Introduction to Scientific Programming
The C Language
More Pointers
Arrays

• An array in C is a group of elements of the same type.

• Arrays use square brackets like so:

```c
int some_nums[200];
char bunch_o_chars[45];

some_nums[3] = 5;
bunch_o_chars[0] = 'a';
```

In C, we must give the length when we declare the array. *one caveat which we will see later*
Arrays

Passing arrays into functions

• We prototype a function which accepts an array like this:

```c
void process_array (int []);
int calc_array (char[]);
```

• And write the function like this:

```c
void process_array (int all_nums[])
{
    all_nums[1]= 3;
}
```

• And call the function like this:

```c
int some_numbers [100];
process_array(some_numbers);
```
Pointers

Pass by value/reference

• Normally when passing a variable to a function, the compiler makes a **COPY** of the variable in the function.
• Hence changing the value of the argument in the function does not change the original value.
• This is called pass by value.
• Sometimes, like in `scanf()`, we want to change the variable inside the function.
• To do this, we pass a pointer as input argument to the function
• This is called **pass by reference**.
Pointers

Function argument passing by reference

• If we pass a pointer to a variable into a function we can change the value of the variable within the function.

• This is what is going on when we use & in scanf.
Pointers

What are pointers?

- Pointers "point at" areas in your computer's memory.

```c
int *p; /* p is a pointer to an int */
```

What is `++p`?
- 56701014 or 56701017

Addresses
- 56701013
- 56701014
- 56701015
- 56701016

int = 32bits = 4 bytes
Pointers

What are pointers?

- Pointers "point at" areas in your computer's memory.

```c
int *p; /* p is a pointer to an int */
```

What is `++p`?

56701014 or 56701017

Int = 32 bits = 4 bytes
Pointers

Pointer Access

• * on a pointer means “value of”
• & on any variable means “address of”

```c
int *p;
int q = 5;
p = &q;
*p = 10;
```

p is a pointer to an int
q has an int value of “5”
p is assigned the address of q
Therefore *p has the value of “10”
q now has the value of “10”

*see p1.c
Pointers
Common C mistakes

• Remember to assign a valid memory space to the pointer before using it.

```c
int *p;
*p = 5;
```

This will cause problems, we are potentially writing to some invalid Address in memory.
Pointers

Allocating memory

• We can allocate memory with the `malloc` function:

```c
p = (int *)malloc(sizeof(int));
```

Cast to appropriate type

`sizeof(int)` returns how Many bytes an int type requires

```c
*p = 5;
```
Pointers and NULL

• NULL pointer is a reserved value representing no object.
• In many implementations it is the value “0”, but it is not necessary to be so.
• Good practice to initialize pointers to NULL.

```c
int *p = NULL;
if (p == NULL) {
    fprintf(stderr, "p points to no object\n");
}
```
Pointers and Memory

- Declare a pointer with an `*`
- Use an `&` to get the "address of" (and convert a variable to a pointer)
- Use an `*` to get the value "pointed at"

```c
int *p;
int q = 5;
p = &q;
*p = 6;
```

- `int *p;`  
  Pointer to integer
- `int q = 5;`  
  `q` assigned with value 5
- `p = &q;`  
  Variable `p` points to `q`'s memory
- `*p = 6;`  
  `q` now has value 6
Pointer and Arrays

- In C, pointers are often used to manipulate arrays.

```c
//p2_arrays.c
int *p;
int q[7];

p = &q[0];
*p = 3;
p++;
*p++ = 4;
p++;
*p++ = 5;

printf("q[0] %d, q[1] %d, q[2] %d\n",q[0],q[1],q[2]);
```

Array of 7 integers

- `p` points to memory location of `q[0]`
- `q[0] == 3`
- `q[1] == 4`
Pointers and char arrays

- A pointer to a char array is a common way to refer to a string:

```c
char *string = NULL;

string = (char *)malloc(strlen("Hello") + 1) * sizeof(char));

strcpy(string, "Hello");
```

Hello
Arrays of pointers

• We can declare an array of pointers like so:

```c
char *name[] = {"John", "Jay", "Joe"}; /* Creates and initialises 3 names */
```

We can now use the array elements anywhere we could use a string.
Example

```c
int i;
char *people[] = {"John", "Jay", "Joe"};
for (i = 0; i < 3; i++) {
    printf ("String %d is %s\n", i, people[i]);
}
```

Output:

String 0 is John
String 1 is Jay
String 2 is Joe
Structures

- Structures are a way of constructing higher-level types.
  E.g.

  ```c
  struct coordinate {
    int x;
    int y;
  } var;
  ```

  Dot notation "." accesses members in struct

  ```c
  var.x = 1;
  var.y = 2;
  ```

  printf("structure size = %d\n", sizeof(struct coordinate));
Pointers and structures

```
struct coordinate {
    int x;
    int y;
} *var;

var = (struct coordinate *)malloc(sizeof(struct coordinate));
var->x = 1;
var->y = 2;
```

- Variable is a pointer to struct coordinate
- Allocate memory of correct size
- Arrow notation “->” accesses members
Pointers and functions

• Functions can return arrays as pointers:

```c
int *foo(void)
{
    static int array[100];
    ...
    return array; // Returns array to invoker
}
```

Why is this okay?
Pointers and functions

- C passes by value, so there is no direct way for the called function to alter a variable in the calling function

```c
swap(a,b);
.
.
.
void swap(int x, int y) /* WRONG */
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```
Pointers and functions

- Because of call by value, swap cannot affect the arguments a and b.
- We must use pointers!

```c
void swap(int *px, int *py) /* interchange *px and *py */
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
```
Pointers and functions

- Functions can also return structs as a pointer:

```c
struct coordinate *foo(void)
{
    struct coordinate *tmp;
    tmp = (struct coordinate *)malloc(sizeof(struct coordinate));
    return tmp;
}
```

Returns heap allocated struct to invoker
Memory model

• Properties of variable determined by type, scope and life-time

• Life-time of a variable:
  – Static: starts and ends with program
    • Global and static variables
  – Automatic: starts and ends in block
    • Local and stack allocated variables
  – Heap: starts when allocated, ends with freed
    • Dynamically allocated variables
Memory model

- **Static data**: Pre-allocated at compile time.
- **Automatic data**: Variables are allocated on stack as blocks are entered and de-allocated as blocks leave. Usually grows downwards (address-wise).
- **Heap data**: Variables are allocated in area at the bottom of the address space. May be organized specifically to optimize re-use as life-time of variables can be arbitrary.
Automatic memory allocation

```c
void foo(void)
{
    int local_var = 1;
    printf("Local=%d\n", local_var++);
}

foo();
foo();
```

Allocated at function startup and visible only in block

Output:
Local=1
Local=1
int g_var1 = 1;
static int g_var2 = 1;
void foo(void)
{
    static int s_var = 1;
    printf("Local=%d, Global=%d\n", s_var++, g_var2++);
}
foo();
foo();
en.wikipedia.org/wiki/Static_variable
Allocating memory on heap

- We can allocate memory on the heap with the `malloc()` function:

```c
p = (int *)malloc(sizeof(int));
```

- `sizeof(int)` returns how many bytes an int type requires.
- Cast to appropriate type
Resizing memory on heap

• We can resize memory on the heap with the `realloc()` function:

```c
int *p;
double *q;

p = (int *)malloc(sizeof(int));

q = (double *)realloc(p, sizeof(double));
```

Previously allocated heap storage

New size
Freeing memory on heap

You can free memory on the heap with the `free()` function:

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

int *p = (int *)malloc(sizeof(int));

/* use pointer p */
free(p);
```

Function prototype in `stdlib.h`
Allocate memory of size of integer
After use ... free it
Dynamic automatic memory allocation

```c
void foo(void)
{
    int *local_var = (int *)alloca(sizeof(int));
}
```

Memory space automatically de-allocated when execution leaves block (free is not necessary)
char *foo(void)
{
    char *string = "hello world";
    return string;
}

char *mystring;
mystring = foo();
...
printf("mystring = %s\n", mystring);
Common mistake (2)

void foo(int *data) {
    ...
    free(data);
}

int *mydata = (int *)malloc(sizeof(int));
foo(mydata);
free(mydata);
Heap storage free’d twice!
Common mistake (3)

```c
void foo(void)
{
    int *pt = (int *)malloc(sizeof(int));
    ...
    return;
}

foo();
foo();
foo();
```

Memory Leak! May eventually crash.
Allocate a 2-D array

```c
int **A;
int A_rows = 3;
int A_cols = 2;

A = malloc(A_rows * sizeof(int *));
If(A == NULL){
    printf("cannot allocate memory!\n");
    exit(-1);
}

for(i=0; i< A_rows; i++){
    A[i] = malloc(A_cols * sizeof(int));
}
```
A function using a 2-D array argument

//prototype
void 2d_func(int **A);

//function define:
void 2d_func(int **A){
    printf("the value at 0,0 is:  %d\n", A[0][0]);
}

De-allocate a 2-D array

```c
//free allocated memory when finished
for(i=0; i<A_rows; i++){
    free(A[i]);
}
free(A);
```
Homework 6
Using Dynamic Memory allocation & Functions

• Write a program that performs a matrix multiply inside of a function.
• You will have four functions in this program, main(), check(), matmul(), and printResult().
• In the main function you can initialize the matrix by reading values from the keyboard or you can hard-code the values.
• Use the check() function to determine if the two matrices can be multiplied, exit the program if they cannot.
• Pass your matrixA and matrixB to the function matmul(), which will perform the matrix multiply then print the result using the printResult() function.