Pointers and Functions - Chapter Summary

1 Function Invocation and Call by Value

When a function with arguments is called, it is normally only the values of the arguments that are passed to the function and not their actual addresses. This mechanism is known as call by value.

(The one exception to this rule discussed so far occurs when an array is used as an argument to a function, in which case it is the address of the beginning of the array that is passed.)

Data used as arguments in the calling environment are unchanged by any operations carried out in the function that has been called, as is demonstrated by the following program:

```
void alter(int, int, int);

int main(void)
{
     int a = 1, b = 2, c = 3;

     alter(a, b, c);

     printf("\nValues of a, b, and c in main: %d, %d & %d.",
               a, b, c);

     return 0;
}

void alter(int a, int b, int c)
{
     a = a + b + c;
     b = a * b * c;
     c = (a + b) / c;

     printf("\nValues of a, b and c in alter: %d, %d & %d.",
               a, b, c);
}
```

When compiled and run, this program prints out:

```
     Values of a, b and c in alter: 6, 36 & 14.
     Values of a, b and c in main: 1, 2 & 3.
```

The function `alter` is passed the values `1`, `2` and `3`, which it assigns to its own variables called `a`, `b` and `c`. Nothing that is done to the `a`, `b` and `c` in `alter` changes the variables `a`, `b` and `c` in `main`.

In order to change data in the calling environment, a function would have to be passed the addresses of that data as its arguments, so that it could use indirection to alter the values stored at those locations. This can be achieved by using pointers and provides a mechanism known as call by reference.

2 Call by Reference

The following program uses pointers as arguments to the function `decode` to achieve call by reference, and incidentally demonstrates that the name of Arthur C Clarke's fictitious computer HAL in "2001: a Space Odyssey" is a slyly-coded reference to a certain well-known computer company...

```
void decode(char *, char *, char *);

int main(void)
{
     char lt1, lt2, lt3;
     char * pchar1, * pchar2, * pchar3;

     lt1 = 'H';
     lt2 = 'A';
     lt3 = 'L';

     pchar1 = &lt1;
     pchar2 = &lt2;
     pchar3 = &lt3;

     decode(pchar1, pchar2, pchar3);

     printf("\nWhen decoded, 'HAL' spells '%c%c%c'!",
               lt1, lt2, lt3);

     return 0;
}

void decode(char * first, char * second, char * third)
{
     (*first)++;
     (*second)++;
     (*third)++;
}
```

The function decode takes three pointers of type pointer to char as arguments and uses the pointer reference operator * to gain access to the values held at the three addresses that are pointed to by its arguments. These values are each incremented by one, and since they represent letters of the alphabet, which are assigned sequential numbers in the system's character set (ASCII assumed!), the effect is of changing the character stored in each variable to the next letter in the alphabet.

However, note that the function decode is of type void. It does not return a value to main, but uses pointers and indirection to change the values held in the variables lt1, lt2 and lt3 in the main function. This is made possible by the fact that the addresses of these three variables have been assigned to three pointers, of type pointer to char, which are then used as the arguments in the function call to decode from main.

Also note the way that the pointers and pointer types are declared in the function prototype at the top of the program and in the function definition itself: char * in the prototype and char *name in the function definition.

3 Functions Returning Pointers

A function can return a single value of a type specified in its declaration. This may be the value of local data or expression in the function, or it may be the address of data used in the function; that is to say, a pointer. Just as using pointers as arguments in a function invocation gives that function access to data in the calling environment, so, conversely, returning a pointer from a function gives the calling environment access to data in the function that it has called.

The function larger in the following example takes two pointers to integers as its arguments and is of type pointer to int, which means that it returns a value that is the address of an integer data item.

```
int * larger (int *, int *);

int main(void)
{
     int a = 10, b = 5;
     int * plarge;

     plarge = larger(&a, &b);

     printf("\nThe larger of %d and %d is %d.",
               a, b, *plarge);

     return 0;
}
```

```
int * larger (int * lhs, int * rhs)
{
     if (*lhs > *rhs)
          return lhs;
     else
          return rhs;
}
```

The function `larger` receives pointers to two `ints` and compares the integers using indirection. The function then returns a pointer to the larger integer. The expression `*plarge` in the `printf` statement in `main` refers to the value of a or b, whichever is larger.