Overview

- This lecture will:
  - extend the complexity of the C you know;
  - introduce the use of pointers;
  - show how to use pointers for basic vector algebra.
- This will be achieved using practical examples.
- Has some work you must undertake.

Pointer Variables

- A pointer variable contains the address of a piece of memory.
- Assign address with `&` operator:

```
px = &x;
```

- To 'dereference' it, the $*$ operator is used:

```
y = *px;
```

- A pointer variable is declared in the following way:

```
type *var_name;
type* var_name;
```

Simple Use of Pointer Variables

- Example of modifying arguments:

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

- The function call is

```
int x, y;
swap(&x, &y);
```

- Returning multiple values:

```
int x, y;
/* Values of both x and y can be altered by foo */
foo(&x, &y);
```

Arrays

- Pointers and arrays are two ways of viewing one language construct. Array indexing is equivalent to pointer arithmetic.

```
int a[10];
int *pa;
pa = &(a[0]); /* pa and a[0] reference same location */
x = *pa;
x = a[0];
```

- `pa+i` points to `a[i]`.
- This does not depend on the type of `a`.
- String constants are arrays of `char` locations with an additional element at the end containing the null character `\0` (which always has the value 0).

Arrays and Functions

- When an array name is passed to a function, it is the address of the start of the array that is passed.

```
foo(type *name1)
bar(type name2[])
```

- Inside the function

```
x = *(name1+3);
x = name2[3];
```

- Both versions are valid and this gives room for misunderstanding.

Pointers in use – algorithmic complexity.

- Recall (or learn) that we can write the algorithmic (memory or computational) complexity of an algorithm in terms of the order, $O()$, of the number of operations required to process n elements.
- For example $O(n)$ represents linear growth.
- In the labs you will look at code to print a table of the growth in complexity with increasing n.
- This uses one dimensional arrays, and calls a function to print the table passing in the arrays.

Writing a function in C

- Write a function which takes in two vectors, `v1`, `v2` (of length 3), adds them together and returns the result in a third vector called `sum`.
- You can assume that the following header exists:

```
#include <stdio.h>
#include <math.h>
#define VLENGTH 3
typedef float Real;
typedef Real Coordinate;
typedef Coordinate* Vector;
void addVector(Vector v1, Vector v2, Vector sum);
```

Writing a function in C

- Your result should look something like:

```
void addVector(Vector v1, Vector v2, Vector sum)
/* Add two vectors together and return result in sum. */
{
  int i; /* index variable for loop. */
  /* The sum of two vectors is the sum of the elements. */
  for (i=0; i<VLENGTH; i++) {
    sum[i] = v1[i]+v2[i];
  }
  return;
}
```

- There are other versions possible ....
- The call would be `addVector(vec1,vec2,vec3)` for example.

Pointers in use – working with vectors.

- Vectors are fundamental to computer graphics – typically represent vertices or normals. First consider the header:

```
#include <stdio.h> #include <math.h>
#define VLENGTH 3

/* Define a coordinate type to store x,y and z  */
typedef float Real;
typedef Real Coordinate;
/* Define a Vector as a pointer to a coordinate -
   a pointer to first element of array of coordinates.
   Recall that the type Vector is a pointer. */
typedef Coordinate* Vector;

/* Function prototypes */
void printVector(Vector vec);
/* We could equally well write this function header:
     void printVector(Coordinate *vec);
   the type Vector is equivalent to Coordinate* */
void crossProduct(Vector vec1, Vector vec2, Vector vec3);
void scalarTimesVector(Vector vec1, Real scale);
Real vectorLength(Vector vec);
```

Working with vectors.

- The main function is

```
int main(void)
{
  /* Declare three vectors and initialise the first two  */
  Coordinate v1[VLENGTH] = \{1.0,0.0,0.0\};
  Coordinate v2[VLENGTH] = \{0.0,1.0,0.0\};
  Coordinate v3[VLENGTH];
  Real norm; /* To store the length (norm) of a vector. */

  /* Display the initial vectors */
  printf("Vector v1 is: ");
  /* When we call printVector we pass a Vector,
     i.e. a pointer to a Coordinate
     - first element of the Vector v1 in this case. */
  printVector(v1);
  printf("Vector v2 is: ");
  printVector(v2);
```

```
  /* Now compute the cross product v1 x v2, place in v3 */
  crossProduct(v1,v2,v3);

  /* Display the result */
  printf("The cross product result is: ");
  printVector(v3);

  /* Compute the norm of the vector and display it. */
  norm = vectorLength(v3);
  printf("This vector has length: %3.1f \n",norm);

  /* Display the normalised cross product */
  scalarTimesVector(v3,(Coordinate) 1.0/vectorLength(v3));
  printf("The  normalised cross product result is: ");
  printVector(v3);

  return 0; /* ANSI C requires main to return an int. */
}
```

Working with vectors – functions.

Two functions from the program:

```
void crossProduct(Vector vec1, Vector vec2, Vector vec3)
{
  /* The cross product (x1,y1,z1) x (x2,y2,z2) is:
     (y1z2 - z1y2,z1x2-x1z2,x1y2-y1x2). */
  /* Recall C arrays are indexed from 0 not 1,
     and that vec1[0] is the same as *vec1 */
  vec3[0] = vec1[1] * vec2[2] - vec1[2] * vec2[1];
  vec3[1] = vec1[2] * vec2[0] - vec1[0] * vec2[2];
  vec3[2] = vec1[0] * vec2[1] - vec1[1] * vec2[0];
  return;
}
```

Working with vectors – functions.

and

```
Real vectorLength(Vector vec)
{
  /* The length of a vector (x1,y1,z1) is:
     sqrt(x1^2 + y1^2 + z1^2). */
  int i;
  Real length = 0.0; /* Requires initialisation */

  /* Recall C arrays are indexed from 0 not 1 */
  for (i=0;i<VLENGTH;i++) {
    /* Add the squares of the elements together. */
    length += vec[i]*vec[i];
    /* Equivalent to: length = length + vec[i]*vec[i] */
  }
  return (Real) sqrt((double)length);
}
```

Multi-dimensional Arrays

- C does allow the declaration of fixed size two dimensional arrays with the following syntax:

```
type name[num_rows][num_columns];

int a[3][2];
/* stored in order */
a[0][0] a[0][1] a[1][0] a[1][1] a[2][0] a[2][1]
```

- Arrays are stored in row major form, so the rightmost index varies the fastest.
- As we will see OpenGL expects arrays in column major format so we have to beware.

Pointers to Functions

- Sometimes useful to pass functions as arguments in function calls – C uses pointers to functions.
- It can be a little confusing and you really don't need to know much other than it is possible.
- This is used in GLUT– see the labs, where you simply need to use the template provided.

Summary

- Having finished this lecture you should:
  - understand the role of pointers in – to variables (and functions);
  - be able to use pointers in your programs;
  - be able to write more complex C programs (but this is not a programming course);
  - know a little about algorithmic complexity and vector algebra.
- Although this is all the C I will formally teach, in the labs you will gain a lot more experience.