C for Science

Week 2

Pointers and Dynamic Memory, C-Standard Library Functions and Makefiles

Dr Maria Valera(m.valera-espina@doc.ic.ac.uk)

Imperial College London

February 2016

Table of Contents week 1

Table of Contents

Course Content

- Program structure.
- C Types: conversions and casts.
- Control structures: sequence,selection and repetition.
- Conditions: operands, operators and its precendence.
- Arithmetic and Logical expressions.
- C Standard Library Functions.
- string Types.
- Pointers and Arrays Types.
- Memory management.
- Input/Output.
- Structures: Type Data Structure and Dynamic Data Stucture.
- C Standard Library and Scientific C-Libraries.
- Optimisation & Debugging.

Table of Contents

More Mathematical Functions in <math.h>

- Maths functions come with the ANSI Standard C Library, which contains many maths functions. To use them we need a:
- `#include <math.h>`
- Here some example functions:

```
sin(x) asin(x) sinh(x) exp(x)
cos(x) acos(x) cosh(x) log(x)
tan(x) atan(x) tanh(x) log10(x)
sqrt(x) atan2(x,y) pow(x,y) fabs(x)
```

(all the trigonometric functions use radians!)

The `pow(x, y)` function declared in <math.h>

Exponentiation

As you noticed from your exercise, there is no exponentiation operator (e.g. ^) in C. Instead, we have the following:

x^y =`pow(x,y)`

`pow(x,y)` assumes x and y are of type `double.`

Notice

The `pow` function is often implemented as:

`exp (y * ln(x))`

For whole integer powers (i.e. x^2), the multiplication explicitly (`x * x`) should be used.

Table of Contents

This lecture

Pointers:

- We touched on pointers previously when we looked at `scanf()`.
- Using pointers, we can construct references to *any* C variable.
- Pointers in C are much more powerful, and dangerous. If we use a pointer and the value it points to no longer exists, **bad** things will happen.

Dynamic memory allocation:

- We'll be looking at dynamic (at run-time) memory allocation. [1]
- Failure of code to do this is called a *memory leak*.

Incorrect use of pointers and memory handling are major sources of C programming errors.

[1] C's equivalent of the **new** operator in Java.

Pointers

- Memory can be seen an ordered sequence of consecutively numbered storage locations.
  - Variables are stored in memory in one or more adjacent storage locations depending on its type:

```
char str[] = "is"; int i = 1; char c = 'R';
```

i	s	\0	1				R
0xbfffff72	0xbfffff73	0xbfffff74	0xbfffff75	0xbfffff76	0xbfffff77	0xbfffff78	0xbfffff79
str			i				c

  - The **address** (&) of a variable is the address of its firt storage location.

i	s	\0	1				R
0xbfffff72	0xbfffff73	0xbfffff74	0xbfffff75	0xbfffff76	0xbfffff77	0xbfffff78	0xbfffff79
&str==0xbfffff72			&i==0xbfffff75				&c==0xbfffff79

- A pointer in C represents the starting address of a value in memory.

Table of Contents

Declaring pointers

- The syntax for declaring pointers in C can be slightly inconsistent.
- For primitive types, we simply append a "*" to the type to construct the pointer type:

Declare a pointer

```
double* doublePtr; /* A pointer to a double */
int *intptr;       /* A pointer to an int */
char* charptr, strptr;/* A pointer to a char 'charptr'
                       and a pointer to a char 'strptr' */
int  **intPtrPtr;  /* A pointer to a pointer to an int */
```

- After declaring a pointer we assign the first storage location of a variable to a pointer using &

```
strptr = (char *) &str; intptr = &i; charptr = &c;
```

0xbfffff72		0xbfffff75		0xbfffff79	
0xbfffff80	0xbfffff81	0xbfffff82	0xbfffff83	0xbfffff84	0xbfffff85
strptr		intptr		charptr	

The `const` keyword

The `const` keyword allows us to specify that certain values cannot be modified.

```
const int x = 4;
x = 5; /* This is a compile-time error */
```

`const` can also be applied to pointers, though it can become confusing quickly for multiple levels of indirection:

```
int val = 5;
const int *ptr1 = &val;       /* ptr1 can be modified, val cannot */
int *const ptr2 = &val;       /* val can be modified, ptr2 cannot */
const int *const ptr3 = &val; /* neither val nor ptr3 can be modified */
```

Try reading the pointer declarations right to left.

Table of Contents

Pointer Operators

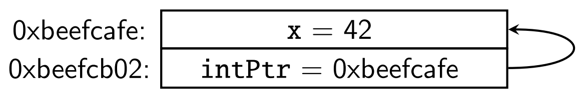
The following two operators are the primary mechanism for performing pointer-related operations in C:

- The address operator (&) is a prefix operator that takes a value and returns its address.
- The indirection operator (*) is a prefix operator that takes a pointer and returns the value it points to. This is called "de-referencing" the pointer.

Pointer Operators

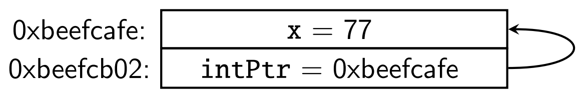
Declare and Assign

```
int x = 42;
int *intPtr = &x;
```



De-referencing

```
*intPtr = 77;
```



Printing pointers

`printf` can also print pointer values using the "%p" specifier. Printing out the addresses of a parameter in a recursive function lets us see what way the stack grows:

```
#include <stdio.h>

void printAddresses(int depth) {
  printf("Address of depth: %p\n", &depth);

  if(depth > 0) {
    printAddresses(depth-1);
  }
}

int main(void) {
  printAddresses(5);
  return 0;
}
```


A word on null pointers

- You can nullify any pointer by assigning the integer `0` to it.
- The header file `stdlib.h` defines the macro `NULL` which is equivalent.
- Checking if a pointer is `NULL` is a good thing to do, but only useful if it was actually set to `NULL` in the first place.

intPtr might not be NULL!

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

void someFunction(int *output) {
  assert(output != NULL);
  *output = 42;
}

int main(void) {
  int *intPtr;
  someFunction(intPtr);
  return 0;
}
```


Table of Contents

Pointer Arithmetic

In C, we have the ability to explicitly manipulate pointer values. We can add and subtract values from them, and use the increment (++) and decrement (--) operators on them.

```
#include <stdio.h>

int main(void) {
  char greeting[] = "Hello world!\n";
  char *currentLetter = greeting;

  while(*currentLetter != '\0') {
    putchar(*currentLetter);
    ++currentLetter;
  }

  return 0;
}
```

Pointer Arithmetic

Incrementing and decrementing pointers does so in multiples of the size of the type being pointed to:

```
#include <stdio.h>

int main(void) {
  char strArr[] = "abcd";
  int intArr[] = {1, 2, 3, 4};

  char *strPtr = strArr;
  int *intPtr = intArr;
  for(int i=0; i<4; ++i) {
    printf("*strPtr = %c, strPtr = %p\n", *strPtr, strPtr);
    printf("*intPtr = %i, intPtr = %p\n\n", *intPtr, intPtr);

    ++strPtr;
    ++intPtr;
  }
}
```


Pointer Arithmetic

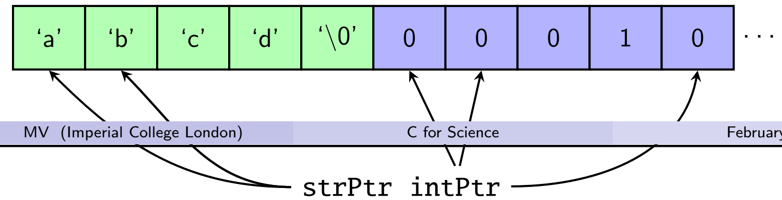
```
char strArr[] = "abcd";
int  intArr[] = {1, 2, 3, 4};

char *strPtr = strArr;
int  *intPtr = intArr;
```

```
strPtr++;
intPtr++;
```

```
intPtr--;

/* Circumventing the type system - don't do this! */
intPtr = (int*) (((char*) intPtr) + 1);
```



Allowed Pointer Operations

- Declaration: double *pA, *pB;
- Assignment: pA = &var;
- Increment: pA = pA +1; /*Incrementing memory position */
- Decrement: pA = pA - 1; /*Decrementing memory position */
- Difference: gap = pA - pB; /*Subtraction is a block of memory */
- Comparison: if(pA == pB) /* comparison of memory positions */
- De-referencing: *pA = val; *pA = *pA + 1; /*Incrementing var variable to value + 1*/

Table of Contents

Pointer to Pointer

```
#include <stdio.h>
int main(){
char str1[]="is"; char str2[]="in";
char *i[2];
i[0] = (char *) &str1;
i[1] = (char *) &str2;
char **ii = (char **) &i;
printf("will print 'is': %s\n",i[0]);
printf("will print 'in': %s\n",i[1]);
printf("will print 'is': %s\n",*ii);
ii++;
printf("will print 'in': %s\n",*ii);
return 0;
}
```


Pointer to Pointer

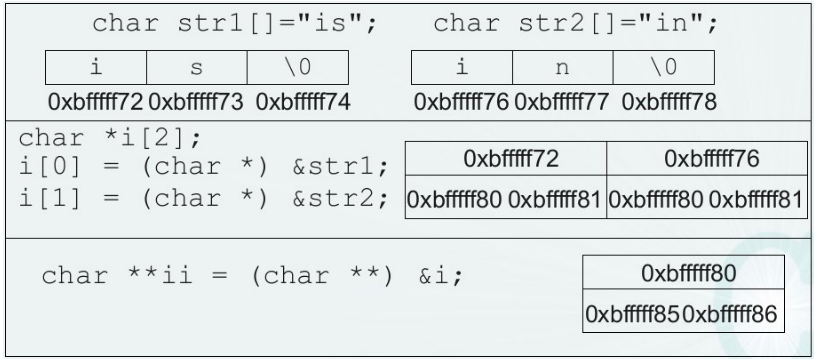


Table of Contents

Pointers and Arrays

- The concepts of arrays and pointers are closely related in C.
- An array can casted *implicitly* to a pointer to the element type.

```
double x[10];
double *xPtr = x;
float *floatPtrArray[10];/* An array of pointers to floats */
int y[15][30];
int (*yPtr)[30] = y;/* A pointer to an array of floats */
```

- What happens if we want a pointer to the first element of a multi-dimensional array in C?

```
float array2D[10][20];
float *array2DPtr = &array2D[0][0];
```

- We simply take the address of the first element.

Pointers using Array Notation

Rather than using pointer arithmetic and the dereference operator, it is possible to use the [] operator in exactly the same way as with arrays:

The following three accesses are identical:

```
void func(char *str) {
  /* Access the eighth letter (we count from 0) */
  char letter1 = *(str+7);
  char letter2 = str[7];
  char letter3 = (str+4)[3];
}
```


Fixed Size Two-Dimensional Arrays

As we saw in week2, 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:

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]

They are allocated from the stack thus large arrays may cause problems.

To access the top left element:

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


Example 2D Array

```
#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;
}
```


Table of Contents

Passing by reference

- The pointer passed by reference is copied but not the data pointed to.

```c
#include <stdio.h>

void swap(int *a, int *b) {
  int temp = *b;
  *b = *a;
  *a = temp;
}

int main(void) {
  int a = 42;
  int b = 77;

  printf("a: %i, b: %i\n", a, b);
  swap(&a, &b);
  printf("a: %i, b: %i\n", a, b);

  return 0;
}
```

```c
#include <stdio.h>

void swap(int *a, int *b);

int main (void){
int a =3;
int b = 4;
printf("a: %i , b: %i\n",a,b);
swap(&a,&b);
printf("a: %i , b: %i\n",a,b);
return 0;
}

void swap (int *a,int *b){
int *temp = b;
*b = *a;
*a = *temp;
}
```


The truth about passing arrays to functions

- C doesn't perform type checking on the leading dimension of an array when passing it to a function.
- C passes arrays by reference.
- This may have seemed a little strange given that the only way to pass by reference in C is to use a pointer.
- In fact, C always converts the leading dimension of an array to a pointer when passing it to a function.
- This is the true reason why arrays are passed by reference, and why the leading dimension is never checked.

The truth about passing arrays to functions

We can see the artifacts of this behaviour if we look close enough. What do you think this will print?

```c
#include <stdio.h>

void printSizeArray(char array[100]) {
  printf("sizeof(array) = %li\n", sizeof(array));
}

void printSizePtr(char *ptr) {
  printf("sizeof(ptr) = %li\n", sizeof(ptr));
}

int main(void) {
  char buffer[100];
  printf("sizeof(buffer) = %li\n", sizeof(buffer));
  printSizeArray(buffer);
  printSizePtr(buffer);
  return 0;
}
```


Command Line Arguments

The `main` function of a C program can also have a type signature where it receives arguments passed to it from the command line. It takes two parameters:

- `argc` - The number of parameters passed.
- `argv` - An array of C strings.

We can print them as follows:

Prints each argument

```c
#include <stdio.h>

int main(int argc, char **argv) {
  for(int i=0; i<argc; ++i) {
    printf("argv[%i] = %s\n", i, argv[i]);
  }

  return 0;
}
```

atoi()

You will need to convert a command line argument to a number for your tutorials. The easiest way to do this is with the `atoi()` function from `stdlib.h`.

```
int atoi(const char *nptr);/* atoi() does not detect errors*/
```

Counts from one to the supplied argument

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

int main(int argc, char** argv){
int count =0;
if(argc >1) {
  count = atoi(argv[1]);
}
printf("the number of count is %d\n", count);
for(int i=0; i< count; i++){
 printf("%d\n",i);
}
return 0;
}
```


Pointers as function types

- We also can declare pointers as function types:

  `char *doSomething()`

- `doSomething()` will return a `char` pointer. However, this is only really useful if dynamic memory is allocated.

Table of Contents

String Utilities

The header `string.h` contains a number of useful utility functions:

- String length:

```
size_t strlen(const char *s);
```

- String concatenation:

Requires dest to have size `strlen(dest)+n+1`!

```
char *strncat(char *dest, const char *src, size_t n);
```

- String comparison:

```
int strncmp(const char *s1, const char *s2, size_t n);
```

- String copying:

Both unsafe!

```
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
```

On Unsafe String Functions

The C API contains many unsafe string functions. Whenever you use them, check the documentation for the following:

- Whenever a buffer is being written to, the requirements on the size of the destination buffer.
- If the function writes a string, under what circumstances it terminates the string with a '\0'.

If the source string is long enough, `strncpy` will not terminate the destination string with '\0', possibly causing later code to run off the end of the string. This is why it is considered unsafe.

Have a look at `strlcpy()` and `strlcat()` for an example of safer functions. They both come from BSD, and are therefore unfortunately non-portable.

String Utilities

String comparison:

```
int strcmp(const char *s1, const char *s2);
```

Lexicographically compares the strings `s1` and `s2`. The return value is

- `<0` - if `str1` is less than `str2`.
- `=0` - if `str1` is equal to `str2`.
- `>0` - if `str1` is greater than `str2`.

There is also a function `strncmp` which only compares up to `n` characters:

```
int strncmp(const char *s1, const char *s2, size_t n);
```


`strcmp()` example

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

int main(void) {
  char a[] = "astring";
  char b[] = "astring";
  char c[] = "astr ing";

  if (!strcmp(a, b)) {
    printf("Strings a and b are the same\n");
  }

  if (!strcmp(a, c)) {
    printf("Strings a and c are the same\n");
  }

  return 0;
}
```


Table of Contents

Function Pointers

C also supports pointers to functions. Here's how we can take a pointer to a sum function:

```
static int sum(int a, int b) {
  return a + b;
}

int main(void) {
  int (*sum_ptr)(int, int);
  sum_ptr = &sum;
  return 0;
}
```

We've written the declaration of sum_ptr the same way we'd have written a function declaration except we replaced the function name with (*sum_ptr).

Function Pointers

It's possible to invoke a function pointer in exactly the same way as normal function.

```
#include <stdio.h>

static int sum(int a, int b) {
  return a + b;
}

int main(void) {
  int (*sum_ptr)(int, int) = &sum;

  printf("The sum of 39 and 73 is %i.\n", sum_ptr(39, 73));
  return 0;
}
```


Function Pointers

We can pass them to other functions as well.

```
#include <stdio.h>

static int sum(int a, int b) { return a + b; }
static int mul(int a, int b) { return a * b; }

static void print_result(int (*func)(int, int), int a, int b) {
  printf("func(%i, %i) = %i\n", a, b, func(a, b));
}

int main(void) {
  int a = 42;
  int b = 37;

  print_result(&sum, a, b);
  print_result(&mul, a, b);
  return 0;
}
```


Table of Contents

Dynamic Memory

- Up until this point, we've only used stack allocated values – values which are destroyed as soon as they go out of scope.
- There is another area of memory called the *heap* which can hold dynamic memory.
- In Java, destroying unreferenced dynamically allocated values was done though a process of *garbage collection*.
- In C, you will need to design your own strategy for freeing dynamic memory, depending on the context.

malloc() example

A C program

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

int main(void) {
  int size = 100;
  int *arr = malloc(size * sizeof(int));


  for(int i=0; i<size; ++i){
    arr[i] = i;
  }
  free(arr);
  return EXIT_SUCCESS;
}
```

A C equivalent

- &arr - The address of the arr in the stack
- *arr - The contents of what is in the heap
- arr - The address allocated in the heap

malloc() example

- `malloc()` is completely unaware of how the returned memory will be used.
- `malloc` takes its size parameter in bytes, *not elements*. We need to use `sizeof()` to work out how much memory to use.
- `malloc` returns a type of `void*` (a pointer to an unknown type). Any value pointer can be implicitly converted to `void*`, but `void*` must be explicitly cast to another pointer type.

We could have written:

```c
void *memory = malloc(size * sizeof(int));
int *arr = (int*) memory;
```


Using dynamic memory safely

- `malloc()`, `realloc()` and `calloc()` all return `NULL` if the allocation fails. This is *not* a fatal error, so you must check for it.
- The pointer passed to `free()` *must* come from `malloc` (or its relatives) or be `NULL`.
- Except for `calloc()`, the memory allocation routines return uninitialised memory.
- If you can't find the `free()` corresponding to a memory allocation in your code, you may have a bug. It won't cause your code to crash (unless you exhaust RAM) but can cause your program's memory use to become bloated.
- **Remember to free!**

The Dynamic Memory API

The main allocation-related functions in `stdlib.h` are:

- `malloc()` - allocates a region of memory of `size` bytes and returns a pointer to the allocated memory.

```
void *malloc(size_t size);
```

- `calloc()` - allocates a region of memory that can hold `nmemb` elements of `size` bytes. The region is initialised to 0.

```
void *calloc(size_t nmemb, size_t size);
```

- `realloc()` - reallocates a region of memory to the supplied size, preserving the contents.

```
void *realloc(void *ptr, size_t size);
```

- `free()` - frees a memory region previously allocated using the above.

```
void free(void *ptr);
```


When `malloc()` fails

- In the code you write, you probably don't have much choice except to exit if a `malloc()` fails. This might not be acceptable in other cases (e.g. a kernel).
- The `perror()` function defined in `stdio.h` prints the last error encountered by a system or library routine to standard error, prefixed by the supplied string (allowed to be NULL).
- The `exit()` function defined in `stdlib.h` immediately (but relatively cleanly) terminates the process with the supplied status code.
- `EXIT_SUCCESS` and `EXIT_FAILURE` are error codes defined in `stdlib.h` and are slightly more portable to non-POSIX systems than using 0 and non-zero values for exit codes.

A Quick Recap

After executing this code:

```
/* Allocates space for two integers */
int *intPtr = malloc(sizeof(int) * 2);
```

- `&intPtr` is the address of the pointer on the stack, and has the type `int**`.
- `intPtr` is a stack-allocated value which contains the starting address of the region allocated on the heap and has the type `int *`.
- `*intPtr` or `intPtr[0]` is the value (uninitialised) of the first integer in the heap-allocated region.
- `*(intPtr+1)` or `intPtr[1]` is the value (uninitialised) of the second integer in the heap-allocated region.

Clearing Memory

Since both stack and heap allocated memory may contain uninitialised values, it's useful to be able to zero large regions quickly. We can do this with the `memset()` method from `string.h`.

```
void *memset(void *s, int c, size_t n);
```

The `n`-byte region pointed to by `s` is set to the value `c`. Although `c` is an `int`, it is converted to an `unsigned char` first. `s` is returned.

`memset()` example

```
char quote[] = "To be or not to be";
memset(quote, '.', 9);
printf("%s\n", quote);
```

Output

```
.........not to be
```

Copying Memory

Copying regions of memory may be done using the `memcpy()` method from `string.h`.

```
void *memcpy(void *dest, const void *src, size_t n);
```

Copies `n` bytes from `src` to `dst`, returning `dest`. The source and destination regions must not overlap. If they do, use `memmove`.

`memcpy()` example

```
char str[] = "Morning World!";
char time[] = "Evening";
memcpy(str, time, 7);
printf("%s\n", str);
```

Output

```
Evening World!
```


Valgrind

- Valgrind is a GPL-licensed framework for debugging and profiling tools that can run under Linux and Mac OS.
- It functions by disassembling the application at run-time, adding instrumentation instructions and then converting back to machine code.
- You can expect Valgrind to result in a slowdown of around 5-100x.

Valgrind

Valgrind comes with a number of tools that use the core framework. A few of them are:

Memcheck Detected invalid memory accesses, use of uninitialised memory, memory leaks, invalid uses of free and other errors.

Callgrind Provides detailled call-graph information, and with the "–simlate-cache" option, estimated values of cache hits/misses and cycle counts.

Massif Produces information on the heap usage of a program.

Helgrind Locates data races in multi-threaded programs. Specifically, it locates values that are accessed by multiple threads that do not appear to have an associated lock.

Memcheck

- Invoking memcheck:

```
$ valgrind --tool=memcheck ./executable
```

- If we compile with the `-g` option to the C compiler, Valgrind will give us line numbers.
- Supplying `--leak-check=full` to Valgrind will give details of individual memory leaks.

An example

The following example overruns the bounds of the heap-allocated region and fails to free it afterwards.

```
#include <stdlib.h>

int main(void)
{
  double *squares = malloc(100 * sizeof(double));

  for(int i = 0; i <= 100; ++i)
    squares[i] = i * i;

  return EXIT_SUCCESS;
}
```


Example Memcheck output

```
==974== Memcheck, a memory error detector
==974== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==974== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==974== Command: ./broken_memory
==974==
==974== Invalid write of size 8
==974==    at 0x40055A: main (broken_memory.c:8)
==974==  Address 0x51f1360 is 0 bytes after a block of size 800 alloc'd
==974==    at 0x4C2B3F8: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==974==    by 0x40052D: main (broken_memory.c:5)
==974==
==974==
==974== HEAP SUMMARY:
==974==     in use at exit: 800 bytes in 1 blocks
==974==   total heap usage: 1 allocs, 0 frees, 800 bytes allocated
==974==
==974== 800 bytes in 1 blocks are definitely lost in loss record 1 of 1
==974==    at 0x4C2B3F8: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==974==    by 0x40052D: main (broken_memory.c:5)
==974==
==974== LEAK SUMMARY:
==974==    definitely lost: 800 bytes in 1 blocks
==974==    indirectly lost: 0 bytes in 0 blocks
==974==      possibly lost: 0 bytes in 0 blocks
==974==    still reachable: 0 bytes in 0 blocks
==974==         suppressed: 0 bytes in 0 blocks
==974==
==974== For counts of detected and suppressed errors, rerun with: -v
==974== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 2 from 2)
```


Table of Contents

Constructing Matrices with Pointers

Allocate Dynamic Memory

```
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?

Another way of Allocating Matrices

Allocate Dynamic Memory

```
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;
}
```


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:

Free Dynamic Memory

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


Matrices utility functions

Let's define some utility functions to:

- Allocate memory for the matrix (`allocMatrix`) - done.
- Free a matrix (`freeMatrix`) - done.
- Print a matrix (`printMatrix`).
- Create a random matrix (`randomMatrix`).
- Add matrices together (`addMatrix`).

printMatrix, randomMatrix and addMatrix

printMatrix

```
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");
  }
}
```

randomMatrix

```
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;
    }
  }
}
```

addMatrix

```
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 < rows; j++){
      matrixR[i][j] = matrixA[i][j]+matrixB[i][j];
    }
  }
}
```


The main function

main not completed..

```
int main(void)
{
  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

Output

```
Enter rows cols: 4 4


matrix A =
 0.84019  0.39438  0.78310  0.79844
 0.91165  0.19755  0.33522  0.76823
 0.27777  0.55397  0.47740  0.62887
 0.36478  0.51340  0.95223  0.91620


matrixB =
 0.63571  0.71730  0.14160  0.60697
 0.01630  0.24289  0.13723  0.80418
 0.15668  0.40094  0.12979  0.10881
 0.99892  0.21826  0.51293  0.83911


matrixA + matrixB =
 1.47590  1.11168  0.92470  1.40541
 0.92795  0.44044  0.47245  1.57241
 0.43445  0.95491  0.60719  0.73768
 1.36371  0.73166  1.46516  1.75531
```


Table of Contents

Header files

- `stdio.h` is one of a number of *header* files defined by the C standard library.
- On Unix-like systems, you can usually find it at the location `/usr/include/stdio.h`. You can open it in a text editor.
- Header files contain information about functions, types and global variables that a library (or other C source files) want to export.
- Header files use exactly the same syntax as C source files, only the ".h" extension distinguishes them as header files.
- We'll look at defining our own headers later.

Splitting Code Across Multiple Files

- Until now, we've only considered how to write programs whose source code resides in a single file.
- When a program is split across multiple files, the compiler compiles each source file *independently*.
- *Headers* provide just enough information about available functions and variables to type-check.
- Care must be taken to avoid accidentally creating duplicate symbols (names of functions or variables).

A simple example

add.h

```c
#ifndef ADD_H
#define ADD_H

int add(int a, int b);

#endif
```

add.c

```c
#include "add.h"

int add(int a, int b) {
  return a + b;
}
```

sum.c

```c
#include "add.h"

int main(void) {
  int sum = add(5, 4);
  return 0;
}
```


Compiling Multiple Source Files

If we assume `add.h`, `add.c` and `sum.c` are in the current directory, we can compile our program as follows:

```
$ gcc -Wall -pedantic sum.c add.c -o sum
```

If we only modify `add.c`, we don't want to have to recompile everything. We can instruct the compiler to create object files (`.o` files) with the `-c` option.

```
$ gcc -Wall -pedantic -c sum.c -o sum.o
$ gcc -Wall -pedantic -c add.c -o add.o
```

These can then be combined into the final executable though a process called *linking*:

```
$ gcc sum.o add.o -o sum
```

Including Headers

- The #include "file.h" directive searches the source file directory first, before looking at other paths.
- The #include <file.h> only looks at predefined paths and paths given to the compiler on the command line.
- All headers should be surrounded by *include guards*. They have the form:

```
#ifndef __SOME_UNIQUE_TOKEN__
#define __SOME_UNIQUE_TOKEN__
/* Your code */
#endif
```

- This prevents the header file content from being included more than once.

Table of Contents

Summary

In this set of slides, we looked at:

- C's pointer types.
- Using the address (&), indirection (*) and array subscript ([]) operators with pointers and values.
- Using arithmetic with pointers.
- Why C arrays are passed by reference.
- The command line arguments.
- String Utilities.
- File Manipulation.
- Makefiles.
- Dynamic memory allocation.
- `perror()` and `exit()`.
- `memset()` and `memcpy()`.