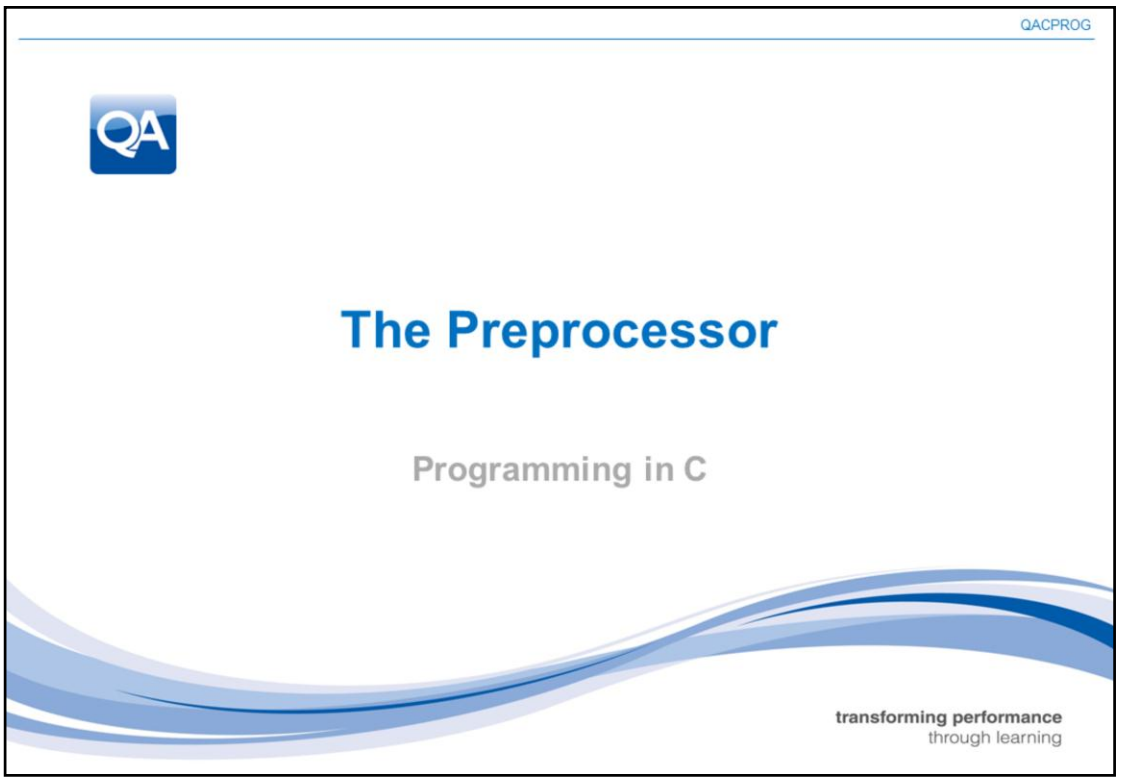
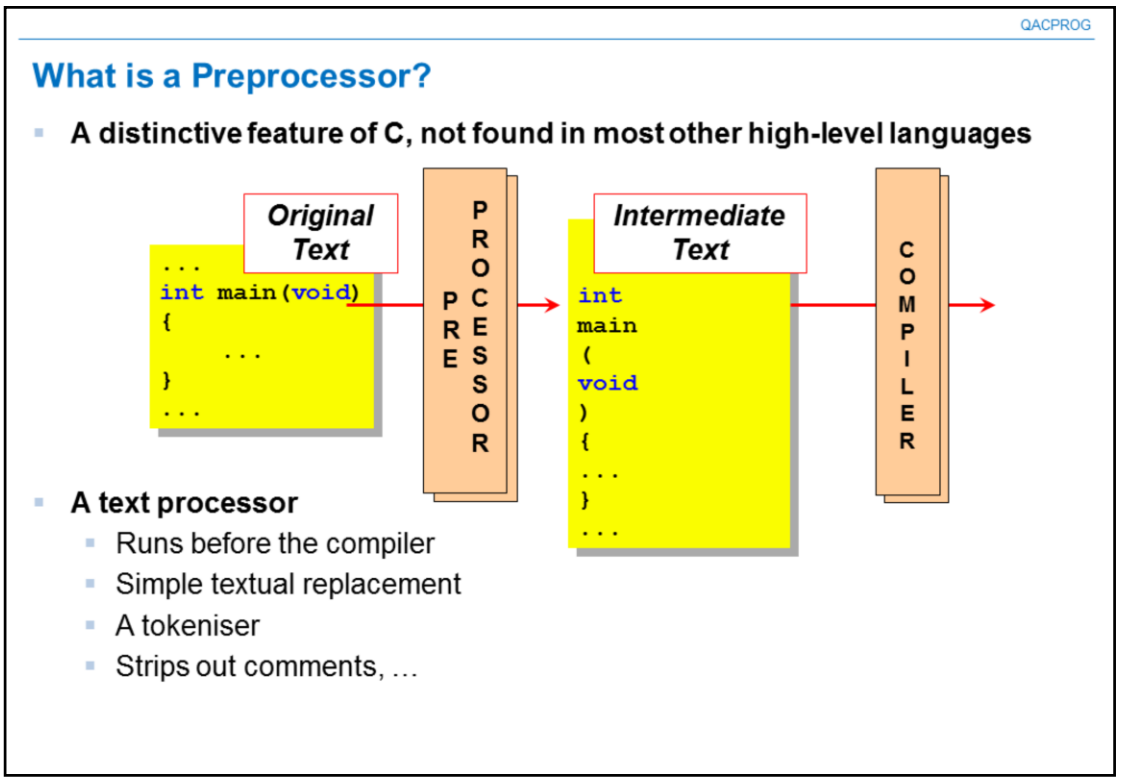

QACPROG
The Preprocessor
Programming in C
transforming performance
through learning

QACPROG

The Preprocessor

- **Objective**
  - To improve readability, maintenance and portability?
- **Contents**
  - Constant definition
  - Macro substitution
  - Macro pitfalls
  - File inclusion
  - Conditional compilation
- **Summary**
- **Practical**
  - Using #define and #include to help program organisation

The objective of this chapter is to cover the three major uses of the preprocessor. Other facilities are mentioned at the end of the chapter.



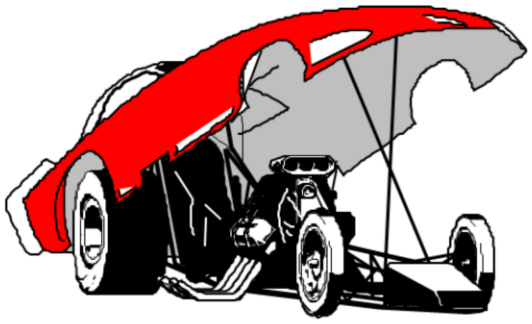
The preprocessor is part of the compilation phase. It is, effectively, a first parse of the source code. During the parse, it performs some processing. All of this processing is geared towards changing our well-documented, clear and portable C source code into 'compiler acceptable' input. It provides the layer that is more 'programmer friendly'. Most compilers give you the option of viewing the resultant output. The output from the preprocessor, which is the input to the compiler, is either displayed on standard output or placed in a temporary file that usually has the filename extension `.i`.

QACPROG

Preprocessor Directives

- Preprocessor directives may be embedded in C code
- All directives start with a #
- These facilities can aid code
  - Readability
  - Maintainability
  - Portability

```
...
#.....
...
int main(void)
{
    ...
}
```

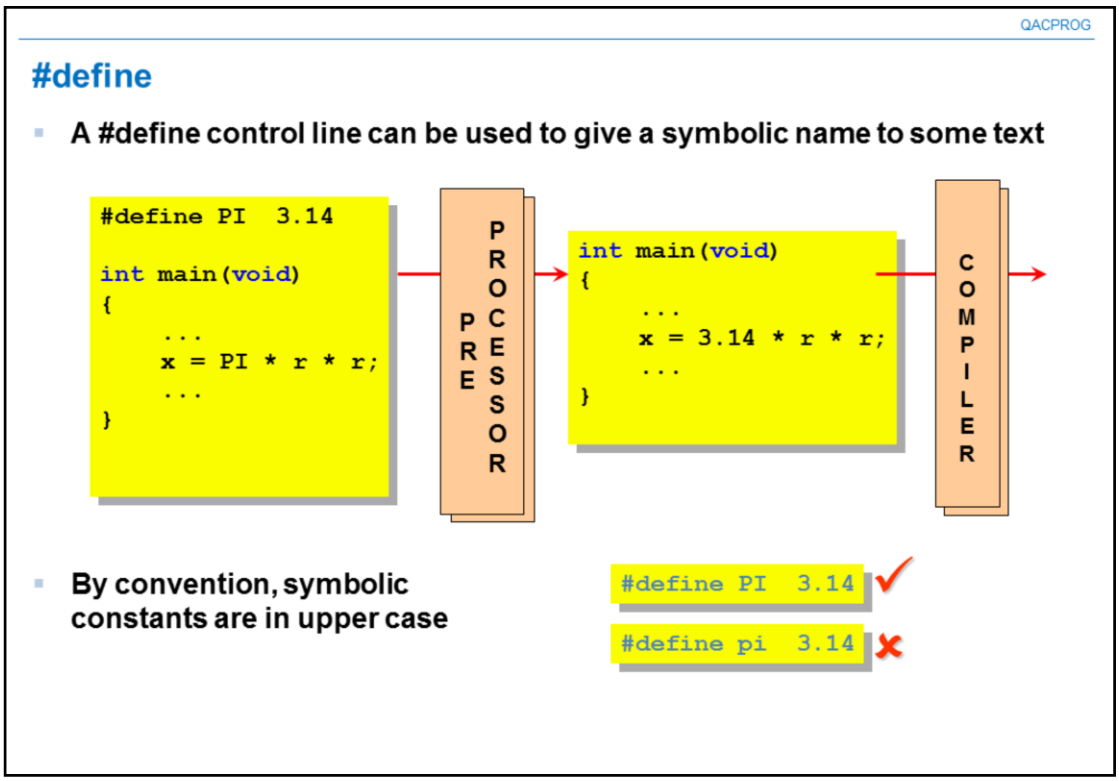


Preprocessor commands are line based and can be placed anywhere in the source file. Without exception, they begin with the # character. Syntax is similar, but not identical, to C.

The preprocessor is regarded as part of the C environment and is controlled by the ISO standard. The following directives are supported:

```
#define and #undef
#include
#if  #elif   #else    #endif
#ifdef       #elif    #else    #endif
#ifndef      #elif    #else    #endif
#line
#pragma
#error
```

As well as these, there are other facilities: trigraphs, the `defined` keyword, predefined identifiers and the two operators, # and ##.



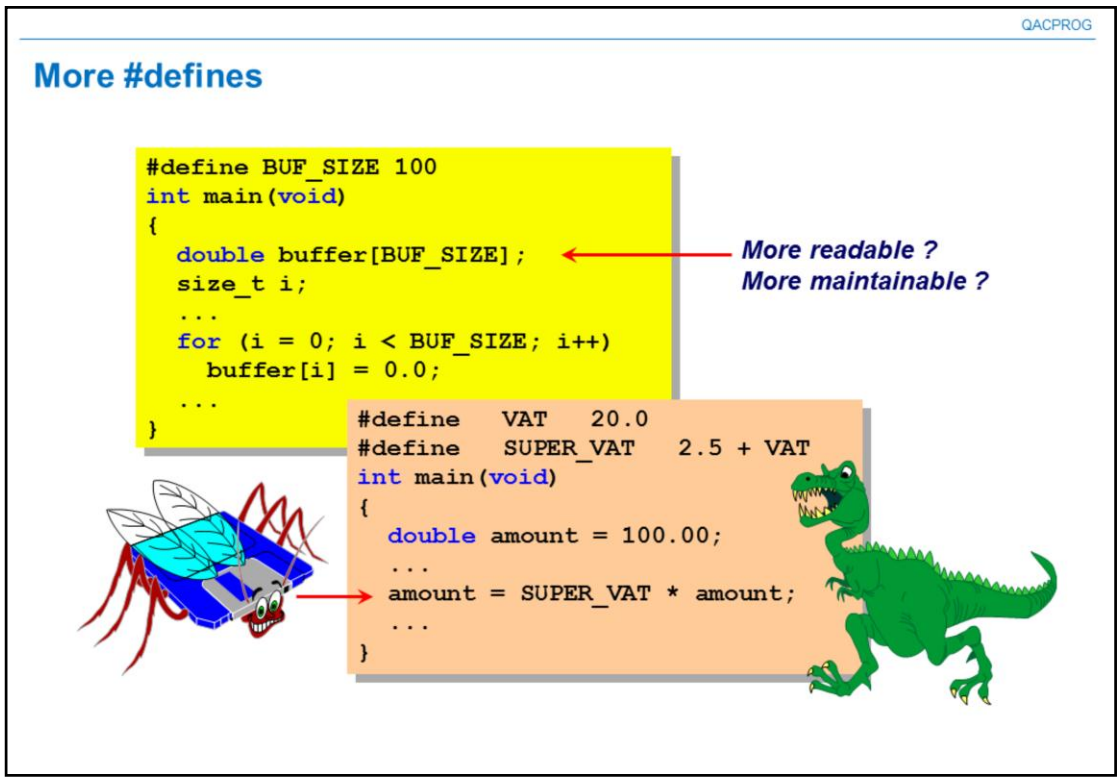
The expansion text of the directive could be anything. It starts immediately at the first non-white space character after the name. It is delimited by the end of line, though this can be overcome by placing a \ character immediately before the newline, thus extending the expansion text to two or more lines. This is more useful when defining macros, which tend to be more than one line long.

The `#undef` directive is used to wipe out a previously `#defined` name. The syntax is simply:

```
#undef name
```

This is usually used when the name is to be given a new value in the source that follows.

The name used can be any legal C identifier.



Because the preprocessor effectively performs a single parse, the scope of its `#define` tokens are global and take effect on the line after the definition. This means that you may define tokens using tokens already defined. However, you must be careful with precedence. Brackets are required to make the code shown above work, i.e.:

```
#define  SUPER_VAT  (2.5 + VAT)
```

Extreme care is required when defining names whose text are other `#defines`.

A `const` *cannot* be used for array sizing.

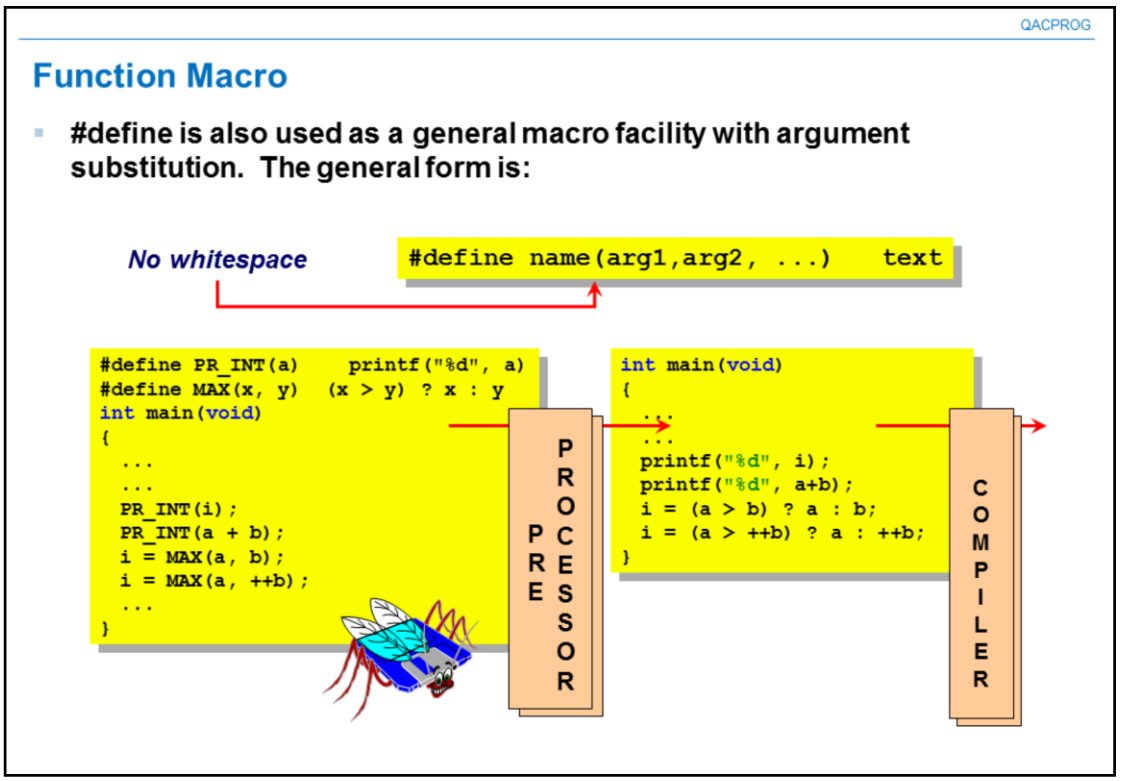
```
const size_t buf_size = 100;
double buffer[buf_size];
```

However an `enum` literal *can*:

```
enum { buf_size = 100 };
double buffer[buf_size];
```

Many experienced programmers (who have already been bitten by the many gotchas covered in this chapter) try and avoid the preprocessor and hence prefer the `enum` mechanism. Note that the change of context (away from the preprocessor) makes the change from uppercase `MAX_SIZE` to lowercase `max_size` a sensible one - we do not want the `enum` literal clobbered by a `#define`!

```
#define MAX_SIZE 100
enum { MAX_SIZE = 100 }; /* ouch */
```



Be careful with the syntax of the macro. There must not be any space before the '`(`' introducing the argument list. For example:

```
#define PR_INT  (a) printf("%d", a)    /* WRONG! */
```

The preprocessor would take the line `PR_INT(i) ;` and make it:

```
(a) printf("%d", a)(i) ;
```

Not a pretty sight!

The version of `MAX` given in the example is not the best. It will work for the example shown above, but not for other, more complex, expressions.

An improved version is given on the next page.

QACPROG

Exercise: Problem Macros

- What went wrong? Can you correct it?
- What are the pros and cons of macros over functions?

This is not the correct algorithm for a leap year test

```
#define   IS_LEAP(y)       y % 4 == 0
#define   DAYS_IN_FEB(y)   IS_LEAP(y)?29:28
int main(void)
{
    int day, year = 2012;
    double amount;
    ...
    if (IS_LEAP(year))
        printf("Leap!!");
    ...
    day = DAYS_IN_FEB(year);
    ...
    amount = DAYS_IN_FEB(year+2) * 0.1;
    ...
}
```

The preprocessor performs text manipulation for `#define` substitution. For macros, it adds the facility of replacing arguments on a one-to-one basis. The preprocessor has no knowledge of the rules and syntax of C expressions. It will not add integrity to an expression, i.e. it will not add precedence brackets. This is the responsibility of the designer of the macro.

Solution

There are two simple rules:

Place brackets around individual arguments in the macro's text.
Place brackets around the *entire* expansion text.

The macros should be:

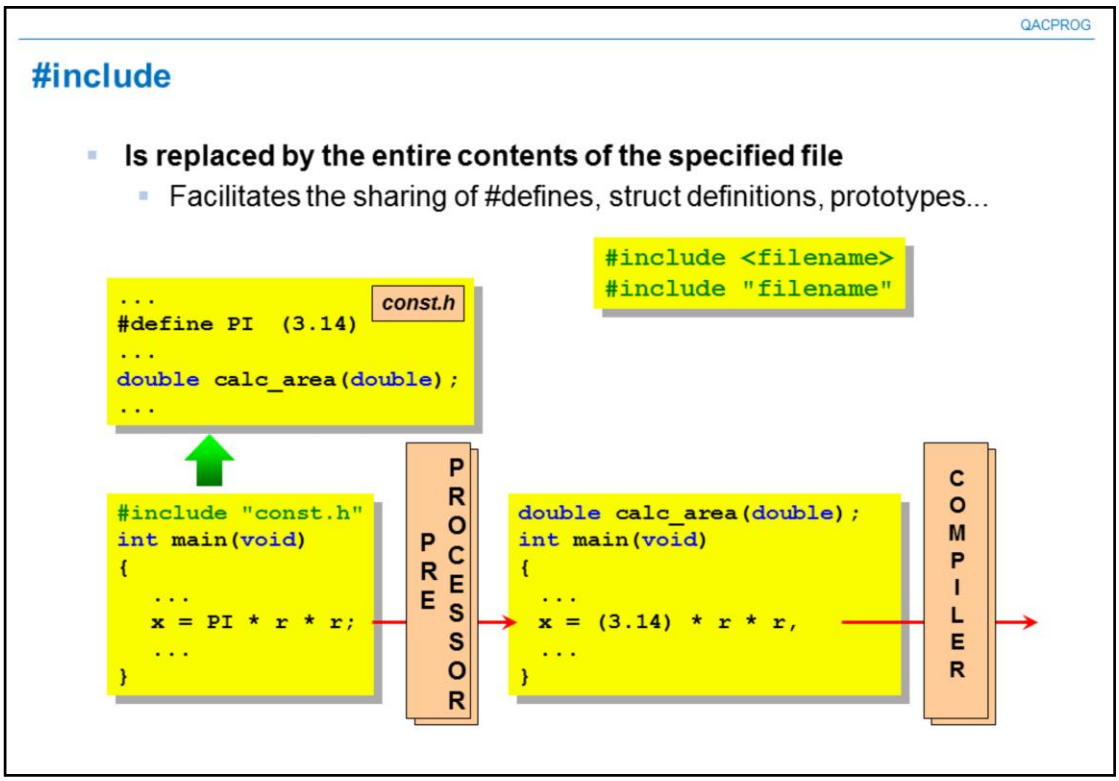
```
#define IS_LEAP(y)      ((y) % 4 == 0)
#define DAYS_IN_FEB(y) (IS_LEAP(y) ? 29 : 28)
```

`MAX` (see previous page) should be:

```
#define MAX(x,y) ((x) > (y) ? (x) : (y))
```

Unfortunately, there are still operations where macros are inadequate. The section at the end of this chapter covers the most common problem.

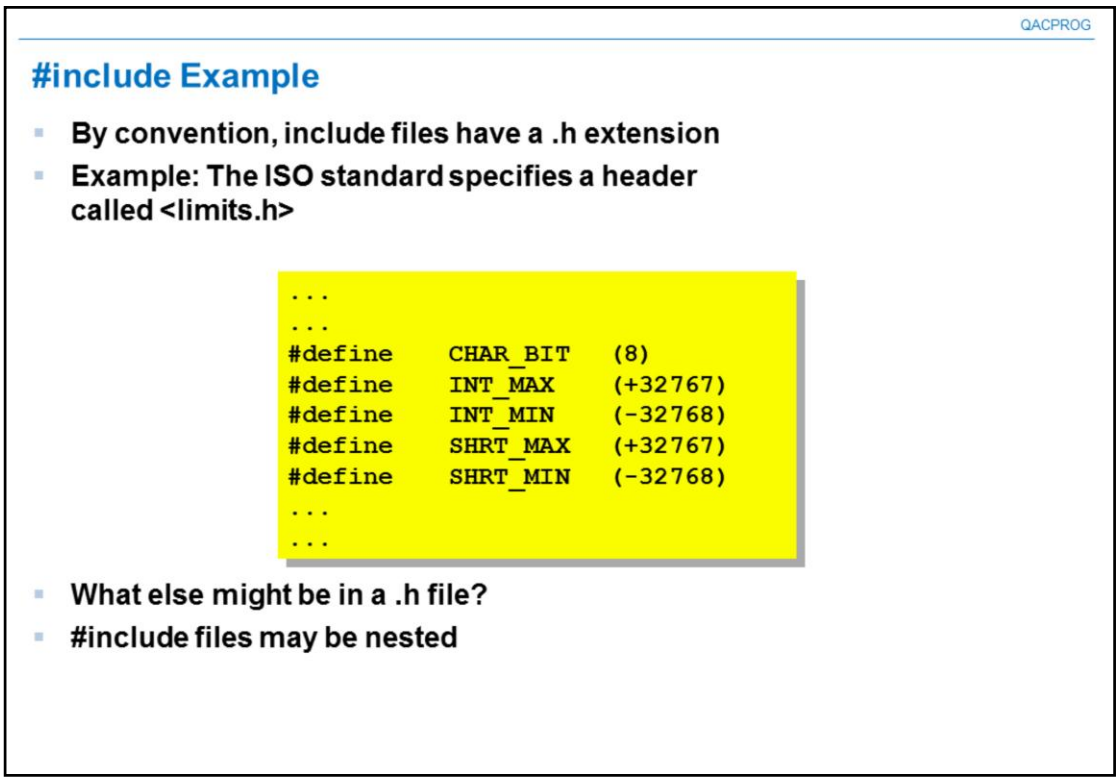
Pros: fast (no function call overhead) and generic (hence no type checks)
Cons: no type checking, larger code, hard to debug, lots of brackets, its not C



`#include` finds the location of its files to include in one of two ways:

If the filename is enclosed in angle brackets (< and >), the header file is searched for in the places specified by the compiler. A special include directory is normally searched first.

If the filename is enclosed in double quotation marks, the compiler uses the name as an absolute or relative pathname starting from the current directory. If that search fails, the compiler adopts the search rules for the angle-bracket notation.



A header file is an ordinary text file. It could contain anything that, when `#included`, will make sense to a compiler. You are, however, advised to adhere to certain rules:

> Do not include function definitions, i.e. bodies. This could lead to multiple function definitions, which could cause a problem for the linker.
>
> Do not include global data definitions. Each will be a different data item with scope restricted to that of the individual source file. This will lead to a communication breakdown.

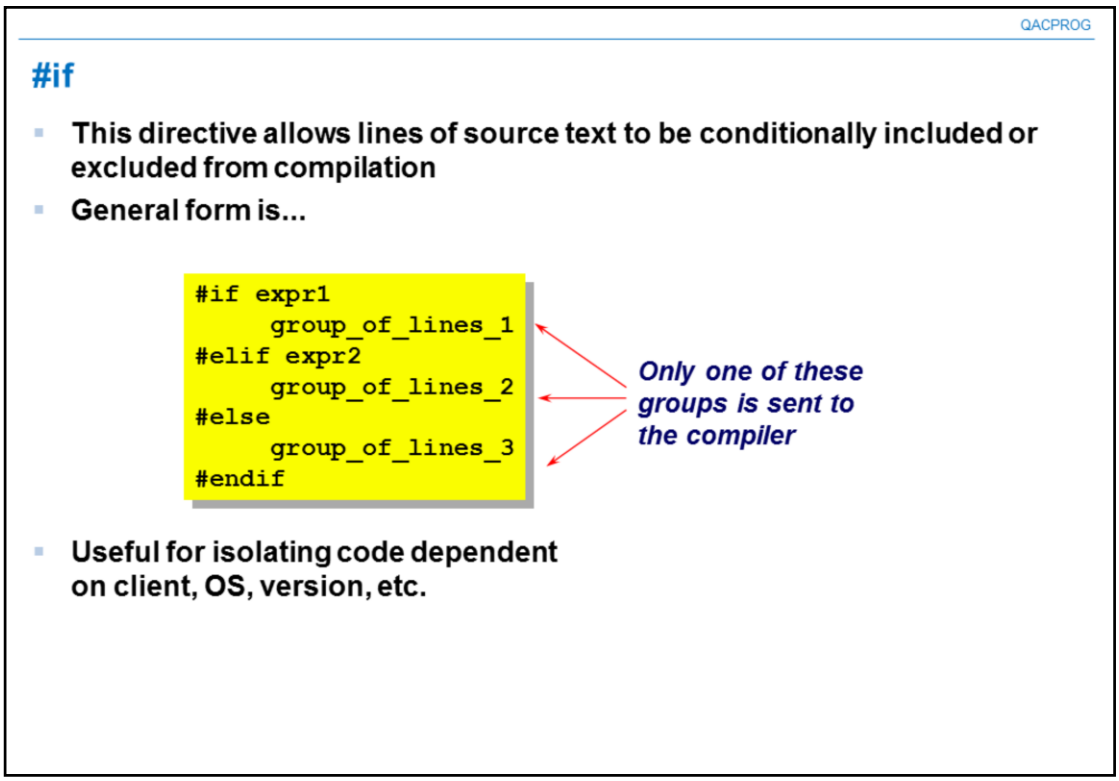
You are actively encouraged to put anything else in, especially if it is useful to the entire program. More information will be found in the *Working with Large Programs* chapter.

Including system files (such as `stdio.h`) within your own header files requires careful management and is not encouraged without safety mechanisms, such as `#if` constructs.

Note: You should note that all programs that use I/O routines, such as `printf` and `scanf` should have the following directive at the top of the source:

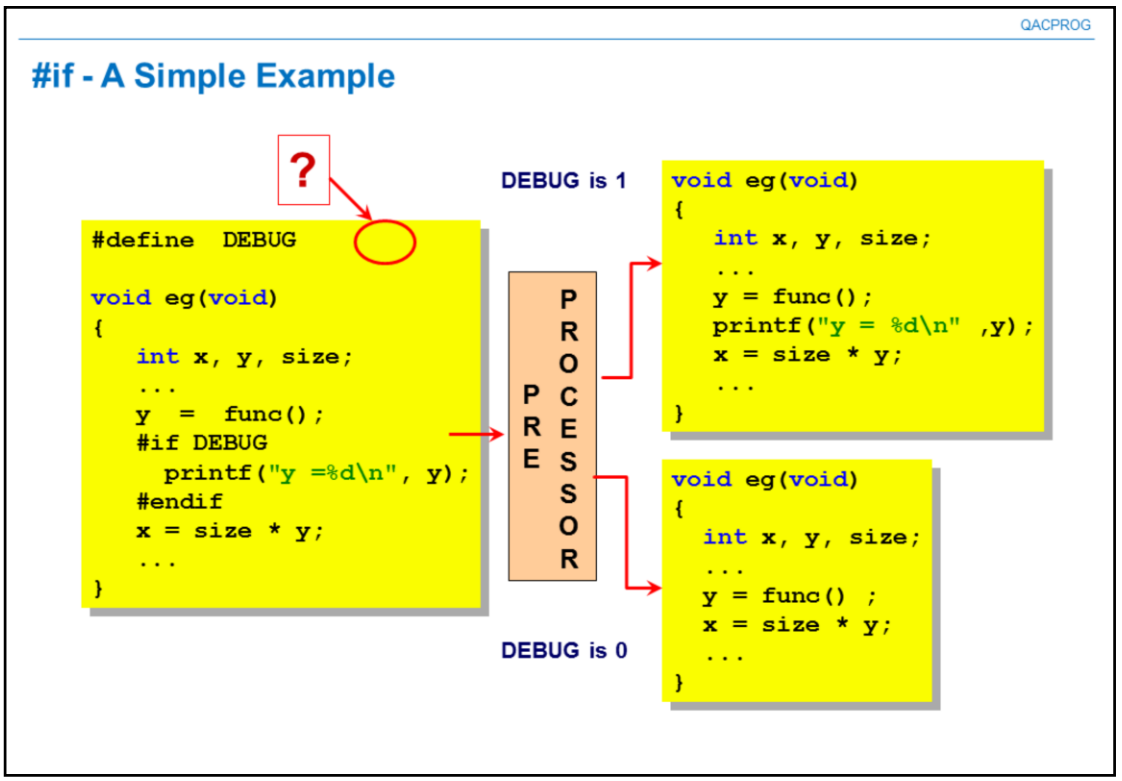
```
#include <stdio.h>
```

This will ensure that the compiler has the appropriate prototypes available.



The `#if` construct has many forms. The form of the example happens to be the neatest version. The `expr1`, `expr2`, etc. could be of the form `GROUP == 1`, `GROUP == 2`, etc., where `GROUP` would be `#defined` as the required integer, depending on the compiler/implementation. The line `#define GROUP` ... would be the only line changed for the mechanism to work.

A typical example could be that the `group_of_lines_N` would be function definitions, each with the same interface but containing implementation-dependent code statements.



The alternative `#if` construct could be used in this example.

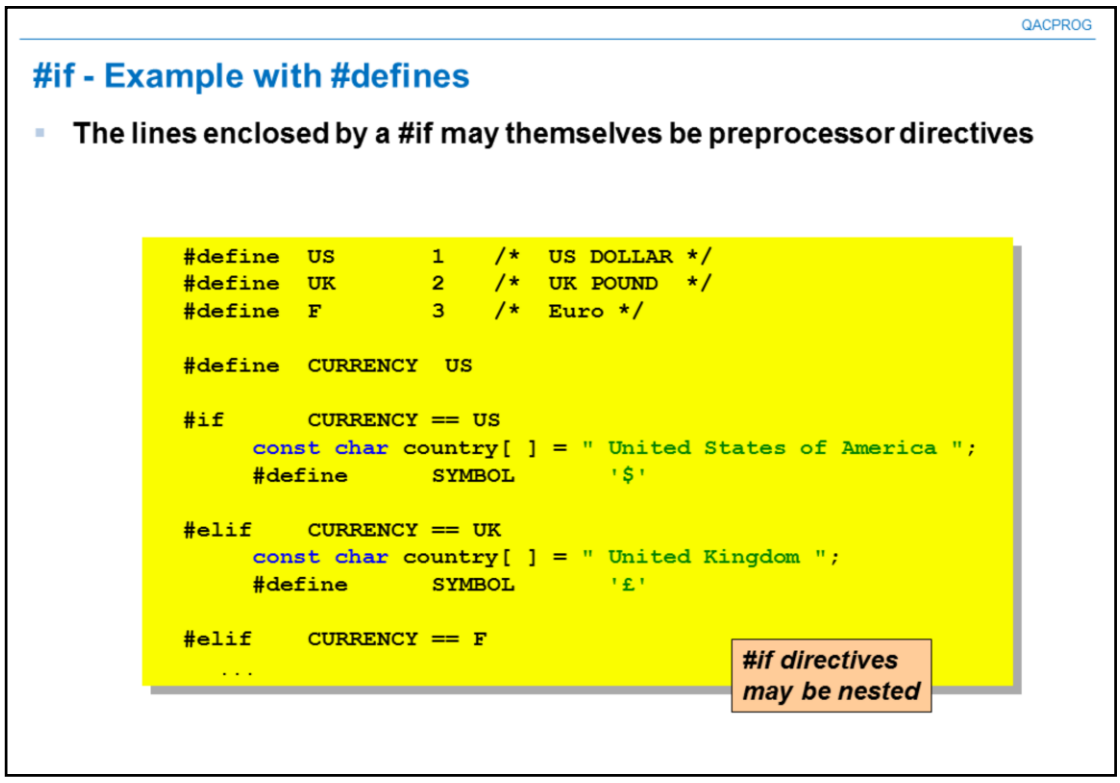
Change `#define DEBUG 1`
to `#define DEBUG_FLAG`
...
or `#define DEBUG 0`
to `/* Nothing */`
or `#undef DEBUG_FLAG */`

then
change `#if DEBUG`
to `#ifdef DEBUG_FLAG`

Using the preprocessor's `defined` keyword, an alternative to `#ifdef DEBUG_FLAG` is:

```
#if defined DEBUG_FLAG
```

Note that the `#if DEBUG` is true for all values of `DEBUG` other than `0`.

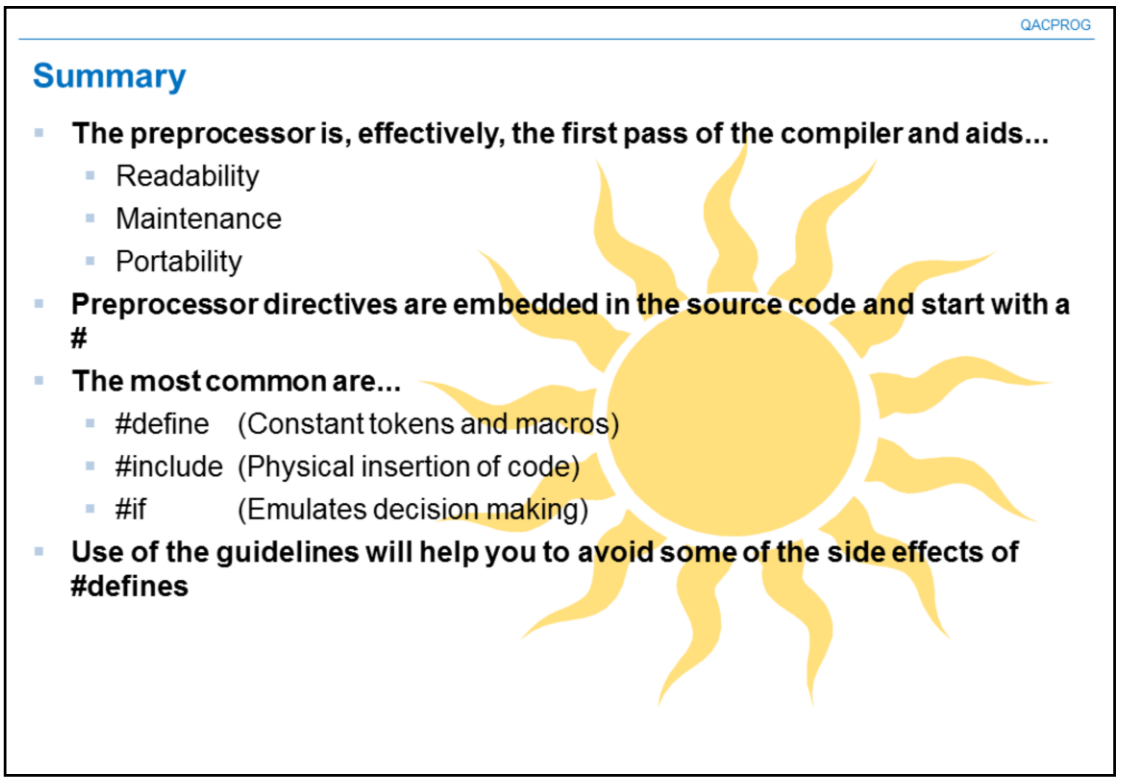


The example illustrates that there is a strict ordering of the preprocessor's tasks.

Trigraphs are replaced (see below).
CR characters are replaced by CR LF characters, if required.
The \ character will splice adjoining lines (as in multiline `#defines`).
Comments are replaced by a single space character.
directives are obeyed.
Macros are expanded.
Escape sequences in character and string constants are replaced.
Adjacent constant strings are concatenated (see advanced example).

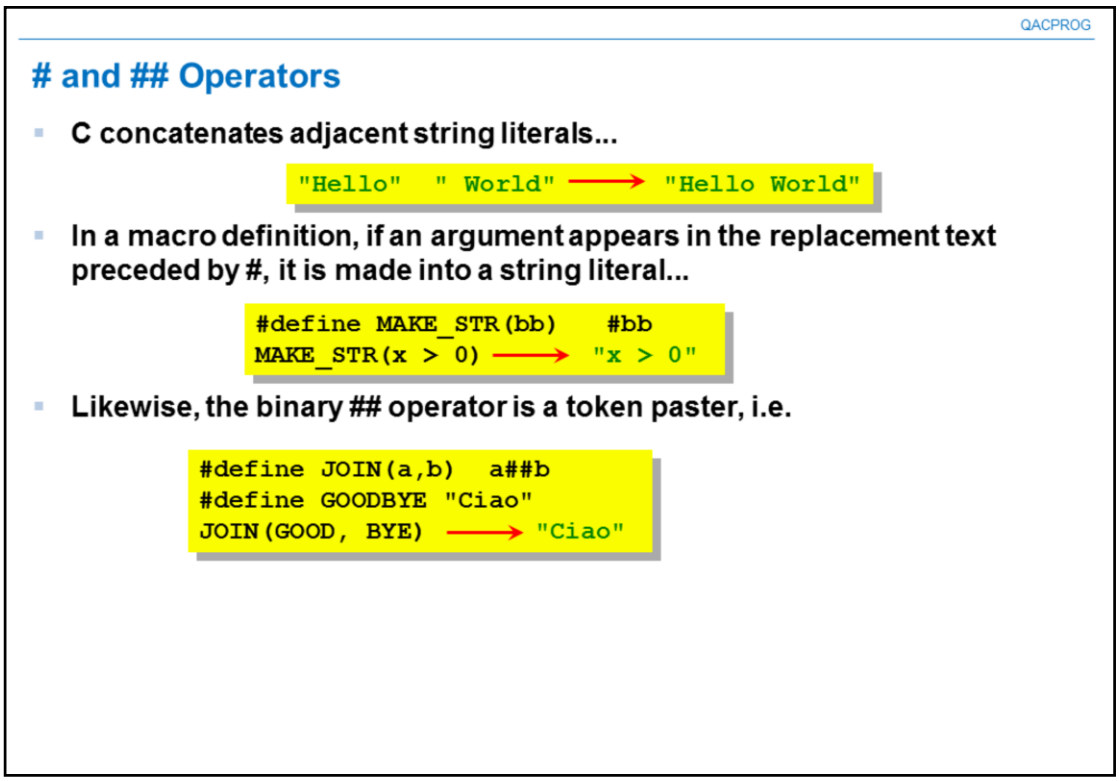
A trigraph is a sequence of three characters. They were new with the ISO standard. They provide a uniform way of generating certain essential characters that cannot be generated easily from a non-standard keyboard. They are as follows:

TRIGRAPH	C character	TRIGRAPH	C character
??=	#	??!	|
??/	\	??<	{
??'	^	??>	}
??(	[	??-	~
??)	]		



The preprocessor is a very powerful facility. It has been improved for the ISO C standard in order to ease debugging and portability. Everybody uses `#includes` and `#defines`, but few developers get beyond this. Try to develop a style that uses the techniques covered in the chapter.

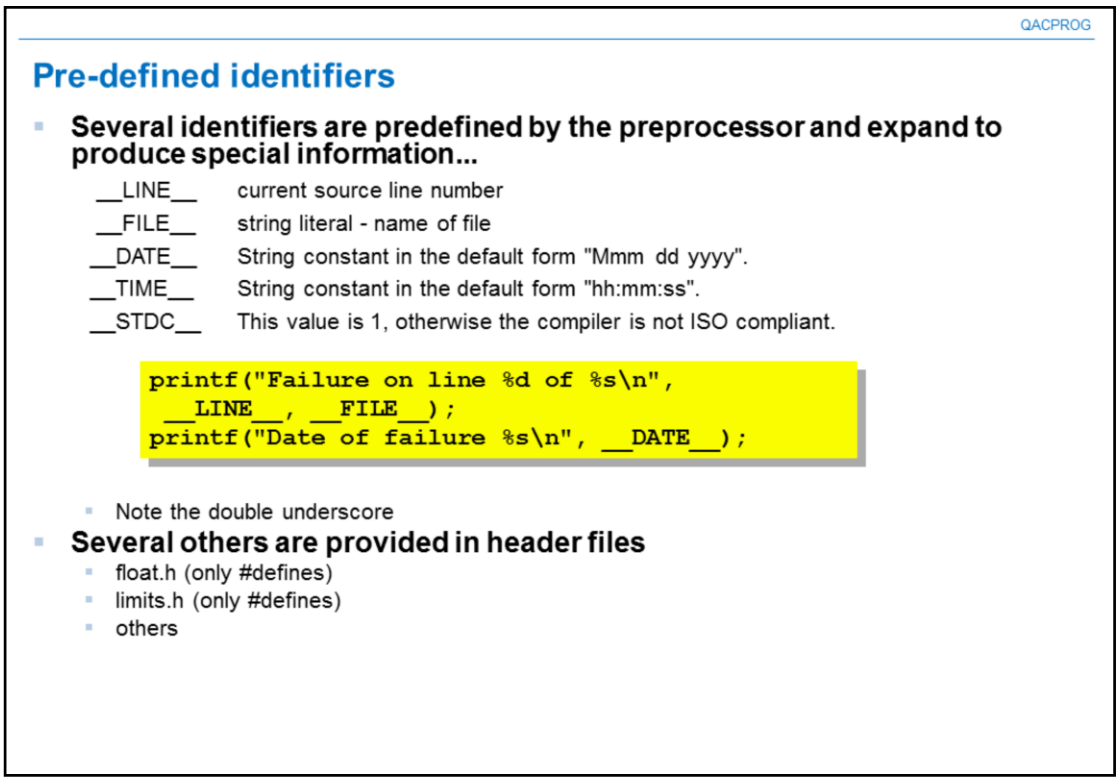
The preprocessor is not perfect, especially in the implementation of macros. Precedence and the absence of both type checking and placing breakpoints can present themselves as major problems. To a certain extent, we can overcome precedence problems, but there are occasions when we are totally in the hands of the code using the macro.



The preprocessor splices together adjacent string constants. `"Hello" " World"` will be joined to become a single literal `"Hello World"` (note that the space is a character from the second string). The compiler will implement this as a read-only anonymous char array. This mechanism is useful when writing long string constants, usually in `printf` statements. String splicing was brought in by the ISO standard.

The `#` operator creates a string from its single argument. The argument must be a name substituted by the preprocessor, as in the example shown above. This, in combination with the string splicing and the identifiers `__LINE__` and `__FILE__` provides a good debugging tool (see next page).

Also useful is the token pasting binary operator `##`. This is used to build up a pre-processor token which is also defined in the parse.



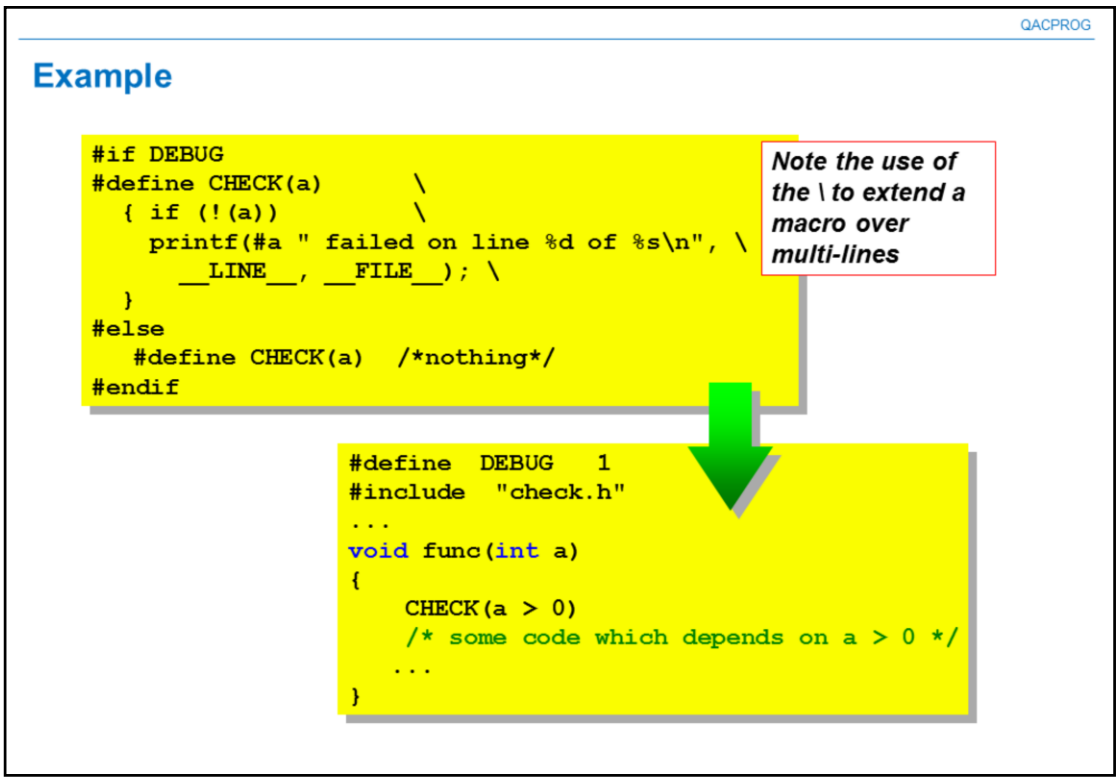
These five pre-defined pre-processor macros are standard. The first and last are `int`s and the others are `string`s.

Please note that these identifiers start and finish with *two* underscore characters, and cannot be undefined or re-defined.

The floats.h header supplies some magic `double` and `long double` numbers. These will be compiler | platform specific.

Similarly, limits.h provides numbers for the other primitives in the form of MIN and MAX numbers.

Others, like NULL for example, are defined in appropriate header files.



The `CHECK` macro is a gentler version of the library `assert` macro. `assert` can cause the program to abort, which is its main facility, and covered in the Large Programs chapter.

Macros are typically small and are delimited by the EOL characher (sequence). However, the \ can be used to extend the macro over multiple lines.

Intentionally left blank