More on Data Types - Chapter Summary

1 The Range of Data Types in C

C offers a very wide range of data types and operators to work on them.

There are the scalar data types, comprising integers (including `int`, `short`, `long` and `unsigned` integers, plus `unsigned short` and `unsigned long` integers), characters (of type `char`, which are a special form of integer data in which integer numbers are translated into characters according to one of the standard tables of association, and which also include `signed char` and `unsigned char` types) and floating-point values of various degrees of precision (represented by types `float`, `double` and `long double`).

There are also the aggregate data types: arrays (which store items of the same type in adjacent locations in memory), structures (which store logically-related items that may be of different types) and unions (which can, at different times, store data of different but previously-declared types in the same location in memory).

C programs may also use other kinds of data. A range of bitwise operators gives powerful low-level access to individual bits within a byte or a word. Enumerated data types (mentioned in the Advanced Language Topics Appendix) allow programmers to define their own data types and the range of values that they may take, while the `typedef` statement allows a programmer to rename an existing scalar or aggregate data type, thus improving the readability of a program and achieving a higher level of abstraction.

2 Background on Bit Manipulation

2.1 Bits, Bytes and Words

The smallest unit of computer data or storage is the bit, so-called because it is a `BI`nary digi`T`. Bits are usually arranged into groups of eight (in 8-bit machines) to form a unit known as a byte. It is the byte that forms the basic unit of data storage; each byte in a computer's memory being identified by a numerical address.

The amount of memory that can be processed by the Central Processing Unit in a single operation is known as a word, and the size of this unit will depend on the computer being used. In 16-bit machines based around the Intel 8088 microprocessor, for instance, a word is 2 bytes long, whereas in a 32-bit machine using the Intel 80386 chip, a word is 4 bytes long.

Implementations of C use the size of the machine word as the size for type `int`.

2.2 Numbers and Switches

A single bit may have a value of either 0 or 1. Conceptually, this range of values means that a bit may either be regarded as a store for a number, or as a toggle switch for some computer operation or device that is either off or on. The device controlled by the bit is switched off when the bit has the value 0, and on when it has the value 1.

The 8 bits in a byte may similarly be regarded as either the stores for the digits in a binary number or as a bank of eight switches. As a binary number, successive bits from the least significant bit (LSB) to the most significant bit (MSB) represent ones, twos, fours, eights, sixteens, thirty-twos, sixty-fours and hundred-and-twenty-eights respectively:

```
 128   64   32   16   8    4    2    1
[   ][   ][   ][   ][   ][   ][   ][   ]
 msb                                 lsb
```

This means that a byte can store any number between 0, when all eight bits have the value 0, and 255, i.e. (1 + 2 + 4 + 8 + 16 + 32 + 64 +128), when all eight bits have the value 1:

```
 128   64   32   16   8    4    2    1
[ 0 ][ 0 ][ 0 ][ 0 ][ 0 ][ 0 ][ 0 ][ 0 ]  /* 0 decimal */
 msb                                 lsb


 128   64   32   16   8    4    2    1
[ 1 ][ 1 ][ 1 ][ 1 ][ 1 ][ 1 ][ 1 ][ 1 ]     /* 255  */
 msb                                 lsb      /* decimal */
```

Considering a byte as a bank of eight switches, this means that each possible combination of on and off switches is uniquely represented by a single decimal number between 0 and 255.

The eight switches in a byte are conventionally numbered from 0 to 7, switch 0 being the least significant bit, and switch 7 the most significant bit.

If a certain operation controlled by a byte of switches requires, for instance, that switches two and six are on, while all the others are off, then storing the number 68 in that byte would set the switches to exactly the right configuration. The decimal number 68 is (64 + 4), and is stored as follows:

```
 128   64   32   16   8    4    2    1
[ 0 ][ 1 ][ 0 ][ 0 ][ 0 ][ 1 ][ 0 ][ 0 ] /* 68 decimal */
 msb                                 lsb
```

If a device or operation is controlled by a two-byte-integer word of switches, then each possible combination of on and off switches is uniquely represented by one of 65,536 decimal numbers. An `unsigned` integer uses all 16 bits in a word to store a number, and so has a range of values from 0 to 65,535. A `signed` integer uses only 15 bits for storing a number, and uses the remaining bit for storing the sign of the number. Signed integers therefore have a range from -32,768 to +32,767.

2.3 Ports

A computer's central processing unit can communicate with the various physical devices in the computer system, such as the keyboard, monitor, disk drive motor and loudspeaker, by means of both addresses in memory and dedicated input/output ports. Machines using the Intel 8088 chip, for instance, have 65,536 numbered I/O ports.

Physical devices are connected to interfaces that plug into these ports. The interfaces possess small units of memory, known as registers, which are connected to specific ports, and the number stored in each of these registers determines the setting of the individual bit switches in the registers that control the operation of the physical devices.

The functions available for reading the number stored in a register connected to a port, and for writing a number to a given port, are not defined by the ISO Standard, but are compiler dependent. Microsoft C, for instance, provides four functions to perform these tasks and gives their declarations in the `conio.h` header file:

`inp` reads a byte from the port passed as its argument, returning an `int`.

`inpw` reads a word from the port passed as its argument, returning an `unsigned int`.

`outp` writes a byte to a port and takes two arguments: the number of the port and byte to be written. The function returns the value written as an `int`.

`outpw` writes a word to a port and takes the number of the port and the word to be written as its two arguments, returning the value written as an `unsigned int`.

All of these functions take the number of the port as an `unsigned int`.

Details of the actual device operations controlled by the bits in the registers connected to each port will be given in a system's technical manual, which should always be consulted before port I/O operations are attempted.

These operations may be controlled by writing the appropriate byte- or word-sized numbers to the relevant ports using the port I/O functions supplied by a compiler.

3 Bit Operators

C supplies a range of bitwise operators that act directly on individual bits in a byte or word and may be used in routines to control devices or actions controlled by bit switches, as described above.

These operators may be used with any of the integer types of data, including the `char` type, which is a special form of integer.

4 Bitwise Logical Operators

4.1. Bitwise AND operator &

The bitwise AND operator & is a binary operator that compares corresponding bits in its two operands, producing a result in which each bit is set to 1 if both corresponding bits in the operands are set to 1, but otherwise setting the bit to zero. Thus, for example:

```
23 & 168 == 0
145 & 233 == 129
```

These results may be explained by considering each number as a binary number stored in a word of memory (a word will be assumed to be two bytes long):

```
00000000 00010111              /* unsigned int 23 */
00000000 10101000              /* unsigned int 168 */
```

No pair of corresponding bits in these two words are both set to 1, so the result produced by using bitwise AND on them must be:

```
00000000 00000000              /* zero! */
```

The decimal numbers 145 and 233 would be stored like this, however:

```
00000000 10010001        /* unsigned int 145 */
00000000 11101001        /* unsigned int 233 */
```

Bits 0 and 7 in both numbers are set to one, so that bitwise AND will produce:

```
00000000 10000001        /* unsigned int 129 */
```

4.2. Bitwise OR Operator |

The binary bitwise OR operator | compares corresponding bits in its two operands, producing a result in which bits are set to 1 if either of the corresponding bits in the two operands is 1.

Thus, the expression `145 | 233` produces the result `249`:

```
00000000 10010001        /* unsigned int 145 */
00000000 11101001        /* unsigned int 233 */


00000000 11111001        /* unsigned int 249 */
```

4.3 Bitwise Exclusive OR Operator ^

The bitwise Exclusive OR operator ^ is a binary operator that compares corresponding bits in its two operands, producing a result in which bits are set to 1 if one but not both of the corresponding bits in the two operands has the value 1. Thus, the expression `145 ^ 233` produces the result `120`:

```
00000000 10010001        /* unsigned int 145 */
00000000 11101001        /* unsigned int 233 */


00000000 01111000        /* unsigned int 120 */
```

4.4 Bitwise Negation: ONES Complement Operator ~

The unary bitwise operator ONES Complement ~ changes each bit in its operand, setting a bit to 1 if it is set to 0, and setting it to 0 if it is set to 1. This has the effect on an unsigned operand of giving the result of subtracting the operand from the highest value that can be stored in a word. Thus, the expression ~233, where 233 is stored as an `unsigned` integer, produces the result `65302`, which is also (`65535 - 233`):

```
00000000 11101001        /* unsigned int 233   */
11111111 00010110        /* unsigned int 65302 */
```

When 233 is stored as a `signed` integer, the expression ~233 produces the answer -32534

```
00000000 11101001        /* signed int    233 */
11111111 00010110        /* signed int -32534 */
```

5 Using Bitwise Logical Operators: An Example

On an IBM PC computer based on the Intel 8088 chip, the loudspeaker is controlled via an 8255 Programmable Parallel Interface Controller. This interface has three registers, and the one that contains the switches that turn the loudspeaker on and off is a byte sized register connected to port 97.

The particular bits that operate the loudspeaker are bits 0 and 1, and the default setting for these switches is 0 in each case, meaning that the loudspeaker is off. Setting these bits to 1 results in the loudspeaker producing a beep.

The other bits in port 97 control other operations, such as enabling and disabling the keyboard, so it is important that the values of these bits should not be changed when it is only the loudspeaker that needs to be turned on and off.

The bitwise OR operator | produces a result in which the bits have identical values to those in one of its operands if the corresponding bits in the other operand are set to zero. This means that to change the values in bits 0 and 1 in an operand from 0 to 1, leaving all the other bits set as they were before, the other operand for bitwise OR should be a number that has every bit set to 0 except bits 0 and 1, and this is the decimal number 3:

```
 128   64   32   16   8    4    2    1
[ 0 ][ 0 ][ 0 ][ 0 ][ 0 ][ 0 ][ 1 ][ 1 ]      /* decimal 3 */
[ 0 ][ 1 ][ 0 ][ 0 ][ 1 ][ 1 ][ 0 ][ 0 ]      /* port 97: LS off */
[ 0 ][ 1 ][ 0 ][ 0 ][ 1 ][ 1 ][ 1 ][ 1 ]      /* (port 97) | 3 */
```

Similarly, applying the bitwise AND operator to an operand can produce a result in which all the bits have identical values to the corresponding bits in that operand if the other operand has every bit set to 1. This means that to change the values of bits 0 and 1 from 1 to 0 while leaving the values of all the other bits unchanged, an operand needs to be acted on by the & operator and a number that has every bit set to 1 except for bits 0 and 1, and this would be the decimal number 252:

```
 128   64   32   16   8    4    2    1
[ 1 ][ 1 ][ 1 ][ 1 ][ 1 ][ 1 ][ 0 ][ 0 ]      /* decimal 252 */
[ 0 ][ 1 ][ 0 ][ 0 ][ 1 ][ 1 ][ 1 ][ 1 ]      /* port 97: LS on */
[ 0 ][ 1 ][ 0 ][ 0 ][ 1 ][ 1 ][ 0 ][ 0 ]      /* (port 97) & 252 */
```

If a Microsoft C compiler is being used, the normal setting for the bits in port 97 can be determined by a call to the `inp` function, and applying the bitwise OR operator and the

number 3 to the value returned, then writing the result to port 97 with the `outp` function, should change only bits 0 and 1 in the interface register, leaving all the others bit settings unchanged:

```
outp(97, (inp(97) | 3));        /* BEEP! - turns on loudspeaker */
```

The loudspeaker can then be turned off by applying the bitwise AND operator and the number 252 to the contents of port 97:

```
outp(97, (inp(97) & 252));      /* turns off loudspeaker */
```

Compilers other than the Microsoft C compiler may have other functions similar to `outp` and `inp` to achieve these tasks, and programmers should consult their compiler manuals for details.

6 Bitwise Shift Operators

6.1 The Left-Shift Operator <<

The left-shift operator << is a binary operator that shifts the bits in the left operand by the number of places given by the value of the right operand:

```
00000000 00000011   /* unsigned int 3 */
00000000 00000110   /* 3 << 1 ==  6 */
00000000 00110000   /* 3 << 4 == 48 */
```

Vacated bits are given the value 0, and bits moved past the end of the left operand are lost:

```
10101100 10011011   /* unsigned int 44187  */
10110010 01101100   /* 44187 << 2 == 45676 */
```

Use of the left-shift operator provides a quick and efficient method of multiplying by powers of two, provided that the problems of overflow are avoided by ensuring that sufficient space is allowed for the result.

6.2. The Right-Shift Operator >>

The right-shift operator is a binary operator that shifts the bits in the left operand by the number of places indicated by the value of the right operand. Again, vacated bits are replaced by 0, and bits past the end of the left operand are lost:

```
0 0 0 0 0 0 0 0  0 0 1 1 0 0 1 0   /* unsigned int 50 */
0 0 0 0 0 0 0 0  0 0 0 1 1 0 0 1   /* 50 >> 1 == 25   */
```

```
0 0 0 0 0 0 0 0  0 0 0 0 1 1 0 0   /* 50 >> 2 == 12   */
```

The right-shift operator provides a quick and efficient method of dividing by powers of 2, although non-integer results are always "rounded down" to the next integer below the true answer, as in the final example above, where 50 divided by 4 returns the answer 12.

7 typedef

The `typedef` statement allows the programmer to give a new name to an existing data type:

```
typedef char * STRING;
```

7.1 Aliases

This statement defines `STRING` as the name of a pointer to `char`, so that the type name `STRING` may be used in the declaration of data and functions that would otherwise be declared as being of type pointer to `char`:

```
STRING reply;                 /* char *reply; */
STRING answer(STRING reply); /* char *answer(char *reply); */
```

A new typename defined by a `typedef` statement is conventionally written in upper-case letters to identify it clearly as a user-defined type.

The scope of a `typedef` definition is local if the `typedef` statement is made within a function and global if it is made outside a function.

7.2 typedef and #define

The `typedef` statement is superficially similar to the `#define` directive, but whereas a `#define` is implemented by the preprocessor, `typedef` statements are enacted by the compiler itself. Although `typedef`, unlike `#define`, is limited to defining names for data and function types, it is much more flexible than `#define`. Function and array types, for instance, can only be defined by `typedef` statements.

Some type names are composed of more than a single identifier, and a `#define` statement cannot conveniently be used to give these types symbolic names, e.g.:

```
int * point1, * point2;
```

can be rewritten:

```
typedef int * ADDRESS;
ADDRESS p1, p2;
```

Attempts to use a `#define` to achieve the same result tend to come to grief, e.g.:

```
#define ADDRESS int *
ADDRESS p1, p2;
```

would result in:

```
int * p1, p2;        /* p2 is an int not a pointer to int */
```

7.3 Type Checking

C does not type check data types defined by a `typedef` statement. Whereas an attempt, for instance, to assign a return value from a `float` type function to an `int` type variable will stimulate a warning message from most compilers, no such warning will appear if `typedef` statements have been used to assign new names to these standard types. This means that additional caution is called for on the part of the programmer to ensure that violations of type do not inadvertently occur.

7.4 Advantages: Readability

The `typedef` statement can be used to improve the readability of code by giving data types relevant and self-explanatory names.

A program to handle accounts, for instance, might only see `float` variables to represent sums of money. In this instance, a `typedef` statement might be used to redefine the type `float` as `POUNDS`:

```
typedef float POUNDS;
POUNDS income, expenditure;
```

The code itself makes it clear that `float` data will be supplied monetary values in this program.

7.5 Advantages: Maintainability and Portability

A change to the range of values likely to be assigned to data items would normally require all relevant declarations throughout a program to be changed to a type appropriate to handle the new size of values without errors through overflow. If a `typedef` statement has been used to name the type of all relevant data, however, a single change to the type given in the `typedef` definition is all that is needed to allow the data throughout the program to accommodate the new size of values.

The size allocated to particular data types is machine dependent. If it is important that a particular group of data items be assigned a type of a specific size, then use of the `typedef` statement again means that if the program is to be used on a different machine, where the data type originally assigned will be implemented with a different and inappropriate size, only a single change to the type defined in the `typedef` statement is needed to ensure that all relevant variables in the program are assigned the correct amount of memory storage.

7.6 Advantages: Increased Abstraction

C permits all meaningful combinations of data types. This facility allows a programmer to create data objects that accurately model a vast range of different kinds of data, but it can also result in declarations and definitions of great detail and complexity.

An array of pointers to structures whose single member is a pointer to arrays of pointers to `float` is, no doubt, a highly useful data type in some applications, but this level of complexity is distracting, if not actually confusing, even to the experienced programmer.

i.e.

```
#define PFSIZE      100       /* for pointers to float */
#define PASSIZE     20        /* for pointers to arrays of structs */


typedef float * APFLOAT[PFSIZE] ;
                    * APFLOAT is an array of 100 pointers to float */


typedef struct {APFLOAT * ftable}S ;
                    /* S is a struct containing a pointer to APFLOAT */


typedef S1 * A[PASSIZE] ;      /* A is an array of 20 pointers to S */
```

The `typedef` statement can hide this complexity behind a simple symbolic name and allows the programmer to concentrate on the larger issues of planning and organizing the program that will actually use the data types that `typedef` has defined. True, the type definitions, in all their complexity, have to be written out somewhere, but this may be done in a header file, where the information is not only hidden from the programmer working on the program design, but may also be conveniently `#included` in any program that requires it.