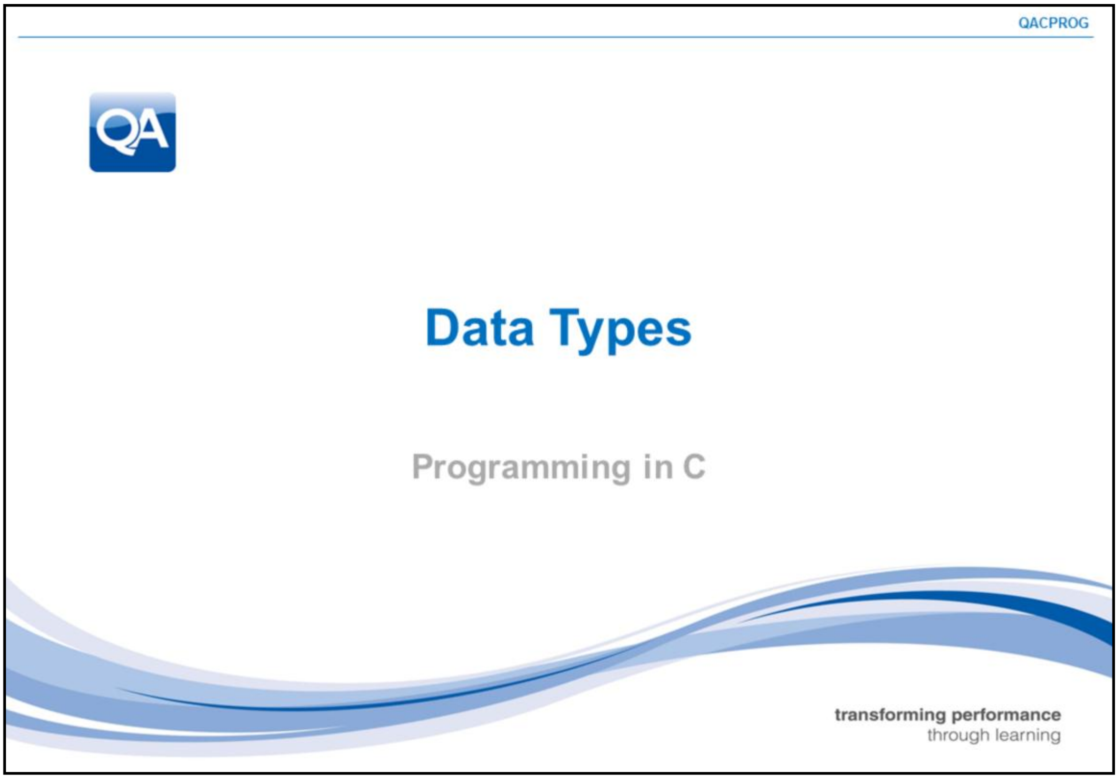
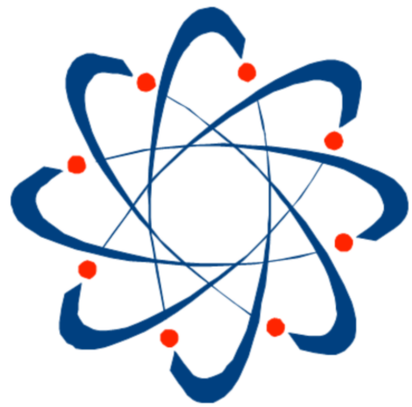

QACPROG
Data Types
Programming in C
transforming performance
through learning

QACPROG

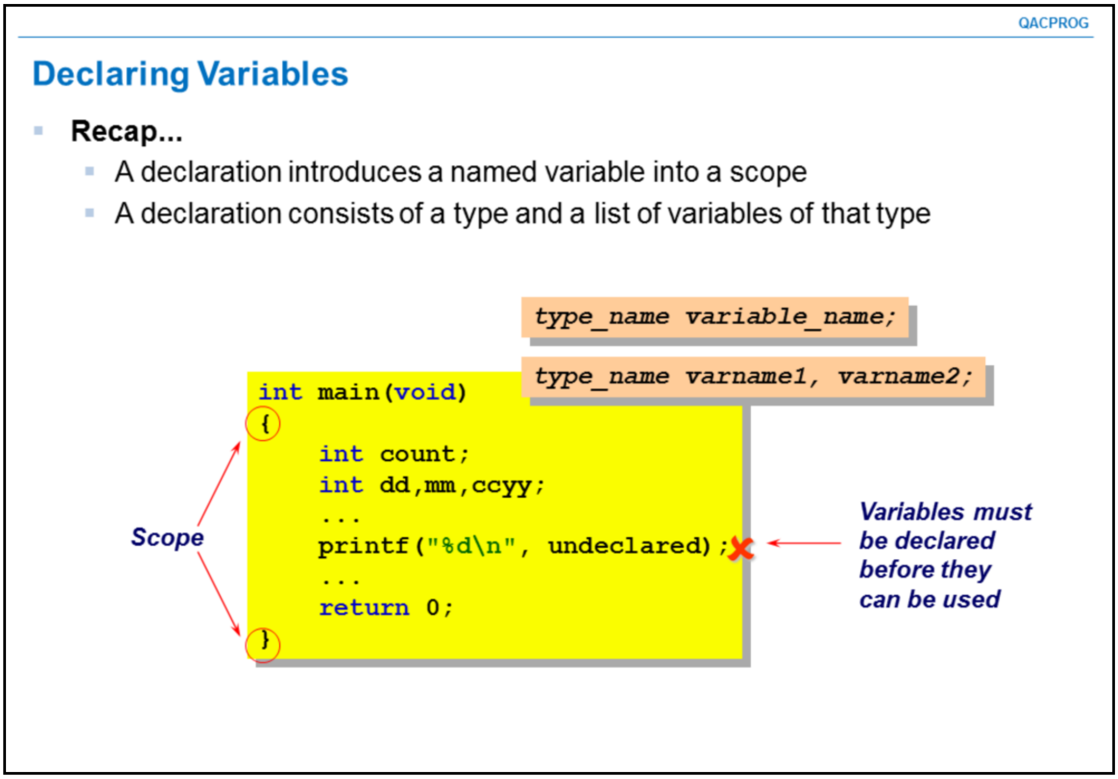
Data Types

- **Objective**
  - To design simple data using C's built-in scalar types
- **Contents**
  - Declaration recap
  - Initialisation
  - Integer numbers
  - Enumerations
  - Floating point numbers
  - Characters
  - Constants
  - typedef
- **Summary**



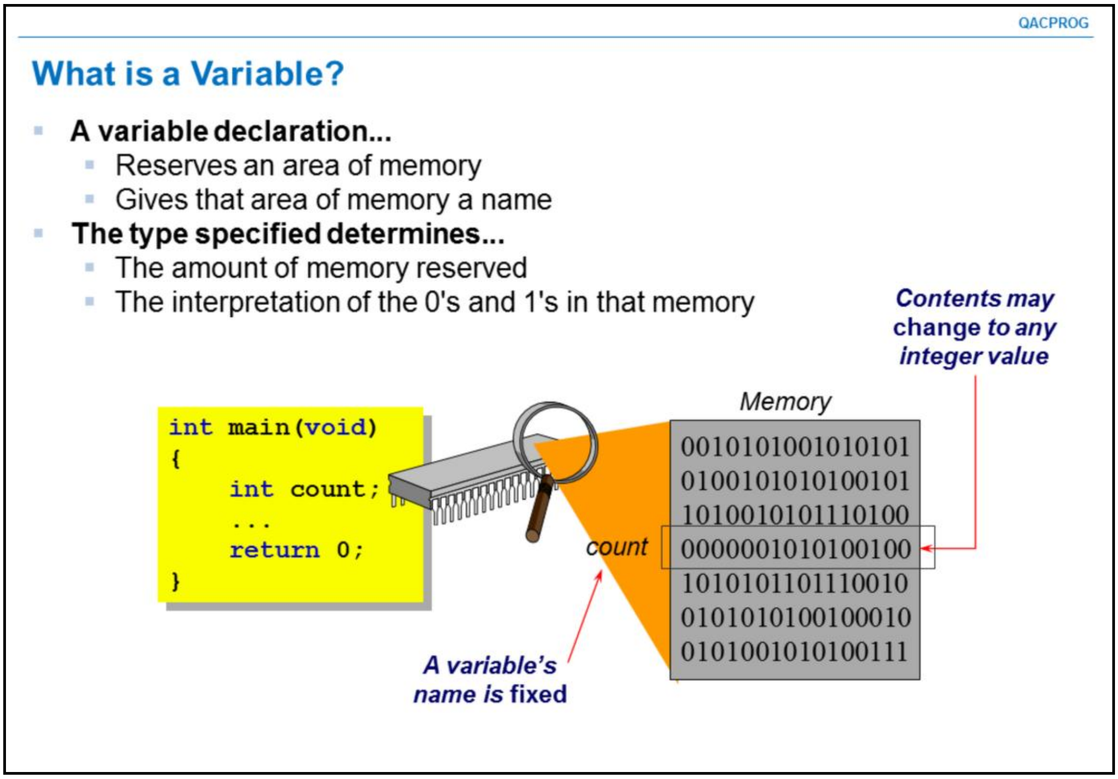
The objective of this chapter is to introduce C's fundamental data types. It is the first of two chapters that cover the details of the syntax and functionality of C statements. This chapter looks at declaration statements. It covers all of C's scalar data types. It introduces syntax for constant declarations. It also introduces data initialisation at declaration time. This enables programmers to choose, and hence guarantee, the initial value of a data item; a good programming practice.

The exercise at the end gives you some practice with character and floating-point data.



The format of a declaration must adhere strictly to C's exact syntax. For scalar variables, it consists of the type name, followed by a list of one or more comma-separated identifiers. Some coding standards prefer to restrict declarations to one data item per line.

In the slide, the printf statement will not compile. The compiler will rightly complain that it does not know what `undeclared` is. You cannot just pluck identifiers out of thin air and expect the compiler to magically know what they refer to!



A { block } is the formal name for a section of code between an opening brace and a matching closing brace. Data can be declared at the beginning of any block. By doing so, the programmer has named some data, i.e. variables and constants, which are used within that block.

The programmer supplies a typename and a variable name. The typename can be the name of a built-in types (such as int) or the name of a user-defined type (mechanisms for creating user-defined types are covered in later chapters). The variable name must be a legal sequence of alphanumeric characters.

The compiler prepares the storage requirements using predefined compiler-specific size information. The information includes the physical size of each data item in bytes and its representation, i.e. bit pattern. Neither is specified in the language; both are features of the implementation. The compiler is also aware of the usage of the data, i.e. which subset of the language's 45 operators can be used.

QACPROG

Initialising Variables

- **Variables may be *initialised* in their *declarations***
  - Initialising variables is good programming practice
  - It ensures that when the variable is used it has a valid value

```
...
int initialised = 42;
int also_initialised = initialised;
int not_initialised;
...
printf("%d\n", initialised);
printf("%d\n", also_initialised);
printf("%d\n", not_initialised);
...
```

By default, all local variables are created on the stack. Their initial value is indeterminable, i.e. garbage. C provides a way around this. The *initialisation* of variables is very flexible. Values used for the *initialisation* of scalar variables can be very complex runtime expressions. The syntax is neat and concise:

```
type variable = expression;
```

Initialisation is carried out during data declaration. This is slightly more efficient than runtime assignment (which is covered in the *Expressions* chapter). It is also better programming practice!

In the slide the first and second printf statements print the values of `initialised` and `also_initialised`, which are both 42. The third printf statement attempts to print the value of `not_initialised`, which as its name suggests, is not initialised! It has been declared, but it has not been initialised. Or to put it another way, the memory for it has been allocated, but the bit pattern inside that memory has not been set. Strictly speaking, this results in undefined behaviour (it is possible that the bit pattern is not a legal bit pattern for that type).

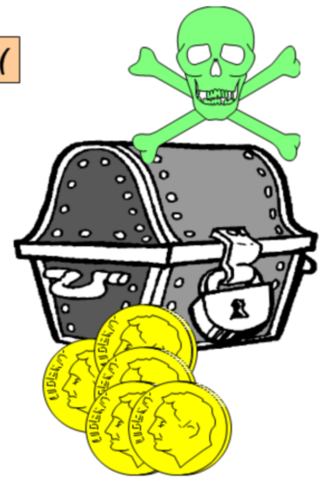
QACPROG

Integer Data

- **The common integer type is represented by int**

```
int curse;
int pieces;
int seas;
```

no initialisation :-(



- **Integer literals may be...**
  - Hexadecimal; prefixed with a `0x` or `0X`
  - Octal; prefixed with a `0`
  - Decimal;

```
int curse = 0x000D;  /* thirteen */
int pieces = 010;    /* eight */
int seas = 7;        /* seven */
```

initialisation :-)

octal digits

0 1 2 3 4 5 6 7

Octal is not as commonly used now as it used to be, but it is convenient to set file permissions on Unix using octal.

decimal digits

0 1 2 3 4 5 6 7 8 9

Decimal is the most commonly used written number representation — which should hopefully not come as too much of a surprise!

hexadecimal digits

0 1 2 3 4 5 6 7 8 9
A B C D E F
a b c d e f

Hex is commonly used when dealing with byte values, where each byte has the range `0` to `15`. One example is specifying colours in RGB (red-green-blue). The value `0xff00ff`, for instance, is magenta.

QACPROG

Other Integer Types

- ***long*** **or** ***long int***
  - Used in preference to plain *int* when a large integer value is needed
  - *long* also has its own literal constant form with an appended *L* or *l*

```
const long c = 299792458L;
```

- ***short*** **or** ***short int***
  - Used when a smaller integer range is needed and space is an issue
  - *short* does not have its own literal constant form

```
short day_in_year = 361;
```

- ***unsigned*** **can be specified for** ***long***, ***int*** **and** ***short*** **types**
  - *unsigned* types have no negative range
  - Literal form uses an appended *U* or *u*

```
unsigned long four_gig = 0xffffffffUL;
```

Additional specifiers can be used to declare variants of the basic `int` type. `signed` is the default signedness for `int` but is very rarely written explicitly. The legal combinations are listed here. Type names on the same line are synonyms, with the most commonly used form on the left hand side.

signed integer types:

```
short          short int          signed short int
int            signed int
long           long int           signed long int
```

unsigned integer types:

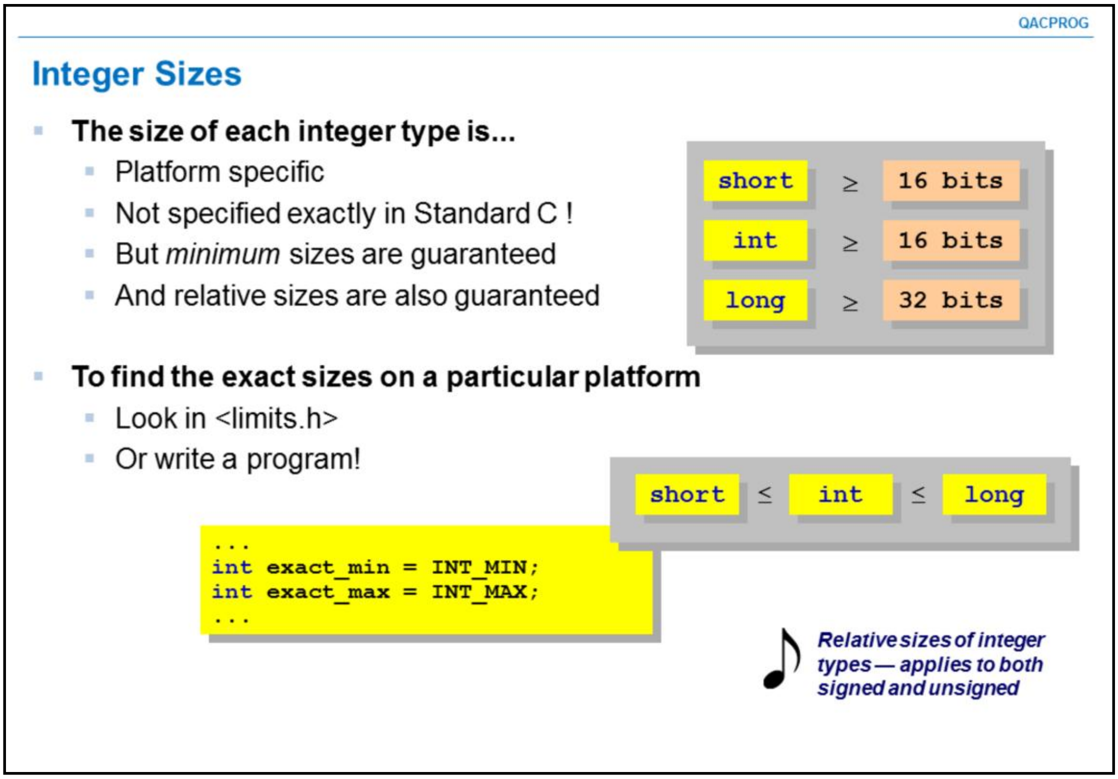
```
unsigned short       unsigned short int
unsigned             unsigned int
unsigned long        unsigned long int
```

character types:

```
char
unsigned char
signed char
```

The plain int type is the int type to use unless you have a good reason not to. It will correspond directly with the natural word size of the computer architecture you are compiling on.

The keyword `unsigned` is intended to convey more meaning to the code. Some variables in the program represent properties that cannot have negative values. For example, the number of sweets in a bag, the height of a person, the age of a person etc.



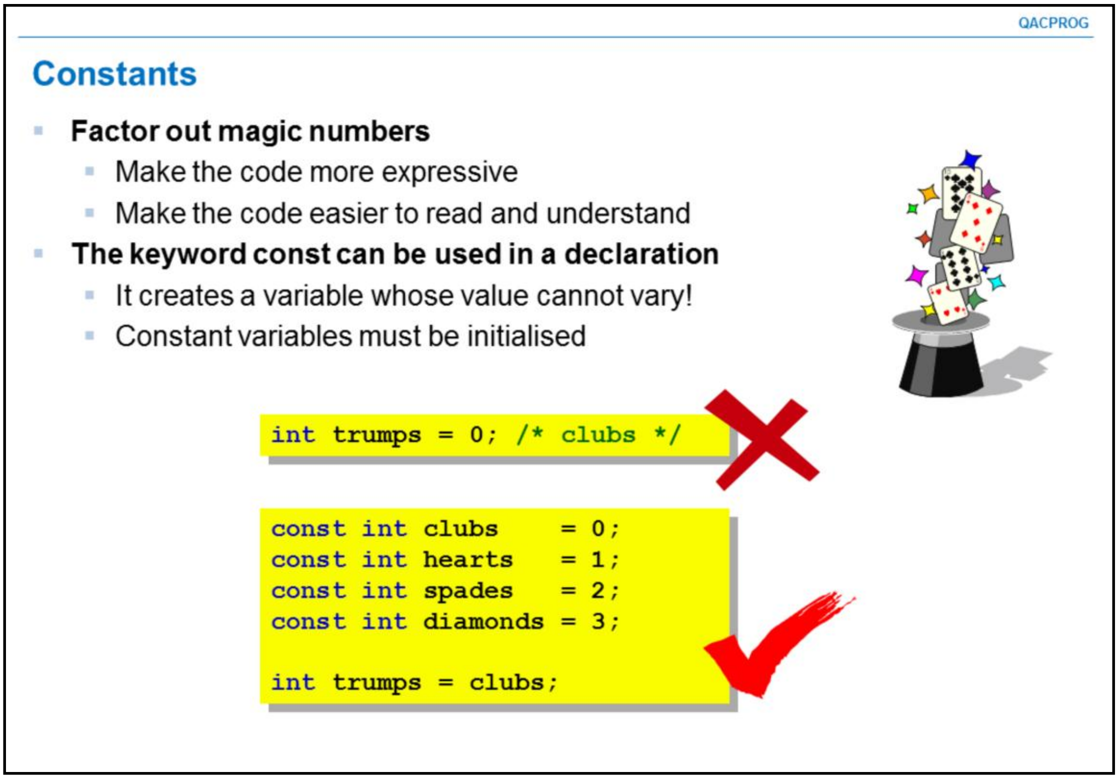
The C standard specifies the sizes of the integer types via minimum limits. For example, `int` must have *at least* 16 bits. This means that the representation of an `int` on *all* platforms will be at least 16 bits, but that *specific* platforms are allowed to represent `int` using *more* than 16 bits. This is an interesting approach to portability that makes even the fundamental integer types a little bit "abstract".

The C standard also guarantees the *relative* sizes of the integer types. The size of a `long` is guaranteed to be greater than *or equal* to the size of an `int`, and the size of an `int` is guaranteed to be greater than *or equal* to the size of a `short`. This means that a platform can legally represent all the integer types using 32 bits; and some platforms do indeed do this.

A running program will of course be executing on a specific platform, and may well need to know the *exact* integer sizes on that *specific* platform. The program can find this information in a number of ways. There is a standard header file, <limits.h>, that specifies that minimum and maximum values for the integer types. The program can include the <limits.h> specific to that compiler/platform to find the values specific to that compiler/platform. Hence the full range of values can be determined.

To determine the number of bits the integer type uses to represent all the values in this range the programmer can use of the `sizeof` operator. The `sizeof` operator returns the number of *bytes* used to represent the type of its argument. In combination with CHAR_BITS which is also defined in <limits.h> and defines the number of bits in a *byte*, the number of bits in an integer type can be calculated. An example is shown in the *Integer Representation* slide.

It is also possible to write a small algorithm that uses the left bitshift operator to determine the number of bits used to represent an integer type. The left bitshift operator is mentioned in the *Bitwise Operators* slide in the *Expressions* chapter.



In C, constants are treated as read-only variables. The two major differences between variable data and constant data are in the declaration. The keyword `const` is placed just in front of the type name, and the data must be initialised. The rules for initialisation are the same as for variables. The compiler will check for attempts to update a constant variable at runtime.

Constant variables have a major advantage over raw literals. It is simply that they have a name! Compare:

```
int month = 0;
```

with:

```
const int january = 0;
...
int month = january;
```

The latter is better for a number of reasons. An expressive identifier like `january` (which happens to be `const`) is far easier to understand than a plain value such as zero. Also the use of an identifier removes potentially awkward questions about the *particular* values to use. For example, should you use the value zero or the value one for "january"? If you leave the choice out in the open then it's almost certain that some programmers will choose zero and some will choose one. Debug time. And even if you document that zero is the correct choice bugs will still creep in. After all, the last thing programmers do (literally) is read the documentation! The best solution is to unask the question. Get rid of the raw literals and use meaningful identifiers instead (possibly via the enum construct; see *Enumeration Type* slide)

The use of constants is actively encouraged.

QACPROG

Enumeration Type

- **A distinct type taking a set of named constant values**
  - Enumerations have integer values
  - Start at *0* and go up in steps of *1* unless otherwise specified
  - Implicit conversion from *enum* to integer types
  - Implicit conversion from integer to *enum* types :-}

```
enum suit { clubs, hearts, spades, diamonds };
enum season { spring, summer, autumn, winter };
```

```
enum suit trumps = hearts;
enum season discontent = winter;
int chicken = spring;
enum season fall = 2;
```

```
hearts = -1;
++winter;
```

Error

There are some types whose purpose is to take only one of a number of named values. For instance traffic lights have a fixed set of colours.

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. Although better than simply using a plain int (a "magic type") and some arbitrary number constants ("magic numbers"), the creation of a higher level enum type raises the abstraction of the code and makes it more declarative.

`enum`s 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.

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`. If particular values are preferred these can be specified in the definition. 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++).

When output an enumeration 'degenerates' to its underlying integer value, but there is no simple way to read an enumeration from input. Lookup tables represent a useful alternative way of tackling this (see the *Lookup Tables* slide in the *Arrays* chapter). Note also that an enum literal value is a true constant, and *can* be used to declare the size of an array.

QACPROG

Character Data

- **Single characters can be represented with *char***
  - The literal form uses single quotes around the character
  - *Escape characters* permit control characters to be specified easily
- ***char* is a kind of integer**
  - Implicit conversion from char to integer
  - Implicit conversion from integer to char :-}
  - Some conversions from *int* look suspicious and may lose precision

```
char lower_a = 'a';
printf("%c", 'q');
printf("%c", lower_a);
```

Escape character examples

```
const char tab = '\t';
const char nl  = '\n';
const char cr  = '\r';
const char nul = '\0';
const char sqt = '\'';
const char bsl = '\\';
```

```
int  space_code  = ' ';
char what_value  = 256;
```

`char` is a keyword for the type that holds single characters. Like many modern languages, C makes a distinction between single characters and sequences of many characters — strings are introduced in the *Arrays* chapter. This distinction is also found in Java and Ada, but not in VB or Fortran. The `char` is the fundamental unit of storage or memory in a C program. It is often equated with a *byte* when dealing with systems programming.

The actual character set used is defined by the compiler and operating system implementation. Common examples include ASCII, Latin-1, Latin-2, etc. Programmers should be careful not to make unnecessary assumptions about alphabetic ordering and character ranges.

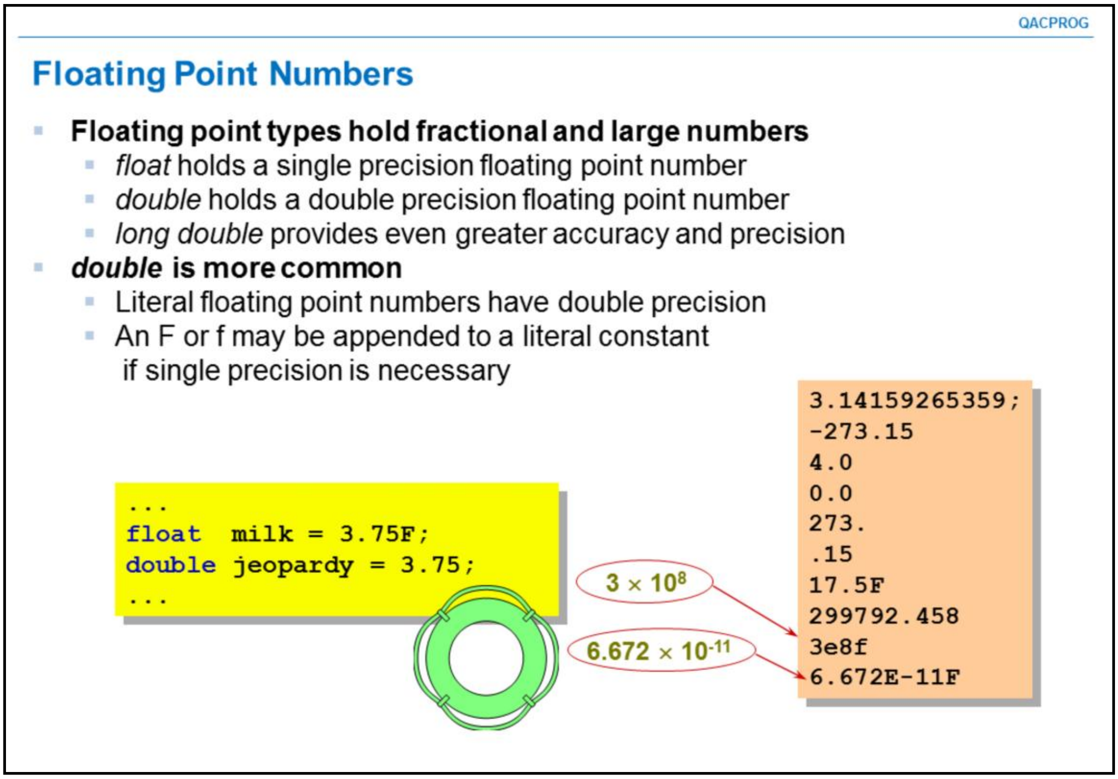
Similarly, although `int` and `char` may be used interchangeably, a `char` is in practice smaller than an `int`: on a typical 32 bit system a `char` will have 8 bits and an `int` will have 32 bits. Assigning values like `256` or `65535` to a `char` will result in truncation. It is generally advised to stick to using `int` for numeric data and `char` for character data.

Some characters cannot be written easily in source code. Escape characters provide a convenient way of specifying such control characters:

`'\a'`	*alert (e.g. bell)*
`'\b'`	*backspace*
`'\f'`	*form feed*
`'\n'`	*newline*
`'\r'`	*carriage return*
`'\t'`	*horizontal tab*
`'\v'`	*vertical tab*

`'\\'`	*backslash*
`'\''`	*single quote*
`'\"'`	*double quote*

`'\ddd'`	*character with octal code* `ddd`
`'\xdd'`	*character with hex code* `dd`



The compiler thinks in terms of doubles, which is the default for floating-point arithmetic. To overcome this, the programmer uses 'F' and 'L' to force the compiler to use float and long double manipulation, respectively. These are illustrated in the example below which also shows the scanf/printf format specifiers. Note that the printf format specifier for a double is different to the scanf format specifier for a double.

```
int main(void)
{
        float f;
        double d;
        long double ld;
        scanf("%f", &f);
        printf("%f", f);

        scanf("%lf", &d);
        printf("%f", d);


        scanf("%Lf", &ld);
        printf("%Lf", ld);


        return 0;
}
```

typedef

- **A typedef provides a new name for an existing type**
  - Creates a synonym for a type
  - Does not create a new type

```
typedef existing_type synonym;
```

- **Reduces the need for magic type names**
  - More *meaningful* names can be provided based on intended use
  - Abbreviations can be given for complex and long type names

```
typedef unsigned char byte;
typedef unsigned int  size_t;
typedef unsigned long ulong;
```

```
byte    lo, hi;
size_t length;
ulong   key;
```

```
typedef enum suit       suit_t;
```

```
suit_t trumps;
```

size_t is part of Standard C

New type names may be used for abbreviation of long type names or to give a more meaningful name for the types of certain variables, i.e. something more descriptive than simply `int` or `unsigned`. The scope of a `typedef` declared type name is the same as that for other variables if declared within a block: from its point of declaration to the end of the block. However, it is more typical to declare types at file level outside functions, e.g. before `main` or in a header file for use by others.

In the example above, the `size_t` type is a type that is provided by the standard library for use in declaring size related values, e.g. array sizes or indices. The other types are ones often provided by the programmer or by a third party library. Naming convention can vary, so depending on the system and preferred style you may find `byte`, `Byte` or `BYTE`.

QACPROG

Summary

- **Declarations**
  - Initialisation is a Good Thing
  - Constants are declared using the const keyword
- **Integers**
  - *int* is most common, short and long types exist too
  - *unsigned* to declare non negative values
- **Enumerations**
  - ints in disguise, help make code more expressive
- **Floating point numbers**
  - Large or fractional values
  - *double* is most common, float and long double exist too
- **Characters**
  - Single quote literals
  - Escape characters such as '\n'

QACPROG

Common Pitfalls

- **Literals**
  - Using a leading *0* for a decimal number
  - Using double quotes for *char* literals
  - Lower case *l* can look like *1*
- ***typedef***
  - Doing it the wrong way round
- **Declarations**
  - Using a type name as an identifier
  - Using a keyword as an identifier

```
march  = 03;     ✓
august = 08;     ✗

char yes = 'k';  ✓
char no  = "k";  ✗

misleading = 654321;

typedef real double;   ✗

typedef double real;   ✓
int real;              ✗

int register;          ✗
```

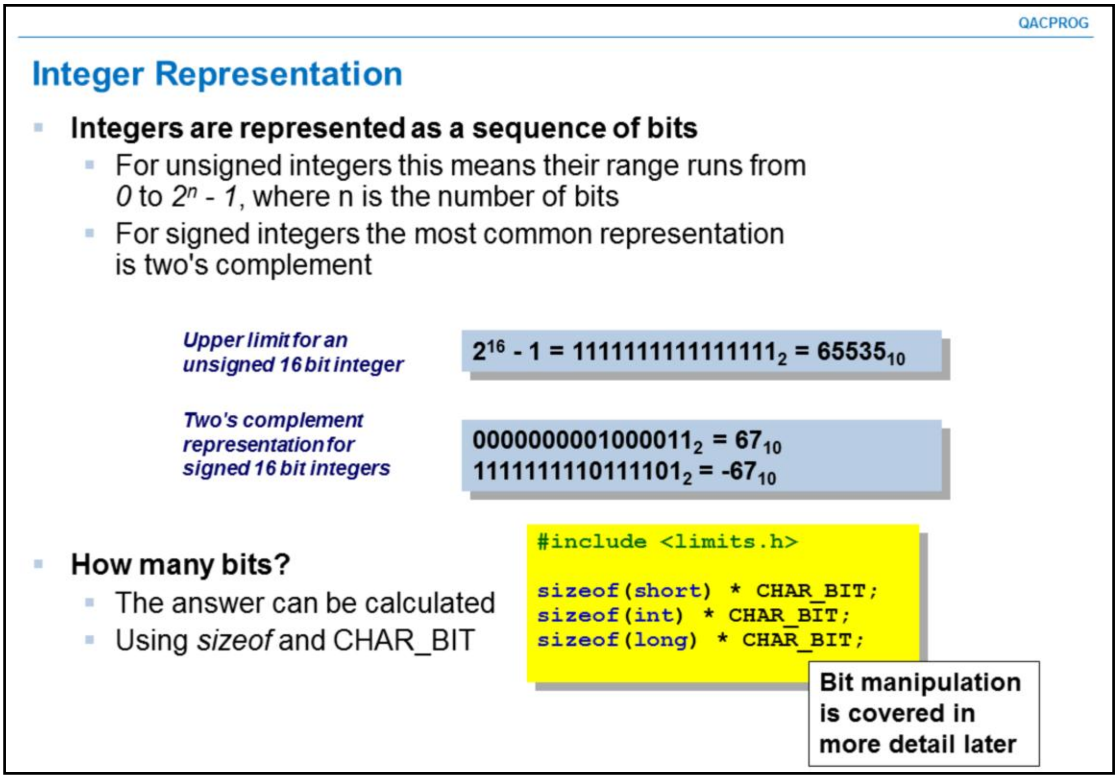
A leading `0` on a number literal indicates that it is an octal number. The only digits possible for base `8` are `0` through `7`, inclusive..

The `char` type in C represents a single character rather than a sequence or string of characters. The literal form for this uses single quotes, e.g. `'k'`. This is not the same as using double quotes around a single character, i.e. the compiler will treat `"k"` as a full string data type rather than a single character. This will lead to a type mismatch error.

`long` literals use a suffixed `L` or `l`. It is clearer to use the upper case form as the lower case form can be easily confused with the digit `1`: `65432L` versus `65432l`.

The `typedef` syntax follows the standard C declaration order; i.e., type followed by the identifier you are introducing. Therefore the new type name is on the right and not on the left hand side.

There are several lesser used keywords that you might accidentally use as an identifier. For example, `register` is a keyword, and also a word in English with several meanings. It is worth learning the names of all the keywords.



The number of bits used in the representation of integer types determines their range, as does their signedness. Typically a `char` has 8 bits, a `short` has 16 bits and a `long` has 32 bits. The size of an `int` is often, but not always, the word size of a machine. Therefore on 16 bit systems an `int` will have 16 bits and on 32 bit systems an `int` will have 32 bits. Although these are the most common sizes, all that is guaranteed is that a `char` will have at least 8 bits, a `short` at least 16, and a `long` at least 32 with the additional constraints imposed on relative sizes (shown in the slide above). This allows for machines which may support 64 bit versions of `long`, and indeed `char`!

The plain `char` type may use either signed or unsigned representation. Which is used is platform specific. If you definitely need one or other representation for some purpose, the types `signed char` and `unsigned char` should be used.

The upper and lower limits for each of these types are listed as constants in `<limits.h>`. For example:

```
printf("int: %d to %d\n", INT_MIN, INT_MAX);
```

The number of bits in a `char` is also defined in `<limits.h>`:

```
printf("bits in a char: %d\n", CHAR_BIT);
```

The `sizeof` operator can be used to determine the size in bytes of a given type.

```
printf("sizeof(double):\t%d\n", (int)sizeof(double));
```

The number of bits used in representing a type can be calculated:

```
int bits = (int)(sizeof(long) * CHAR_BIT);
printf("bits in an long:%d\n", bits);
```

Bit manipulation is discussed in more detail in the Further Data Types chapter.