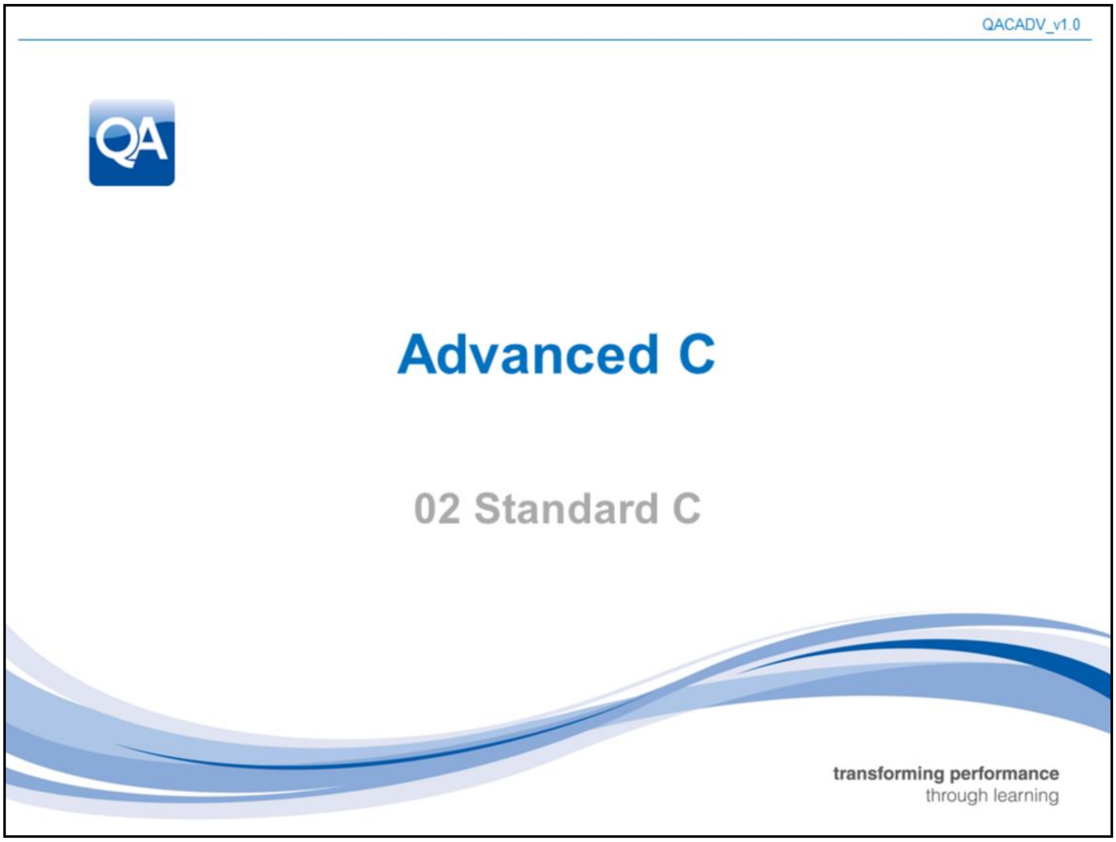

QACADV_v1.0
Advanced C
02 Standard C
transforming performance
through learning

QACADV_v1.0

Standard C

- **Objectives**
  - Study differences between K&R and Standard C
  - Learn of changes in Standard C
- **Contents**
  - K&R and STDC Function differences
  - Keywords: inline, const and volatile
  - Enumerated and Boolean types
  - Changes to string literals
  - Floats, doubles and long doubles
  - Finding limits
- **Summary**

X3J11
X3.159-1989
JTC1/SC22/WG14
ISO 9899:1999

The objective of this chapter is to discuss the changes made to the language by the ANSI and ISO committees whose goals were to produce "an unambiguous and machine-independent definition of the language C". C had a life before the standardisation which stretched out over nearly two decades.

In the early days C was affectionately known as "K & R" C, a reference to the language's creators. This chapter covers the major discrepancies between K & R and Standard C; the chapter covers the changes to the language implementation and the facilities which have been added. Further changes introduced with the C99 standard are mentioned here and throughout the course.

QACADV_v1.0

Standards

- **C is not a static language!**

	Known as:
Kernigham & Ritchie Edition 1	**K&R1**
Kernigham & Ritchie Edition 2	
ANSI X3.159-1989	**C89**
ISO/IEC 9899:1990	**C90**
TC (Technical Corrigenda) 1 & 2, Amendment AMD1	**C95**
ISO/IEC 9899:1999	**C99**

Similar, sometimes called **ANSI C** or **STDC**

Also (incorrectly) known as C9X

C99 defines __STDC_VERSION__ 199901

- **Features are backward compatible**
  - "*Existing code is important, existing implementations are not*"

The various standards committees and working parties are academic to most programmers, we just want to get on with it!

The C programming language has been remarkably stable, but has seen improvements over the years. The standards committees have tried to avoid "change for change's sake", using the following guiding principles:

Existing code is important, existing implementations are not

By "existing implementations" they mean compilers

C code can be portable

C code can be non-portable

Avoid "quiet changes"

Existing code should always do the same thing

A standard is a treaty between implementer and programmer

Limits specified by the standard are only minimums

Keep the spirit of C

In addition, in 1994, the following were added:

Support international programming

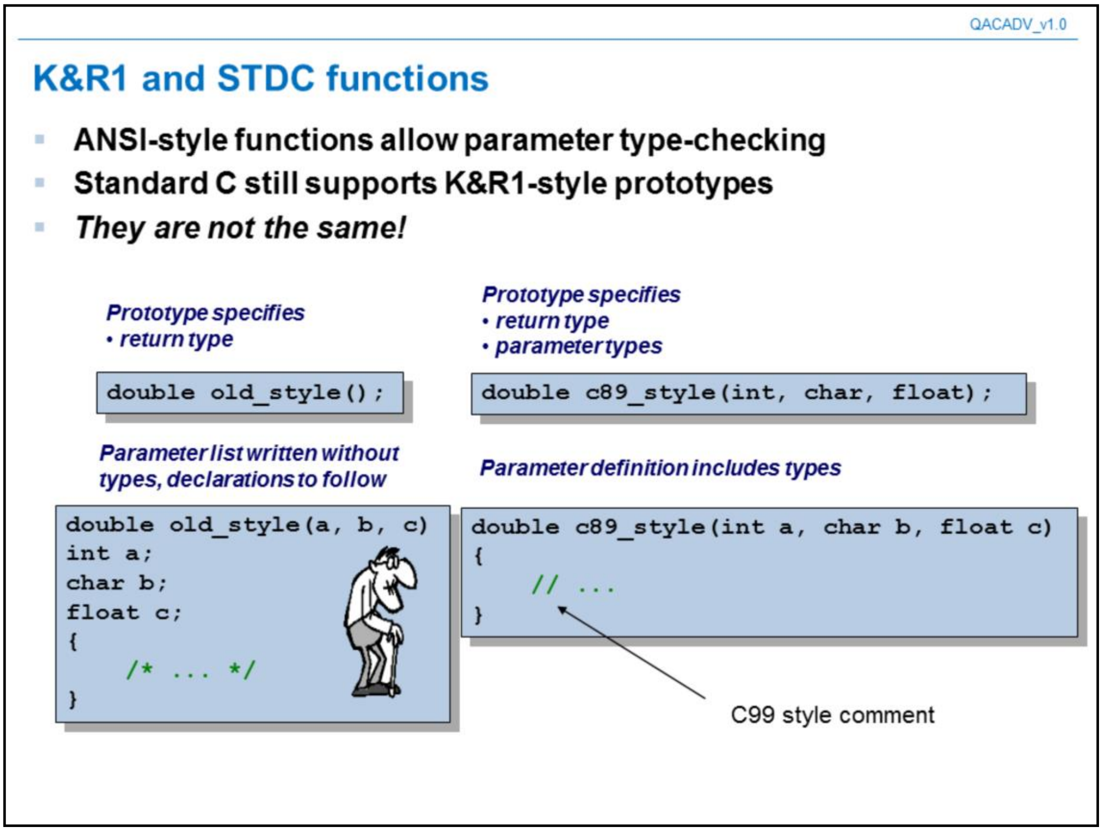
Codify existing practice to address evident deficiencies

Minimize incompatibilities with C90

Minimize incompatibilities with C++

Although there is no intention to compete

Maintain conceptual simplicity



The K&R1 style function declaration provides no means of argument / parameter checking. The parameter list is left empty - `void` did not officially exist in the language.

In the definition, the parameters are named and ordered within the parentheses in the function heading. They are defined immediately before the opening `{` of the function body.

The existence of more than two adjacent parameters of the same type are treated differently in each style. In the old style, `int x, y` is acceptable, as in conventional definitions. In the new style, it has to be `int x, int y`.

The changes to the function definition were mainly cosmetic; they have a different look and feel. Unfortunately C89/C90/ANSI C supported the old style for backwards compatibility. They are not the same and the old style should not be used.

Unfortunately the K&R1 style can be used by mistake! For example:

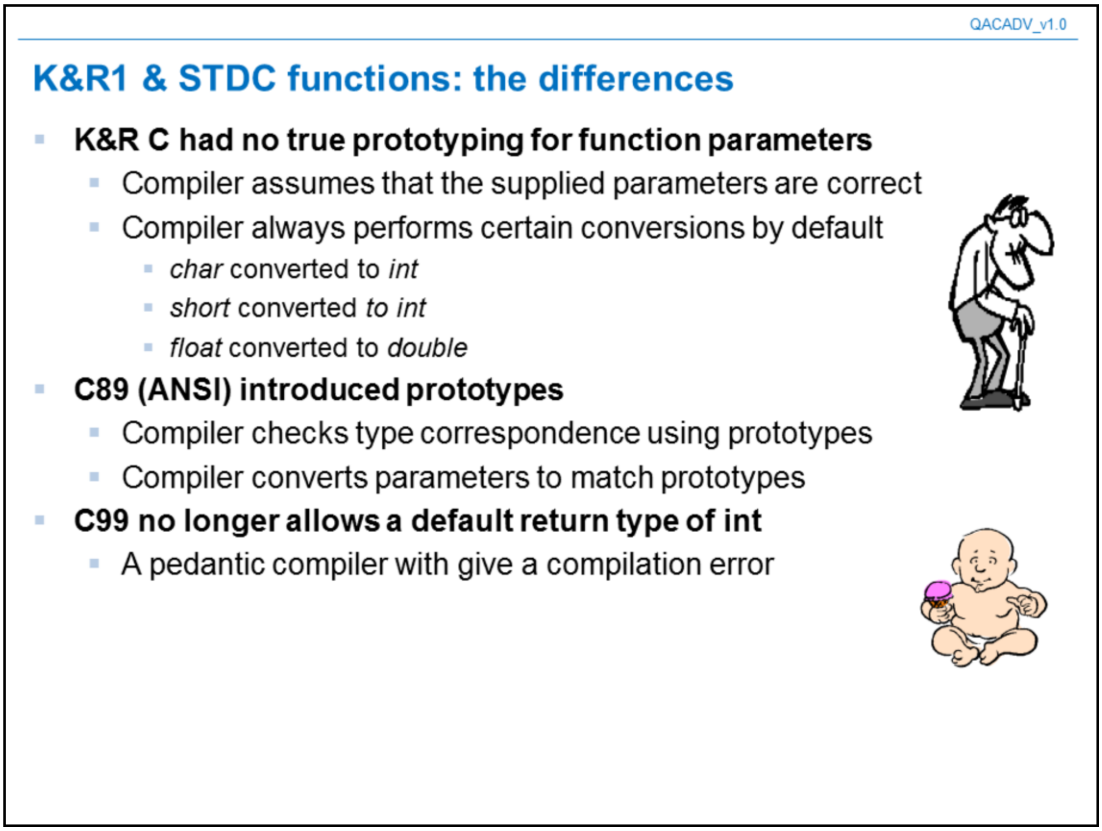
```
void myfunc (void)
```

means "myfunc returns nothing, and takes no arguments". Whereas:

```
void myfunc ()
```

means "myfunc returns nothing, and *has no argument checking*"!

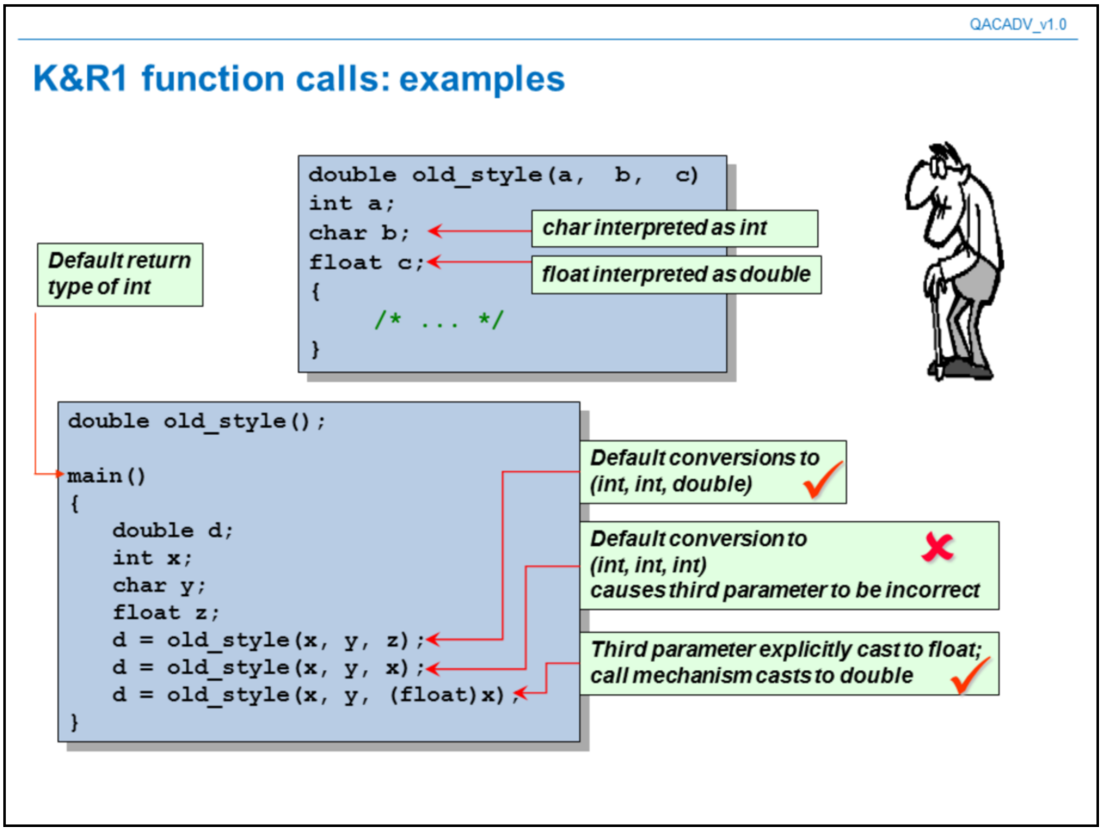
We have thrown in a further change formalised with C99, the adoption of the C++ comment prefix, //. Many implementations supported this before the adoption of C99.



In old-style functions, the compiler is never able to check the types of arguments used during the call to the function. Thus, argument promotions are performed automatically. Specifically, `short`s and `char`s are promoted to `int`s, and `float`s are promoted to `double`s. Casts within the call can be used to create user-controlled conversions.

If an ANSI prototype is used, the compiler will convert the argument to that of the type in the prototype parameter-list, as appropriate.

C99 has sewn up another hole in the standard left over from K&R days where a function return type need not be specified, the default being an int. Most compilers will issue a warning for that, but a strict C99 compiler may refuse to allow it.



The example illustrates the conversions and assumptions which take place when using the old style of the function declaration and definition.

In the function definition, the `char` and `float` arguments are interpreted as the "wider" types `int` and `double`. The function calls have nothing to go on; there is an old-style declaration which tells the compiler nothing about the types. The arguments are treated in isolation and are widened automatically as appropriate. Note the use of the cast in the last example.

Stay away from the K&R1 function declaration and definitions. Providing the compiler with information about argument types will endorse the integrity of the code. You will also be informed about any conversions performed due to the compiler's assumptions.

QACADV_v1.0

C99 inline functions

- **Functions can be defined as inline**
  - A request to the compiler to imbed the code at the function call
  - No guarantee that the compiler will implement it

```
inline int max (const int a, const int b)
{
   return (a > b)?a:b;
}
```

Keep inline functions short

Prototype and call in the usual way

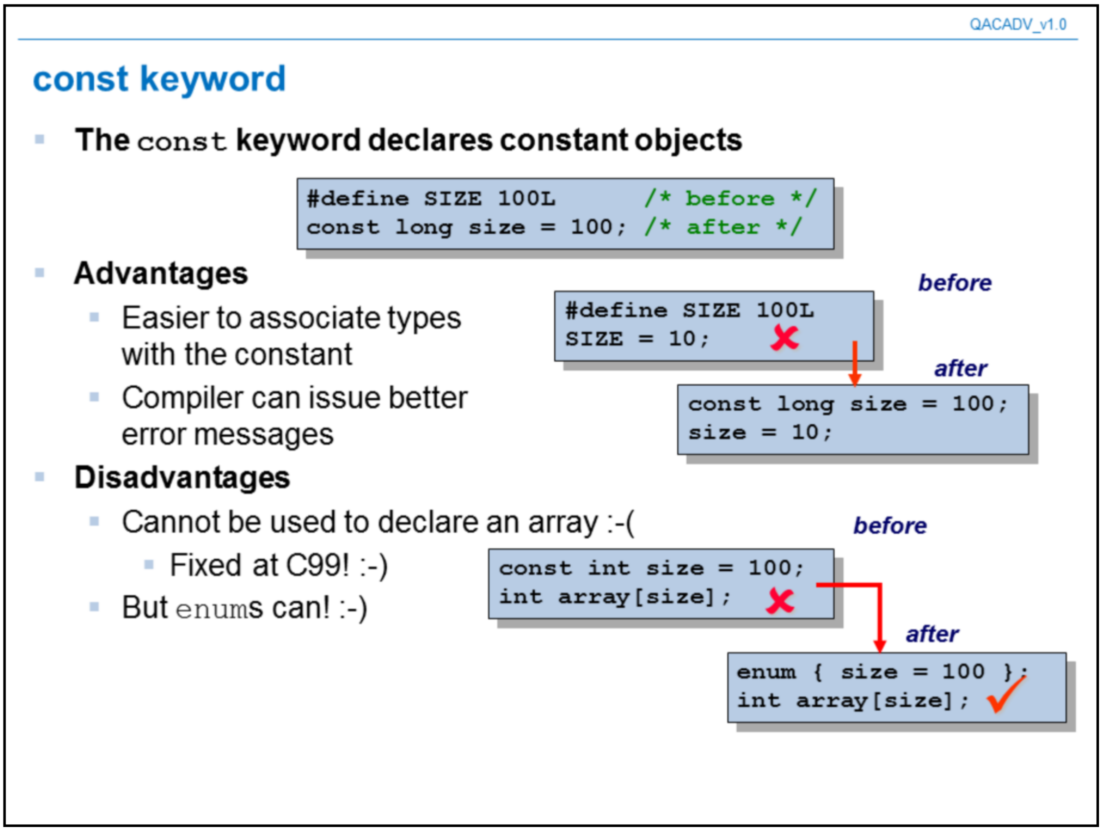
```
inline int max (const int a, const int b);
...
int x,y;
...
printf ("Maximum number is: %d\n", max (x,y));
```

- **Not used for external functions**
  - The linker would not be able to resolve the call

At C99 a function can be defined as **inline**. The compiler should then make the function call as fast as possible, which *might* involve imbedding the function code instead of making a function call. This is intended for short, fast, functions, where the overhead of a function call is significant. There can be considerable performance improvements, but beware that inline functions, if badly managed, can lead to code bloat. If you wanted code bloat you would be using C++ (which is where inline came from).

If an extern function is inline then there might be two versions of the function generated by the compiler. One would be a 'normal' function which can be called from another compilation unit (source code file) and the other would be an inline function used within the same compilation unit. The compiler is not guaranteed to behave in this way, it could just drop the inline attribute if the function is used externally.

In GNU gcc you should enforce the local rule by making inline functions static.



Introduced with C89, the **`const`** keyword speaks for itself. It indicates that the named data item is not a variable, but has a constant value which may not be updated at any time in the code block. The compiler will check that no assignment is attempted.

This has provided a more robust alternative to the preprocessor `#define`. It is more robust in that the `const` data item is a "typed" item and the compiler is able to produce more informative error messages if misused. Unfortunately, the `const` value cannot be used to specify the size of an array. This restriction is lifted in C99, which we shall see later. However even in C89 an enumerated value can be used to specify the size of an array:

```
enum { size = 100 };
int a[size];
```

In Standard C constants do not have to be initialised. The following is valid:

```
const int c;
```

If it is outside a function, it becomes a tentative definition and if no full definition is specified anywhere else, then the value is 0 by language definition. If there is a definition in another file, then it takes that definition's initialising value.

QACADV_v1.0

const pointers

- `const` *can* **be used in pointer declarations**

```
int i = 55;                         /* variable int */
const int ci = 66;                  /* constant int */
const int * pci = &ci;              /* variable ptr to constant int */
int const * pci2 = &ci;             /* variable ptr to constant int */
int * const cpi = &i;               /* constant ptr to variable int */
const int * const cpci = &ci;       /* constant ptr to constant int */
```

```
size_t slen(const char * literal)
{
    size_t result = 0;
    while (*literal != '\0')
    {
        literal++;
        result++;
    }
    *literal = '\0';
    return result;
}
```

variable pointer to constant character(s)

compiler spots mistakes like this

The `const` keyword is a qualifier, but it can also stand as an abbreviation for `const int`. When used together, the keywords `const` and `<type name>` can be interchanged (as in the third and fourth examples above).

Difficulties arise when differentiating between the third and fifth examples:

> `pci` can be assigned another address, but `(*pci)` cannot be assigned a new `int` value
>
> `cpi` cannot be assigned another address, but `(*cpi)` can be assigned a new `int` value

The former is used frequently as a function parameter-type. The function has access to the data via the pointer but cannot update the data. This is C's answer to a safe call by reference.

QACADV_v1.0

volatile keyword

- References to *volatile* data items are not optimised by the compiler. Each reference in the source code corresponds to one reference in the compiled code
- The *volatile* qualifier tells the compiler that the object could be modified by means other than the current unit of code

Main Process or Primary thread

```
volatile int interrupt_flag;

interrupt_flag = 0;

while (!interrupt_flag)
{
    /* Do nothing */
    /* wait for interrupt */
}
```

Asynchronous Process or Thread

```
extern volatile int interrupt_flag;
/* .. */



interrupt_flag = 1;

/* ... */
```

Compilers have a habit of optimising "out" items which are better placed "elsewhere". This usually refers to the object being placed in a register or moved somewhere else more convenient during the processing of an algorithm. The `volatile` object has to remain in a fixed place throughout its entire existence.

Note that any type of object can be `volatile`, such as an array, `struct`, `union`, pointer, etc. They can also be `const`. The semantics for `volatile` are implementation dependent, and the keyword was introduced with C89.

Enumerated types

- **Enumerated types were introduced with C89(ANSI)**

```
enum day { sun, mon, tue, wed, thu, fri, sat, };
enum day weekday = mon;
enum day today = weekday; /* :-( */
```

Trailing comma allowed with **C99** (wow!)

Although the compiler allows such types to be declared, there is no other support (no predecessor or successor functions)

- **Actually implemented as integers**

```
today++;
today--;
```

```
if (today == mon)
    go_to_work();
```

compares "today" with 1

The `enum` data item is an alias for an `int`. The token names must be distinct, can be given any `int` value and can be used freely in the source code when expected. However, the internal symbol is an `int` and all processing is performed on that `int`. There is no runtime checking performed by the compiler and no library support whatsoever; names cannot be inputted directly into the `enum` variable and are outputted as `int`s.

`enum`s are used to enhance readability and as alternatives to the `#define`. The latter use is attractive because the compiler is able to generate distinct values for the token names automatically if required.

Note that like `struct` and `union`, the `enum` keyword is required in data definitions:

```
enum day { sun, mon, tue, wed, thu, fri, sat };
enum day today;        /* correct   */
day tomorrow;          /* incorrect */
```

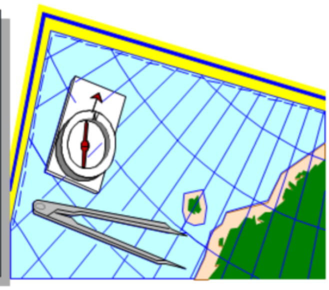
C99's contribution to enums was to allow a trailing comma in the initializer, a feature that most implementations supported anyhow!

QACADV_v1.0

Choosing enum values

- **Identifiers can be given specific values:**

```
enum direction
{
  north = 0,    north_east = 45,
  east = 90,    south_east = 135,
  south = 180,  south_west = 225,
  west = 270,   north_west = 315
};
```



- **enums may only be printed as integers**

```
enum direction heading = north_east;
printf("heading is %d degrees\n", heading);
```

The example used here illustrates the control the programmer has over the values assigned to identifiers in an `enum` type. It also illustrates the improvement in readability. The names are grouped together neatly, which is an improvement on the eight `#defines` required as an alternative.

Note that in `printf`, the `enum` variable `heading` is handled as an integer.

It is, however, possible to write the following code statements with no warnings coming from the compiler; a price we have to pay for the simplistic implementation:

```
enum direction somewhere = east;
somewhere *= 3;
if (somewhere == west)
     printf("I got away with it!\n");
```

QACADV_v1.0

Boolean type

- **C99 introduced _Bool**
  - Only guaranteed to be large enough to store 0 or 1
- **Header file <stdbool.h> provides 4 macros:**

bool	***defined as***	**_Bool**
true	**"**	**1**
false	**"**	**0**
__bool_true_false_are_defined		**1**

  - Since these are macros, a program can redefine them!
- **Using an int as a Boolean has been standard practice**
  - Don't mix them up, they might be different!

At long last, C programmers have a Boolean data type. **_Bool** is a built-in type from C99 onwards, which is different from the name **bool** used in C++. To use that, and **true** and **false**, we need to **#include <stdbool.h>**, although this header file is not required in order to use _**Bool** (just a C99 compiler).

For years, C programmers have worked around the lack of a Boolean data type by using typedef or #define, and usually that has been (for no good reason) an integer. For example, Microsoft use a type named BOOL, which is an unsigned int. The new type _**Bool** might not be as large as an int, in fact most implementations will use a single byte, so be careful of introducing the new type into existing code. Also, if an API uses its own version of a Boolean type then use it, do not be tempted to use _**Bool** unless the documentation allows it.

Another trap for the unwary is in using a _**Bool** type inappropriately. For example, let us say we have a function, myfunc, that returns an integer value. A return value of zero indicates success, and any non-zero value is failure. Conventionally we might call myfunc like this:

```
if (myfunc())
    /* it failed! */
```

There is an argument for saying this is unreadable and bad practice. In C99 you might be tempted to do this:

```
#include <stdbool.h>
if (myfunc() == true)
    /* it failed! */
```

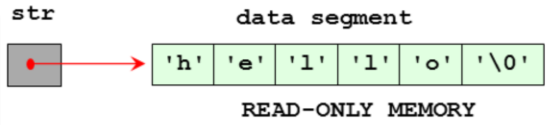
Unfortunately this is **wrong!** The value of `true` is **1**, not "any non-zero value".

QACADV_v1.0

Changes to strings

- **A C compiler may place strings into read-only memory**

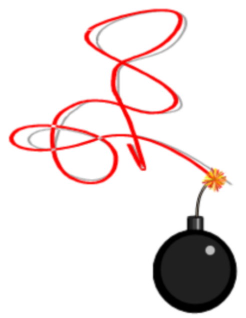
```
char * str = "hello";
*str = 'H';
str[4] = 'O';
```



- **Unfortunately, many OS's have no read-only segments, and mistakes go undetected**
- **The declaration *should* be...**

```
const char * str = "hello";
```

- **The compiler does not enforce this**
  - At best you will get a warning



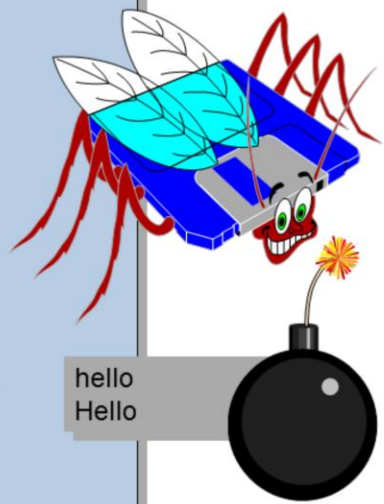
The string, which is referred here, is the string literal within the double quotation marks and accessed by a `char` pointer. The literal should be regarded as a constant object, and may be placed on some platforms in read-only memory. The important word is "may"! The programmer should never assume that the string can be written to. The `const` qualifier will consolidate the definition and prevent accidental damage.

QACADV_v1.0

Example

```c
#include <stdio.h>

void  change(void);

int main(void)
{
     change();
     change();
     return 0;
}

void  change(void)
{
     char * str = "hello";

     printf("%s\n", str);
     *str = 'H';
}
```

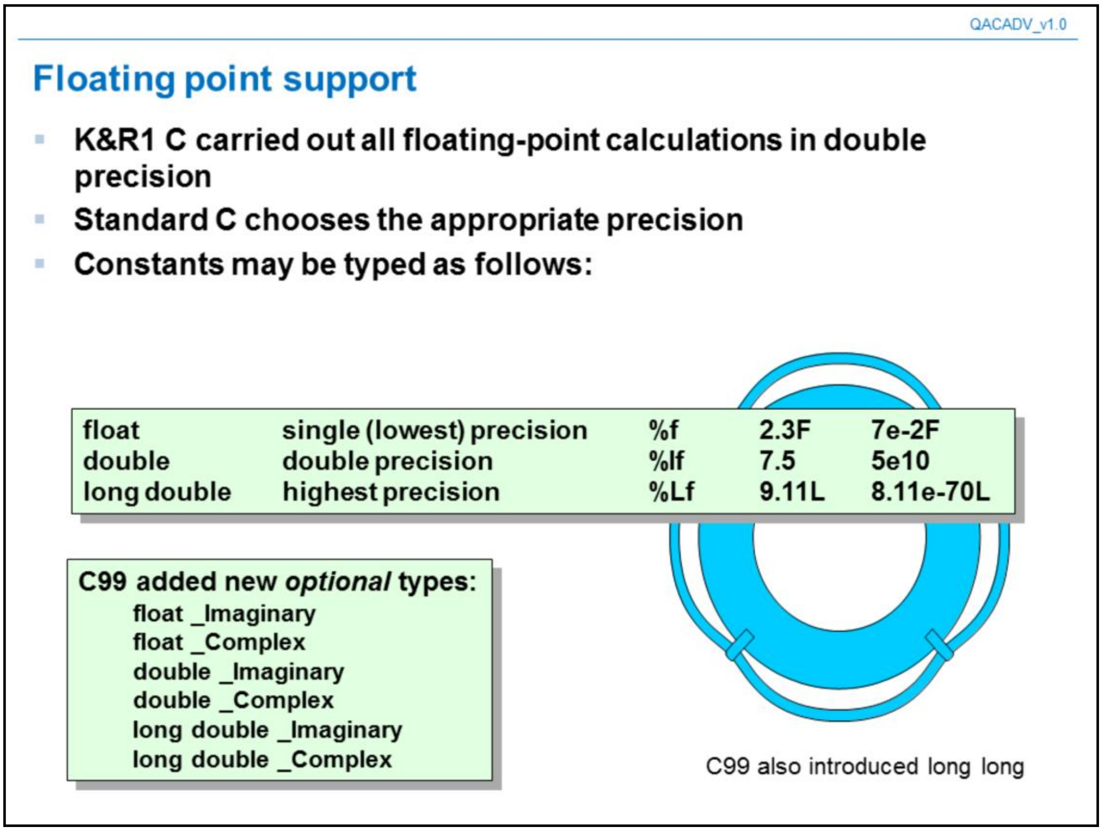
```
hello
Hello
```



This program illustrates a consequence of the storage of strings in the data segment. The first time the `change` function is called the string is altered. When called a second time, the *altered* string is printed.

It should be pointed out that the program will only work under compilers and environments which do not used read-only memory. For example, this "works" on Microsoft Developer Studio 5.0, but fails correctly with an access violation on version 6.0.

Few modern operating systems will tolerate this abuse! When the ‘h’ is overwritten the program will be killed by the operating system, *provided* the compiler applies the correct protection to its constants.



In order to make function communication easier and faster, all floating-point data used to be passed and processed as double precision quantities in K&R compilers. Standard C compilers are more intelligent in this area. They are more likely to do what the programmer requests.

It recognises suffixes in constants which indicate the programmer's desired intentions, i.e. the `F/f` and `L/l` which stipulate `float` and `long double` precision respectively. The compiler still uses a `double` as the default type for a floating-point constant.

It has already been established that a `float` argument is treated as a genuine `float` when it is received by a function using the proper prototyping and definition rules.

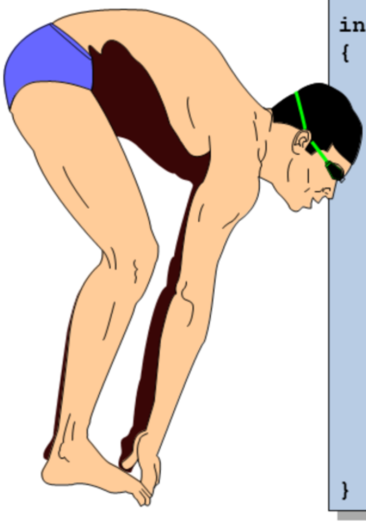
The *usual arithmetic conversion* rules were extended to implement the programmer's request to have a floating-point quantity treated as a single precision or long precision item as required.

At C99 a set of imaginary and complex types were added, but these are only optional. A compiler can consider itself C99 compliant without providing them.

QACADV_v1.0

Floating point exercise

- **How will the compiler handle the following code?**



```
#include <math.h>

int main(void)
{
    int           x = 2, y = 5, z = 5.1;
    float         a, b, c = 7.0;
    double        g, h, i = 7.1;
    long double   m, n, o = 8.5;

    m = x / y;
    a = z * y;
    b = c * 0.45;
    c = c * (float)m;
    h = i * x * 2.7F;
    g = a * sin(a);
    g = a * sin(x);
    n = c * g;

    return 0;
}
```

The program includes examples of the use of suffixes for constants, user-defined conversions and the *usual* automatic conversions. Microsoft Visual C++ compiler, and the GNU compiler (gcc) on Linux, generated the following results when using `printf` with default format specifiers `%f` and `%Lf`:

```
m    0.000000
a    25.000000
b    3.150000
c    0.000000
h    38.340001
g    -3.308794
g    22.732436
n    0.000000
```

QACADV_v1.0

Finding limits

- **Header files are provided to determine the valid maximum and minimum values of different types**

`char`	`CHAR_MIN`	`CHAR_MAX`	
`short`	`SHRT_MIN`	`SHRT_MAX`	
`unsigned short`	`0`	`USHRT_MAX`	
`int`	`INT_MIN`	`INT_MAX`	
`unsigned int`	`0`	`UINT_MAX`	
`long`	`LONG_MIN`	`LONG_MAX`	
`unsigned long`	`0`	`ULONG_MAX`	
`long`	`LLONG_MIN`	`LLONG_MAX`	`C99`
`unsigned long`	`0`	`ULLONG_MAX`	`C99`

`<limits.h>`

`float`	`FLT_MIN`	`FLT_MAX`
`double`	`DBL_MIN`	`DBL_MAX`
`long double`	`LDBL_MIN`	`LDBL_MAX`

`<float.h>`

- **C99 also introduced inttypes.h**
  - Provides portability between different word lengths
  - Many require __STDC_FORMAT_MACROS defined

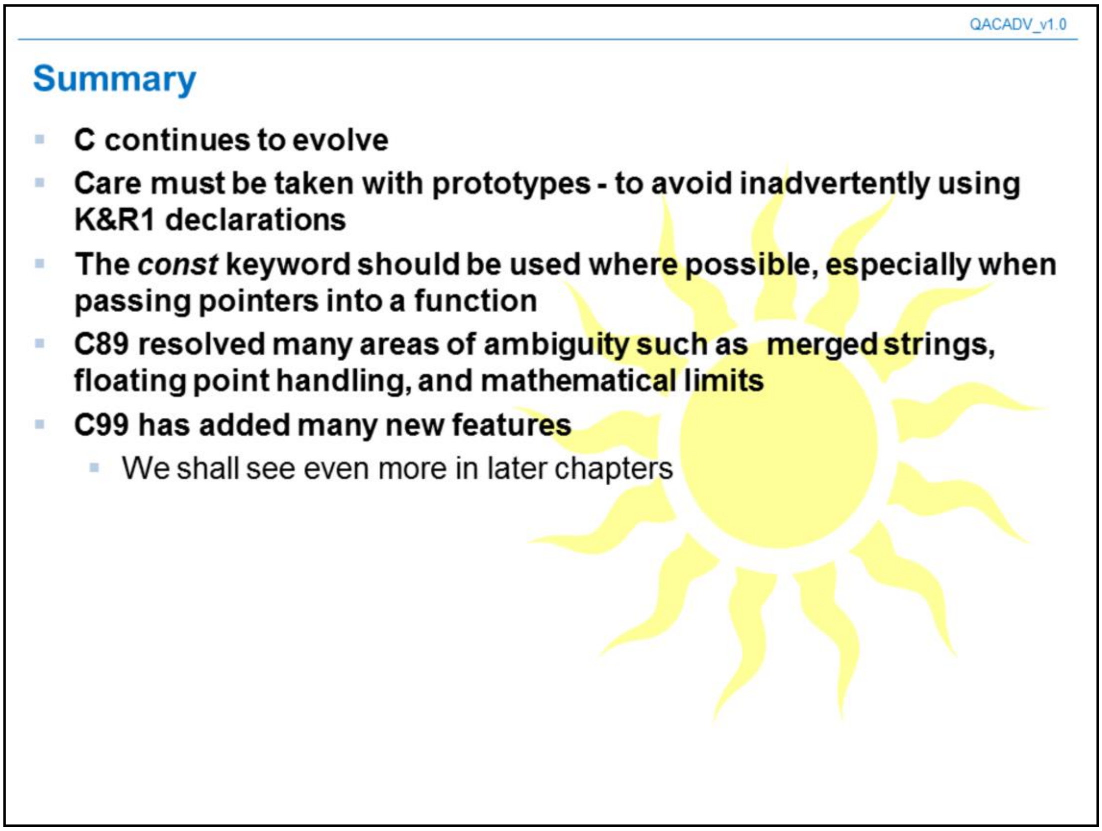
Limits have to be implementation specific. C is famous for its ability to "exploit the use of the underlying system". The only constant setting in Standard C was that of `CHAR_BIT` which is `8`.

All values quoted in the standard are the acceptable minimum magnitudes; larger values are likely to be used. For example, the value `65535U` for `UINT_MAX` is an acceptable smallest value for the maximum `unsigned int` value.

The floating point examples shown above are supplemented by other constants which cover information on the exponent representation and size of granularity.

Two new variable types were introduced at C99, long long and unsigned long long. On most 32-bit implementations these are 64-bits, but what should they be on 64-bit systems? Many ANSI compilers have followed a different path and defined types such as int64, int128, which at least says exactly how big the type is regardless of the hardware. Not everyone is happy with the introduction of the new types!

C99 has attempted to aid portability with the inclusion of inttypes.h, which defines a large set of macros. These cover, among others, the format specifiers for the printf /scanf family.



This chapter has reviewed the most important changes made to C during the standardisation of the language. A significant change has been the adoption of full function prototyping, which was added to C after first appearing in C++ in the mid 1980's.

Another important addition has been the `const` keyword - it is remarkable how much difference such a simple concept can make to a program! Whenever passing pointers into a function, you should ask yourself whether `const` can be used to ensure the data being pointed at does not get corrupted. All of the standard functions in the C library use the `const` keyword religiously to preserve the integrity of incoming pass-by-reference data.

As well as the topics mentioned above, the chapter has also looked at `volatile` variables and enumerated data types, and some of the changes introduced with C99 like `inline` and `bool`. Other significant changes in C99, like using a variable to define an array length, will be seen in later chapters.

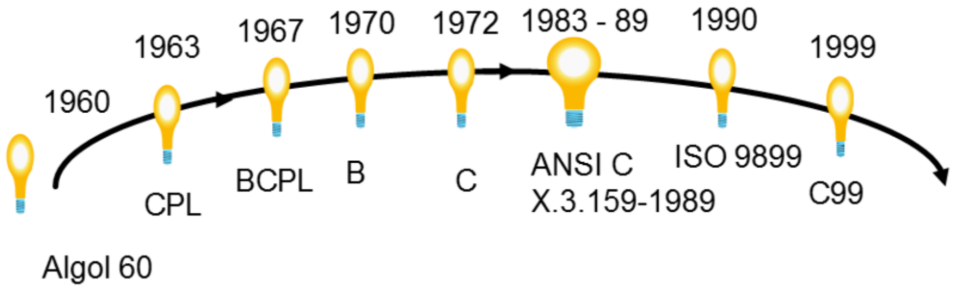
NOTE:

The ANSI and ISO committees have had to cope with many advancements in the computing industry. One of the major areas was the improvements in the internationalisation of computing technology, i.e. the industry was empowered to think about portability across different character sets, time zones, monetary representations, etc. To this end, C90 included support by implementing a "locale" mechanism. See the Appendix on Internationalisation.

QACADV_v1.0

History and evolution of C

- **Designed by Dennis Ritchie at Bell Laboratories in 1972**
- **Its ancestry gives an insight into the nature of C today**



C's major ancestor was the BCPL language; a British language that was basically an assembler language with high-level language syntax. BCPL took many of its constructs and syntax from Algol. C, as it is today, first came to light in 1972; the same year as Pascal!

For over a decade, it ran wild! Every machine had its own version. Only the Unix versions seemed to be controlled in any way. By the time the American National Standards Institute (ANSI) Committee got its hands on it, the problem seemed insurmountable. The creators Kernighan and Ritchie kept tight control of the language, which meant that there was a 'standard' known affectionately as K&R C. Starting from this standard, a new, more realistic and robust language emerged. It took six years to reach a conclusion, but even then, many people thought that the language was still too lax, and others thought that the changes were too drastic.

The C standard is now accepted in all countries that subscribe to ISO, the International Standards Organization. C standard is now known as BS EN ISO 9899, which indicates the acceptance as a British, European and international standard.

It should be noted that the ISO C++ standard was ratified in 1998, and that C++ will not be a strict super set of C.