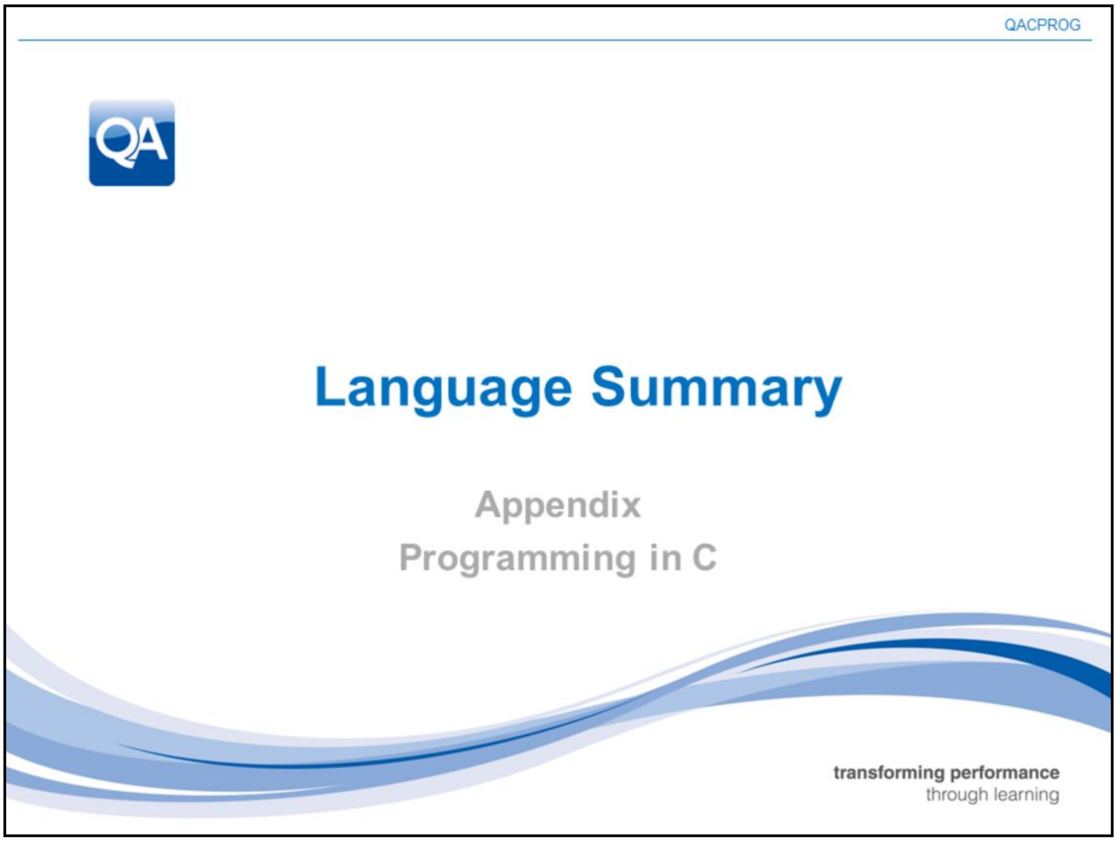


QACPROG

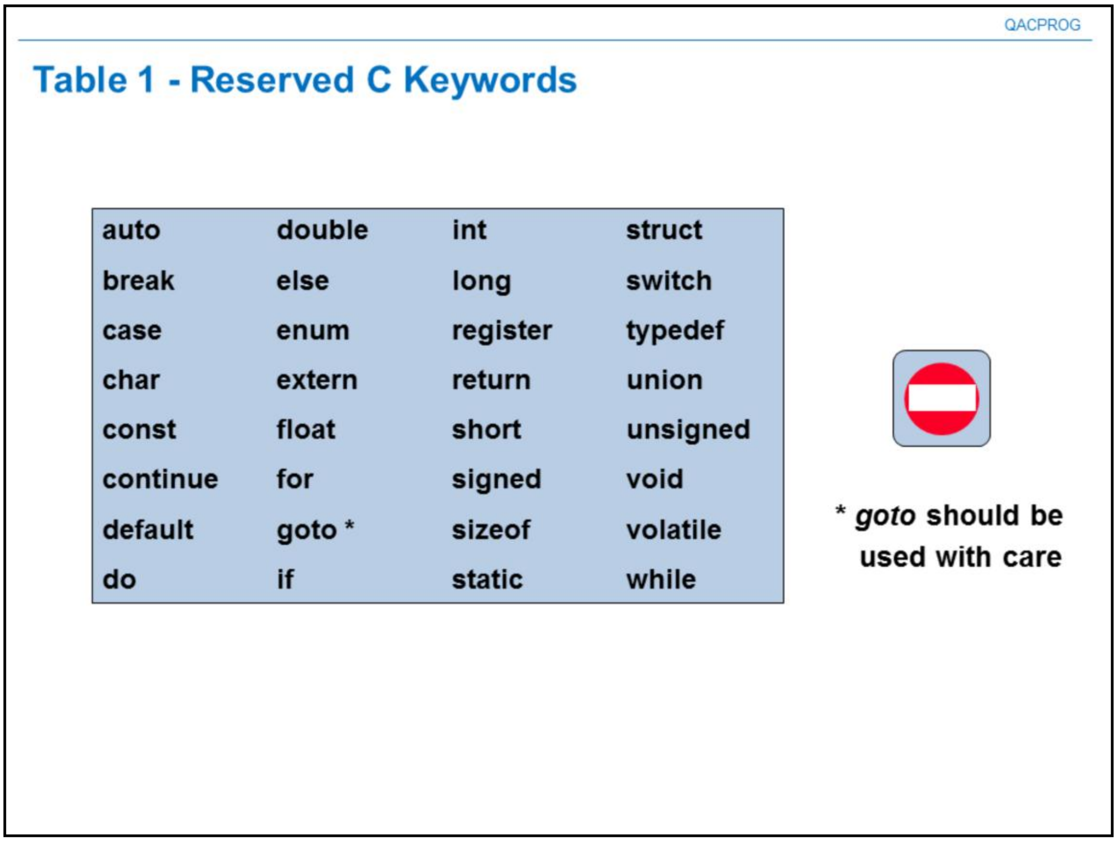
Language Summary

Appendix
Programming in C

transforming performance
through learning

This appendix summarises C’s reserved keywords, built-in data types and operators. It also provides other useful language reference items .

This appendix consists of eleven tables:



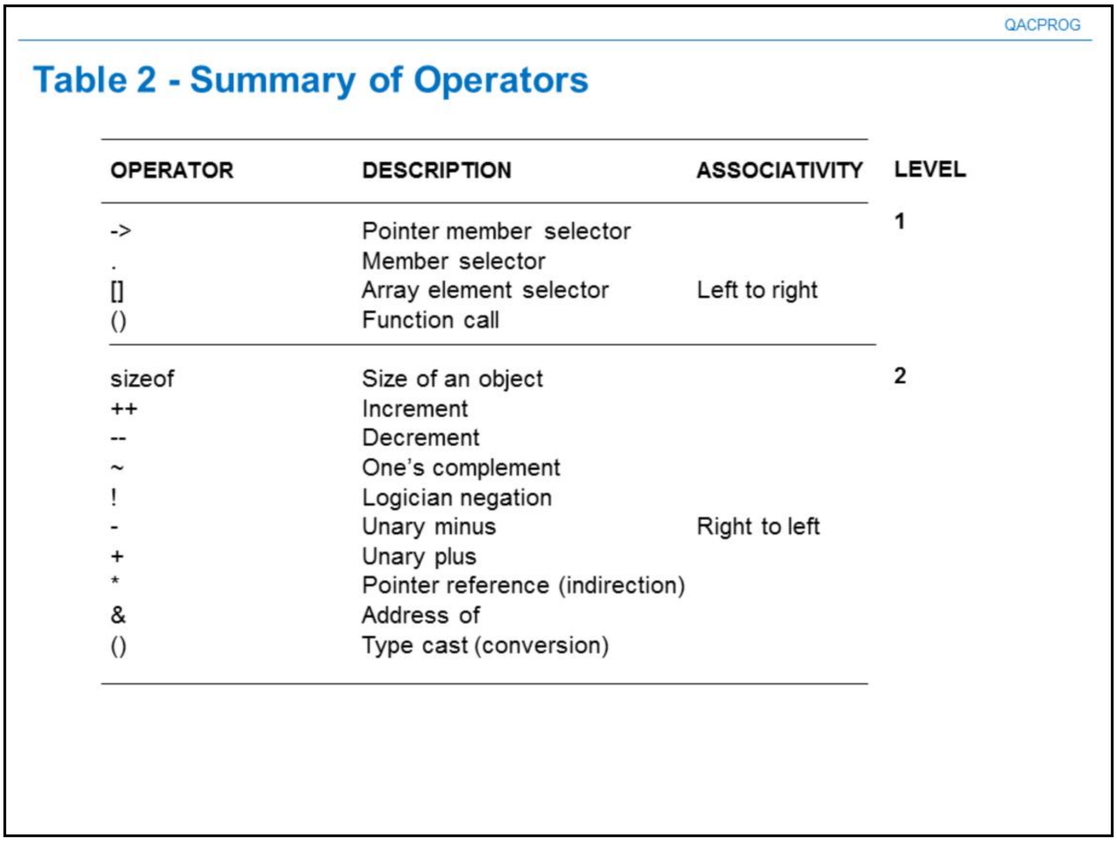
QACPROG

Table 1 - Reserved C Keywords

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto ***	**sizeof**	**volatile**
do	**if**	**static**	**while**

*** *goto* should be used with care**

Some implementations have extra, non-standard keywords such as pascal, asm, etc.



QACPROG

Table 2 - Summary of Operators

OPERATOR	DESCRIPTION	ASSOCIATIVITY	LEVEL
->	Pointer member selector		1
.	Member selector		
[]	Array element selector	Left to right	
()	Function call		
sizeof	Size of an object		2
++	Increment		
--	Decrement		
~	One's complement		
!	Logician negation		
-	Unary minus	Right to left	
+	Unary plus		
*	Pointer reference (indirection)		
&	Address of		
()	Type cast (conversion)		

The level figure in the table indicates the strength of the operator in its binding with the operand or operands. The operators within the same level band have identical precedence. An expression involving a mixture of operators with the same level need to be evaluated according to the association direction.

Parentheses can be used to override precedence. They can and should be used for clarity.

The four examples given below illustrate some common problems and techniques:

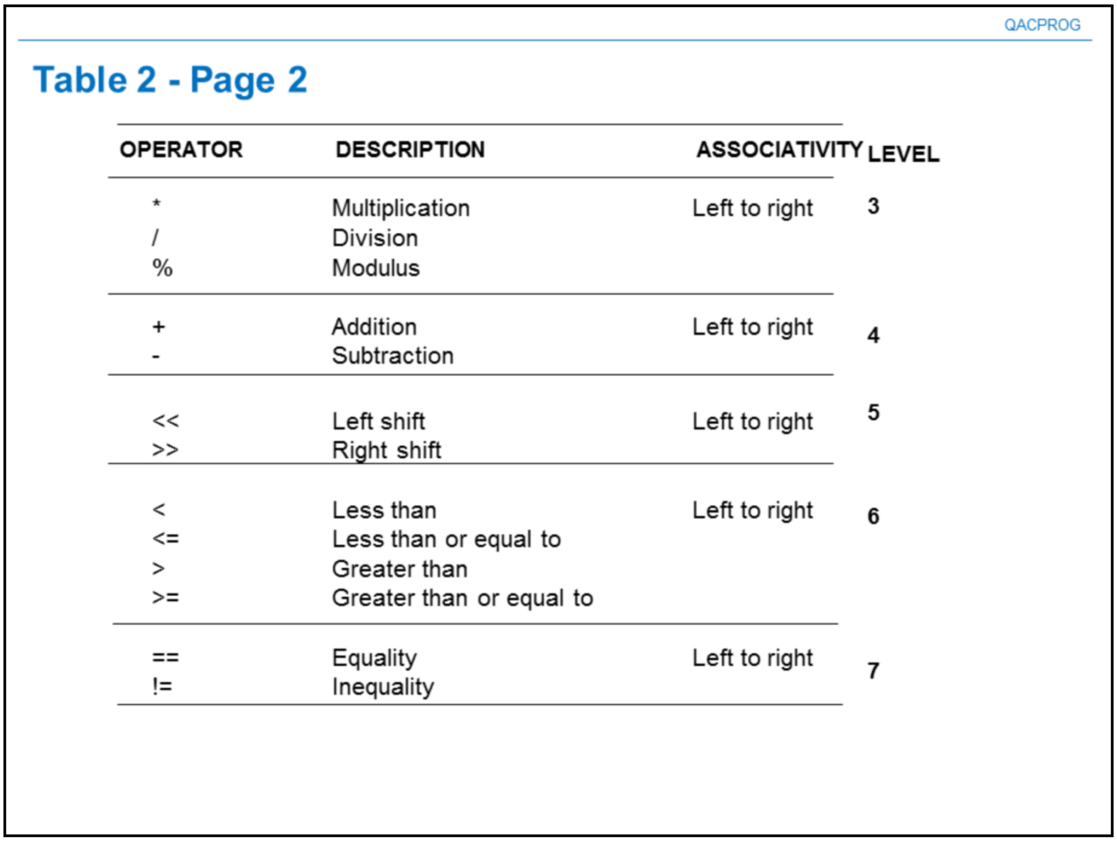
*ptr++ means increment the pointer, but after accessing the data pointed at. It is better to use *(ptr++).

-++index means pre-increment the index, then apply unary minus. Again, it is better to use -(++index).

arrstruc[2]->mem means get element 2 of arrstruc and access its member called mem.

!index-- presents an interesting (and unreadable) case. The decrement operation is performed first (right -> left), then the logical operator is performed. However, the decrement is postfix, so the current version of index is used to perform the logic negation. The index is decremented after the logical operation.

It is worthwhile developing and using a style which clarifies expressions using these operators.



QACPROG

Table 2 - Page 2

OPERATOR	DESCRIPTION	ASSOCIATIVITY	LEVEL
*	Multiplication	Left to right	3
/	Division		
%	Modulus		
+	Addition	Left to right	4
-	Subtraction		
<<	Left shift	Left to right	5
>>	Right shift		
<	Less than	Left to right	6
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
==	Equality	Left to right	7
!=	Inequality		

Most of these operators are conventional.

The bit-shifts can cause precedence problems because they are higher than the other binary bit manipulations and lower than the + and - operators. The two examples following illustrate these problems.

if(x << 2 && x & 2) looks bad enough anyway, but is actually if((x << 2 && x) & 2).   Clearly, parentheses should be put around the bit & operation, but it is still better to put them round both bit operations, i.e. if((x << 2)&&(x & 2)).

x = y * 8 + 4 is often replaced by x = y << 3 + 4 in order to speed things up. The precedence rules would force the addition to be performed first, so the expression should be
x = (y << 3) + 4. A simple rule is do not use shift operators to perform swift arithmetic, but that would be too prohibitive!

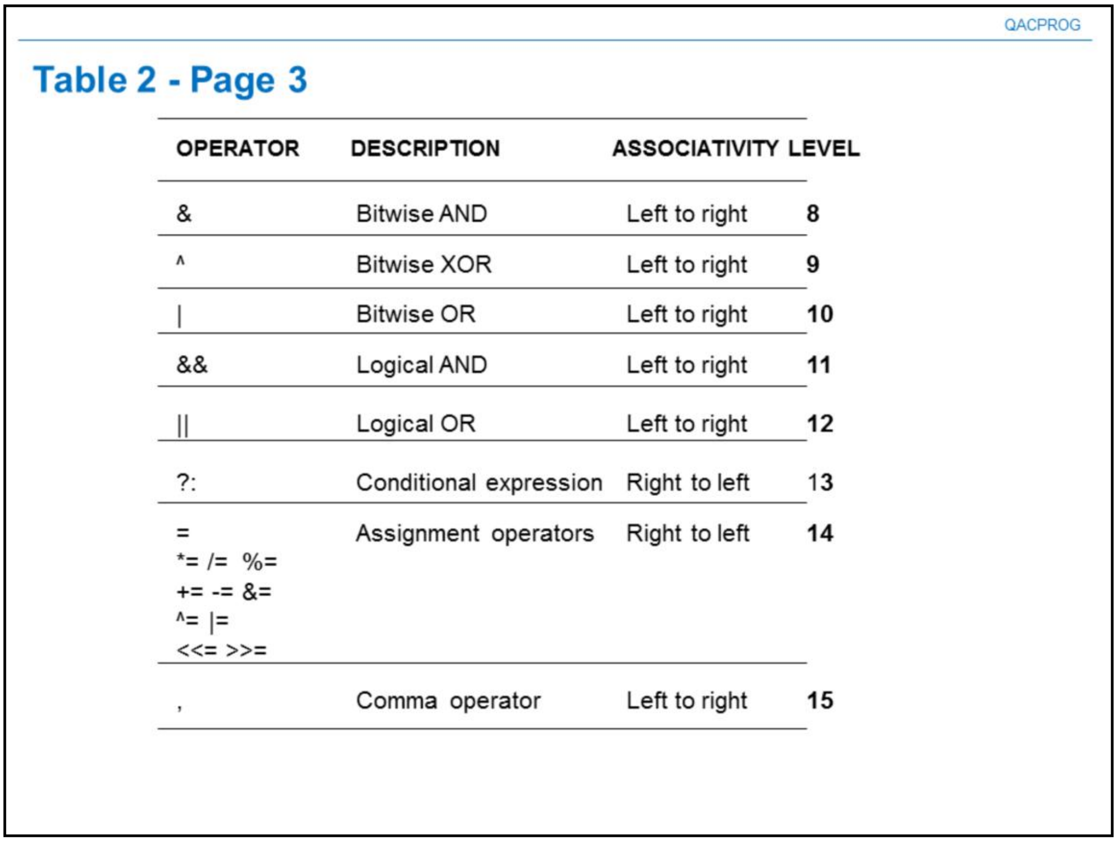

QACPROG

Table 2 - Page 3

OPERATOR	DESCRIPTION	ASSOCIATIVITY	LEVEL
&	Bitwise AND	Left to right	8
^	Bitwise XOR	Left to right	9
|	Bitwise OR	Left to right	10
&&	Logical AND	Left to right	11
||	Logical OR	Left to right	12
?:	Conditional expression	Right to left	13
= *= /= %= += -= &= ^= |= <<= >>=	Assignment operators	Right to left	14
,	Comma operator	Left to right	15

The low precedence operators include some strange cases! The conditional operator has its own problems of readability, and the comma operator is unheard of in other languages. Here are a few examples of common problems facing the users of these low-level operators:

if (bool1 & bool2) should presumably be if (bool1 && bool2).

VAT = base_rate + year < 1985 ? 2.5 : 3.5 will have the addition evaluated first! It's pretty gross without brackets anyway. Clearly, the rvalue should be base_rate +(year < 1985 ? 2.5 : 3.5).

The assignment operators have their own problems. The richest source of errors involves using assignment instead of equality checking, e.g. if(x = 5) instead of if(x == 5).

Because assignment is an operator, it is possible to write expressions like x = y + (z = 3) + 5. This is horrific style, even with parentheses, but possible. This is true for the assignment/updaters as well. It is even possible to write horrific expressions like x += y + (z *= 3) + 5!

The comma operator deserves special attention. Its function is described in the notes accompanying the description of the for loop in the Looping Constructs chapter. It can, however, be confused with the comma used to separate data declarations and parameter lists in functions.

The comma may also present a problem in function calls. The call funct1(a, b) is ALWAYS a call with two arguments, not a single argument with the comma expression, a,b. The prototype of funct1 should sort out any problems, e.g. consider the prototype int funct2(int). To call this with an argument involving a comma expression a,b would be achieved thus: funct2((a,b)). This is not encouraged.

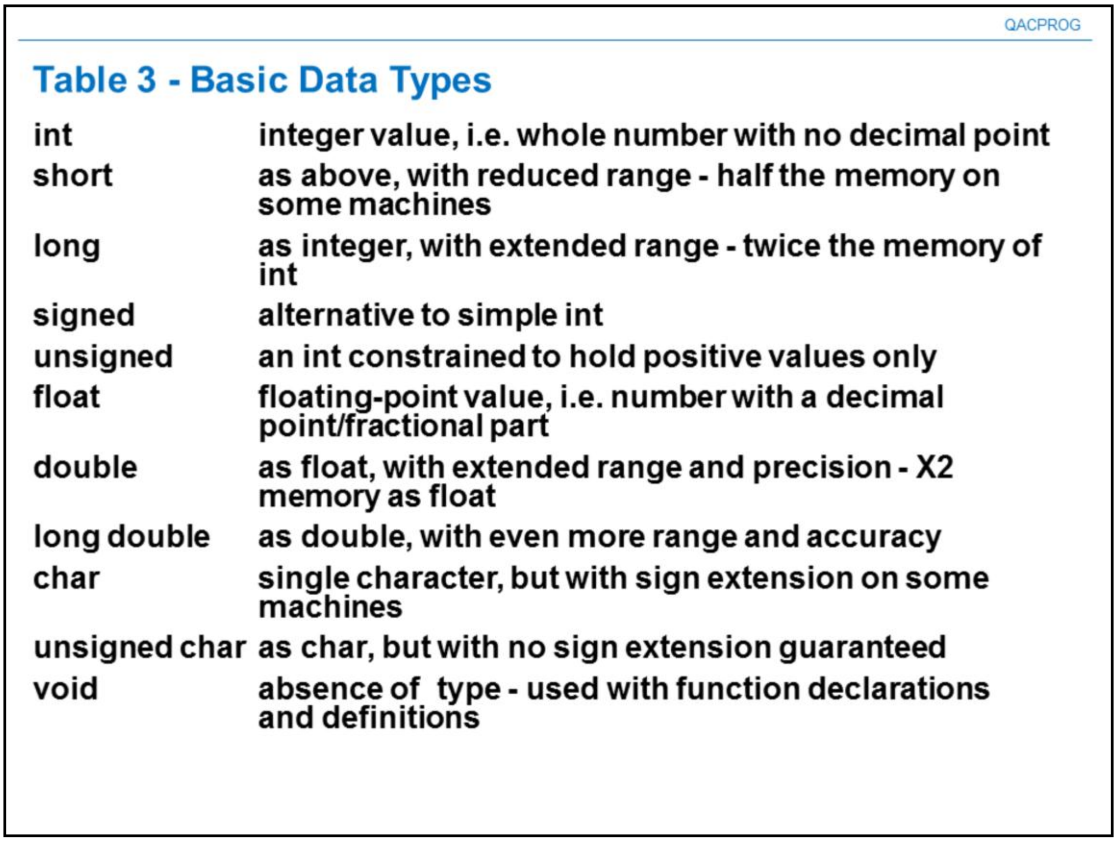

QACPROG

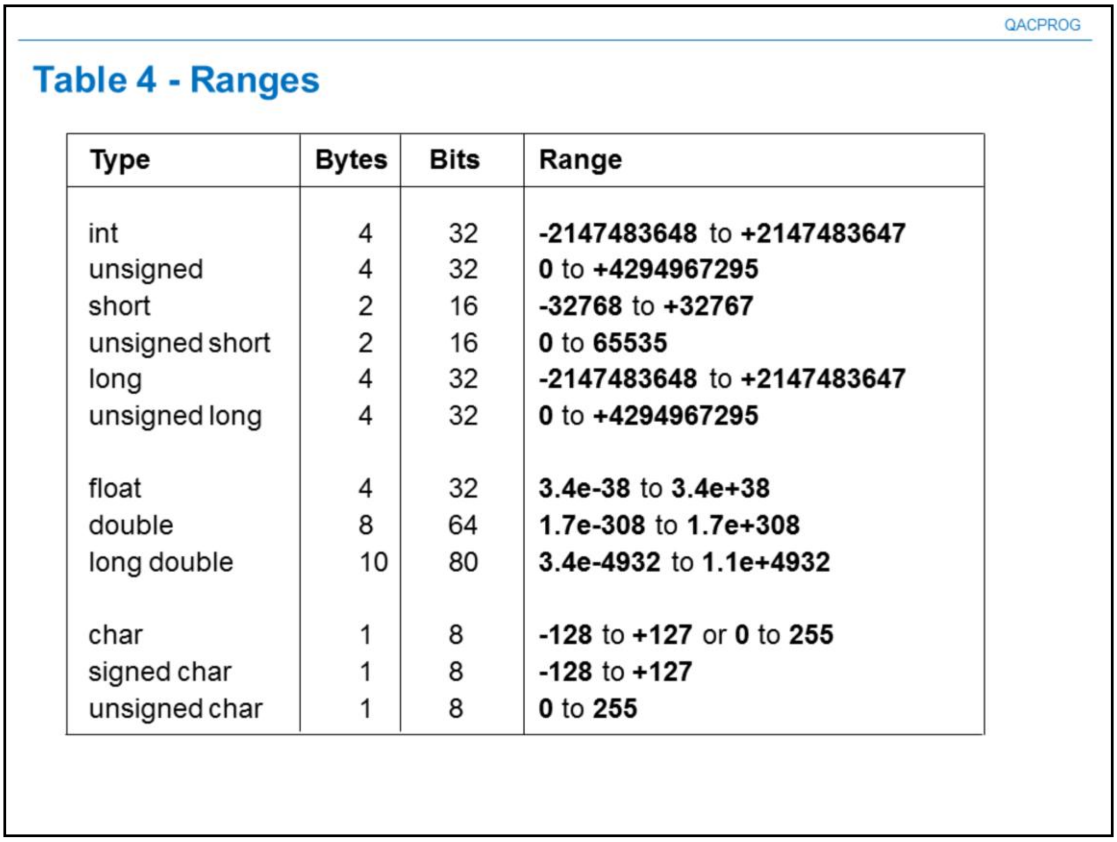
Table 3 - Basic Data Types

int	integer value, i.e. whole number with no decimal point
short	as above, with reduced range - half the memory on some machines
long	as integer, with extended range - twice the memory of int
signed	alternative to simple int
unsigned	an int constrained to hold positive values only
float	floating-point value, i.e. number with a decimal point/fractional part
double	as float, with extended range and precision - X2 memory as float
long double	as double, with even more range and accuracy
char	single character, but with sign extension on some machines
unsigned char	as char, but with no sign extension guaranteed
void	absence of type - used with function declarations and definitions

All built-in types, except void, can be used as building blocks for both types of aggregate, i.e. arrays and structure types.

All of the above, without exception, can be used as a basis for pointer declarations, i.e. pointers to int, pointers to unsigned long, pointers to void, etc. Pointers can be declared to any depth, e.g. pointers to pointers to char.

Operators are predefined for the built-in types. Some operations require conversion from one type to another if the operands are not the appropriate types (refer to the Expressions, Assignments and Operators chapter).



QACPROG

Table 4 - Ranges

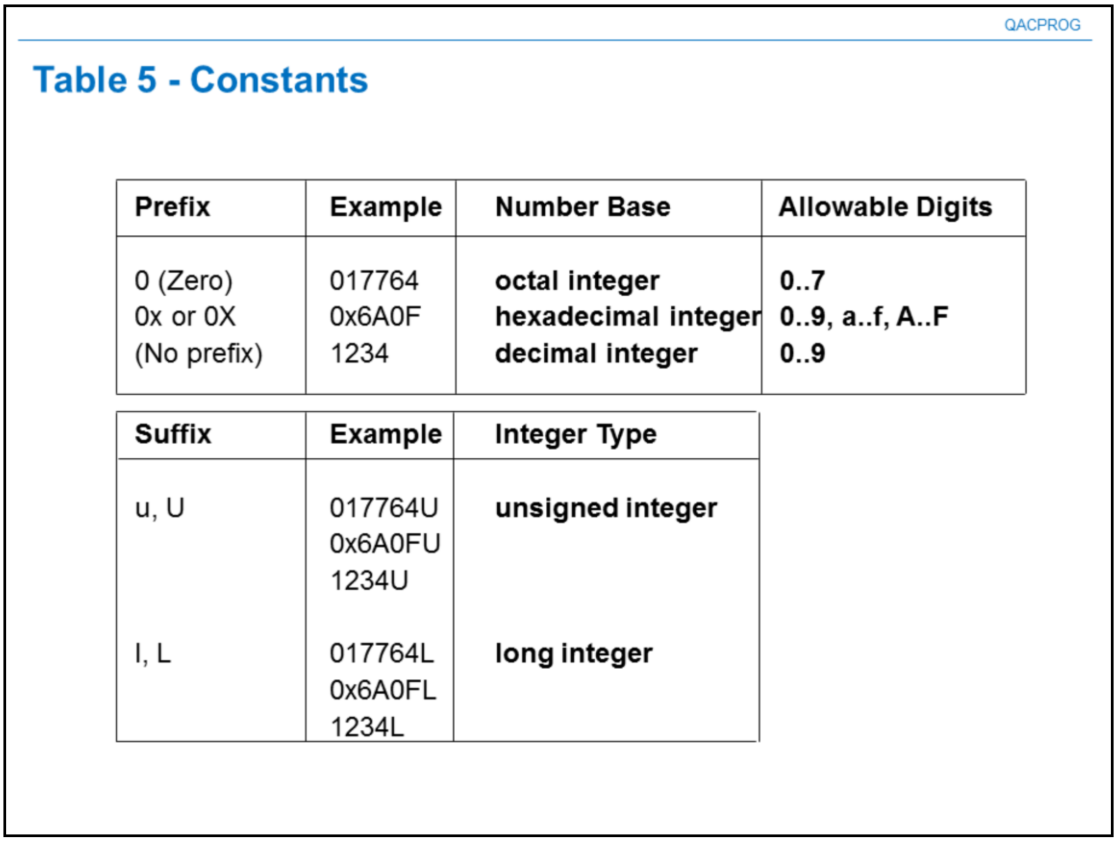
Type	Bytes	Bits	Range
int	4	32	**-2147483648** to **+2147483647**
unsigned	4	32	**0** to **+4294967295**
short	2	16	**-32768** to **+32767**
unsigned short	2	16	**0** to **65535**
long	4	32	**-2147483648** to **+2147483647**
unsigned long	4	32	**0** to **+4294967295**
float	4	32	**3.4e-38** to **3.4e+38**
double	8	64	**1.7e-308** to **1.7e+308**
long double	10	80	**3.4e-4932** to **1.1e+4932**
char	1	8	**-128** to **+127** or **0** to **255**
signed char	1	8	**-128** to **+127**
unsigned char	1	8	**0** to **255**

The size of the basic types is not defined in the language specification. You may rely on the fact that the char will be a byte (but see the note at the bottom of the page) and that int is the size of the machine word.

The alternative range for the char confirms that a char is represented in an implementation-specific way. It could be a signed or an unsigned char. The internal representation is only guaranteed if one uses the signed or unsigned qualifier.

NOTE:

Wide chars are supported in many modern implementations and are receiving world-wide acclaim. They will probably be included in the next standard. The support extends to strings of wide chars and library functions which manipulate them. They are represented as 2-byte versions of the char and support both signed and unsigned versions. The extended range will enable a representation of characters outside the conventional ‘European’ characters.



QACPROG

Table 5 - Constants

Prefix	Example	Number Base	Allowable Digits
0 (Zero)	017764	**octal integer**	**0..7**
0x or 0X	0x6A0F	**hexadecimal integer**	**0..9, a..f, A..F**
(No prefix)	1234	**decimal integer**	**0..9**

Suffix	Example	Integer Type
u, U	017764U 0x6A0FU 1234U	**unsigned integer**
l, L	017764L 0x6A0FL 1234L	**long integer**

There are four types of constants: integer, character, floating point and string.

By default, integer constants are decimal. However, by using a prefix, both octal and hexadecimal constants can be used. A suffix is used to specify unsigned or long constants.

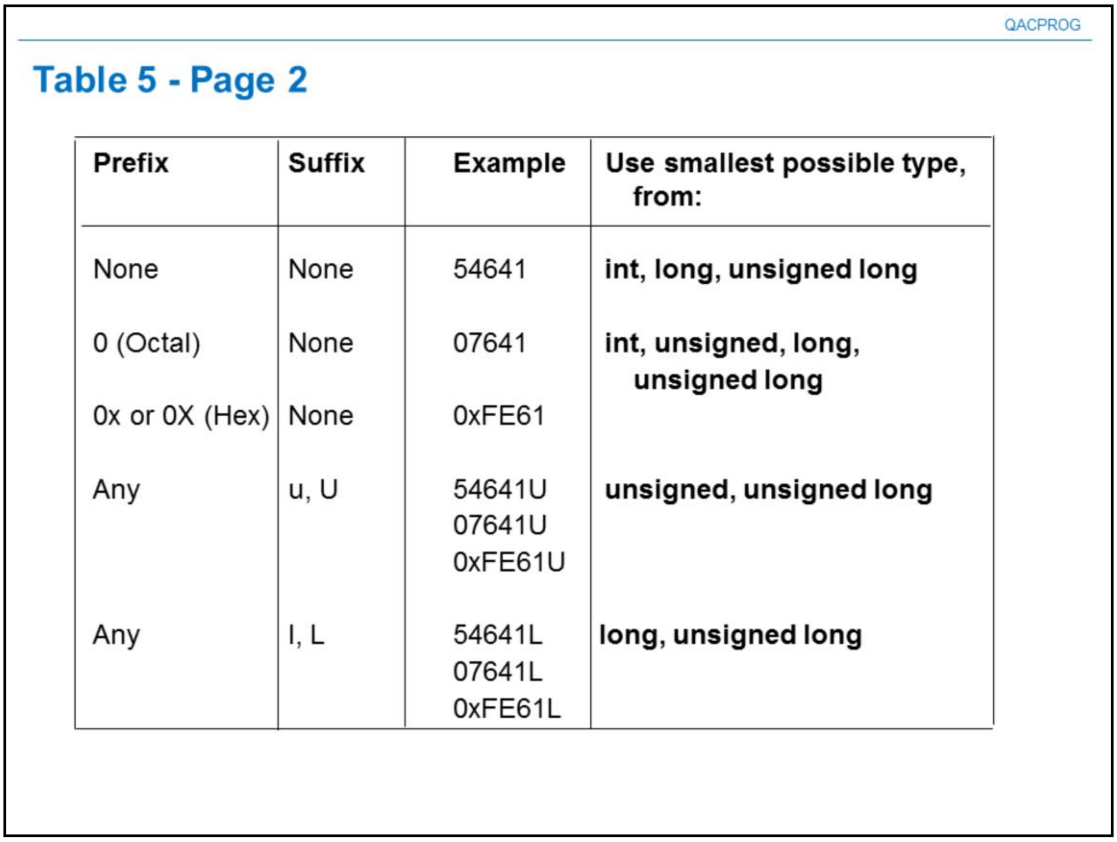
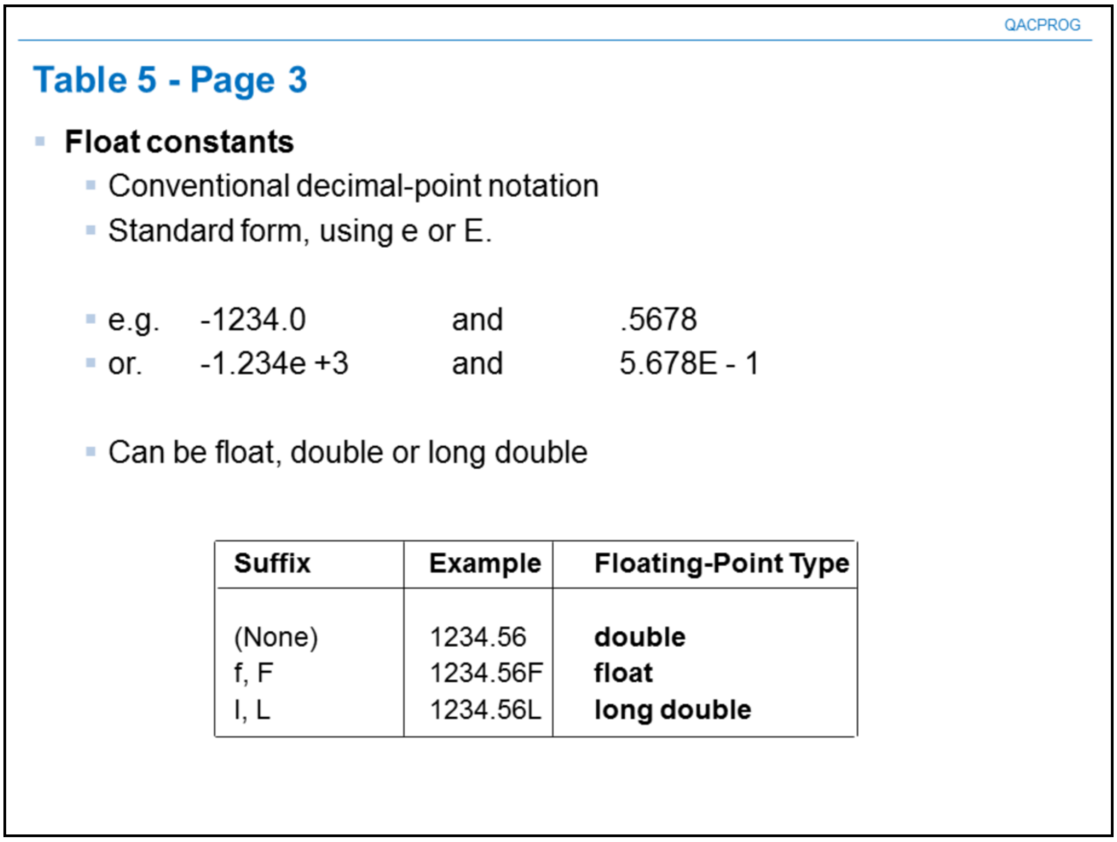

QACPROG

Table 5 - Page 2

Prefix	Suffix	Example	Use smallest possible type, from:
None	None	54641	**int, long, unsigned long**
0 (Octal)	None	07641	**int, unsigned, long, unsigned long**
0x or 0X (Hex)	None	0xFE61	
Any	u, U	54641U 07641U 0xFE61U	**unsigned, unsigned long**
Any	l, L	54641L 07641L 0xFE61L	**long, unsigned long**

The compiler will attempt to associate an integer constant with the smallest type that will accommodate it. The suffixes U and L (lower or UPPERcase) indicate a desire to ‘promote’ the default (an int) to an unsigned or long respectively.



QACPROG

Table 5 - Page 3

- **Float constants**
  - Conventional decimal-point notation
  - Standard form, using e or E.

  - e.g. -1234.0 and .5678
  - or. -1.234e +3 and 5.678E - 1

  - Can be float, double or long double

Suffix	Example	Floating-Point Type
(None)	1234.56	**double**
f, F	1234.56F	**float**
l, L	1234.56L	**long double**

By default, a floating-point constant is processed as a double. The suffices f,F and l,L are used to inform the compiler that you would like the constant treated as a float and long double, respectively.

Floating-point constants use both conventional 'decimal point' and scientific formats.

The decimal notation is:

<sign> mmmm . dddddd < suffix>

The standard form notation is:

<sign> mmmm . dddddd < E sign exp > <suffix>

<sign > is + or - and can be omitted.

<suffix> is f, l, F or L and can be omitted.

E can be E or e.

exp is an integer exponent in the range dictated by type of floating point (see Table 4).

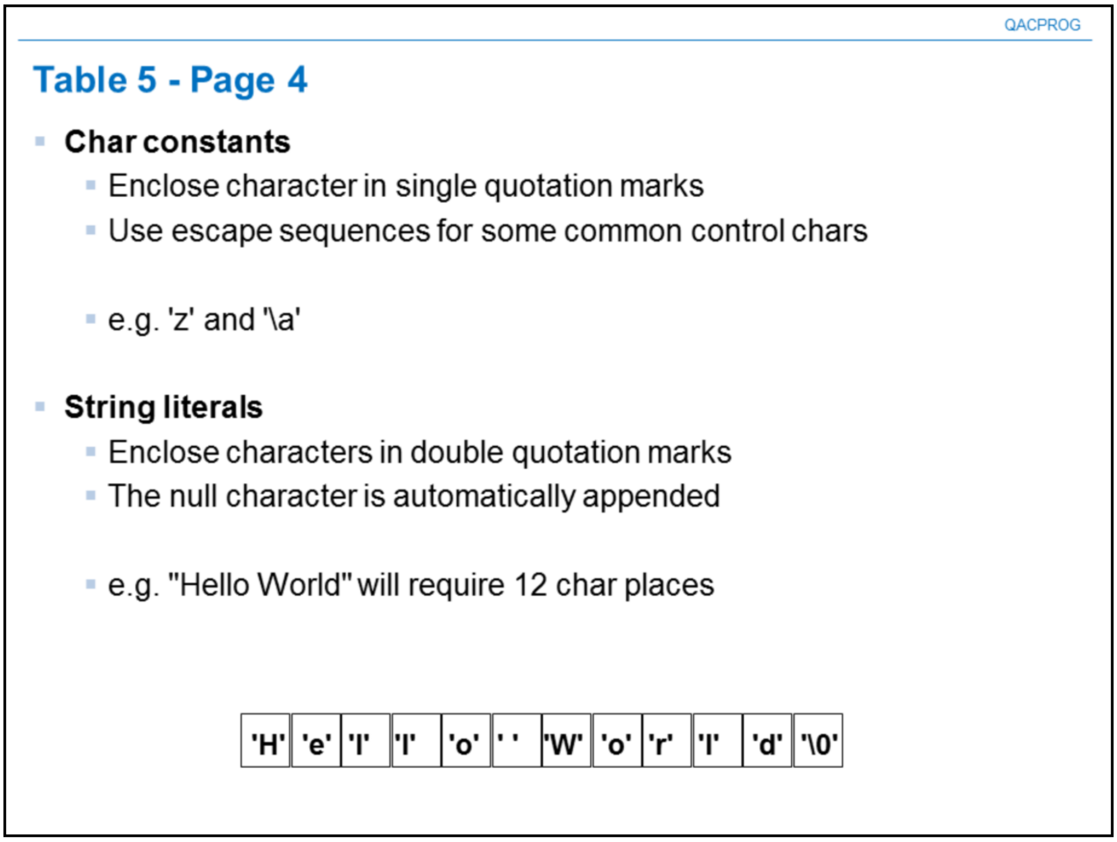

QACPROG

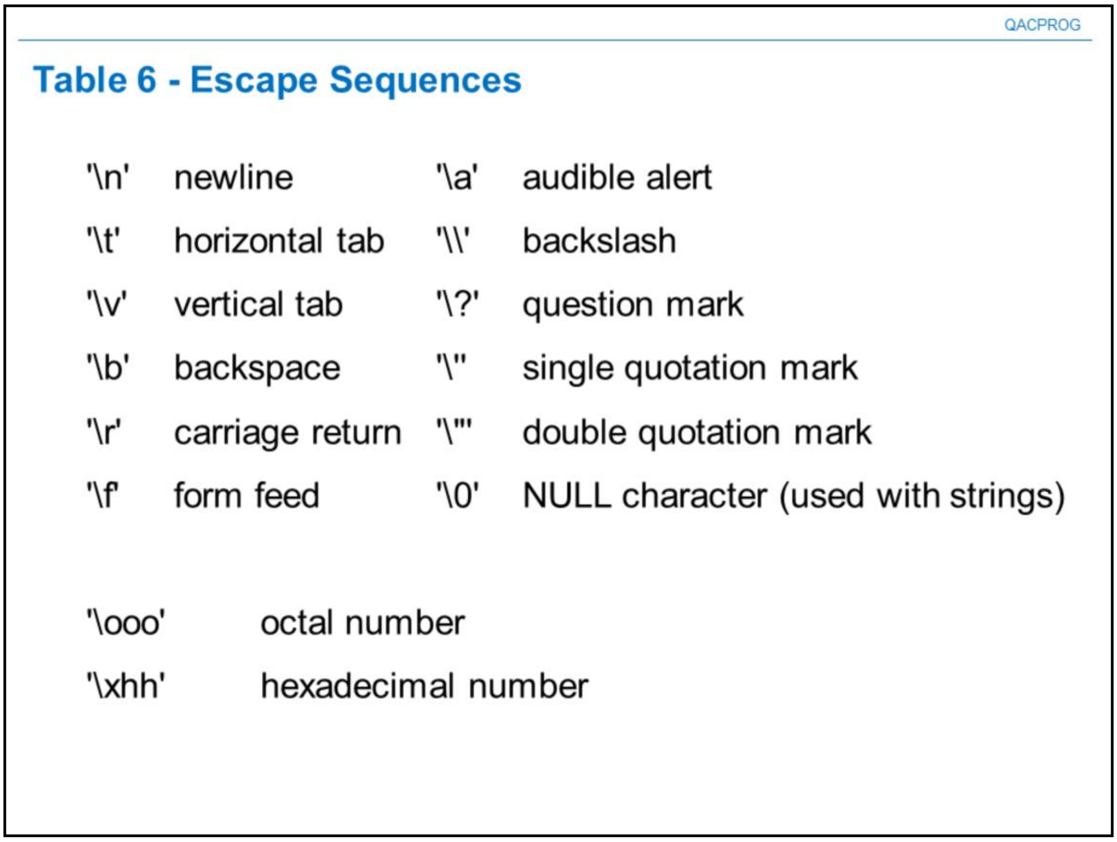
Table 5 - Page 4

- **Char constants**
  - Enclose character in single quotation marks
  - Use escape sequences for some common control chars
  - e.g. 'z' and '\a'
- **String literals**
  - Enclose characters in double quotation marks
  - The null character is automatically appended
  - e.g. "Hello World" will require 12 char places

'H'	'e'	'l'	'l'	'o'	' '	'W'	'o'	'r'	'l'	'd'	'\0'

Character constants are enclosed in single quotation marks. Certain unprintable characters are represented as ‘escape’ sequences. These are shown in Table 6.

A string literal is a sequence of characters enclosed in double quotation marks. A string has type ‘array of chars’ and may be initialised with such a character sequence. A null terminator ('\0') is automatically appended to the string literal.



QACPROG

Table 6 - Escape Sequences

'\n'	newline	'\a'	audible alert
'\t'	horizontal tab	'\\'	backslash
'\v'	vertical tab	'\?'	question mark
'\b'	backspace	'\''	single quotation mark
'\r'	carriage return	'\"'	double quotation mark
'\f'	form feed	'\0'	NULL character (used with strings)

'\ooo'	octal number
'\xhh'	hexadecimal number

The table presents a full list of escape sequences supported by ANSI C.

The characters '\ooo' and '\xhh' represent octal and hex patterns, as required. The values are usually those of unprintable characters which are not included in the above list.

'\ooo' is the backslash followed by 1,2 or 3 octal digits (0..7). The sequence is then interpreted as an internal representation of a character.

'\xhh' is the backslash followed by any number of hexadecimal digits (0..9, a..f, A..F). Again, the sequence is interpreted as an internal representation of a character.

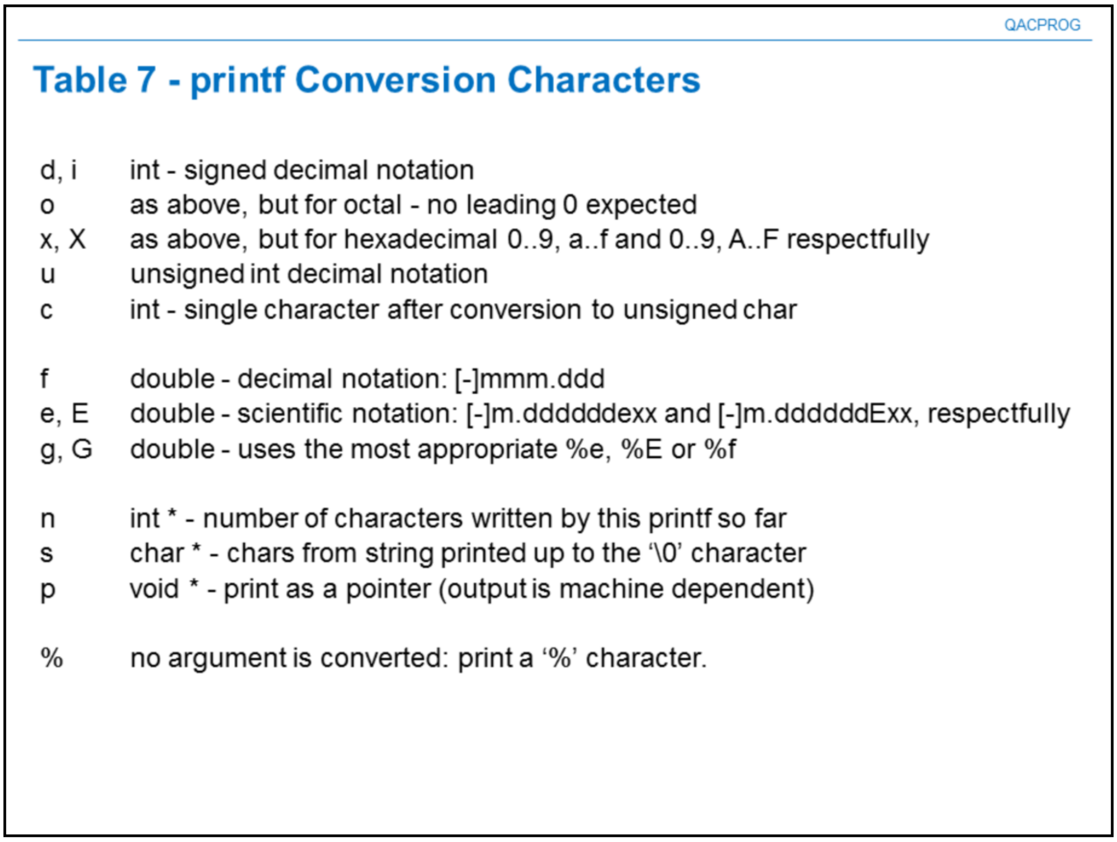

QACPROG

Table 7 - printf Conversion Characters

d, i	int - signed decimal notation
o	as above, but for octal - no leading 0 expected
x, X	as above, but for hexadecimal 0..9, a..f and 0..9, A..F respectfully
u	unsigned int decimal notation
c	int - single character after conversion to unsigned char
f	double - decimal notation: [-]mmm.ddd
e, E	double - scientific notation: [-]m.dddddexx and [-]m.dddddExx, respectfully
g, G	double - uses the most appropriate %e, %E or %f
n	int * - number of characters written by this printf so far
s	char * - chars from string printed up to the '\0' character
p	void * - print as a pointer (output is machine dependent)
%	no argument is converted: print a '%' character.

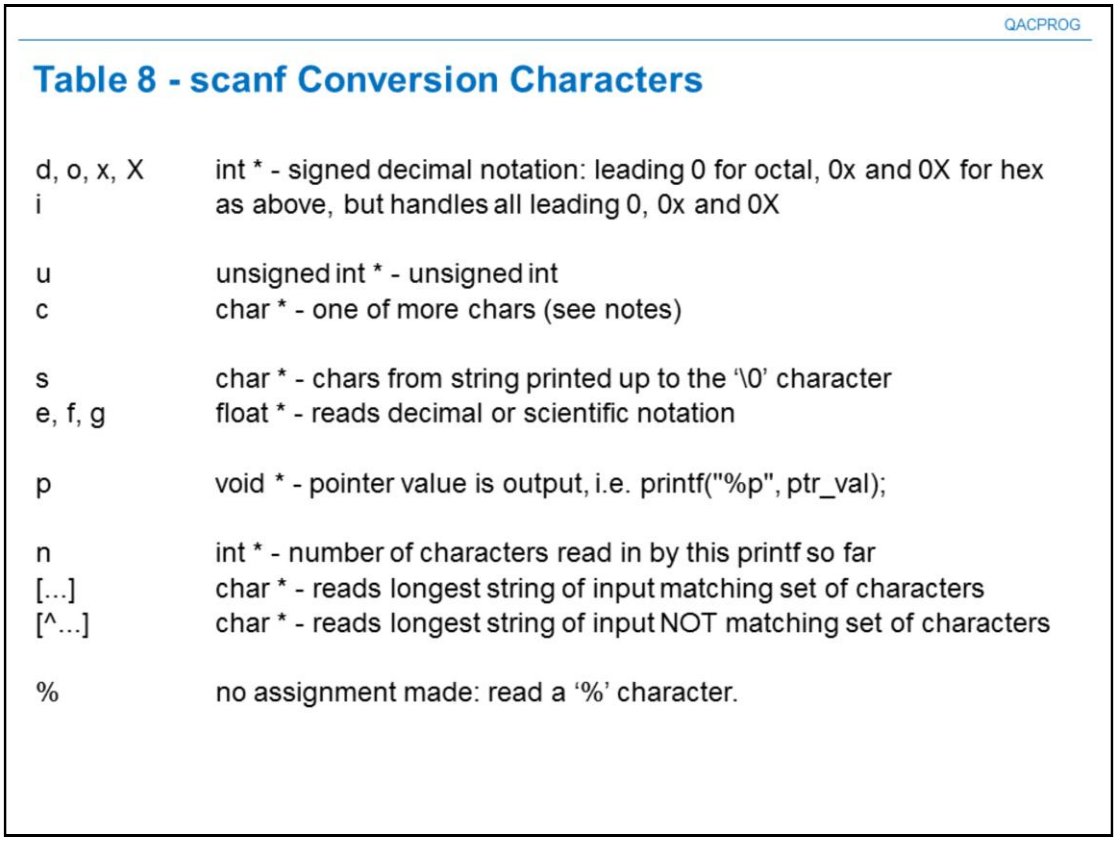
The conversion characters and their meanings as summarised in the table shown above are standard to the printf library function, including fprintf, sprintf, vfprintf, vprintf and vsprintf.

printf is a variadic function which takes a const char * as the only fixed first argument and returns an integer indicating the number of characters put onto the output stream.

If i, o, d, x, X or u is preceded by l, it specifies a long int.

If i, o, d, x, X or u is preceded by h, it specifies a short int.

If e, E, f, g, or G is preceded by L, it specifies a long double.



QACPROG

Table 8 - scanf Conversion Characters

d, o, x, X	int * - signed decimal notation: leading 0 for octal, 0x and 0X for hex
i	as above, but handles all leading 0, 0x and 0X
u	unsigned int * - unsigned int
c	char * - one of more chars (see notes)
s	char * - chars from string printed up to the '\0' character
e, f, g	float * - reads decimal or scientific notation
p	void * - pointer value is output, i.e. printf("%p", ptr_val);
n	int * - number of characters read in by this printf so far
[...]	char * - reads longest string of input matching set of characters
[^...]	char * - reads longest string of input NOT matching set of characters
%	no assignment made: read a '%' character.

scanf is a variadic function which takes a const char * as the only fixed first argument and returns an integer that indicates the number of format specifiers correctly matched.

The format string may contain blanks and tabs (ignored by all except %c and %s); ordinary characters, which are expected to be on the input stream exactly as stated in the string (except %); and the conversion specifiers beginning with %.

Specifiers d, i, n, o, x and X my be preceded by:

l (letter ell) if a long int is to be read

h if a short int is to be read

e, f and g may be preceded by:

l (letter ell) if a double (rather than a float) is to be read

L if a long double to be read

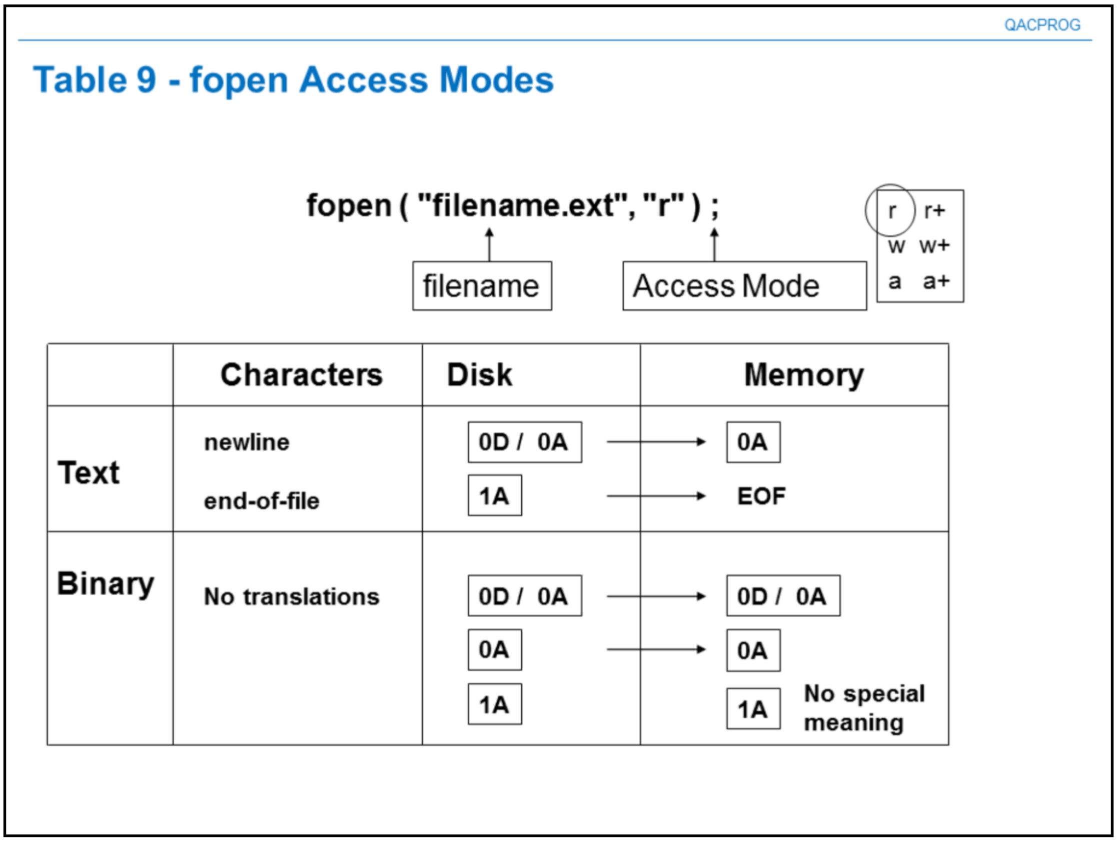
The c specifier can be used to place characters into the buffer, up to a field width. If no width is specified, one character is read. The blank and tabs are NOT skipped and no '\0' character is appended, e.g.:

```
char buffer[10]; char single_ch;

scanf("%10c", &buffer[0] );   /*  read 10 chars from input stream into buffer  */

scanf("%c", &single_ch);          / * read 1 char only and place it in single_ch */
```

The s specifier must be used with a char * large enough to accommodate the input. Unlike c, a '\0' is appended. s can be used with a numeric field-width specifier.



QACPROG

Table 9 - fopen Access Modes

```
fopen ( "filename.ext", "r" ) ;
           ↑                ↑
       filename        Access Mode
```

[✓ r]	r+
w	w+
a	a+

	Characters	Disk	Memory
Text	newline	0D / 0A →	0A
	end-of-file	1A →	EOF
Binary	No translations	0D / 0A →	0D / 0A
		0A →	0A
		1A	1A No special meaning

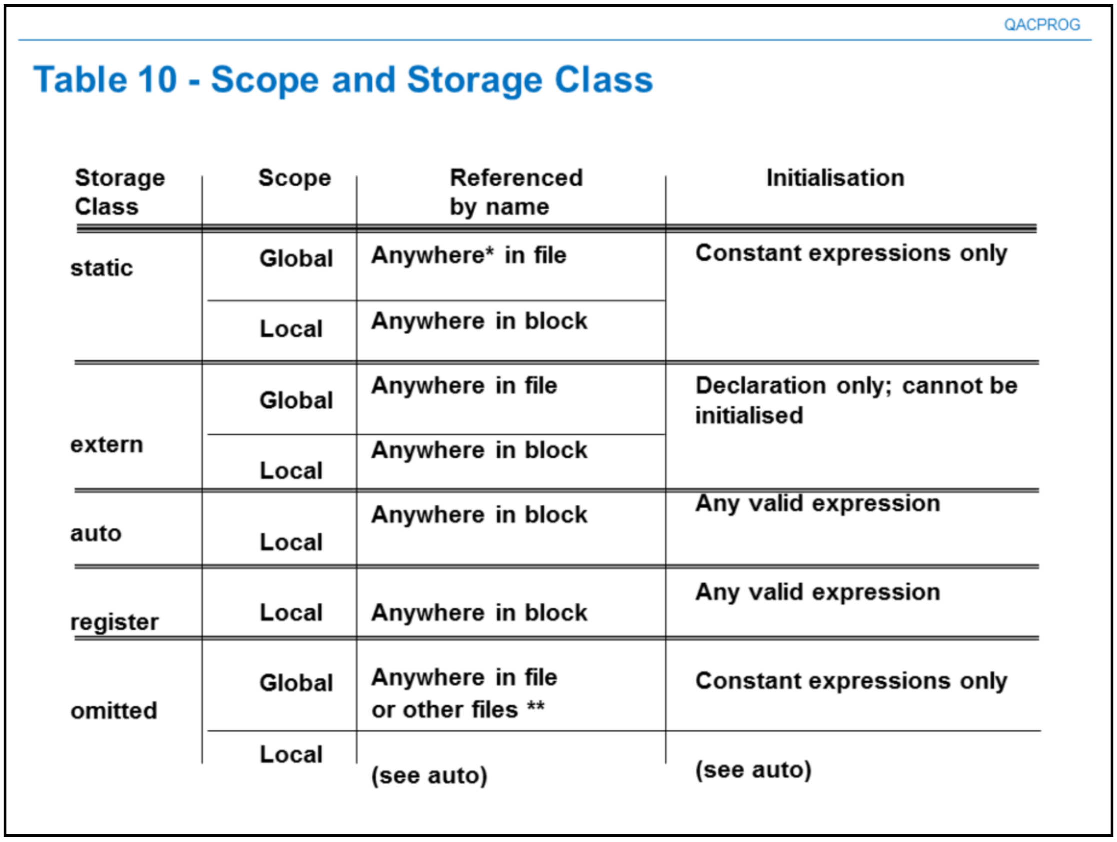
fopen is a standard ANSI function to open a file for buffered input and output. The first parameter is a string which is taken as the filename. The second, also a string, represents the ‘opening’ mode. A synopsis of modes is:

r	Open file for reading; file must exist.
w	Open file for writing; overwrite existing file or create a new file if necessary.
a	Open file for appending; same as w, if file does not exist.
r+	Open for reading and writing; file must exist.
w+	Open for reading and writing, then same as w.
a+	Open for reading and appending, then same as a.

The file is opened in ‘text’ mode by default. In text mode, the character sequence <CR><LF> read from the file is translated to <NL> (i.e. 0x0A) when written to the program and vice versa. Also, the file end-of-file character 0x1A is translated to EOF, which is locally represented, usually by (-1).

The other mode recognised by fopen is binary mode, where no translations are performed. The fopen function will process the file in binary mode if the mode string includes the character b.

The mode could include either t or b in its string, e.g. rt or wb+. Input and output is then performed in 'text' or 'binary;' mode respectively.



QACPROG

Table 10 - Scope and Storage Class

Storage Class	Scope	Referenced by name	Initialisation
static	Global	Anywhere* in file	Constant expressions only
	Local	Anywhere in block	
extern	Global	Anywhere in file	Declaration only; cannot be initialised
	Local	Anywhere in block	
auto	Local	Anywhere in block	Any valid expression
register	Local	Anywhere in block	Any valid expression
omitted	Global	Anywhere in file or other files **	Constant expressions only
	Local	(see auto)	(see auto)

* Anywhere refers to definition ‘downwards’.

initialisation

The methods used for initialisation depend on the category of type. There are three categories: built-in scalar, arrays and structures. The last two take a very restricted set of initial values.

static

Data is initialised only once at the start of program execution; the default value for non-const being zero. Values of non-globals are still retained through function calls.

extern

The data must be defined exactly once, i.e. without the extern specifier.

auto

The data is initialised each time the function or block is entered. There is no default value.

register

Like the auto, the data is initialised each time the function or block is entered, and there is no default value. The use of a register is not guaranteed; it may be treated as an auto. There are some language restrictions; only built-in scalars can be qualified as register and the address of a register item cannot be taken.

omitted

** In global scope, the data item is potentially available across the application. Any module can gain access to the item if it is declared with extern. Again, the data must defined exactly once, and like the static global, data is initialised only once at the start of program execution. The default value for non-const is zero.

Local data is treated as auto.

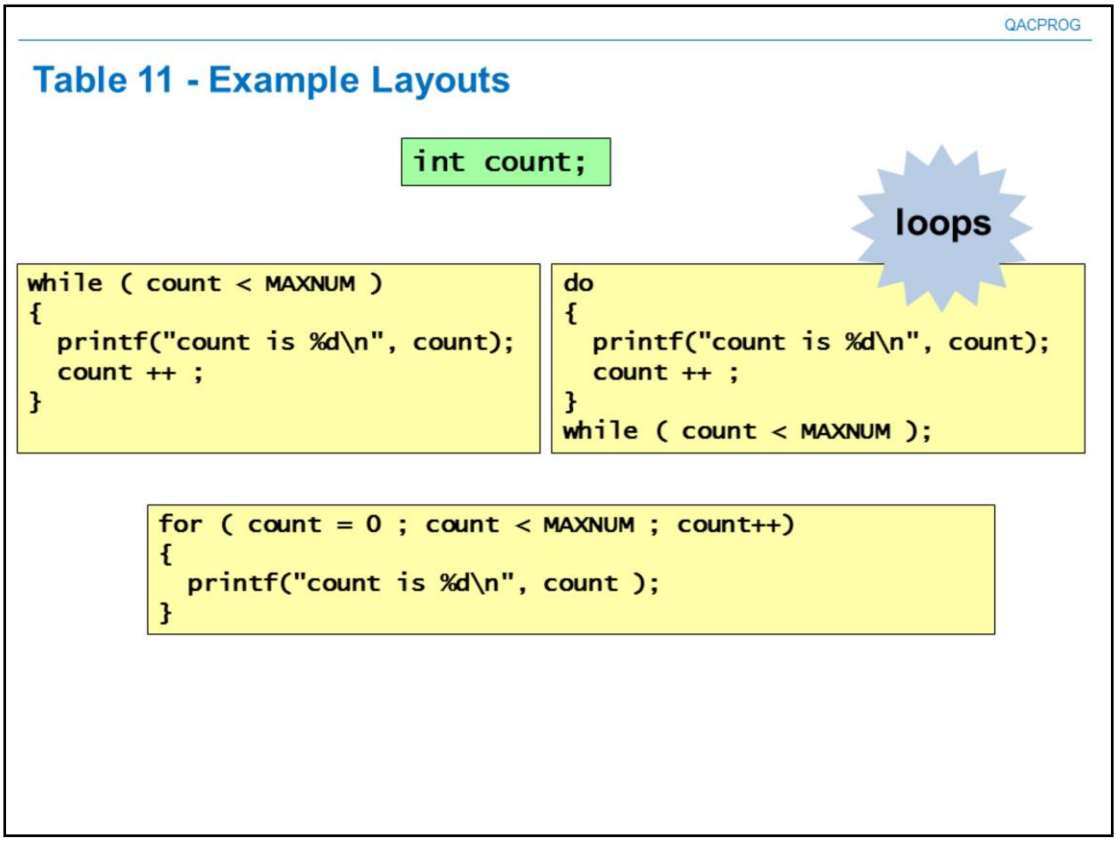

QACPROG

Table 11 - Example Layouts

```
int count;
```

```
while ( count < MAXNUM )
{
  printf("count is %d\n", count);
  count ++ ;
}
```

```
do
{
  printf("count is %d\n", count);
  count ++ ;
}
while ( count < MAXNUM );
```

```
for ( count = 0 ; count < MAXNUM ; count++)
{
  printf("count is %d\n", count );
}
```

COMMENTS:

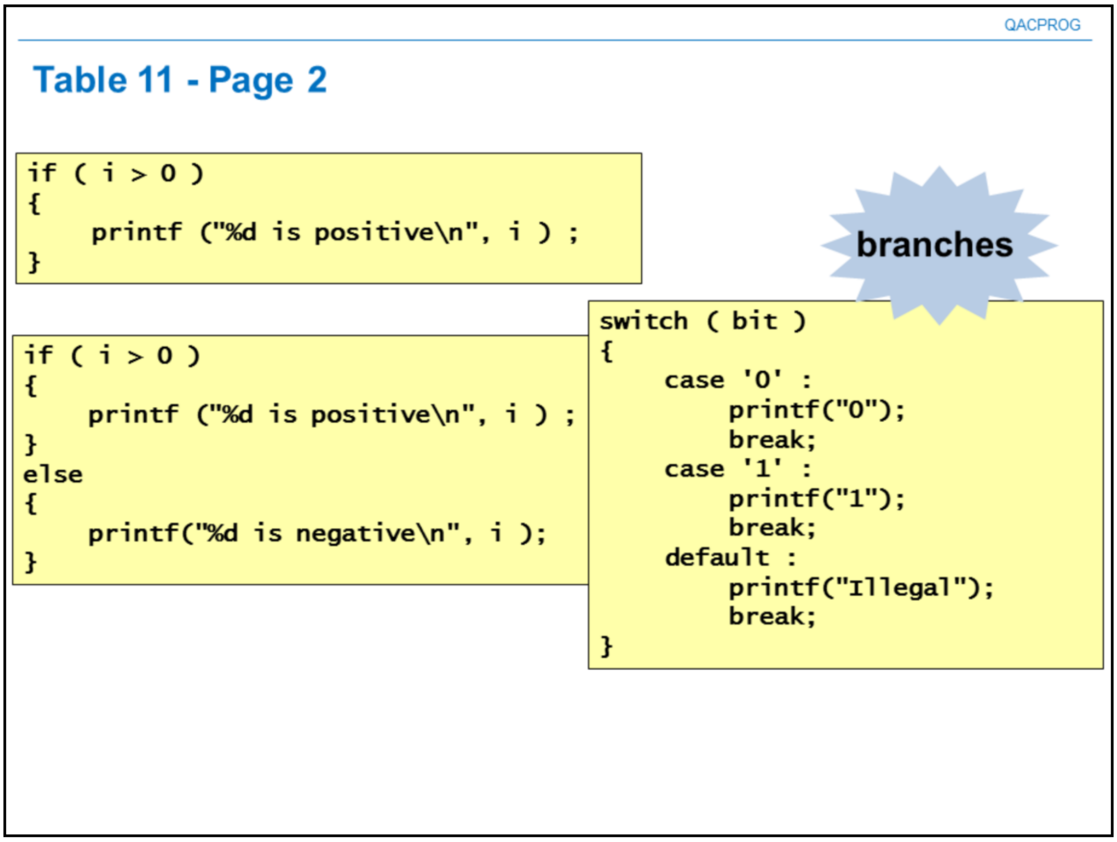
If data is defined, it must be declared at the top of the compound statement. The scope will be that of the compound statement and nested compound statements within.

NOTES on style:

The braces are not mandatory for single statements. You are encouraged to use them always.

We have adopted the 'PASCAL' style, i.e. the BEGIN/END braces are on lines of their own. Again, this is not mandatory. Indeed, many styles exist where the opening brace is placed after the test's final ')' , thus:

```
while(count < MAXVAL){
        etc ..          /* This is the style adopted by
        many American writers  */
}
```



QACPROG

Table 11 - Page 2

```
if ( i > 0 )
{
     printf ("%d is positive\n", i ) ;
}
```

```
if ( i > 0 )
{
     printf ("%d is positive\n", i ) ;
}
else
{
     printf("%d is negative\n", i );
}
```

```
switch ( bit )
{
     case '0' :
          printf("0");
          break;
     case '1' :
          printf("1");
          break;
     default :
          printf("Illegal");
          break;
}
```

COMMENTS:

The breaks in the case statements need not be there. However, the absence of a break will result in control dropping through into the next sequence of statements after a subsequent case line. Break or the closing { is the only cause of exiting the switch.

The last case, which is often the default, does not require a break.

In the switch statement, there is no requirement to make the statements into a single compound statement.

NOTES on style: /* Same as loops */

Again, the braces are not mandatory for single statements and we have adopted the ‘PASCAL’ style, i.e. the BEGIN/END braces are on lines on their own. The ‘American’ style if/else would appear thus:

```
if(i > 0){
etc.
}
else{
etc.
}
```

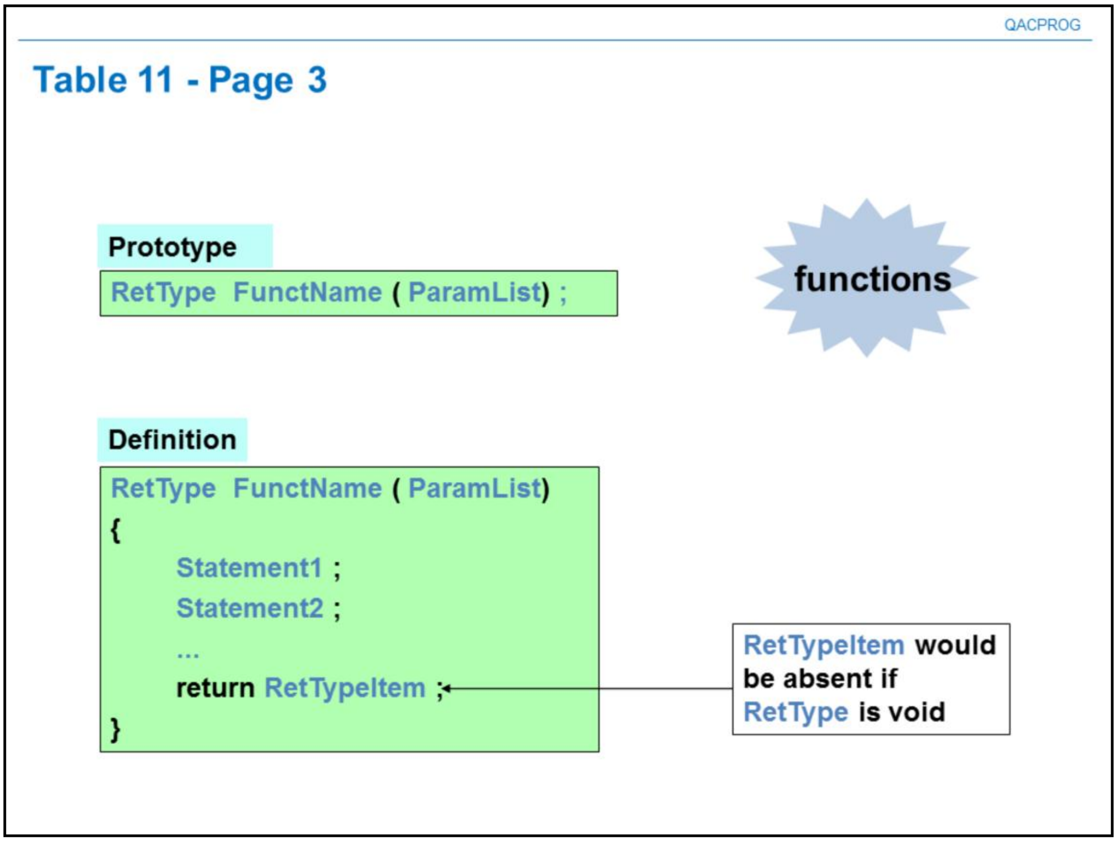


Table 11 - Page 3

Prototype

```
RetType  FunctName ( ParamList) ;
```

Definition

```
RetType  FunctName ( ParamList)
{
        Statement1 ;
        Statement2 ;
        ...
        return RetTypeItem ;
}
```

COMMENTS:

ParamList is a comma-separated sequence of <type> <parameter name> pairs, e.g.:

(int index, char idcode)

<type> may be built in or defined by the programmer. It could also be a pointer. Absence of the ParamList implies no input parameters are expected. The last, or only pair in the list could be ... , i.e. three full stops with no comma following. This tells the compiler to switch off parameter checking, which indicates that a variable number of items is expected at this point.

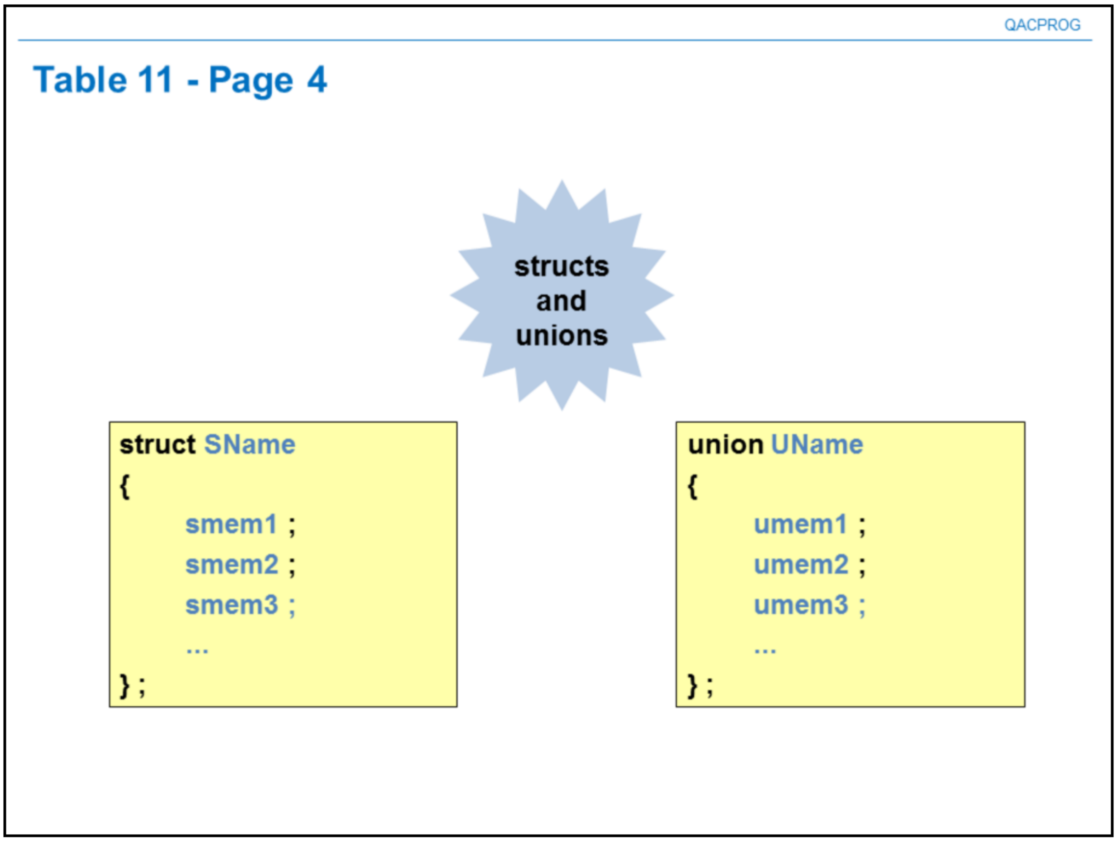
A function with a single-line body still requires a compound statement.

For functions returning void, the return statement may be omitted; the final } will signal a return of control.

NOTES on style:

Some writers prefer to have the return type on a line of its own, i.e.:

```
RetType
FunctName(ParamList)
{
        etc.          /* Particularly useful for long-line function headings */
```



```
struct SName
{
        smem1 ;
        smem2 ;
        smem3 ;
        ...
} ;
```

```
union UName
{
        umem1 ;
        umem2 ;
        umem3 ;
        ...
} ;
```

COMMENTS:

The struct/union name is not mandatory if data is being defined at the same time as the type. This is poor style, but there are occasions where it may be acceptable.

The smem and umem definitions share the same syntax as data definitions, i.e.

```
typename dataname;
```

Initialising is achieved by placing a comma-separated list of constant values enclosed in braces, e.g.:

```
struct SName sdata
        = {val1, val2,val3, ...};      /* valN must be constants */
```

unions can only be initialised with the value for the first member.

NOTES on style:

Writers who favour the 'American' style adopt the same strategy for these types, i.e.:

```
struct SName{
        etc.
}
```