C for Science

Lecture 4 of 5

Sam Bott
http://www2.imperial.ac.uk/~shb104/c
s.bott@imperial.ac.uk

17^{th} April 2013

Imperial College London

Last Week...

- A char is the smallest addressable data type (1 byte). It is used as either an `ascii` character or a very small integer.
- `stdin, stderr and stdout` are the standard console streams. File streams are opened with `fopen`.
- Memory can be allocated dynamically using `malloc`. This is useful for large (or unknown size) arrays and matrices.
- `free` is used to release previously allocated memory back to the stack; forgetting to 'free' memory causes memory-leaks.

Matrices - Details

For matrices allocated in the previous lecture

`matrix`	points to a matrix - *more specifically, it points to an array holding the addresses of each row*
`matrix[0]`	points to the first row of the matrix
`matrix[0][0]`	The top left element of the matrix

Which have the following types:

`matrix`	`double **`
`matrix[0]`	`double *`
`matrix[0][0]`	`double`

Higher Dimensional Arrays

```c
double *** alloc3Tensor(unsigned int dim)
{
   unsigned int i, j;
   double *** tensor;
   tensor = (double ***) malloc(dim*sizeof(double **));
   if (!tensor) return NULL;
   tensor[0] = (double **) malloc(dim*dim*sizeof(double *));
   if (!tensor[0])
   {
      free(tensor);
      return NULL;
   }
   tensor[0][0] = (double *) malloc(dim*dim*dim*sizeof(double));
   if (!tensor[0][0])
   {
      free(tensor[0]); free(tensor);
      return NULL;
   }
   for (i = 1; i < dim; i++)
      tensor[0][i] = tensor[0][i-1]+dim;
   for (i = 1; i < dim; i++)
   {
      tensor[i] = tensor[i-1] + dim;
      tensor[i][0] = tensor[i-1][0] + dim * dim;
      for (j = 1; j < dim; j++)
         tensor[i][j] = tensor[i][j-1] + dim;
   }
   return tensor;
}
```

More Multi-Index

- The 3D array behaves as expected:

```
tensor[i][j][k] = 0.5;
```

- The array can be freed with:

```c
void free3Tensor(double *** tensor)
{
    free(tensor[0][0]);
    free(tensor[0]);
    free(tensor);
}
```

More Elaborate Data Structures

- In C we are able to manipulate data directly, this has allowed us to partition a contiguous array of memory into matrix rows.
- We needn't restrict ourselves to structures where the number of elements per row is constant, one example is Pascal's triangle:

$$\begin{array}{ccccccccc} & & & & 1 & & & & \\ & & & 1 & & 1 & & & \\ & & 1 & & 2 & & 1 & & \\ & 1 & & 3 & & 3 & & 1 & \\ 1 & & 4 & & 6 & & 4 & & 1 \\ & & & & \vdots & & & & \end{array}$$

```c
int main()
{
   unsigned int size = 10, r, c;
   int ** pasc;
   pasc = (int **) malloc(size*sizeof(int *));
   pasc[0] = (int *) malloc(size*(size+1)*sizeof(int)/2);
   /* check the mallocs */
   for (r=1; r < size; r++) pasc[r] = pasc[r-1]+r;
   pasc[0][0] = 1;
   for (r = 1; r < size; r++)
   {
      for (c = 1; c < r; c++)
         pasc[r][c] = pasc[r-1][c-1]+pasc[r-1][c];
      pasc[r][0] = 1; pasc[r][r] = 1;
   }
   for (r=0; r < size; r++)
   {
      for (c = 0; c < r+1; c++)
         printf("%3d ", pasc[r][c]);
      printf("\n");
   }
   free(pasc[0]);
   free(pasc);
   return 0;
}
```

Pascal's Triangle

- The program outputs:

```
1
1    1
1    2    1
1    3    3    1
1    4    6    4    1
1    5   10   10    5    1
1    6   15   20   15    6    1
1    7   21   35   35   21    7    1
1    8   28   56   70   56   28    8    1
1    9   36   84  126  126   84   36    9    1
```

- Whilst in memory, the data structure looks like:

1	1	1	1	2	1	1	3	3	1	···

Custom Data Types

Recall, when dealing with files we used:

```
FILE * file;
file = fopen ("myfile.txt", "r");
```

`FILE` is in fact a custom data type, with its own size:

```
#include <stdio.h>

int main()
{
    printf("sizeof(FILE) = %d\n", sizeof(FILE));
    return 0;
}
```

Except from MS - `<stdio.h>`

(From Visual Studio 2008)

```
struct _iobuf {
    char *_ptr;
    int   _cnt;
    char *_base;
    int   _flag;
    int   _file;
    int   _charbuf;
    int   _bufsiz;
    char *_tmpfname;
    };
typedef struct _iobuf FILE;
```

`typedef` Structures : Custom Data Types

- `FILE` is a custom data type defined in `<stdio.h>`.
- It is comprised of elements of different, known, types.
- Each element of the `FILE` *structure* has its own name.

Let's consider a structure of our own, one of the simplest examples is a complex number:

```
typedef struct
{
    double real;
    double imag;
} complex;
```

C99

C99 fully supports its own `_Complex` type.

typedef Structures - II

Definitions of structures take the following form:

```
typedef struct
{
    elementType elementName;
    elementType elementName;
    :
} structureTypeName ;
```

Handling Structures

1. A structure may be an argument to a function.
2. A function may return a structure.
3. A pointer may point to a structure.
4. Structures are referenced as normal: `&name`.
5. Elements of a structure are referenced as: `&name.element`.
6. A pointer to a structure may be passed to a function.
7. If `p` is a pointer to a structure, then `p->member` allows us to access it's members.

Example Function Applying to Structures

```c
#include <stdio.h>

typedef struct
{
   double real;
   double imag;
} complex;

void printComplex(complex * mc)
{
   printf("%lg + %lgi\n", mc->real, mc->imag);
}

int main()
{
   complex c1 = {1.0, 0.5}; /* assignment at declaration */
   printComplex(&c1);        /* pass pointer to struct */
   c1.real = 10.0;           /* piecewise assignment */
   printComplex(&c1);
   return 0;
}
```

More Structures

- Passing structures via pointer is usually more efficient.
- Assignment can be done at declaration or after.
- In C we are not allowed to overload operators, so the following won't work:

  `c1 = c2 + c3;` (where `c1` and `c2` are `complex`)

  so we need to do something like:

  ```
  complexAdd(&c1, &c2, &c3);
  ```
- Arrays of `struct`s are allowed (so matrices can consist of complex numbers for example).
- Structures can contain structures as elements.

Complex Number Support in C/C++

C++

Complex numbers are supported in C++ as `class`es, also the operators are overloaded properly too meaning any code which uses them will be concise. (look in `<complex>`).

C99

Complex numbers are supported as a native data type (not `struct`) in C99. Unfortunately not many compilers support this. GNU C, fully supports complex numbers (see `<complex.h>`).

C90

Complex number support in C90 is non-existent. I would recommend either third party libraries or a switch to C99/C++ for heavy complex number use.

Scope Blocks

Variable scope is not restricted to functions, indeed anything contained in braces (*scope blocks*) has it's own scope:

```
#include <stdio.h>

int main()
{
    int i = 1;

    if (i==1)
    {
        int j = 10; /* local to if block */
        printf("i+j=%d\n", i+j);
    }

    /* printf("j = %d\n", j); - error */

    return 0;
}
```

Local Variable Lifetime

The lifetime of a local variable is limited:

```
#include <stdio.h>

void F1()
{
    int i = 1;
    printf("In F1(): i = %d\n", i);
    i = i + 1; /* won't do much */
}

int main()
{
    F1();
    F1();
    return 0;
}
```

Local Variables

- Every time `F1()` is invoked, the value of `i` is reset to 1.
- The variable `i` is a local variable which lives on the stack:
  - it will be destroyed every time we leave `F1()`,
  - and recreated every time we invoke `F1()`.
- We can ask the C compiler to retain the value of `i`:

`static` variables

A variable can be declared to be `static`, this tells the C compiler to set aside memory other than the stack to store the variable. The variable will not be destroyed until the program ends.

Why Use Static Variables?

- To count the number of times a function has been called.
- To have the function remember something between calls so on subsequent calls it can calculate what has changed (time, storage, the date, etc...), or resume from where it left off (reading from a list...)

Worked Example: Elapsed Time

- We write a function `timer` which returns the time since it was last called.
- Our program will contain two `.c` files to demonstrate linking.

Timer Example: `timer()`

```c
#include <time.h> /* for clock function */

double timer()
{
    static double oldTime = 0.0;
    double newTime, diff;

    newTime = clock();
    diff = newTime - oldTime;
    oldTime = newTime;
    return diff/CLOCKS_PER_SEC;
}
```

Timer Example: `main()`

```c
#include <stdio.h>

double timer(); /* declare timer */

int main()
{
   timer(); /* reset clock */
   printf("Wait... and hit return\n");
   getchar(); /* wait for return */
   printf("Elapsed time: %.2f seconds\n", timer());
   printf("Resetting clock, hit enter again\n");
   getchar();
   printf("Elapsed time: %.2f seconds\n", timer());
   return 0;
}
```

Global Variables - Shared between Functions

We can define a *global* variable:

```c
#include <stdio.h>

double global = 42.0;

void F1()
{
   printf("Global = %g\n", global);
}

int main()
{
   F1();
   global = global + 1.0;
   F1();
   return 0;
}
```

Global Variables - Between Multiple Files

Global to All Files

If we want to share a global variable between files, then we define it *once* as:

```
type myglobalvariable = value;
```

Then in every other file we declare it as:

```
extern type myglobalvariable;
```

Global Only to Current File

If we do not wish to share a global variable, we define it as follows:

```
static type myPrivateGlobal = value;
```

Shared/Private Global Variables - File 1 of 2

```
#include <stdio.h>

int globalEverywhere = 42;
static int globalHereOnly = 1;

void F2(); /* defined in file 2 */

int main()
{
   printf("globalEverywhere = %d\n", globalEverywhere);
   printf("globalHereOnly = %d\n", globalHereOnly);

   F2();

   printf("globalEverywhere = %d\n", globalEverywhere);
   printf("globalHereOnly = %d\n", globalHereOnly);

   return 0;
}
```

Shared/Private Global Variables - File 2 of 2

```
#include <stdio.h>

extern int globalEverywhere;
/*if we forget the extern we get a linker error */

static int globalHereOnly = 100;

void F2()
{
   printf("F2(): globalEverywhere = %d\n",
                 globalEverywhere);

   printf("F2(): globalHereOnly = %d\n",
                 globalHereOnly);
}
```

Compilation

- As seen in the first lecture, C programs are *compiled* into a low level (machine specific language).
- This language is called *assembly language*.
- Most C compilers allow you to view the assembler that they generate.

`addMatrices` revisited - C code

From the last lecture we saw a function to add two matrices together:

```
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];
}
```

When we compile this, we get...

addMatrices revisited - when compiled...

```
_addMatrices PROC
; 35   : {
        push    ebp
        mov     ebp, esp
        sub     esp, 216
        push    ebx
        push    esi
        push    edi
        lea     edi, DWORD PTR [ebp-216]
        mov     ecx, 54
        mov     eax, -858993460
        rep stosd
; 36   :        unsigned int i, j;
; 37   :        for (i = 0; i < rows; i++)
        mov     DWORD PTR _i$[ebp], 0
        jmp     SHORT $LN6@addMatrice
$LN5@addMatrice:
        mov     eax, DWORD PTR _i$[ebp]
        add     eax, 1
        mov     DWORD PTR _i$[ebp], eax
$LN6@addMatrice:
        mov     eax, DWORD PTR _i$[ebp]
        cmp     eax, DWORD PTR _rows$[ebp]
        jae     SHORT $LN4@addMatrice
; for (j = 0; j < cols; j++)
        mov     DWORD PTR _j$[ebp], 0
        jmp     SHORT $LN3@addMatrice
$LN2@addMatrice:
        mov     eax, DWORD PTR _j$[ebp]
        add     eax, 1
        mov     DWORD PTR _j$[ebp], eax
```

```
$LN3@addMatrice:
        mov     eax, DWORD PTR _j$[ebp]
        cmp     eax, DWORD PTR _cols$[ebp]
        jae     SHORT $LN1@addMatrice
;matrixR[i][j] = matrixA[i][j]+matrixB[i][j];
        mov     eax, DWORD PTR _i$[ebp]
        mov     ecx, DWORD PTR _matrixA$[ebp]
        mov     edx, DWORD PTR [ecx+eax*4]
        mov     eax, DWORD PTR _i$[ebp]
        mov     ecx, DWORD PTR _matrixB$[ebp]
        mov     eax, DWORD PTR [ecx+eax*4]
        mov     ecx, DWORD PTR _j$[ebp]
        mov     esi, DWORD PTR _j$[ebp]
        fld     QWORD PTR [edx+ecx*8]
        fadd    QWORD PTR [eax+esi*8]
        mov     edx, DWORD PTR _i$[ebp]
        mov     eax, DWORD PTR _matrixR$[ebp]
        mov     ecx, DWORD PTR [eax+edx*4]
        mov     edx, DWORD PTR _j$[ebp]
        fstp    QWORD PTR [ecx+edx*8]
        jmp     SHORT $LN2@addMatrice
$LN1@addMatrice:
        jmp     SHORT $LN5@addMatrice
$LN4@addMatrice:
        pop     edi
        pop     esi
        pop     ebx
        mov     esp, ebp
        pop     ebp
        ret     0
_addMatrices ENDP
```

Optimisation

- A few lines of C becomes > 30 lines of assembler (only three of which are actually floating point instructions!).
- It is possible to write a much smaller assembler routine by hand $\approx$10-20 instructions long.
- This would run $\approx$3 times quicker than the C compiled routine (this is a general rule of thumb).
- Any custom assembly code would only target a very specific chip, however.

Optimisation II

- Rather than rewrite the assembly code, it is easier to ask the C compiler to perform code optimisation itself.
- By default C will compile the code as it appears (the exact order of operations is preserved etc), this aids debugging.
- C compilers can be told to optimise their code in the following ways:

MSVC	Project configuration options can be set, defaults in “Release” build do a good job.
gcc	The `-O` command line flags influence optimisation, `-O0` means “off” whilst `-O3` means “extremely aggressive”.

Optimisation Example - Visual Studio 2008

Debug Build

```
$LN2@addMatrice:
        mov     eax, DWORD PTR _j$[ebp]
        add     eax, 1
        mov     DWORD PTR _j$[ebp], eax
$LN3@addMatrice:
        mov     eax, DWORD PTR _j$[ebp]
        cmp     eax, DWORD PTR _cols$[ebp]
        jae     SHORT $LN1@addMatrice
;matrixR[i][j] = matrixA[i][j]+matrixB[i][j];
        mov     eax, DWORD PTR _i$[ebp]
        mov     ecx, DWORD PTR _matrixA$[ebp]
        mov     edx, DWORD PTR [ecx+eax*4]
        mov     eax, DWORD PTR _i$[ebp]
        mov     ecx, DWORD PTR _matrixB$[ebp]
        mov     eax, DWORD PTR [ecx+eax*4]
        mov     ecx, DWORD PTR _j$[ebp]
        mov     esi, DWORD PTR _j$[ebp]
        fld     QWORD PTR [edx+ecx*8]
        fadd    QWORD PTR [eax+esi*8]
        mov     edx, DWORD PTR _i$[ebp]
        mov     eax, DWORD PTR _matrixR$[ebp]
        mov     ecx, DWORD PTR [eax+edx*4]
        mov     edx, DWORD PTR _j$[ebp]
        fstp    QWORD PTR [ecx+edx*8]
        jmp     SHORT $LN2@addMatrice
```

Release Build

```
;matrixR[i][j] = matrixA[i][j]+matrixB[i][j];
                fld     QWORD PTR [edx+eax]
                fadd    QWORD PTR [eax]
                fstp    QWORD PTR [esi+eax]
        add     eax, 8
        dec     ebx
        jne     SHORT $LL3@addMatrice
```

Bits and Bytes

Bytes

Smallest *addressable* unit of memory, each byte is composed of eight bits.

Bits

These are the smallest units of computer memory, each bit can be either 0 or 1.

- Addressing bits individually requires some extra operations to be carried out.
- There are good reasons for accessing data at the bit-level however.

Efficient Data Packing

Given a 32 million base pair chromosome

It will require: $\approx$128 megabytes to store as `int`
$\approx$32 megabytes to store as `char`
$\approx$8 megabytes to store as bit data.

Computer Graphics

Given a monochrome print image of 2400 dpi rendered over 80 square inches gives $\approx 500,000,000$ dots. This takes up:

$\approx$1 gigabyte if using `short`
$\approx$64 megabytes if using bits.

Bit Manipulation Friendly Data Types

As seen before, the unsigned integer data types have values ranging from 0 to $2^n - 1$ where n is the number of bits in the data type:

Data Type	n
`unsigned short`	16
`unsigned int`	32
`unsigned long int`	32
`unsigned long long int`	64

(Unsigned data types are also desirable for accessing array indices as they can never be negative.)

When using these data types for bit data we ignore the resulting value it stores and instead just focus on the n zeros or ones in the data type.

How to Get Them In and Out of The Computer

- They can be read using `scanf` and `"%u"`, `"%lu"` or `"%Lu"`.
- They can be printed using `printf` and `"%u"` or `"%Lu"`.
- We can output to octal (base 8 numbers) using `printf` and `"%o"`
- Also we can output to hexadecimal (base 16: 0-9 and a-f) using `printf` and `"%x"` or `"%X"`.

Example - byte (`char`)

Binary	10101011
Hexadecimal	AB
Decimal	171
Octal	253

Manipulating Bits within an Unsigned Integer

- C can shift all the bits comprising a number a fixed number of places to the left or right.
- Zeros are propagated in to the vacated spaces.
- Bits that shift outside, disappear. (i.e. the shift is not cyclic).
- Bit shifting is accomplished with the $>>$ (right) and $<<$ (left) operators.

For example:

```
1 << 2 = 4
8 >> 3 = 1
```

Bit shifts are much cheaper than multiplying or dividing by powers of two.

4 Bitwise operators &, |, ∧ and ∼

Assuming 0 is false and 1 is true, we have the following bitwise logical operators.

And Operator(&)

N1	0	0	1	1
N2	0	1	0	1
N1 & N2	0	0	0	1

Or Operator(|)

N1	0	0	1	1
N2	0	1	0	1
N1 | N2	0	1	1	1

Exclusive Or(∧)

N1	0	0	1	1
N2	0	1	0	1
N1 ∧ N2	0	1	1	0

Not Operator(∼)

N1	0	1
∼N1	1	0

Case Study: The Sieve of Eratosthenes

Algorithm

List all odd numbers up to n. Then, starting at the beginning of the list. . .

1. select next available number in the list, i, as prime
2. if $i \leq \sqrt{n}$ remove all multiples of i in the list greater than i^2
3. repeat. . .

We prepend 2 to this list, making it the list of primes $< n$.

The Sieve of Eratosthenes: Algorithm Example

For $n = 51$:
Write out list of odd numbers

3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39	41	43	45	47	49	51

Select, and remove multiples of, 3

3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39	41	43	45	47	49	51

Select, and remove multiples of, 5

3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39	41	43	45	47	49	51

Select, and remove the multiple of, 7

3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39	41	43	45	47	49	51

Select the remaining numbers and prepend 2. This gives the primes:

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47

The Sieve of Eratosthenes: In C

- When implementing this in C it makes sense to use a bit to indicate ‘primeness’ of a number.
- The smallest addressable unit of memory in C is the `char` which consists of 8 bits.
- We therefore need to perform *masking* to isolate individual bits.

Masking

We access the i^{th} bit of a variable `x` as follows:

`if(x & (1 << i))`	Check to see if it’s set
`x |= (1 << i)`	To set the bit
`x &= ~(1 << i)`	To clear the bit

The Sieve of Eratosthenes: Implementation

```c
void findPrimes(char * Prime_List, int max_num)
{
   int current_num, sqmax_num;
   char Mask=1;
   sqmax_num = sqrt((float)max_num);

   for(current_num = 3; current_num <= sqmax_num; current_num += 2)
   {
      int Pnum = current_num / 16; /*Find which char we are on*/
      short Pbit = (current_num - Pnum * 16) / 2; /*Which bit in char*/

      if(~Prime_List[Pnum] & (Mask << Pbit))
      {   /*if the current bit in the current char is a zero(prime)
         * then lets strike out some multiples*/
         int strike;
         for(strike = current_num * current_num; strike <= max_num; strike +=
         {
            int Snum = strike / 16;
            short Sbit = (strike - Snum * 16)/2;
            Prime_List[Snum] |= (Mask << Sbit);
         }
      }
   }
}
```

The Sieve of Eratosthenes : Printing out the Primes

```c
void printPrimes(char * Prime_List, int max_num)
{
    int i;
    char Mask = 1;
    for(i = 3; i <= max_num; i = i+2)
    {
        int Pnum = i / 16;
        short Pbit = (i - Pnum * 16) / 2;
        if (~Prime_List[Pnum] & (Mask << Pbit))
            printf("%d\n", i);
    }
}
```

Summary

- More complex array structures can be created with `malloc` than matrices.
- We can create structs which are our own data types holding multiple values.
- Global variables can be accessed anywhere in your program.
- The lifetime of a variable can be extended by making it static.
- A 'Release' build will optimise your program at compile-time.
- Using the bits that make up individual data types can be an efficient alternative the data types themselves.