science
```



```
Characters and Strings
                  Streams
                                    Allocation Memory
                                                     Matrices
000000000
Some useful char Functions
    For chars
     isalpha(c)
                        True (non-zero) if c is from A-Z, a-z
                       True if c if from 0-9
     isdigit(c)
     isalnum(c)
                       =(isalpha(c) || isdigit(c))
                        True if c is from a-z
     islower(c)
                       True if c is from A-Z
     isupper(c)
                       Convert to lowercase (if isupper(c)), oth-
     d=tolower(c)
                        erwise it returns c
                        Convert to uppercase (if islower (c)), oth-
     d=toupper(c)
                        erwise it returns c
Sam Bott
C for Science
```



```
Characters and Strings
                                                       Allocation Memory
                                                                               Matrices
000000000
A String Demo
        1 #include <stdio.h>
        2 #include <string.h>
        3 #include <stdlib.h>
        4 #include <ctvpe.h>
        6 int main()
            unsigned int loop:
            char * string = "A string in C!", * copy;
            printf("strlen(string) = %d\n", strlen(string));
            copy = (char *) calloc(strlen(string)+1, 1);
            if (!copy)
               fprintf(stderr, "Couldn't allocate buffer!\n");
               return -1;
      16
      18
            strncpy(copy, string, strlen(string));
            printf("strncmp(string, copy) = %d\n",
                  strncmp(string, copy, strlen(string)));
            for (loop = 0; loop < strlen(copy); loop++)
               copy[loop] = toupper(copy[loop]);
            printf("modified copy = \"%s\"\n", copy);
            printf("strncmp(string, copy) = %d\n'
                  strncmp(string, copy, strlen(string)));
            return 0;
Sam Bott
C for Science
```

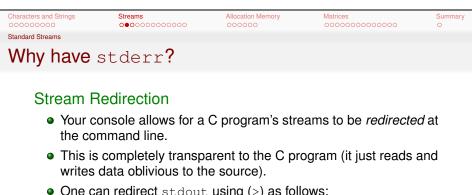
Allocation Memory Matrices Streams •00000000000 Input and Output in C - Streams All I/O in C is accomplished via file streams. • This stems from C's UNIX roots (every device is a file). • We have already seen printf, formatted output to console. • C90 defines three streams which are initialised by default when a program starts. Input from the keyboard. (read only) stdin Text console. (write only) stdout stderr Another text console. (write only) Sam Bott

C for Science

```
Characters and Strings
                                  Allocation Memory
                                                  Matrices
00000000
Results from String Demo
    The program on the previous slide gives the following output:
    strlen(string) = 14
    strncmp(string, copy) = 0
    modified copy = "A STRING IN C!"
    strncmp(string, copy) = 32
```

- Note that strncmp returns 0 for equality and non-zero otherwise.
- Case insensitive string comparisons can be made using: strnicmp.

Sam Bott C for Science



One can redirect stdout using (>) as follows: myprogram > output.txt • stderr can be redirected using (2>): myprogram 2> errorlog.txt stdin can be redirected using (<):</p> myprogram < input.txt</pre> Having a separate *error stream*, allows for important messages to be filtered from data output. Sam Bott C for Science

Standard Streams

printf and scanf - formatted output and input

These are automatically connected to stdout and stdin respectively. The more generalised functions are fprintf and fscanf. As an example:

```
fprintf(stdout, "Text to stdout...\n");
fprintf(stderr, "Text to stderr...\n");
```

We can also read text from a stream:

```
fscanf(stdin, "%lf", &ptrDouble);
```

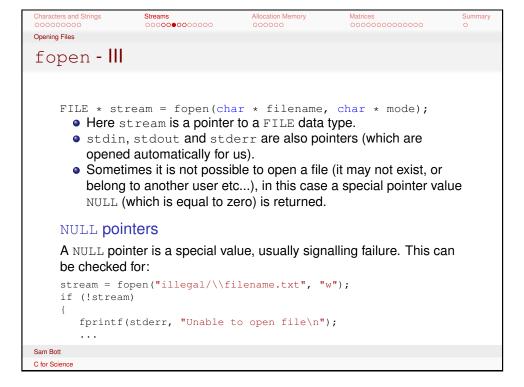
Additional streams can be opened using fopen.

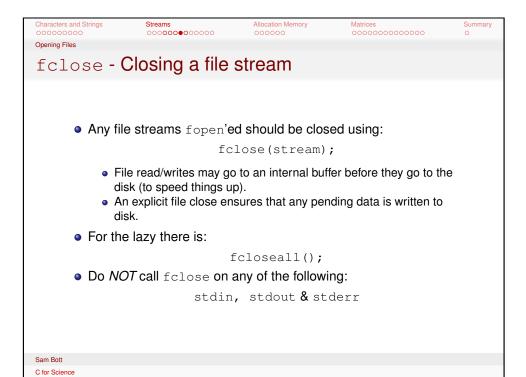
Sam Bott

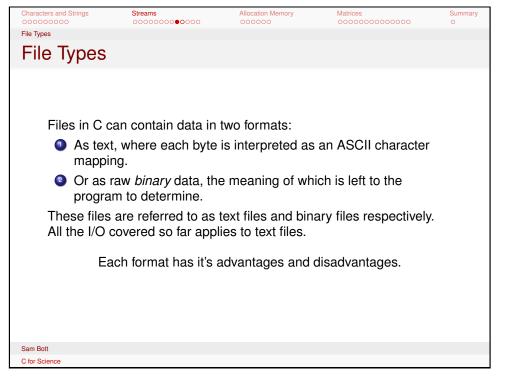
C for Science

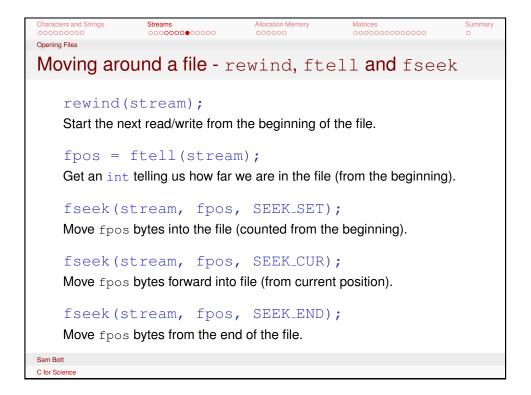
```
Allocation Memory
                                                    Matrices
                  Streams
                  0000000000000
Opening Files
fopen - |
    FILE * stream = fopen(char * filename, char * mode);
    filename
    Must be a legal operating system file name. A safe option would be
    something similar to:
                             "results.txt"
    Absolute Filenames
    We can specify a full directory path to a filename:
    data = fopen("C:\\TEMP\\mydata.dat", "w"); /* Windows */
    data = fopen("/tmp/mydata.dat", "w"); /* UNIX */
    Relative Filenames
    It is much safer to omit the full path:
    data = fopen("mydata.dat", "w"); /* works on most os' */
Sam Bott
C for Science
```

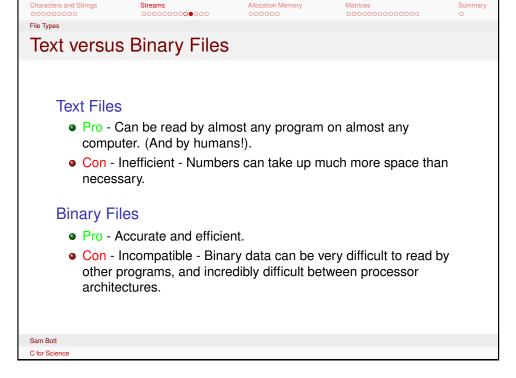
```
Allocation Memory
                      Streams
                      0000000000000
Opening Files
fopen - open a file stream
     FILE * stream = fopen(char * filename, char * mode);
     Working backwards, mode is a string telling C how to open the file:
        mode
                meaning
        "r"
                Open filename for reading. The file must exist or NULL is returned.
                Open filename for writing, starting from the beginning of the file. The
                 file will be created if it doesn't already exist. Any old data will be over-
                Open filename for writing, starting from the end of the file. File will be
                 appended if it does exist and created otherwise.
        "r+" Open filename for reading and writing, starting from the beginning. If
                the file doesn't exist, NULL is returned.
                Open filename for reading and writing, starting from the beginning. If
                the file doesn't exist it's created.
        "a+" Open filename for reading and writing (append if exists).
Sam Bott
C for Science
```











```
Characters and Strings

OCOCCOCCC

Binary Files

Variable sizes

• Different data types take up different amounts of memory.

• Example float is smaller than double.

• The sizeof keyword gives a type's size (in bytes).

#include <stdio.h>

int main()

{
    printf("sizeof(float) = %d\n", sizeof(float));
    printf("sizeof(int) = %d\n", sizeof(int));
    printf("sizeof(short) = %d\n", sizeof(short));
    return 0;
}

Sam Bott

C for Science
```

```
Temporary Files

Permanent versus Temporary Files

Sometimes we may not wish for a file to be permanent.

Temporary files provide a means for storing intermediate computations.

We open a temporary file using the tmpfile() function.

Example

stream = tmpfile();

if (!stream)

{
/* handle failure */
}
/* fread, fwrite, fprintf, fscanf */
fclose(stream);
```

Allocation Memory

Streams

Matrices

```
Characters and Strings

Occossion

Note: Summary Streams
Occossion

Note: Summary Files

Handling Binary Files

Binary files are opened using fopen, we need to add "b" to the file mode though:

stream = fopen("data.dat", "rb");

Data is read from and written to a binary file using fread and fwrite. For example:

double values[10] = {0.0};

/* we have "stream" an open binary file */
read = fread(values, sizeof(double), 10, stream);
if (read != 10)
{ read failed }

Please see < stdio.h > for more details.
```



Dynamic Allocation

The heap

C for Science

- C has set aside a special block of memory known as the *heap*.
- Memory can be requested from the heap.
- If memory is available, it is provided, otherwise a request is made to the operating system.
- The implementation details of the heap are "undefined".

sizeof

- We need to know how much memory to request.
- The sizeof keyword returns the size (in bytes), of a variable or data type.

Sam Bott

```
Allocation Memory
                                        000000
Borrowing From the Heap
How to Borrow Memory
    We use the malloc function (defined in <stdlib.h>) to request
    memory and free to return it to the heap.
     1 #include <stdio.h>
     2 #include <stdlib.h>
     3 #define NCOUNT 10
     4 int main()
     5 {
           int * numbers, loop;
           numbers = (int *) malloc(sizeof(int)*NCOUNT);
     9
              fprintf(stderr, "Unable to allocate memory\n");
    11
    13
           for (loop = 0; loop < NCOUNT; loop++)</pre>
    14
              numbers[loop] = loop;
    16
           for (loop = 0; loop < NCOUNT; loop++)</pre>
              printf("numbers[%d] = %d\n", loop, numbers[loop]);
    18
    19
           free (numbers);
    20
           return 0;
    21 }
Sam Bott
C for Science
```

How to Borrow Memory - More Details

Two more, useful memory management functions are:

realloc Modifies the size of a memory block. calloc Allocates memory, and sets every byte to 0.

Some common heap problems

- Allocating memory is slow and should be done as few times as possible.
- Memory allocation can fail, every malloc/calloc should be checked (or at the very least asserted).
- Every malloc/calloc should have a corresponding free, memory is *leaked* if it's not freed after use.

Sam Bott C for Science

- Any pointer type can be implicitly casted to a void *.
- \bullet A value of ${\tt NULL}$ usually means that a function encountered an error.

What is size t?

Streams

size_t is defined to be an unsigned integer type large enough to hold numbers returned by sizeof.

Allocation Memory

Matrices

Sam Bott C for Science

```
000000
Borrowing From the Heap
1-D Example: Fibonacci (again!)
       1 #include <stdlib.h>
      2 #include <stdio.h>
      4 int main()
           double * fibs:
           unsigned int nfibs=0, loop;
           while (nfibs < 2)
             printf("How many Fibonacci numbers are needed (>1)?\n");
              scanf("%u", &nfibs);
     14
           fibs = (double *) malloc(sizeof(double)*nfibs);
           if (!fibs) /* malloc failed? */
              fprintf(stderr, "Unable to allocate memory!\n");
              return -1;
           fibs[0] = 1.0; fibs[1] = 1.0;
          for (loop = 2; loop < nfibs; loop++)
             fibs[loop] = fibs[loop-1]+fibs[loop-2];
          for (loop = 0; loop < nfibs; loop++)
             printf("fib[%u] = %lg\n", loop, fibs[loop]);
           free(fibs);
           return 0;
      30 }
Sam Bott
C for Science
```

Pointers to Pointers: **ptr

- Pointers in C are very flexible, to the extent that we can form pointers to pointers!
- This is known more formally as the "level of indirection".
- Tensors of arbitrary rank (Fortran 95 is limited to 7) can be formed easily in C.
- The most useful example in C being matrices.

Matrices in C

Matrices in C are commonly represented by the double ** type. This means "a pointer to a pointer to a double".

Sam Bott

C for Science

Allocation Memory Matrices 00000000000000 **Allowed Pointer Operations**

Declaration: double * pA, * pB;

Assignment: pA = &var;Increment: pA = pA + 1;Decrement: pA = pA - 1;Difference: qap = pA - pB;Comparison: if(pA == pB)De-referencing: *pA = val;

Sam Bott C for Science

Streams

Pointers and Arrays

We saw the connection between arrays and pointers in the last lecture. Given the array:

double ar
$$[4] = \{1.0, 2.0, 3.0, 4.0\};$$

Allocation Memory

•000000000000

The elements of this array can be accessed as follows:

```
*(ar) == ar[0] == 1.0
*(ar+1) == ar[1] == 2.0
*(ar+2) == ar[2] == 3.0
```

Pointer referencing is also supported:

```
ar == &ar[0]
(ar+1) == &ar[1]
(ar+2) == &ar[2]
```

Sam Bott

C for Science

Streams Allocation Memory Matrices

Fixed Size Two-Dimensional Arrays

We can declare arrays of dimension higher than one, as follows:

```
double a[2][3] = \{\{1.0, 2.0, 3.0\},
                    \{2.0, 3.0, 4.0\}\};
```

Where the elements of a are denoted as expected:

a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]

To access the top left element:

```
myVal = a[0][0]; /* equal to 1.0 */
```

Sam Bott

C for Science

```
Allocation Memory
                                                      Matrices
                                                      0000000000000
Fixed Sized Matrices
Fixed Size Two-Dimensional Arrays II
     1 #include <stdio.h>
     2 #define COLS 3
     4 void printArray(int matrix[][COLS], int rows)
     5 {
     6
           int i, j;
           for (i = 0; i < rows; i++)
     9
              for (j = 0; j < COLS; j++)
    10
                 printf("%d ", matrix[i][j]);
    11
              printf("\n");
    12
    13 }
    14
    15 int main()
    16 {
    17
           int matrix[2][COLS] = \{\{1, 2, 3\}, \{4, 5, 6\}\};
    18
           printArray(matrix, 2);
    19
           return 0;
    20 }
Sam Bott
C for Science
```

```
Allocation Memory
                                                      Matrices
                                                                          Summary
                                                      000000000000000
Dynamically Allocating Matrices
Constructing Matrices with Pointers
     1 double ** makeMatrix(unsigned int rows, unsigned int cols)
     2 {
     3
           unsigned int i;
           double ** matrix;
           matrix = (double **) malloc(rows * sizeof(double *));
           if (!matrix) return NULL; /* failed */
     8
     9
           for (i = 0; i < rows; i++)</pre>
    10
    11
              matrix[i] = (double *) malloc(cols*sizeof(double));
    12
              if (!matrix[i])
    1.3
                  return NULL; /* lazy, we should really free
    14
                                   all the memory allocated above */
    15
    16
    17
           return matrix;
    18
Sam Bott
C for Science
```

Characters and Strings Streams Allocation Memory Matrices Summary

○○○○○○○

○○○○○○○○○○○

Fixed Sized Matrices

Fixed Size Two-Dimensional Arrays III

We have seen an example of a two dimensional array:

a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]

• In memory it is arranged as follows:

```
      a[0][0]
      a[0][1]
      a[0][2]
      a[1][0]
      a[1][1]
      a[1][2]

      (i.e. as a one dimensional array).
```

- Fixed size arrays are very inflexible as they require the dimensions to be "hard coded".
- They are allocated from the stack thus large arrays may cause problems.
- Dynamically allocated arrays overcome these restrictions.

Sam Bott C for Science

```
Characters and Strings Occiooco Streams Allocation Memory Occiooco Occiooc
```

Accessing Matrix Elements

Usage pattern for makeMatrix

```
double ** matrix = makeMatrix(rows, cols);
for (i=0; i < rows; i++)
   for (j=0; j < cols; j++)
      matrix[i][j] = 0.0;
free the matrix</pre>
```

 Accessing the dynamically allocated array looks identical to the fixed size ones, but "under the hood" things are a little different:

```
matrix[row][col] = *(*(matrix + row) + col)
```

 The makeMatrix code on the previous slide contained a lot of malloc statements, is there a better way to allocate a matrix? (yes!)

Sam Bott C for Science

```
Characters and Strings
                                          Allocation Memory
                                                            00000000000000
Dynamically Allocating Matrices
A Better Way of Allocating Matrices
      1 double ** allocMatrix(unsigned int rows, unsigned int cols)
           double ** matrix:
           unsigned int i;
           matrix = (double **) malloc (rows*sizeof(double *));
           if (!matrix) return NULL; /* failed */
     9
           matrix[0] = (double *) malloc (rows*cols*sizeof(double));
           if (!matrix[0])
     11
               free (matrix); /* we don't need matrix any more */
               return NULL; /* failed */
     14
           for (i = 1; i < rows; i++)</pre>
     16
     17
              matrix[i] = matrix[i-1] + cols;
     18
     19
           return matrix;
     20 }
Sam Bott
```

 Characters and Strings
 Streams
 Allocation Memory
 Matrices
 Summary

 00000000
 0000000000
 0000000
 00000000
 0

orking with Matrices

C for Science

Case Study: Matrix Addition

Let's define some utility functions to:

- Allocate memory for the matrix (allocMatrix) done,
- Free a matrix (freeMatrix) done,
- Print a matrix (printMatrix),
- Oreate a random matrix (randomMatrix),
- Add matrices together (addMatrices)

We drive all these functions using a ${\tt main}$ function.

Sam Bott C for Science

- allocMatrix only uses 2 mallocs whilst, makeMatrix uses cols + 1.
- Meaning there are fewer points of failure (we only check two pointers for NULL),
- It is much easier to free a matrix allocated with the allocMatrix function, all we need to do is:

```
void freeMatrix(double ** matrix)
{
   free(matrix[0]);
   free(matrix);
}
```

Sam Bott

C for Science

```
Streams
                                        Allocation Memory
                                                         Matrices
                                                         000000000000000
printMatrix and randomMatrix
     1 void printMatrix(double ** matrix, unsigned int rows,
                                           unsigned int cols)
          unsigned int i, j;
          for (i = 0; i < rows; i++)
              for (j = 0; j < cols; j++)
                 printf("%8.51f ", matrix[i][j]);
              printf("\n");
    11
    12 }
     1 void randomMatrix(double ** matrix, unsigned int rows,
                                            unsigned int cols)
     3 {
          unsigned int i, j;
           for (i = 0; i < rows; i++)
             for (j = 0; j < cols; j++)
                matrix[i][j] = (double) rand() / RAND_MAX;
     8 }
Sam Bott
C for Science
```

```
Allocation Memory
                                                              000000000000000
Working with Matrices
addMatrices
      1 void addMatrices(double ** matrixA, double ** matrixB,
                          double ** matrixR,
                           unsigned int rows, unsigned int cols)
            unsigned int i, j;
           for (i = 0; i < rows; i++)</pre>
              for (j = 0; j < rows; j++)
                  matrixR[i][j] = matrixA[i][j]+matrixB[i][j];
      9 }
Sam Bott
C for Science
```

```
Allocation Memory
                                                        Matrices
                                                                             Summary
                                                        00000000000000
Working with Matrices
Results
    Enter rows cols: 4 4
    matrix A =
     0.00125 0.56359 0.19330 0.80874
     0.58501 0.47987 0.35029 0.89596
     0.82284 0.74660 0.17411 0.85894
     0.71050 0.51353 0.30399 0.01498
    matrixB =
     0.09140 0.36445 0.14731 0.16590
     0.98853 0.44569 0.11908 0.00467
     0.00891 0.37788 0.53166 0.57118
     0.60176 0.60717 0.16623 0.66305
    matrixA + matrixB =
     0.09265 0.92804 0.34062 0.97464
     1.57353 0.92557 0.46937 0.90063
     0.83175 1.12449 0.70577 1.43013
     1.31227 1.12070 0.47023 0.67803
Sam Bott
C for Science
```

```
Allocation Memory
                                                                             00000000000000
Working with Matrices
The main function
       1 int main()
           unsigned int rows, cols;
           double ** matrivA ** matrivB **matrivC.
           printf("Enter rows cols: ");
           scanf ("%u %u", &rows, &cols);
           matrixA = allocMatrix(rows, cols);
           matrixB = allocMatrix(rows, cols);
           matrixC = allocMatrix(rows, cols);
           if (!matrixA || !matrixB || !matrixC)
           { /* a little lazy, but it does the job */
              fprintf(stderr, "Unable to allocate matrices!\n");
              return -1:
           randomMatrix(matrixA, rows, cols); randomMatrix(matrixB, rows, cols);
           addMatrices(matrixA, matrixB, matrixC, rows, cols);
           printf("\n\nmatrix A = \n");
           printMatrix(matrixA, rows, cols);
           printf("\n\nmatrixB = \n");
           printMatrix(matrixB, rows, cols);
           printf("\n\nmatrixA + matrixB = \n");
           printMatrix(matrixC, rows, cols);
           freeMatrix(matrixC); freeMatrix(matrixB); freeMatrix(matrixA);
           return 0;
Sam Bott
```



Summary

C for Science

- A char is the smallest addressable data type (1 byte). It can be used to represent a text character and a small number (-128 to 127) interchangeably.
- In addition to the three default streams (stdin, stderr, stdout), you can open and interact with file streams by opening them with fopen.
- Memory can be allocated dynamically using malloc. This is useful for arrays or matrices of an unknown, varying or large sizes.
- Considering the structure of the data in memory (and using the examples here) enables you to allocate elaborate data structures more efficiently.
- free is used to release previously allocated memory back to the stack; forgetting to 'free' memory causes memory-leaks.

Sam Bott C for Science