Computing in C for Science

Lecture 3 of 5

Dr. Steven Capper
steven.capper99@imperial.ac.uk
http://www2.imperial.ac.uk/~sdc99/ccourse/

30th November 2011

Imperial College London

Common Problems

1. Misuse of Power!
   x^2 should always be written `x*x`, *NOT* `pow(x, 2.0)`.
   $\sqrt{x}$ should be written `sqrt(x)`, *NOT* `pow(x, 0.5)`.
2. Integer Division
   Remember a fraction such as `1/3` is equal to zero. To get a floating point fraction, this should be rewritten `1.0/3.0`.

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.

`stdin`	Input from the keyboard. (read only)
`stdout`	Text console. (write only)
`stderr`	Another text console. (write only)

Why have `stderr`?

Stream Redirection

- UNIX (and now most other operating systems) allow for a C program's streams to be *redirected* at the command line.
- This is completely transparent to the C program (it just reads and writes data oblivious to the source).
- 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 (<):

```
myprogram < input.txt
```

Having a separate *error stream*, allows for important messages to be filtered from data output.

`printf` and `scanf` - formatted output and input

These are automatically connected to `stdout` and `stderr` 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`.

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.
`"w"`	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 overwritten.
`"a"`	Open `filename` for writing, starting from the end of the file. File will be *appended* if it doesn't exist and created otherwise.
`"r+"`	Open `filename` for reading and writing, starting from the beginning. If the file doesn't exist, `NULL` is returned.
`"w+"`	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).

fopen - II

```
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' */
```

fopen - III

```
FILE * stream = fopen(char * filename, char * mode);
```

- Here `stream` is a pointer to a `FILE` data type.
- `stdin`, `stdout` and `stderr` are also pointers (which are opened automatically for us).
- Sometimes it is not possible to open a file (it may not exist, or belong to another user etc...), in this case a special pointer value `NULL` (which is equal to zero) is returned.

NULL pointers

A `NULL` pointer is a special value, usually signalling failure. This can be checked for:

```
stream = fopen("illegal/\\filename.txt", "w");
if (!stream)
{
   fprintf(stderr, "Unable to open file\n");
   ...
```

`fclose` - Closing a file stream

- Any file streams `fopen`'ed should be closed using:

  ```
  fclose(stream);
  ```

  - File read/writes may go to an internal buffer before they go to the disk (to speed things up).
  - An explicit file close ensures that any pending data is written to disk.
- For the lazy there is:

  ```
  fcloseall();
  ```
- Do *NOT* call `fclose` on any of the following:

  `stdin, stdout` & `stderr`

Moving around a file - `rewind`, `ftell` and `fseek`

```
rewind(stream);
```

Start the next read/write from the beginning of the file.

```
fpos = ftell(stream);
```

Get an `int` telling us how far we are in the file (from the beginning).

```
fseek(stream, fpos, SEEK_SET);
```

Move `fpos` bytes into the file (counted from the beginning).

```
fseek(stream, fpos, SEEK_CUR);
```

Move `fpos` bytes forward into file (from current position).

```
fseek(stream, fpos, SEEK_END);
```

Move `fpos` bytes from the end of the file.

File Types

Files in C can contain data in two formats:

1. As text, where each byte is interpreted as an ASCII character mapping.
2. Or as raw *binary* data, the meaning of which is left to the program to determine.

These files are referred to as text files and binary files respectively. All the I/O covered so far applies to text files.

Each format has it's advantages and disadvantages.

Text versus Binary Files

Text Files

- Pro - Can be read by almost any program on almost any computer. (And by humans!).
- Con - Inefficient - Numbers can take up as much as twice the space as necessary.

Binary Files

- Pro - Accurate and efficient.
- Con - Incompatible - Binary data can be very difficult to read by other programs, and incredibly difficult between processor architectures.

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.

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);
```

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};
```

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]
```

Allowed Pointer Operations

Declaration:	`double * pA, * pB;`
Assignment:	`pA = &var;`
Increment:	`pA = pA + 1;`
Decrement:	`pA = pA - 1;`
Difference:	`gap = pA - pB;`
Comparison:	`if(pA == pB)`
De-referencing:	`*pA = val;`

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 */
```

Fixed Size Two-Dimensional Arrays II

```c
#include <stdio.h>
#define COLS 3

void printArray(int matrix[][COLS], int rows)
{
   int i, j;
   for (i = 0; i < rows; i++)
   {
      for (j = 0; j < COLS; j++)
         printf("%d ", matrix[i][j]);
      printf("\n");
   }
}

int main()
{
   int matrix[2][COLS] = {{1, 2, 3},{4, 5, 6}};
   printArray(matrix, 2);
   return 0;
}
```

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.

Dynamic Allocation

The heap

- 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.

How to Borrow Memory

We use the `malloc` function (defined in `<stdlib.h>`) to request memory and `free` to return it to the heap.

```
#include <stdio.h>
#include <stdlib.h>
#define NCOUNT 10
int main()
{
   int * numbers, loop;
   numbers = (int *) malloc(sizeof(int)*NCOUNT);
   if (!numbers)
   {
      fprintf(stderr, "Unable to allocate memory\n");
      return -1;
   }
   for (loop = 0; loop < NCOUNT; loop++)
      numbers[loop] = loop;

   for (loop = 0; loop < NCOUNT; loop++)
      printf("numbers[%d] = %d\n", loop, numbers[loop]);

   free(numbers);
   return 0;
}
```

How to Borrow Memory - Details

```c
void * malloc(size_t Size);
void free(void * Memory);
```

What is `void *`?

- `void *` is a pointer to an unknown data type, it consists only of an address.
- We can explicitly cast from `void *` to any other pointer type.
- Any pointer type can be implicitly casted to a `void *`.
- A value of `NULL` usually means that a function encountered an error.

What is `size_t`?

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

How to Borrow Memory - More Details

Two more popular 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 `assert`ed).
- Every `malloc`/`calloc` should have a corresponding `free`, memory is *leaked* if it's not `free`d after use.

1-D Example: Fibonacci (again!)

```
#include <stdlib.h>
#include <stdio.h>

int main()
{
   double * fibs;
   unsigned int nfibs=0, loop;
   while (nfibs < 2)
   {
      printf("How many Fibonacci numbers are needed (>1)?\n");
      scanf("%u", &nfibs);
   }

   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;
}
```

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`”.

Constructing Matrices with Pointers

```
double ** makeMatrix(unsigned int rows, unsigned int cols)
{
   unsigned int i;
   double ** matrix;

   matrix = (double **) malloc(rows * sizeof(double *));
   if (!matrix) return NULL; /* failed */

   for (i = 0; i < rows; i++)
   {
      matrix[i] = (double *) malloc(cols*sizeof(double));
      if (!matrix[i])
         return NULL; /* lazy, we should really free
                         all the memory allocated above */
   }

   return matrix;
}
```

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
```

- 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!)

A Better Way of Allocating Matrices

```
double ** allocMatrix(unsigned int rows, unsigned int cols)
{
   double ** matrix;
   unsigned int i;

   matrix = (double **) malloc (rows*sizeof(double *));
   if (!matrix) return NULL; /* failed */

   matrix[0] = (double *) malloc (rows*cols*sizeof(double));
   if (!matrix[0])
   {
      free(matrix); /* we don't need matrix any more */
      return NULL;  /* failed */
   }

   for (i = 1; i < rows; i++)
      matrix[i] = matrix[i-1] + cols;

   return matrix;
}
```

Why is `allocMatrix` Better?

- `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);
}
```

Case Study: Matrix Addition

Let's define some utility functions to:

1. Allocate memory for the matrix (`allocMatrix`) - **done**,
2. Free a matrix (`freeMatrix`) - **done**,
3. Print a matrix (`printMatrix`),
4. Create a random matrix (`randomMatrix`),
5. Add matrices together (`addMatrices`)

We drive all these functions using a `main` function.

printMatrix and randomMatrix

```
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("%8.5lf ", matrix[i][j]);
      printf("\n");
   }
}


void randomMatrix(double ** matrix, unsigned int rows,
                                    unsigned int cols)
{
   unsigned int i, j;
   for (i = 0; i < rows; i++)
      for (j = 0; j < cols; j++)
         matrix[i][j] = (double)rand()/RAND_MAX;
}
```

addMatrices

```
void addMatrices(double ** matrixA, double ** matrixB,
                 double ** matrixR,
                 unsigned int rows, unsigned int cols)
{
   unsigned int i, j;
   for (i = 0; i < rows; i++)
      for (j = 0; j < cols; j++)
         matrixR[i][j] = matrixA[i][j]+matrixB[i][j];
}
```

The main function

```c
int main()
{
   unsigned int rows, cols;
   double ** matrixA, ** matrixB, **matrixC;
   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;
}
```

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
```