An Introduction to HPC and Scientific Computing

Lecture five: A deeper dive into C programming

Wes Armour

Oxford e-Research Centre,
Department of Engineering Science

Overview

In this lecture you will learn about:

- More on arrays.
- Multidimensional arrays.
- An introduction to pointers.
- Characters and strings.
- Variable scope.
- Advanced program control.
- How to work with files.
- Dynamic memory allocation.

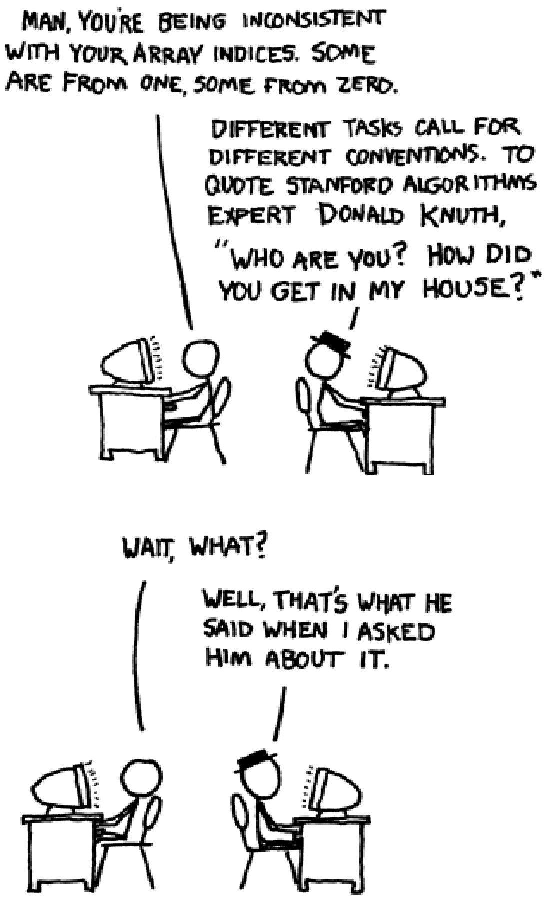
Using Arrays

In lecture two we presented the concept of an array and how to use them. Now lest look a them in some more detail. What characterises an array.

- An array is a collection of data storage locations.
- An array hold data all of the same type.
- Each storage location is called an *array element*.
- C arrays are indexed from 0 to n-1, where n is the number of elements in the array.
- Arrays can be initilaised using braces:

```
int array[3] = {1,3,5};
```

As demonstrated in our first practical arrays are a useful way to organise variables in your program.



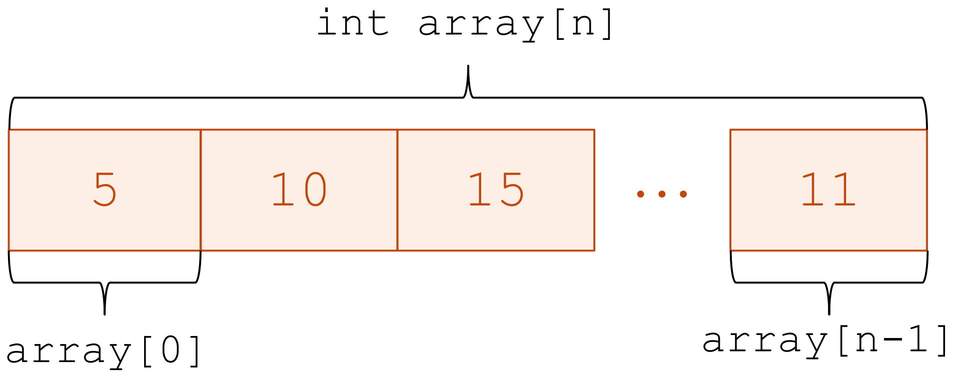
https://xkcd.com/163/

Using Arrays – single dimensional

In practical one we used an array called radius to hold different values for the radius of a circle. This can be represented schematically below.

We have a *contiguous* collection of array elements, starting at zero, increasing to n-1, all holding a single value.

`array[0]` holds the integer value 5, `array[1]` holds the integer value 10, up to `array[n-1]` which holds the integer value 11.



Using Arrays – multidimensional

We can see that arrays can be very useful in helping us create compact and easy to read code. But what about if we want to store something that has more than a single dimension?

Fortunately C has the concept of multidimensional arrays. Using these we can store an entitly that has any dimension.

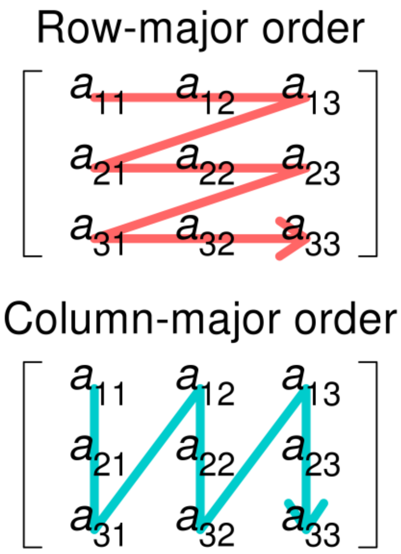
Lets consider the 3x3 identity matrix (right). We can store this as a 2D array which we declare as:

```
int identity[3][3]
```

In C this would tell the compiler to create a linear contiguous area in memory to hold 3x3 = 9 ints.

C uses row-major ordering. What does this mean?

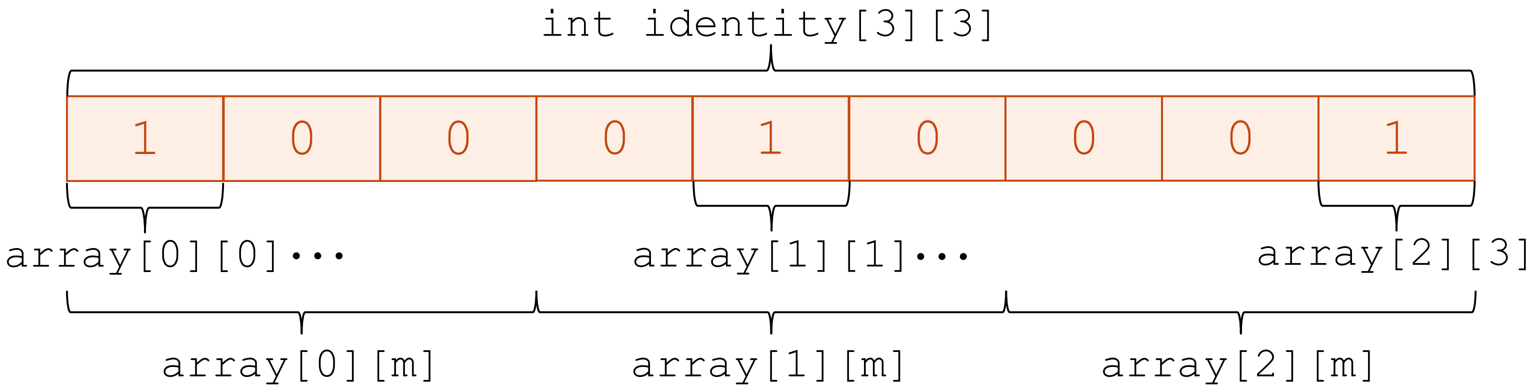
$$I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$



Using Arrays – multidimensional

Row-major ordering means that consecutive elements within a row reside next to each other in memory. Rows are then stored (in order) in memory consecutively.

Lets return to our example of the 3x3 identity matrix (below). We see the first element of the first row is stored first, followed by the second element of the first row, and so on. Then the first element of the second row is stored, followed by the second element of the second row etc.



Using Arrays – initialising

Finally – how do we initialise these arrays in our code?

Previously we showed that a one dimensional array can be initialised as follows:

```
int array[3] = {1,3,5};
```

We can initialise a 2D array in the following way:

```
int identity[3][3] = {{1,0,0},
                      {0,1,0},
                      {0,0,1}};
```

This can be generalised. For example below is the initialisation for a 3D array which has dimensions p x q x r (I've omitted actual values in an attempt to make this clearer).

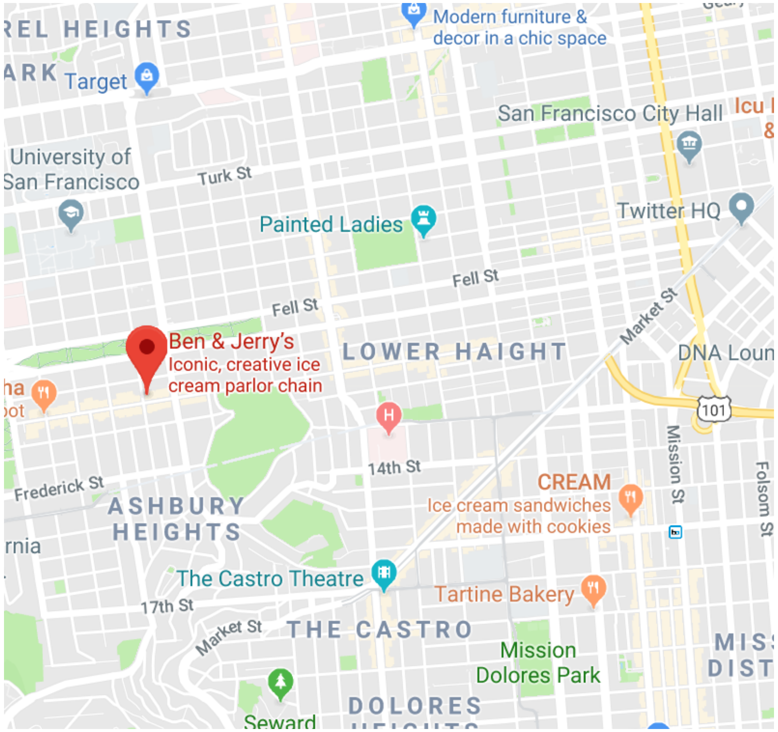
```
int array[p][q][r] = {{{},{},{}},
                      {{},{},{}},
                      {{},{},{}}};
```

Understanding pointers - memory locations

Our previous example of arrays depicted a computers memory (RAM) as a sequence of linear, contiguous storage elements. We looked at how data items are stored in each element.

But how does a computer know in which element the data it wants to access is stored? For example how does it know where `identity[0][0]` is located?

This problem is overcome just as it would be in the real world. Each memory location is given a unique address. This is a simplified view of the actual mechanics of how memory is addressed, however it is sufficient for our needs.

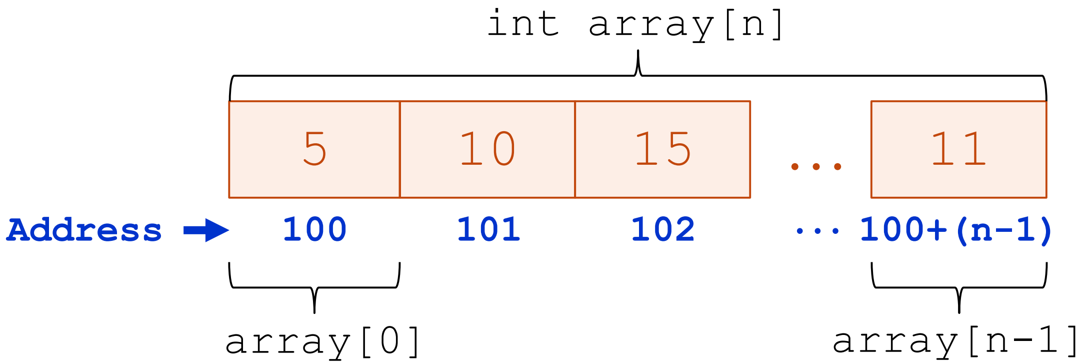


Understanding pointers – addressing memory

Lets go back to our previous description of an integer array.

But now lets add an address. I've chosen to add an integer address where the beginning element our `array` which is `array[0]` has address **100** (again this is simplified, but its sufficient to understand pointers).

Schematically this is represented on the right.



Understanding pointers – creating a pointer

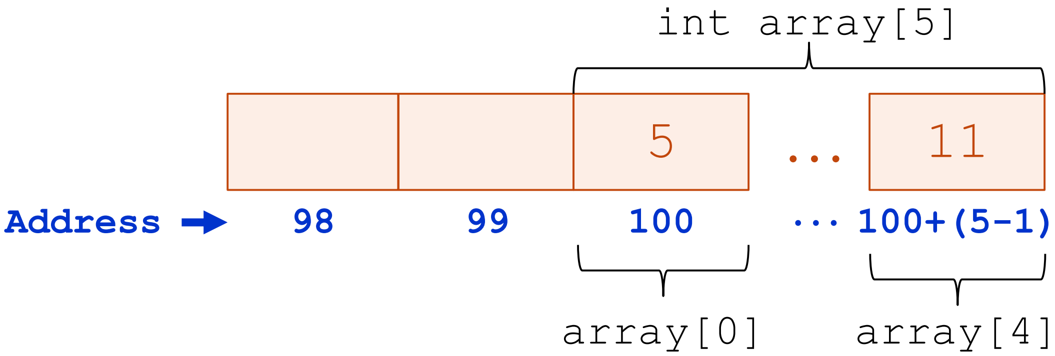
The next thing to note is that each address is a number and so can be treated like any other number in C.

If we know the address of `array[0]` then we could create another variable to store this address.

Lets work through the process for doing this over the next few slides.

Understanding pointers – creating a pointer

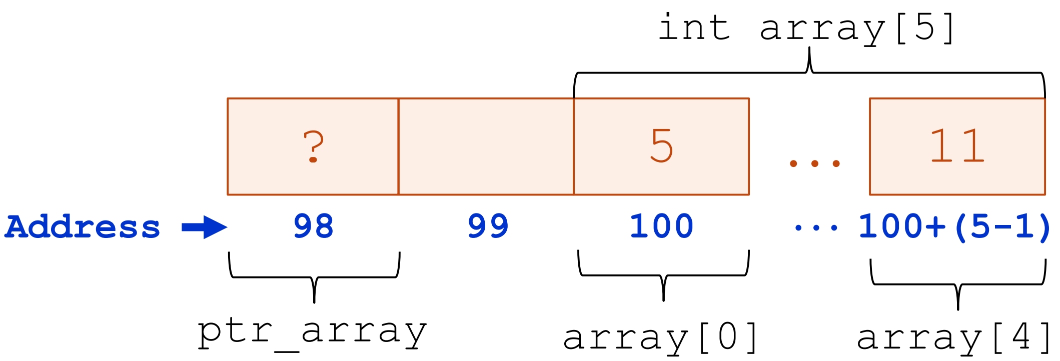
As before we declare our integer array, let's declare space for 5 integers (so n=5)



Understanding pointers – creating a pointer

Next we declare a variable (called `ptr_array`) that happens to live at address 98.

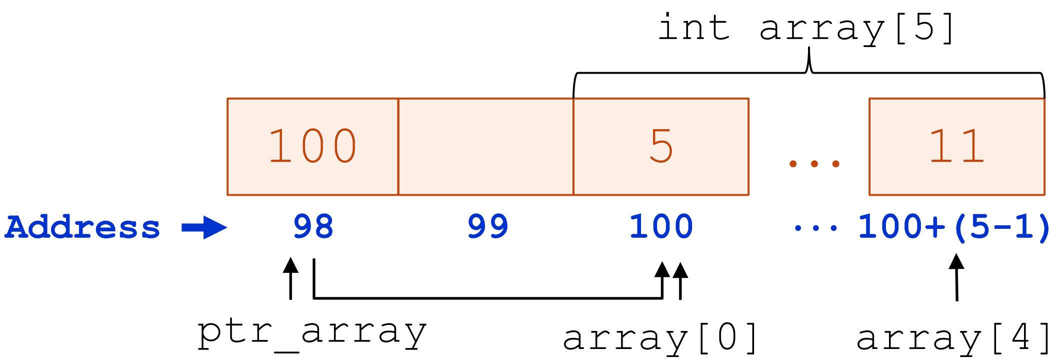
At this point it is uninitialized, so its value is undetermined.



Understanding pointers – creating a pointer

Now we store the address of `array[0]` in the variable `ptr_array`

Because `ptr_array` contains the address of `array[0]` it points to where `array` is stored in memory.



Hence `ptr_array` is a pointer to `array`

Understanding pointers – using pointers

To work with pointers we need to know about two operators. These are:

The indirection operator `*`

The address-of operator `&`

To understand these operators lets return to our simple example of calculating the area of a circle.

We declare a pointer to type float by:

```
float *ptr_radius;
```

The indirection operator tells the compiler that `ptr_radius` is a pointer to type float and not a variable of type float.

```
#include <stdio.h>

#define PI 3.14

int main() {

    float radius = 10.0;
    float area;

    float *ptr_radius;

    area = PI * radius * radius;

    return(0);

}
```

Understanding pointers – using pointers

To initialise the pointer (set it to point to something) we use the address-of operator:

```
ptr_radius = &radius;
```

This tells the compiler to take the address of the variable `radius` and store it in the pointer `ptr_radius`

The code on the right shows how this works and tests the results of using a pointer.

```
#include <stdio.h>

#define PI 3.14

int main() {

    float radius = 10.0;
    float *ptr_radius;
    float area;

    area = PI * radius * radius;
    printf("\nArea:\t%f", area);

    area = 0;
    ptr_radius = &radius;
    area = PI * (*ptr_radius) * (*ptr_radius);
    printf("\nArea with pointer:\t%f", area);

    return(0);
}
```

Understanding pointers – using pointers

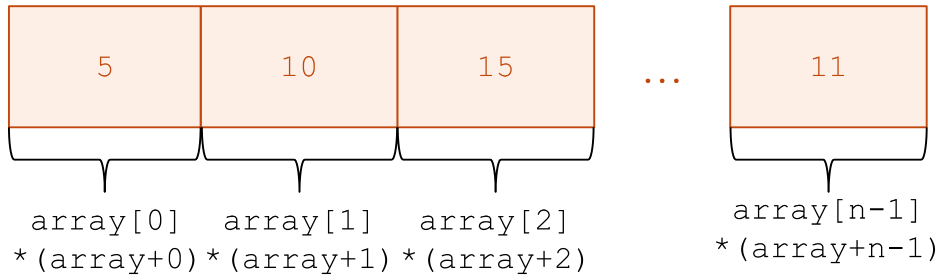
We can use arithmetic on pointers, just like we can any other numbers. A schematic demonstrating this is given below.

We also have increment: `ptr_radius++;`

and decrement operators: `ptr_radius--;`

Finally we can pass pointers as arguments to functions:

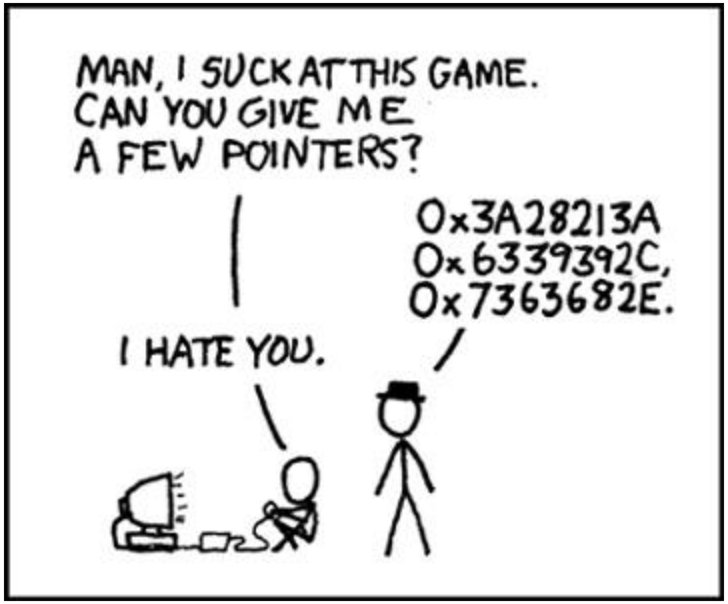
```
float area_of_circle(float *ptr_radius);
```



Understanding pointers

Pointers are (arguably) the most difficult concept in C to understand. However they are a powerful tool that can be used to write versatile and concise code. They also provide a flexible method for data manipulation.

In “Practical examples using the C programming language” we will look at the uses of pointers in more detail.



https://xkcd.com/138/

Characters and Strings

C uses the char variable to store characters and strings.

C uses ASCII encoding to turn integer numbers into characters. An example of this is given on the right.

C decides whether a char holds a character or a number depending on the context of its use.

```
#include <stdio.h>

int main() {

     char one = 70;
     char two = 'q';

     printf("\none as a character:\t%c", one);
     printf("\none as a number:\t%d", one);

     printf("\none as a character:\t%c", two);
     printf("\none as a number:\t%d", two);

     return(0);
}
```

https://en.wikipedia.org/wiki/ASCII

Characters and Strings

C uses arrays of char variables to store strings.

Stings are terminated with the null character which is represented by \0

So a string that has seven characters needs an array of eight elements to store it.

The code on the right gives an example of this and demonstrates two ways to initialise a character array with a string.

Recall the conversion specifier for a string is %s

```
#include <stdio.h>

int main() {

    char one[5] = {'H','a','r','d'};
    char two[5] = "Easy";

    printf("\nString one:\t%s", one);
    printf("\nString two:\t%s", two);

    return(0);
}
```

Characters and Strings

You can also allocate storage space for your string at compile time. To do this use one of the two ways demonstrated in the code on the right.

```
#include <stdio.h>

int main() {

    char one[] = {‘H’,’a’,’r’,’d’};
    char *two  = “Easy”;

    printf(“\nString one:\t%s”, one);
    printf(“\nString two:\t%s”, two);

    return(0);
}
```

Characters and Strings

For a user to interact with your program they need to be able to pass it input. C has two methods to read strings from the keyboard.

The first is the gets() function. This simply reads all input to the keyboard until a user presses the Enter key.

The second is the scanf() function, this requires the programmer to specify the format of the input using conversion specifiers.

Both methods are demonstrated in the code on the right.

```
#include <stdio.h>

int main() {

    char one[256];
    char two[256];
    char three[256];

    int count;

    printf("\nType some text and press Enter:\n");
    gets(one);
    printf("\nYou typed:\t%s", one);

    printf("\nType two words and press Enter:\n");
    count = scanf("%s%s", &two &three);
    printf("\nYou enterd %d words:\t%d", count);
    printf("\nYour words are: %s and %s", two, three);

    return(0);
}
```

Variable scope – Global variables

Variable scope refers to the extent to which different parts of your C program can "see" a variable that you declare.

The concept of scope allows a programmer to truly separate out (structure) their code into independent self contained routines or functions.

Doing this helps reduce bugs in code and makes for more reusable code. For example if a variable can only be seen by the function that is operating on it another function cannot mistakenly corrupt its value.

In some instances it is desirable to share a variable amongst the whole code. This can be done with the `extern` keyword.

```
#include <stdio.h>

void print_number(void);

float one = 3.0;

int main() {

    extern float one;

    print_number(void);

    return(0);
}

void print_number(void) {

    extern float one;

    printf("\nYour number is:\t%f\n", one);
}
```

UNIVERSITY OF OXFORD

Oxford e-Research Centre

Variable scope – Local variables

External variables are sometimes called global variables. Their scope is the whole program, so `main()` and any other `functions()` that you define.

This is opposite to *local variables*. A local variable is defined within a function. As such its scope is within the function (remember `main()` is a function and so we can have local variables in `main()`).

Local variables are *automatic*, meaning they are created when the function is called and destroyed when it exits. So an automatic variable doesn't retain its value in between function calls.

To remember the value of a variable between function calls we can use the *static* keyword.

```
#include <stdio.h>

void print_number(int x);

int main() {

    for(int x=0; x<3; x++) {
        print_number(x);
    }
    return(0);
}

void print_number(int x) {

    static int y = 0;
    printf("\nx,y are:\t%d %df\n", x, y);
    y--;
}
```

Advanced program control

C provides some additional tools for advanced program control.

Three of the most useful are:

- `break`
- `continue`
- `switch`

Both `break` and `continue` provide additional control within loops (in fact they can only be placed within the body of a `for()` or `while()` loop).

The `switch` statement takes an argument and then executes code based on this.

```
#include <stdio.h>

void print_number(int x);

int main() {

    for(int x=0; x<5; x++) {
        print_number(x);
        if(x == 2) break;
    }

    int x = 0;
    while(1) {
        if(x == 3) {
            break;
        }
        print_number(x);
        x++;
    }
    return(0);
}

void print_number(int x) {
    static int y = 0;
    printf("\nx,y are:\t%d %df\n", x, y);
    y--;
}
```

Advanced program control

An example of how the `switch` statement can be used is given on the right.

```
#include <stdio.h>

void print_number(int x);

int main() {

    int choice;
    printf("\nEnter a choice: 1,2 or 3 to exit: ");
    scanf("%d", &choice);

    switch(choice) {
        case 1:
            printf("You entered 1");
            break;
        case 2:
            printf("You entered 3");
            break;
        case 3:
            printf("You entered 3. Exiting.");
            exit(0);
    }
    return(0);
}
```

Using files

C provides a number of different tools for interacting with files stored on your computer.

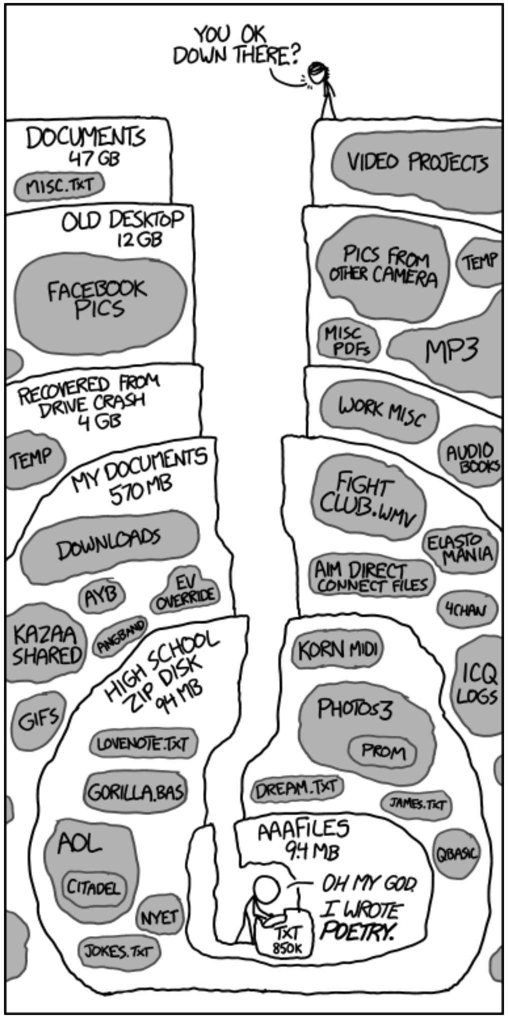
In this lecture we'll cover four basic functions for this.

fopen()

fclose()

fprintf()

fscanf()



https://xkcd.com/1360/

Using files – fopen and fclose

The code on the right declares a pointer to type FILE. Then two char arrays are declared to hold a file name and a mode.

The file name is the name of the file you want to use and the path to it.

We use `gets()` to read strings from the keyboard and store these in our char arrays.

We then use `fopen()` to try and open the requested file. If the file opens successfully then the pointer fp is initialised. If not the pointer is set to NULL and we print an error.

Finally we close the file using the `fclose()` statement.

```
#include <stdio.h>

int main() {

    FILE *fp;
    char filename[200], mode[4];

    printf("\nEnter your file: ");
    gets(filename);
    printf("\nEnter a file mode: ");
    gets(mode);

    if((fp=fpoen(filename, mode)) != NULL) {
        printf("\nOpened %s in mode %s", filename, mode);
    } else {
        printf("\nERROR: File not recognised");
    }
    fclose(fp);
    return(0);
}
```

Using files – fopen and fclose

In the previous slide we introduce the concept of opening a file in different *modes.*

The table on the right outlines different modes and the and the associated return value of `fopen()`

Mode	Meaning	Return value from fopen if the file:	
		Exists	**Doesn't exist**
r	Reading	–	NULL
w	Writing	Overwrite if file exists	Create new file
a	Append	New data is appended at the end of file	Create new file
r+	Reading + Writing	New data is written at the beginning of the file overwriting existing data	Create new file
w+	Reading + Writing	Overwrite if file exists	Create new file
a+	Reading + Appending	New data is appended at the end of file	Create new file

Using files – fprintf and fscanf

The code on the right gives examples of using the `fprintf()` and `fscanf()` functions (it's a bit squashed).

We open a file as before (note the bad coding practice, the code fails silently if `fopen()` fails).

We use `fprintf()` to print 10 numbers and their squares to a file. Note how `fprintf()` works like `printf()`

We close the file and reopen it.

We then use `fscanf()` to read in our outputted numbers.

Finally we perform a difference of these and print out to screen.

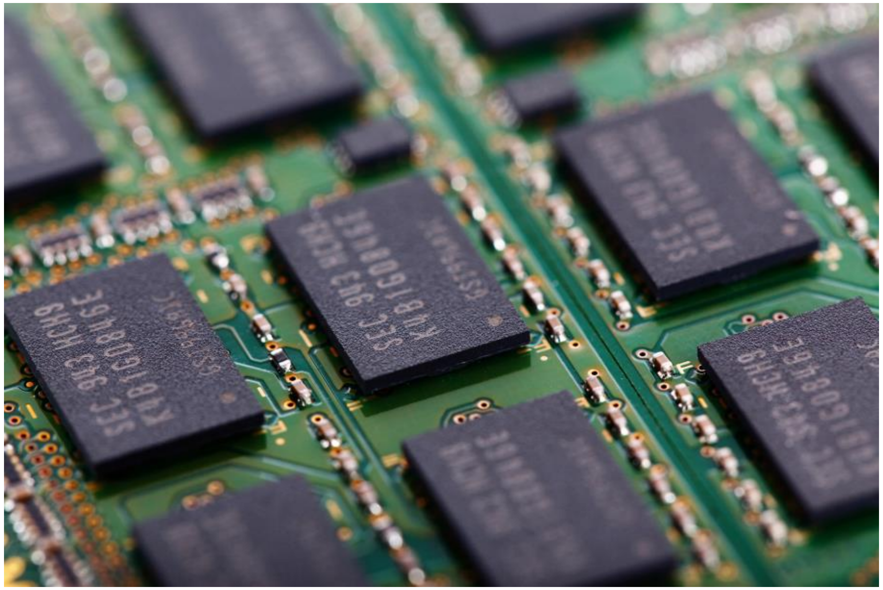
```
#include <stdio.h>

int main() {

    FILE *fp;
    char filename[200], mode[4];
    int index[10], square[10];

    printf("\nEnter your file: "), gets(filename);
    printf("\nEnter a file mode: "), gets(mode);
    if((fp=fpoen(filename, mode)) == NULL) exit(1);

    for(int i=0; i<10; i++) {
        fprintf(fp, "%d %d", i, i*i);
    }

    fclose(fp);
    fp=fpoen(filename, mode);

    for(int i=0; i<10; i++) {
        fscanf(fp, "%d %d", &index[i], &square[i]);
    }
    for(int i=0; i<10; i++) {
        fprintf(fp, "%d %d", i-index[i],(i*i)-squared[i]);
    }
    return(0);
}
```

Working with memory

Up until now all of our example codes have allocated *static memory.*
By this we mean that when we write our code we declare how much memory we need for particular variables or arrays.

However there might be instances where we don't know how much memory we need until we run the code. For example we might want to read a file into our code that is updated on a daily basis, the files length might be different on different days.

C provides a process for allocating memory at *runtime*. This is called *dynamic memory allocation*.



https://en.wikipedia.org/wiki/C_dynamic_memory_allocation

Working with memory

Lets return to our example code from Lecture two. We can modify this code so that is askes the user how many areas they want to calculate and then dynamically allocates memory for them.

We start by declaring two pointers:

`float *radius;`

We then use `scanf()` to get the number of circles the user wants to work with.

We then use `malloc()` to allocate the memory that we need.

After this we use `scanf()` to get the radii from the user.

This time we accumulate directly to `total_area`

Finally we `free()` our allocated memory.

```
#include <stdio.h>
#include <stdlib.h>
#define PI 3.14

float area_of_circle(float radius);

int main() {
    int i=0, number_of_circles=0;
    float *radius;
    float total_area=0;

    printf("\nEnter the number of circles to calculate:\t");
    scanf("%d", &number_of_circles);

    radius=(float *)malloc(number_of_circles*sizeof(float));

    printf("\nEnter the radii:\n");
    for(i=0; i<number_of_circles; i++) scanf("%f", &radius[i]);

    i=0;
    while(i<number_of_circles) {
        total_area += area_of_circle(radius[i]);
        i++;
    }
    printf("\nTotal area is:\t%f\n", total_area);

    free(radius);
    free(area);
    return(0);
}

float area_of_circle(float radius) {
    float area = PI * radius * radius;
    return area;
}
```

Working with memory – malloc()

Lets take a closer look at `malloc()`

Both `malloc()` and `free()` are defined in `stdlib.h`, so this must be included into your code to use them.

The function prototype for `malloc()` is:

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

`size_t` is an unsigned integral type and is used to represent the size of an object in bytes. The return type of the `sizeof()` operator is `size_t`

`malloc()` will return NULL if `num` bytes cannot be allocated (for example the computer doesn't have enough memory space left)

$$\overbrace{\mathrm{m}}^{\text{memory}}\underbrace{\mathrm{alloc(size)}}_{\text{allocation}}$$

Working with memory – free()

Once we've finished working with the memory that we've allocated using `malloc()` we should free it so that it can be used again.

This is done using the `free()` function. Its function prototype is:

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

Calling `free()` releases the memory that is pointed to by `ptr`

Working with memory – multidimensional arrays

Finally lets return to our identity matrix.

`int identity[3][3]`

How can we allocate memory for this using `malloc()`?

The code snippet on the right shows how to do this and then free the allocated memory using `free()`

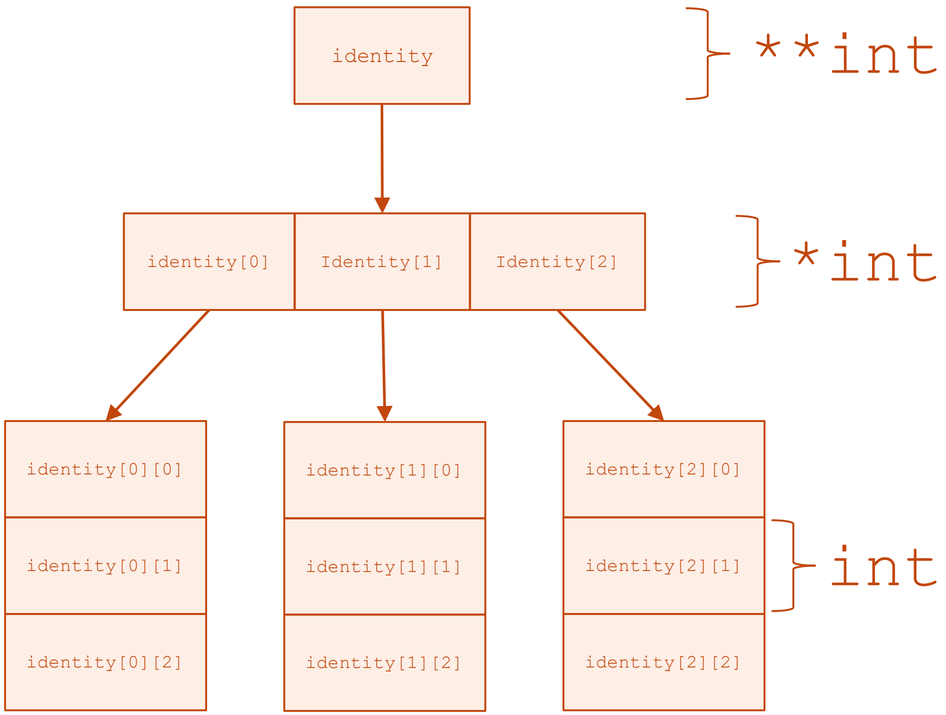
```
int ** identity;

int num_rows = 3;
int num_cols = 3;

// To allocate

identity = (int **)malloc(num_rows * sizeof(int*));
for(int i=0; i<num_rows; i++) {
    identity[i] = (int *)malloc(num_cols * sizeof(int));
}

// To free

for(int i=0; i<num_rows; i++) {
    free(identity[i]);
}
free(identity);
```

Working with memory – multidimensional arrays

Schematically this can be represented by the diagram on the right.

We begin by allocating a double pointer to type `int**` This in turn points to an array of pointers of type `int*`

We then point each of these at a single dimensional array of type `int`.



What have we learnt?

In this lecture you have learnt about some of the advanced features of the C programing language.

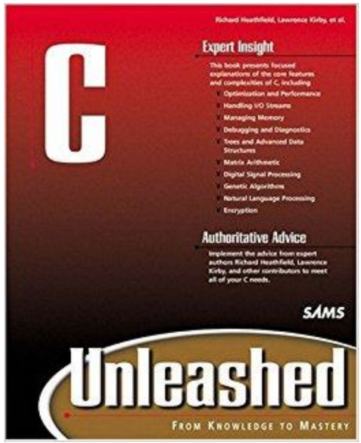
We have covered multidimensional arrays, pointers and characters and strings.

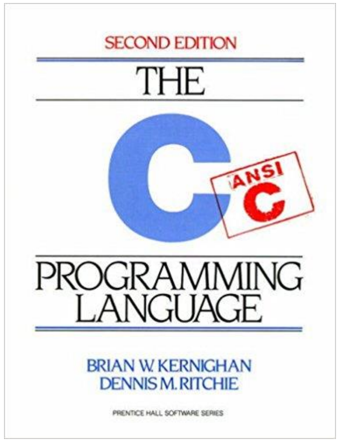
You have learnt about variable scope, some of the functions that provide advanced program control and how to work with files.

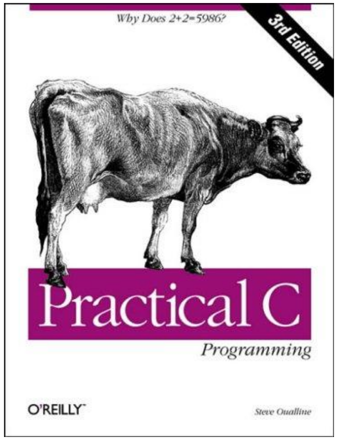
Finally we have covered the basics of dynamic memory allocation.

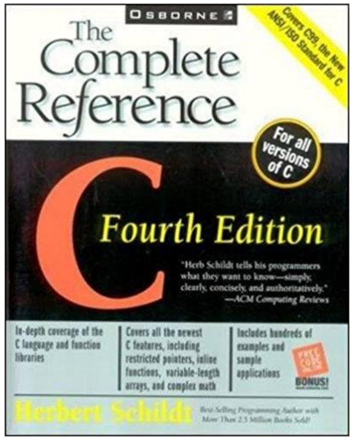
You should now be in a position to write your own advanced C program.

Further reading









http://www.learn-c.org/

https://www.cprogramming.com/tutorial/c-tutorial.html

https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html

In the next lecture…

In the next lecture you will learn about multi-tasking on CPUs using OpenMP. You will learn about parallelism and concurrency. You will get to know OpenMP and how to use it to share work across cores in a CPU and across CPUs in a server.

UNIVERSITY OF OXFORD

Oxford e-Research Centre