OUTLINE

- Review
- Functions
- Pointers
  - Function calls
  - Array Pointers
  - Pointer Arrays
- Data Structures and dynamic Memory
- Function Pointers
  - quicksort example

FUNCTIONS

- Functions enable grouping of commonly used code into a reusable and compact unit.
- In programs containing many functions main should be implemented as a group of calls to functions undertaking the bulk of the work
- Become familiar with rich collections of functions in the ANSI C standard library
- Using functions from ANSI standard library increases portability

STANDARD LIBRARY FUNCTIONS

Header	Description
<stdio.h>	Functions for standard input and output
<float.h>	Floating point size limits
<limits.h>	Contains integral size limits of system
<stdlib.h>	Functions for converting numbers to text and text to numbers, memory allocation, random numbers, other utility functions
<math.h>	Math library functions
<string.h>	String processing functions
<stddef.h>	Common definitions of types used by C

The University Of Sheffield.

The University of Sheffield Research IT

FUNCTIONS AVAILABLE WITH THE LIBRARY MATH.H

Function	Returns
sqrt(x)	Square root
exp(x)	Exponential function
log(x)	Natural logarithm (base e)
log10(x)	Logarithm (base 10)
fabs(x)	Absolute value
pow(x,y)	X raised to the power of y
sin(x)	Trignometric sine (x in radians)
cos(x)	Trignometric cosine (x in radians)
tan(x)	Trignometric tangent (x in radians)
atan(x)	Arctangent of x (returned value is in radians)

USING FUNCTIONS

- Include the header file for the required library using the preprocessor directive
  - #include <libraryname.h>
  - Note no semi colon after this
- Variables defined in functions are local variables
- Functions have a list of parameters
  - Means of communicating information between functions
- Functions can return values
- printf and scanf good examples of function calls
- Use the –lm option to compile an application using math library functions e.g.
  - pgcc myprog.c –o myprog -lm

USER DEFINED FUNCTIONS

- Format of a function definition

```
Return-value-type function-name(parameter-list)
{
        declarations

        statements
}
```

A return value of type void indicates a function does not Return a value.

FUNCTIONS: RETURN

- Return control to point from which function called
- 3 Ways to return
  - Function does not return a result (void) control is returned when function right brace } is reached.
  - Execute the statement
    - return;
  - If the statement returns a value the following statement must be executed
    - return expression;

FUNCTION PROTOTYPES

- Tells compiler
  - type of data returned by function
  - Number and types of parameters received by a function
- Enable compiler to validate function calls
- Function prototype for a RollDice function
  - int RollDice(int iPlayer);
  - Terminated with ;
  - Placed after pre-processor declarations and before function definitions

USING FUNCTIONS

- Declare function using prototype
- Define source code for function
- Call the function
- See program functions.c for an example of function declaration, definition, usage

HEADER FILES

- Standard libraries have header files containing function prototypes for all functions in that library
- Programmer can create custom header files
  - Should end in .h e.g. myfunctionlib.h
- Programmer function prototypes declared using the pre processor directive
  - #include “myfunctionlib.h”

Demonstration

- Build and run the example function1.c
  - •Add more calls to the blorf() function in the main pprogram
- Build and run function2.c
  - •Note this avoids the use of the function prototype
  - •Move the soup function after the main function compile and run what happens?
  - •Add a prototype and build and run again

POINTERS AND ARRAYS

- Pointers are a powerful feature of C which have remained from times when low level assembly language programming was more popular.
- Used for managing
  - Arrays
  - Strings
  - Structures
  - Complex data types e.g. stacks, linked lists, queues, trees

VARIABLE DECLARATION

- A variable is an area of memory that has been given a name.
- The variable declaration
  - float f1;
  - is command to allocate an area of memory for a float variable type with the name f1.
- The statement
  - f1=3.141
  - is a command to assign the value 3.141 to the area of memory named f1.

WHAT IS A POINTER

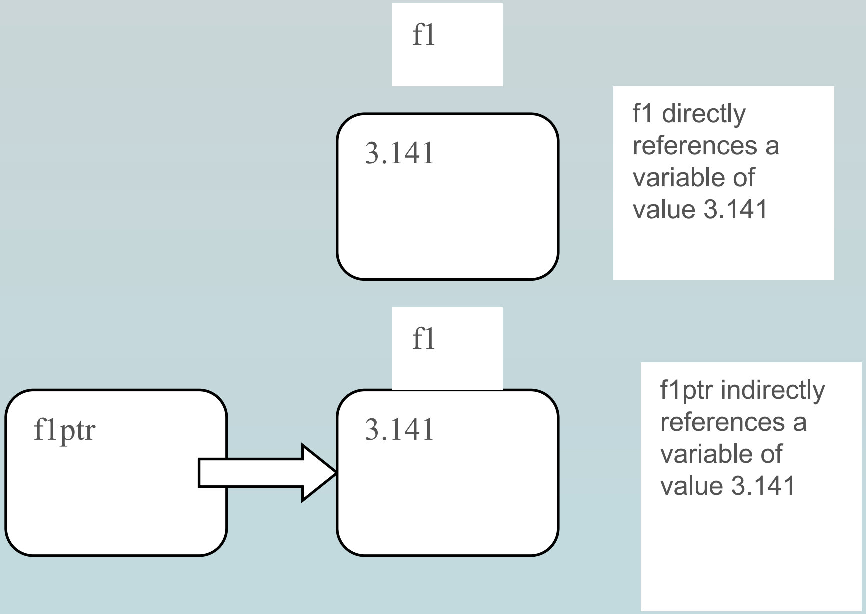
- **Pointers are variables that contain memory addresses as their values.**
- Pointer declared using the indirection or de-referencing operator *.
- Example
  - float *f1ptr;
- f1ptr is pointer variable and it is the memory location of a float variable

POINTER EXAMPLE

- The redirection operator returns the address of a variable
- & applied to f1 returns the address of f1

```
float f1;
float *f1ptr;  /* Declare a pointer variable to an integer*/
f1=3.141;
f1ptr=&f1; /*f1ptr is set to the address of f1*/
```

Pointer Variables



USING THE * AND & OPERATORS:

```
int some_var;   /*1*/
int *ptr_to_some_var; /*2*/
ptr_to_some_var = &some_var; /*3*/
printf ("%d\n\n", *ptr_to_some_var); /*4*/
```

- /*1*/ Declare an integer
- /*2*/ Declare a pointer
- /*3*/ Assign a value to the pointer variable
- /*4*/ Use the pointer in a function (dereference the value)
- Compile and run the example pointers.c

FUNCTION CALLS

- Call by value
  - Copy of variable passed to function
  - If that variable is modified within the function then upon return from the function since only the copy has been modified, the actual variable is not modified
- Call by reference
  - Pass the address of a variable (i.e. a pointer) to a function
  - The variable pointed to can be modified within that function

CALL BY VALUE EXAMPLE

- finval=FuncByValue(finval);
- The FuncFyValue function

```
float FuncByValue(float fval)
{
        return fval*fval;
}
```

CALL BY REFERENCE EXAMPLE

- FuncByReference(&finref), Use & to pass the address of a variable to the function;
- The FuncByReference function
- Value of the referenced variable passed to the function is modified after returning from the function.

```
void  FuncByReference(float *fvalptr)
{
        *fvalptr = *fvalptr * *fvalptr;
}
```

ARRAYS

- Initialisation
- int iarray[5]={1,2,3,4,5};
- Or… initialise elements individually
- Note first element is referenced using 0
  - iarray[0]=1;
  - ……..
  - iarray[4]=5;

FURTHER EXAMPLES OF FUNCTION CALLS

- Putting it all together
  - Function calls
  - Simple array examples
- Numerical Method Examples
  - Numerical differentiation
  - Numerical Integration

MULTIDIMENSIONAL ARRAY

- The declaration for a multi dimensional array is made as follows:
  - *Type variable[size1][size2];*
- To access or assign an element to an element of a multidimensional array we use the statement:
  - *variable[index1][index2]=avalue;*

MATRIX INITIALISATION EXAMPLE

- Alternatively the bracket initialisation method can be used, for example the integer matrix[2][4] can be initialised as follows:

```
int matrix[2][4]
{
 {1,2,3,4},
 {10,20,30,40}
};
```

ARRAYS ARE POINTERS

- The array variable is a pointer whose value is the address of the first element of the array.
- For a one dimensional array access a value using the following pointer notation:

  *int ielement= *(iarray+1);*

- This assignment increments the array pointer to the second element in the array (the first element is always index 0)
- uses the * operator to dereference the pointer

POINTER ARRAYS

- A string is a pointer to an array of characters
- An array of strings is an array of pointers
- Multidimensional array is essentially an array of pointer arrays.

MEMORY LEAKS

- TAKE VERY SPECIAL CARE IN USE OF POINTERS AND MANAGEMENT OF ARRAYS
- A common problem when using arrays is that the program might run off the end of the array particularly when using pointer arithmetic.
- When passing an array to a function it is good practice to pass the size of that array making the function more general.

DATA TYPES AND STRUCTURES

- Features for representing data and aggregations of different data types.
  - structures,
  - type definitions,
  - enumerations and
  - unions.

DATA STRUCTURES

- Arrays and structures are similar
  - pointers to an area of memory that
  - aggregates a collection of data.
- Array
  - All of the elements are of the same type and are numbered.
- Structure
  - Each element or field has its own name and data type.

FORMAT OF A DATA STRUCTURE

```
struct structure-name {
        field-type field-name; /*description*/
        field-type field-name; /*description*/
        …….
}  variable-name;
```

DECLARING STRUCTURES AND ACCESSING FIELDS

- *struct structure-name variable-name;*
- A pointer to a structure
  - *struct structure-name *ptr-variable-name;*
- Accessing a field in a structure
  - *variable-name.field-name*
- For a pointer to a structure a field is accessed using the indirection operator ->
  - *ptr-variable-name->field-name*

STRUCTURE EXAMPLE

```
struct node {
        char *name;
        char *processor;
        int  num_procs;
        };
```

DECLARING AND INITIALISING STRUCTURES

```
struct node n1;
struct node *n1ptr;

n1.name="Titania";
n1.processor ="Ultra Sparc III Cu";
n1.num_procs = 80;
n1ptr = &n1;
```

ACCESSING STRUCTURE DATA

- Direct access

```
printf("The node %s has %d %s processors\n",
    n1.name, n1.num_procs, n1.processor);
```

- Access using a pointer

```
printf("The node %s has %d %s processors\n",
    n1ptr->name, n1ptr->num_procs, n1ptr->processor);
```

- Dereferencing a pointer

```
printf("The node %s has %d %s processors\n",
    (*n1ptr).name, (*n1ptr).num_procs, (*n1ptr).processor);
```

TYPE DEFINITIONS

- *typedef float vec[3];*
- Defines an array of 3 float variables a particle position may then be defined using:
- *vec particlepos;*
- Defined structure types
- *typedef struct structure-name mystruct;*
- *mystruct mystructvar;*

ENUMERATIONS

- enum enum-name {tag-1, tag-2, ….} variable-name;
- enum months {JAN=1, FEB, MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC};
- Same as
  int JAN=1;

  int FEB=2;

  ..

  int DEC=12;

USING AN ENUMERATION

```
enum months month;

for(month=JAN; month<=DEC;
  month++)

     statement;
```

DECLARING AND USING AN ENUMERATION

```
int main ()
{
  enum compass_direction {
            north,
            east,
            south,
            west
};
enum compass_direction my_direction;
my_direction = west;
return 0;
 }
```

DYNAMIC MEMORY ALLOCATION

- Allocate Memory
  - malloc
  - calloc
- Free memory
  - Free
- Size of memory used by variabletype
  - sizeof

USING MALLOC

```
struct node *newPtr;

newPtr = (struct node *)malloc(sizeof(struct node));
```

- The *(struct node *)* before the malloc statement
  - used to recast the pointer returned by malloc from (void *) to (struct node *).

FREE ME!

- *free(newPtr);*

RESERVING MEMORY FOR AN ARRAY

```
int n=10;

struct node *newPtr;

newPtr = (struct node *)calloc(n, sizeof(struct node));
```

Avoid memory leaks Free the memory!

```
free(newPtr);
```

POINTERS TO FUNCTIONS

- Pointer to function
  - Address of the function in memory
  - Starting address of the code
- Quick sort function has prototype (see 2 slides later)
  - **int (*fncompare*)(const void *, const void *)**
  - fncompare is function pointer
  - Tells quicksort to expect pointer to a function receiving 2 pointers to void
- Note without the brackets around *fncompare the declaration becomes a declaration of a function

DECLARATION OF FUNCTION USING A POINTER TO A FUNCTION

- Declare mysortfunc as
  - mysortfunc(int *data, int size, int (*compare)(void * , void *))
- Call mysortfunc in the following way
  - mysortfunc(data,size,ascending)
  - mysortfunc(data,size,descending)
- Ascending and descending are functions declared as
  - Int ascending(void *a, void *b)
  - Int descending(void *a, void *b)

ARRAY OF POINTERS TO FUNCTIONS

- Declare functions
  - void function1(int);
  - void function2(int);
  - void function3(int);

- void (*f[3])(int)={function1,function2, function3};
- Called as follows
  - (*f[choice])(myintergerinput);
  - Choice and myintegerinput are both integers

APPLICATION OF FUNCTION POINTERS

- Numerical recipes
  - E.g. function which is dependent on a differential which has yet to be defined
- Quick sort function see next slides
- Make C object oriented
  - Include pointers to data methods in struct data types…. See later!

USING THE QUICK SORT FUNCTION

- Implementation of quick sort algorithm
- Qsort function in <stdlib.h>
- Features
  - Pointer to void *
  - Pointer to a function

SYNTAX FOR QSORT FUNCTION

- implementation of the quicksort algorithm to sort the *num* elements of an array pointed by *base*
  - each element has the specified *width* in bytes
  - method used to compare each pair of elements is provided by the caller to this function with *fncompare* parameter (a function called one or more times during the sort process).
- **void qsort (void * *base*, size_t *num*, size_t *width*, int (**fncompare*)(const void *, const void *));**

EXAMPLE USING QSORT

```
/* qsort example */
#include <stdio.h>
#include <stdlib.h>
 int values[] = { 40, 10, 100, 90, 20, 25 };
int compare (const void * a, const void * b)
{ return ( *(int*)a - *(int*)b ); }
int main ()
{
        int * pItem;
        int n;
        qsort (values, 6, sizeof(int), compare);
        for (n=0; n<6; n++)
        {
           printf ("%d ",values[n]);
         }
        return 0;
 }
```

EXAMPLES

- Compile and run the following programs
  - Program array.c initialising and using arrays with pointers
  - Program bubblesort.c is a bubble sort example, using call by reference to manipulate data passed into a function
  - Program arrayref.c uses pointer notation to manipulate arrays
  - Modify the bubblesort program to use the qsort routine

PRACTICAL EXAMPLES

- Compile and run the following programs
  - Numerical Differentiation
    - 2 and four point methods
  - Numerical Integration
    - Trapezium method
    - Simpsons rule (includes lagrange interpolation function)