# Data Types - Chapter Summary

## 1 Functions and main

No item of data can be introduced into a C program unless its name and type are specifically declared before it is used. This declaration reserves an area of memory for storing the current value of the data, and associates a symbolic name (i.e. the name of the data) with that area of memory. This name is then fixed, although the value assigned to the item may change.

The type of data specified determines the amount of memory reserved and the sort of values that may be subsequently stored there.

The design philosophy of C is that the language should combine easily-readable source code with compact and efficient executable code. The declaration of data items helps to achieve both these aims. Declaring the names and types of the data all together at the top of a function makes it easier to read the program and understand what the function does, and how it does it, especially if the data items are given meaningful names. Again, the fact that the compiler is informed how many data items are used, and how much memory each will require, means that exactly enough memory is allocated and no more, so that the compiled code is as efficient as possible.

A declaration statement consists of a type and the name of one or more data items belonging to that type. Each data name is separated from the next in the list by a comma:

```
int year, population, births, deaths;
char initial, grade, gender;
```

## 2 Names of Data Items

Data names consist of letters and digits: `costs` and `income89` are both valid names, but `profit.90` is not, because it contains `'.'`, which is neither a letter nor a digit.

The first character in the name must be a letter; `year1` is a valid name, `1year` is not.

The underscore character `'_'` counts as a letter; `poll_tax` and `_vat` are both valid names for data items (although some old compilers will not accept an underscore as the first character in a data name, and would therefore reject `_vat`).

Upper-case and lower-case letters are differentiated; `fred` and `Fred` are the names of two different data items.

The names given to data items and functions created by the user are known as identifiers. The names of data items that are used only in the files in which they are declared are called internal identifiers, whereas the names of global identifiers, which

are shared between files, are referred to as external identifiers. (The concept of global data items will be covered when we discuss functions).

The number of characters in an identifier that are recognised by the compiler varies with different compilers. C is intended to be a portable language, so that code written on one machine or for one compiler should work equally well on another machine or with another compiler. The ISO standard for C therefore states that identifiers may be of any length, but that the first six characters of external identifiers, and the first thirty-one characters of internal identifiers, must be distinct.

(IBM's alternative SAA standard for C specifies that the first seven characters of an external identifier should be distinct.)

In practice, the number of significant characters allowed in the names of internal identifiers will cause the programmer few problems, but a compiler that recognises only the first six characters of an external identifier will interpret names such as `washington` and `washing_powder` as referring to the same item of data, because the first six characters in the names are the same. There must be a difference in the first six characters of the names if the two data items are to be distinguishable. (For instance, the second item of data in the above example might be renamed `wash_powder`.

It is a convention that the names of data in C are lower case, although other conventions are also coming into use.

A data name may not be the same as a C keyword, nor may data have the same name as any function included in the program, whether the function was written by the programmer or came from the C library.

This means that you cannot call a data item `main` or `int`. It also means that extra care must be taken to avoid using a name already used for a function when adapting an existing program, especially if it was originally written by someone else (such as `printf`).

## 3 Introduction to C Data Types

The fundamental data types in C are:

type `int`, which is used for integers, i.e. whole numbers, whether positive or negative

type `char`, which is used for characters, such as the letters of the alphabet, both upper and lower case, and all the other punctuation marks and symbols included in a computer's character set, such as characters used to represent the numbers from 0 to 9 on screen

types `float`, `double` and `long double`, which are used for numbers that include fractions (i.e. include decimal points), and are used for floating-point, double-precision floating-point and extended-precision floating-point numbers respectively

# 4 Integers

## 4.1 Type `int`

Data of type `int` can be used to store a whole number, whether positive or negative, within a range that depends on the system being used. Typically, data of type `int` will be allocated memory space equal to the word size on the machine concerned (e.g. 16 bits on a 16-bit IBM PC), as the word is the natural unit of memory for a computer, and the one that its central processor will process most efficiently.

## 4.2 `short ints`

If the programmer knows that a variable will have only a limited range, it may be more efficient to declare that variable as a `short int`, providing that the range of values the data will have will be within the range for short integers. Depending on the system, a `short int` will be assigned less or the same amount of memory as data of type `int`, and will accordingly have a smaller range or the same range of values. Like data of type `int`, `short ints` can be either positive or negative.

## 4.3 `long ints`

In the same way, if the programmer knows that data will have very large values, outside the range for data items of type `int`, it should be declared as a `long int`. Again, depending on the system being used, a `long int` will be assigned the same amount of memory or more memory than data of type `int`, and will accordingly have the same range or a longer range than `int` data.

Note the importance of `long ints`; if the value of data of type int exceeds the range for this type, it will be assigned an incorrect value. On a 16-bit IBM PC, for instance, data of type `int` can have values in the range -32,768 to +32,767. If, on such a machine, data of type `int` has the value 32,769, two more than the maximum permitted, the value will wrap around and an incorrect value of -32767 will be assigned. In a similar way, data of type `int` with excessively large negative values can be incorrectly assigned large positive values. The appropriate declaration of `long ints`, which, on a 16-bit PC have a range from -2,147,483,648 to +2,147,483,647, can avoid this problem!

Remember, however, that `long` integers may require more memory than data of type `int` and may also slow down the execution of a program.

## 4.4 `unsigned ints`

Whereas data of type `int` can have either positive or negative values, `unsigned ints` can have only positive values. This means, however, that the range of positive values for an `unsigned int` variable is twice that for data of type `int`.

As stated above, on a 16-bit IBM PC, an `int` variable can have values in the range -32,768 to +32,767. An `unsigned int` variable on the same machine can have a value in the range 0 to 65,535.

In addition to `unsigned ints`, C also allows `unsigned short ints` and `unsigned long ints`.

The ISO standard introduces the keyword `signed` to declare `signed ints` (`shorts` and `longs`) explicitly.

### 4.5 Declarations

The keywords `short`, `long` and `unsigned` can be used as qualifiers in declarations of integer data items. They can be placed in front of the word `int` or instead of `int`, which will be automatically assumed if no other type is specified.

The two forms of each of the following declarations of data items are equivalent:

```
short int legs;              OR   short legs;
long int time;               OR   long time;
unsigned int cheque;         OR   unsigned cheque;
unsigned short int sleeves;  OR   unsigned short sleeves;
unsigned long int johns;     OR   unsigned long johns;
```

## 5 Type char

By their nature, computers can really deal with numbers only, which are stored and processed in binary form as sequences of high-voltage and low-voltage current. Therefore, in order to handle characters, such as letters of the alphabet and punctuation marks, computer software designers have devised conventions that uniquely associate a number with each character to be used. Two of the most common conventions are EBCDIC, which is used on many IBM mainframes, and ASCII (American Standard Codes for Information Interchange), which is used on nearly all personal computers.

To store character data items, C uses a special kind of integer data called type `char`. Although in practice distinguished from the other kinds of integer data, a `char` may be used to contain a single character, and the value stored in memory is the integer value of that character in the machine's character set. A single character constant is formed by enclosing the character in single quotes.

If, for instance, `char` data is given the value `'y'` on a system using ASCII, the actual value stored in memory will be 121, the ASCII code for a lower-case `'y'`. Note that if, on the same system, the value of `char` data has a digit character, such as `'7'`, the value stored in memory will be the ASCII code for that digit, which in the case of the digit `'7'` is 55.

This may be demonstrated by using the `printf` function and the conversion specifications `%c` for characters and `%d` for decimal integers:

```
main()
{
     char testy;
     char test7;

     testy= 'y';
     test7= '7';

     printf(Letter %c has ASCII code %d.\n, testy, testy);
     printf(Number %c has ASCII code %d.\n, test7, test7);
}
```

When compiled and run, the above program will print out:

```
Letter y has the ASCII code 121.
Number 7 has the ASCII code 55.
```

Note that the control characters for escape sequences can also be assigned to `char` type data items, as demonstrated in the example program in this chapter.

## 6 Type enum

There are some types whose purpose is to take only one of a number of named values. For instance the seasons are one of a set of 4 values spring, summer, autumn, and winter. To communicate these ideas more clearly a separate type name will make code easier to understand. In principle these discriminated sets could `typedef` an `int` to an appropriate type, and provide a set of `const` variables for use. This is the approach adopted in some languages. For example:

```
#define SEASON_T int
#define SPRING 0
#define SUMMER 1
#define AUTUMN 2
#define WINTER 3
SEASON_T coldest = WINTER;
```

or better still:

```
typedef int season_t;
const season_t spring = 0;
const season_t summer = 1;
const season_t autumn = 2;
const season_t winter = 3;
season coldest_t = winter;
```

This is clearly better than simply using a plain int (a "magic type") and some arbitrary number constants ("magic numbers"). For example:

```
int coldest = 3;
```

However, the creation of a higher level enum type raises the abstraction of the code and makes it even more declarative.

## 6.1 Setting up an enum

enums are an example of *user defined types* – UDTs. They provide a mechanism for introducing a new type with a name and a related set of constant values that can be used with them. Each enum definition defines a new type that is separate from other enum types.

```
enum season_t { spring, summer, autumn, winter };
```

An enum value is like an integer in that it is represented like one and its value can be implicitly converted to one. By default the enumeration constants are numbered from 0 upwards in steps of 1. So in the example above, spring has the value zero, summer the value one, autumn the value two, and winter the value three. If particular values are preferred these can be specified in the definition. For example:

```
enum season_t { spring=1, summer=2, autumn=3, winter=4 };
```

More precisely, the value of enumeration constants default to one more than the previous enumeration constant. So in the above example, the explicit specifications of summer, autumn, and winter are not required.

```
enum season_t { spring=1, summer, autumn, winter };
```

Initialising an enum constant value is one of the few places that C requires a compile time constant value. Integers can be implicitly converted to enum values but this is not a good idea (and is not allowed in C++ which might an issue if C++ compatibility is required). When output an enumeration 'degenerates' to its underlying integer value.

There is no simple way to read an enumeration from input. Lookup tables represent a useful way of tackling this.

Enumeration constants are true compile time constants. This means that an alternative to specifying the size of an array using the preprocessor:

```
#define MAX_BUFFER 32
char buffer[BUFFER_SIZE];
```

is

```
enum buffer_t { buffer_size = 32 };
char buffer[buffer_size];
```

The latter is preferred as it avoids the preprocessor, and hence, the structure is visible not only to the reader but to the compiler as well. For example, the following does *not* cause a duplicate definition compiler error:

```
#define MAX_BUFFER 32
…
#define MAX_BUFFER 42
```

whereas, the following *does* cause a duplicate definition compiler error:

```
enum buffer_t { buffer_size = 32 };
…
enum limit_t { buffer_size = 42 };
```

## 6.2 Declaring enum variables

The syntax for declaring an enum variable follows the regular syntax for a declaration. First name the type, then name the variable. For example:

```
int tally; /* declares tally, an int */
enum season coldest; /* declares coldest, a season */
```

As usual an enum can be initialised at its point of declaration, and assigned to in an expression.

```
enum season coldest = winter;
coldest = winter;
```

# 7 Floating-Point Data

## 7.1 Real Numbers

Types `float`, `double` and `long double` are used for data items with non-integer values; that is to say, for numbers including decimal points, such as 3.14159, 9.8 or 100.0, which are sometimes referred to as real numbers. Note that in C, 100 is an integer, but 100.0 is a floating-point number.

## 7.2 Scientific Notation

In scientific applications, floating-point numbers are often written in either standard form, or in exponential notation (or E numbers). Confusingly, both standard form and exponential notation are sometimes called scientific notation, but in this course the phrase is used to refer to exponential notation only.

In either form of notation, a given number is represented by a floating-point number between one and ten, known as the mantissa, multiplied by ten raised to the appropriate exponent (ten to the power of...). The exponent is zero or a positive or negative integer. Exponents greater than or equal to zero indicate that the number represented is greater than or equal to one, while negative exponents indicate a number less than one.

The following, for instance, are three ways of writing the value of the Faraday constant in coulombs per mole:

| Decimal | Standard Form | Exponential/Scientific Notation |
|---|---|---|
| 96487.0 | 9.6487 x 104 | 9.6487E4 |

## 7.3 Type `float`

On most systems, data items of type `float` are assigned 4 bytes (32 bits); roughly one byte for the exponent part of the number and three bytes for the mantissa. In Microsoft C, for instance, 32 bits are assigned as follows: 8 bits for the exponent, 1 bit for the sign of the exponent (positive or negative) and the remaining 23 bits for the mantissa. This gives data items of type float a `range` of approximately 3.4E-38 to 3.4E+38.

This is equivalent to a digit precision of approximately six decimals. Note that digit precision is not the same as significant places. If a value with more than seven significant digits is assigned to data of type `float`, the additional digits will not be rounded off, they will be simply lost.

## 7.4. Type `double`

Data items of type `double`, for double precision, have twice the digit precision of data items of type `float`, approximately twelve decimals, and use twice as many bits for storage. Typically, data items of type double are assigned 64 bits, but how this is divided between the mantissa and the exponent depends on the system being used.

Microsoft C, for instance, assigns 11 bits to the exponent, one bit for the sign of the exponent, positive or negative, and the remaining 52 bits for the mantissa. This gives data items of type double a range of approximately 1.7E-308 to 1.7E+308. Other systems may assign more bits to the exponent, which gives a greater range of numbers, but reduces the digit precision available.

### 7.5 Type `long double`

The ISO standard for C also defines data type `long double`, which on most systems would use 80 bits and give approximately 24 digits of precision.

### 7.6 Overflow

Exceeding the permitted range for data of type `float`, `double` or `long double` may produce an error message, and will almost certainly produce a wrong answer.

### 7.7 Output of Floating-Point Numbers

The value of floating-point data can be displayed by using the `printf` function and the `%f`, `%e` and `%g` conversion specifications.

## 8 Initialising Data Items

Data items may be initialised to specific values in their declarations by using the = initialisation operator. Just as it is possible to declare several data items of the same type in a single declaration, so it is also possible to initialise several data items of the same type in the same declaration. Note that data items may be assigned the value of either a constant or an expression.

For example, the following are all legal declarations of data items:

```
int       cricket = 11,
          football = 11,
          rugby = 15;
long      secs_in_yr = 60 * 60 * 24 * 365;
double    speed_of_light = 2.9979E8,
          Earth_to_Sun = 1.496E11;
char      tab = '\t',
          backspace = '\b',
          big_a = 'A';
```

# 9 Constants

Constants in C are handled in one of two ways: as `const` data items or as constant literals. The preprocessor directive `#define` is used to organise the latter, which although covered in the Preprocessor chapter, is mentioned briefly in this section.

## 9.1 The C `const`

So far, the data items defined in our data section at the top of our function blocks have been variables (i.e. potentially, they can be updated in the code section). This is the default; all items defined in this way may be assigned to or altered by function calls. If, however, the programmer decides that a data item will not need to change during its existence, it can be qualified with the keyword `const`. This indicates, to the compiler amongst others, that there is no intention to change the data. Indeed, the compiler will check that you have not updated it in the code.

The `const` was introduced in the ISO standard after its success in C++ and has been adopted by many programmers who wish to increase code integrity. However, there is one rule: the `const` data item MUST BE initialised at definition, i.e. using the syntax described in Section 7 above. The keyword `const` is placed in front of the conventional definition. Some examples follow.

```
const int Triangle_sides = 3;        /* Capital letter is used */
const float Pi = 3.14159;            /* for clarity            */
```

## 9.2 Constant Literals

A constant literal is a pure number, integral or floating-point type, a character or a string (covered later). Some examples of integral constants include :

```
4          /* int or short int */
04         /* octal int or short int */
0x4        /* hex int or short int */
40000L     /* long int*/
40000U     /* unsigned int */
...        /* See Appendix A for more */
```

Floating-point constants, used anywhere in C expressions, may be given in either standard decimal or in exponential notation. Note that if the latter notation is used, there must be no spaces in the number. For example, `2.3E4` would be legal, but `2.3 E4` would not.

It is not strictly necessary to put the zero before the decimal point when representing a number less than one, but including the zero improves readability.

The following are all possible forms of floating-point constants:

```
9.8   0.23      23    9.3E6     7.678E-3        /* type double */
9.8F  0.23F     23F   9.3E6F    7.678E-3F       /* type float */
```

Floating-point constants are stored as `doubles`, unless the suffix `F` is appended to denote a single-precision `float` constant or `L` for a `long double`.

`chars` are simpler. All characters are single symbols enclosed in single quotation marks, with the exceptions of special control characters and for octal or hexadecimal constants, e.g.:

```
'\t'          (a tab)
'\007'        (octal 7)
'\0xFF'       (hex FF)
```

## 9.3 Symbolic Constants

As will be explained further when we look at the C preprocessor, constants may be given symbolic names, which are usually written in capitals, where this improves the readability of a program. The value of the constant will then be defined by a `#define` statement at the top of the program or in a separate header file, e.g.:

```
#define GRAVITY (9.81)     /* acceleration due to gravity in ms-1 */


main()
{
     double acceleration = GRAVITY;
     ...
}
```

# 10 Storage Requirements for Data Types

The amount of memory allocated to each data type, and hence the range of values it can represent, is not defined in the C standard, and so is dependent on the system being used. The following table shows the number of bytes allocated to each data type on a sample range of systems:

| | IBM PC | DEC PDP-11 | Intel 80386 | DEC VAX |
|---|---|---|---|---|
| int | 2 | 2 | 4 | 4 |
| short | 2 | 2 | 2 | 2 |
| long | 4 | 4 | 4 | 4 |
| char | 1 | 1 | 1 | 1 |
| float | 4 | 4 | 4 | 4 |
| double | 8 | 8 | 8 | 8 |

The `sizeof` operator can be used to determine the amount of memory allocated to a data type or variable. It returns a value representing the number of bytes used to store that data type.

For example, the availability of the `long double` type could be checked as follows:

```
main ()
{
     int ldble_size;

     ldble_size = sizeof (long double);
     printf ("\n long double: %d bytes", ldble_size);
}
```