Objective

- To learn further features of C particularly how to use arrays and modules (functions) and structures
- To introduce the use of the Debugger.

What is the difference and why? How they are stored in the computer?

The implications

- Standard
- Size of file
  - Storage
  - Processing
- How does it affect the program design and system performance?

Format of GIF87 file

<table>
<thead>
<tr>
<th>Bin</th>
<th>Screen descriptor</th>
<th>Global colormap</th>
<th>Image descriptor</th>
<th>Local colormap</th>
<th>Raster data</th>
<th>GIF terminator</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Screen Descriptor comprises a set of attributes that belong to every image in the file. According to the GIF87 standard, it is defined as in Fig. 3.13
Arrays in C

- The array data structures is used to store a collection of similar data items.
- All arrays in C store elements starting at index (or subscript) 0. E.g., the declaration
  ```c
  int my_array[10];
  ```
  declares my_array as an array of 10 elements, referred to as my_array[0] to my_array[9].

Defining Arrays

- The following lines of code define a type to represent an array of characters (known as a string) of a certain length, and then declares two arrays of this type:
  ```c
  typedef char astring[16];
  astring name;
  ```
- The array ‘name’ declared above can hold a string of up to 15 characters. The last array element (index position 15) is used to hold the so-called null character which indicates the end of the character string.

Example: strings

```c
#include <stdio.h>
#include <string.h>    /* for strcpy */

main()
{
    static char greetings[] = "Hello";
    char subject1[] = {'C','O','M','P','U','T','I','N','G','\0'}; /* Error! */
    char name[20], name2[20];
    strcpy(name, "William Shakespeare");
    printf("Please enter your name:
");
    scanf("%s", name2);
    printf("%s %s
", greetings, name2);
    printf("My name is %s
", name);
    printf("I study %s
", subject1);
    printf("Oops! this string was not null terminated: %s
", subject2);
}
```

Array in C

- An array may be initialised at declaration time as follows:
  ```c
  int days_in_month[] = {31, 28, 31, 30, 31, 30, 31, 30, 31};
  ```
- Note the empty pair of square brackets after the array name. The size of the array is missing! Once initialised, the size becomes fixed to the number of values used to initialise - in the example above, size of days_in_month is fixed to be 12.
String Functions

• The C library `<string.h>` provides many useful functions, eg,
  
  `strcpy` copies strings
  `strcmp` compares strings
  `strcat` concatenates one string onto another string
  `strlen` returns the length of a string

Character Handling Functions

• A number of character handling functions are defined in `<ctype.h>`, eg,
  
  `isalnum` tests for alphanumeric characters
  `isalpha` tests for alphabetic characters
  `isascii` tests for ASCII characters
  `isdigit` tests for 0 to 9
  `islower` tests for lowercase characters
  `isupper` tests for uppercase characters
  `tolower` converts characters to lowercase
  `toupper` converts characters to uppercase

Use of arrays - another example:

/* File: test2-b.c ICT106 test program
   Purpose: To demonstrate array input/output; stores 10 numbers and outputs the numbers in reverse order */
#include <stdio.h>
#define SIZE 10
main()
{
    int numArray[SIZE];
    int i;
    /* store numbers in the array */
    for (i = 0; i < SIZE; i++)
        numArray[i] = i * i;

    printf("The contents of the array are:\n");
    for (i = 0; i < SIZE; i++)
        printf("%d, ", numArray[i]);
    printf("\n");

    /* output contents of the array in reverse order */
    printf("The contents of the array in reverse order are:\n");
    for (i = SIZE-1; i >= 0; i--)
        printf("%d, ", numArray[i]);
    printf("\n");
} /* end main */

Functions

A function definition has the following form in C:

```
return_type function_name ( parameter_list )
{
    local declarations
    statements
}
```
Example
/* Function to input a user choice, ensuring valid input. */
int UserChoice()
{
    /* local vars */
    char correctChoice;
    int choice;

    /* function body */
    correctChoice = 'n';
    while (correctChoice == 'n') {
        PrintMenu(); /* call to another function */
        scanf("%d", &choice);
        if (choice == 1)
            correctChoice = 'y';
        else if (choice == 2)
            correctChoice = 'y';
    } /* end while */
    return choice;
} /* end function UserChoice */

Function Calls
• Function calls require the name of a function followed by a list of actual parameters (arguments), if any.
• On execution of a function call, control passes to the called function F.
• When F finishes, control goes back to the calling function.
• Before a function is called, it must be defined or its prototype must appear in the program.
  A function prototype has the form:
  return_type function_name ( parameter_type_list )

Example
• Prototype of function:
  int gcd ( int n, int d )
  { /* function body */ }
  is
  int gcd ( int, int );
  or
  int gcd ( int n, int d );

Order of functions in a program
• A function definition (including its body) or its prototype (declaration) must appear before any call to the function.
Parameters of Functions and actual arguments of Calls

- The term **parameter** is usually used for a variable named in the parenthesized list in a function definition, **argument** for the value used in a call of the function.
- In C, all function arguments are passed "by value", which means the called function is given the value of its arguments in temporary variables rather than originals.

Arguments are constants, variables or expressions in a function call that correspond to its parameters.

- Pass-by-value initializes the parameter to the corresponding argument value, eg,
  - \( n = 30, \ d = 75 \)

The function cannot alter the value of a pass-by-value argument.

- Pass-by-value is used for a one-way flow of information (into the function) only - INput parameter, eg, arguments \( x \) and \( y \) passed to function \( \text{gcd} \) remain unchanged.

Example:

```c
/* File: test2-c.c ICT106 test program
Purpose: To demonstrate pass-by-value arguments to a function */
#include <stdio.h>
int gcd ( int, int ); /* prototype */
void main ( ) {
  int x = 30;
  int y = 75;
  printf("GCD of 30 and 75 is %d
", gcd (x, y));
  printf( "\n\nx = %d ",  x);
  printf( "\ny = %d ",  y);
} /* end main */
```

```c
int gcd ( int n, int d)  {
  if ( n<=0  ||  d<=0 )      /* check precondition */
    return 0;
  while (n != d) {  
    if (n>d)
      n = n - d ;
    else
      d = d - n ;
  } /* end while */
  return n;
} /* end gcd */
```

Changing the value of arguments in a function

- It is possible for a called function to modify a variable in a calling function.
- This is achieved by the calling function providing the address of that variable and the called function declaring the parameter to be a pointer and then accessing the variable indirectly through this pointer.
- The "address-of" operator `&` is used in front of the argument name when the function is called.
Example

/* File: test2-c.c  ICT106 test program
   Purpose: To demonstrate pass-by-reference arguments to a function
   */
#include <stdio.h>
void exchange (int *a, int *b)
/* a and b are declared as pointers to int */
{
    int temp;
    temp = *a;   *a = *b;   *b = temp;
}

Example cont’d

void main ( )
{
    int x = 4;   int y = 5;
    exchange ( &x,  &y );
    /* pass pointers to the values to be swapped */
    printf("%d %d\n", x , y);
}

Address-of operator

• Since the address-of operator & produces the address of a variable, &x and &y in the above example are pointers to the variables x and y respectively.
• In the function exchange, the parameters are declared to be pointers, and the operands are accessed indirectly through them.

Array Parameters

• In C, arrays are passed differently from variables.
• When the name of an array is used as an argument, the value passed to the function is the address of the beginning (first element) of the array.

• When an array A is used as a formal parameter of a function F1, the array name must be followed by [], e.g.,
   int F1 ( int A[] )
   { /* function body */}
• The size of the array can be passed as another parameter
   int F1 ( int A[], int size )
   { /* function body */}
• When a function with an array parameter is called, the argument should be an array name without [], e.g.,
   total = F1 ( coins, 5 );
   /* call to function F1 */
• Since the address of the array is passed to the called function, the array elements can be altered by this function.
Example

/* File: test2-d.c  ICT106 test program  
 Purpose: To demonstrate passing array arguments to a function  */
#include <stdio.h>
main ( )
{
    int coins[6] = { 5, 10, 20, 50, 100, 200 ];
    int sumOfCoins;
    sumOfCoins = Sum(coins, 6);  
    printf("The sum of coins is %d\n", sumOfCoins);
}

int Sum(int A[ ], int size )
{
    int i = 0;
    int total = 0;
    while (i < size) {
        total = total + A[i];
        i = i+1;
    }  /* end while */
    return total;
}  /* end Sum */

Another Example

// File: arrays1a.c  
// A program to get random numbers in the range 0..10 and store them in two arrays.  
// The corresponding elements of the two arrays are then added and stored  
// in a third array. Finally the contents of the three arrays are printed out.
#include <stdio.h>
#include <stdlib.h>   // for srand, rand
#define MAX_SIZE  10

// function prototypes
void ReadInto (int anArray [ ], int size);  
void AddArrays (int arrayOne[ ], int arrayTwo[ ], int resultArray[ ], int size);  
void PrintArrays (int arrayOne[ ], int arrayTwo[ ], int resultArray[ ], int size);

int main ()
{
    int arrayOne[MAX_SIZE];  
    int arrayTwo[MAX_SIZE];  
    int resultArray[MAX_SIZE];  
    int n;
    char ch;
    n = MAX_SIZE; // n <= MAX_SIZE
    printf("Reading into first array .....\n");
    ReadInto (arrayOne, n);
    printf("Reading into second array .....\n");
    ReadInto (arrayTwo, n);
    AddArrays (arrayOne, arrayTwo, resultArray, n);
    PrintArrays (arrayOne, arrayTwo, resultArray, n);
    printf("End of Program - Press enter to exit.\n");
    ch = getchar();
    return 0;  
}  // end main

void ReadInto (int anArray [ ], int size )
{
    int rand_num, i;
    int seed;
    printf("Enter an integer seed: ");
    scanf("%d", &seed);
    printf("Enter integer seed: %d", seed);
    srand (seed);
    for (i = 0; i < size; i++) {  
        if (i < size) {
            rand_num = rand() % 11;
            anArray[i] = rand_num;
        }
    }  // end for
}  // end ReadInto
void AddArrays (int first[], int second[], int result[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        result[i] = first[i] + second[i];
} // end AddArrays

void PrintArrays (int first[], int second[], int result[], int size)
{
    int i;
    printf("The contents of the arrays are:

");
    printf("%15s %15s %15s
", "First Array", "Second Array", "Sum Array");
    for (i = 0; i < size; i++)
        printf("%10d %15d %17d
", first[i], second[i], result[i]);
} // end PrintArrays

Structs in C

• Arrays are used to group data elements of the same type, eg, an array of integers, an array of floats, etc.
• Structs are like records and can be used to group data elements of different types, i.e., a struct is (usually) a collection of mixed type items.

Example

struct Employee {
    char surname[10];
    char firstInit;
    int age;
    float wage;
};

Accessing elements in a struct

• The elements (members) of a struct are accessed using the dot notation:
  worker1.firstInit = 'J';
  strcpy(worker1.surname, "Blogg");
  worker1.age = 22;
  worker1.wage = 30000.00;
  manager.wage = 2 * worker1.wage;

• A variable of a struct type can be assigned to another of the same type:
  struct Employee worker1, worker2;
  /* aggregate assignment */
  worker2 = worker1;
Main differences with arrays

1. Data items in a struct are (usually) heterogeneous
2. Items are accessed using (field) names rather than indexes
   • Most real-world data are represented using arrays and records

Data structures as members of a struct

• A member of a struct can be
  – Any built-in type - simple or structured
  – Any user-defined type - including other structs

Another example

```c
struct Date {
    int day;
    int month;
    int year;
};

struct Date today, tomorrow;
today.day = 29;
today.month = 7;
today.year = 2000;
printf("Today's date is: %d-%d-%d",
    today.day, today.month, today.year);
```

Arrays of structs

Arrays can have instances of a struct type as elements

```c
typedef char String20[20];
struct EmployeeType {
    String20 name;
    long workNumber;
    float hourlyRate;
    float hoursWorked;
    float grossPay;
    float netPay;
    float tax;
}; /* end definition of struct */

struct EmployeeType employee[100];
printf("%f", employee[99].netPay;
/* prints net pay of the 100th employee */
for ( int i = 0; i < 100; i++ )
    printf("%s\n", employee[i].name);
```