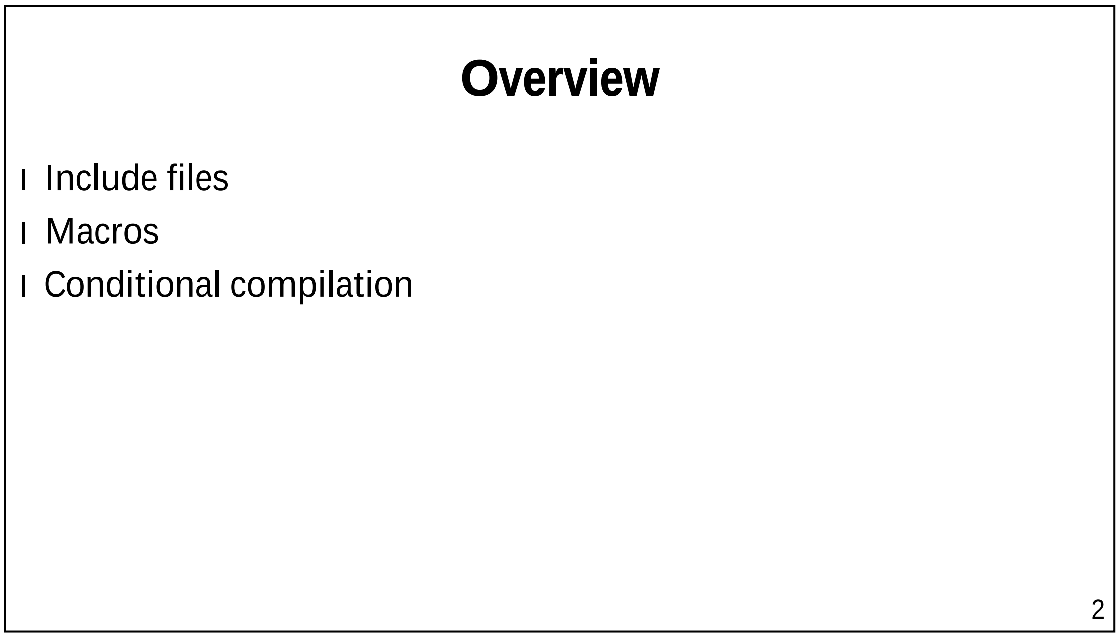
The Preprocessor

This material is intended for students of the "C++ Programming for Financial Engineering" certificate
A joint project by the Baruch MFE program, Datasim Education BV and Quant Network LLC
© Datasim Education BV 2011. Unauthorized distribution is prohibited.



Overview

- Include files
- Macros
- Conditional compilation

2

O b j e c t i v e s

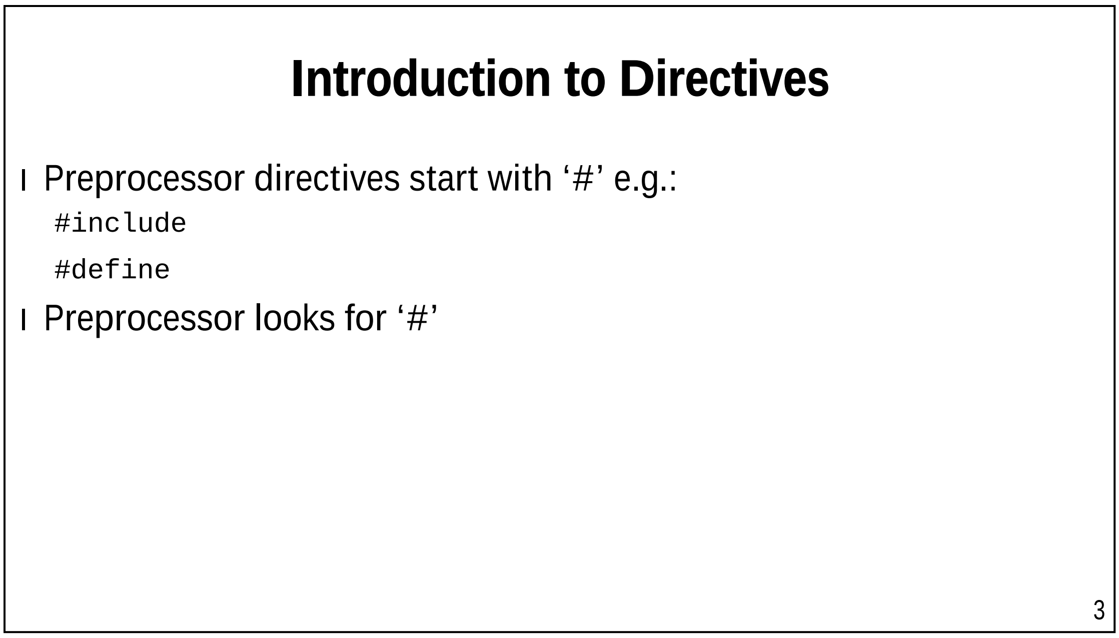
Topics discussed in this unit:

- Include files
- Macros
- Conditional compilation

Introduction

The C preprocessor is based on the ANSI C processor and is not considered to be part of the C compiler as such. Its functions are:
- Macro definition and substitution.
- Conditional compilation.
- Source file inclusion.

The preprocessor has other capabilities but these are not discussed here.



Introduction to Directives

- Preprocessor directives start with ‘#’ e.g.:

```
#include
#define
```

- Preprocessor looks for ‘#’

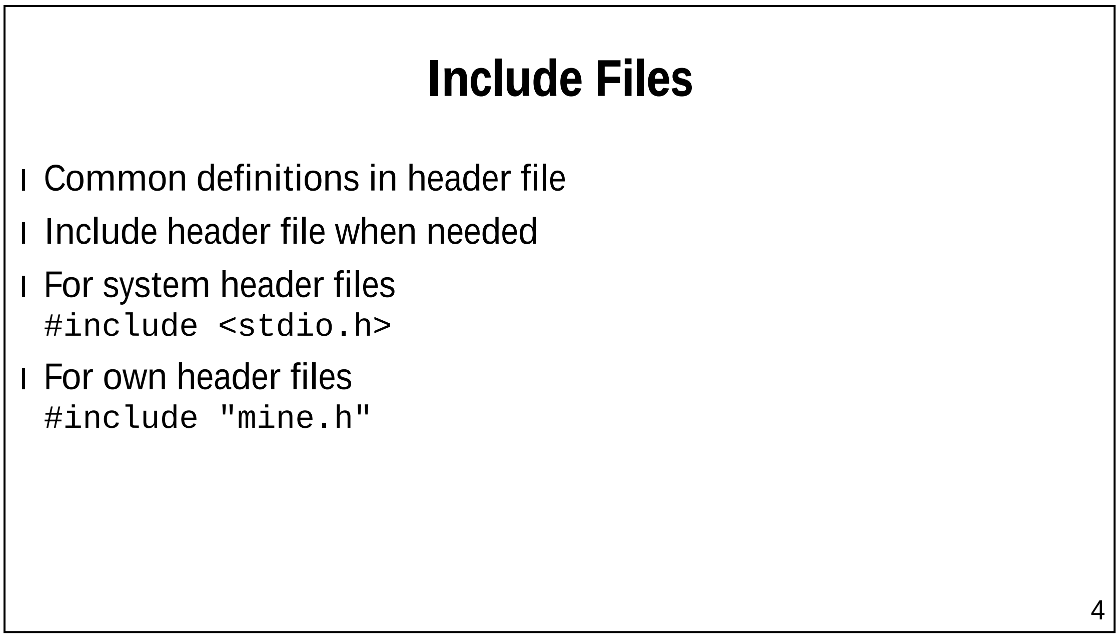
3

P r e p r o c e s s o r D i r e c t i v e s

The first question is: how do we give instructions to the preprocessor? This is achieved by the use of so-called directives. Directives can be recognised by the presence of the character '#' as the first character of a line in the source code. The syntax of the preprocessor is independent of the rest of the language.

We advise on using the preprocessor as little as possible. Much of the functionality can be achieved by C itself. In many cases however, the preprocessor must be used.

This material is intended for students of the “C++ Programming for Financial Engineering” certificate
A joint project by the Baruch MFE program, Datasim Education BV and Quant Network LLC
© Datasim Education BV 2011. Unauthorized distribution is prohibited.



Include Files

- Common definitions in header file
- Include header file when needed
- For system header files
  `#include <stdio.h>`
- For own header files
  `#include "mine.h"`

4

F i l e I n c l u s i o n

It is possible to let the preprocessor include certain files into the current file. This is done by the "include" directive. The syntax of the include directive is:

```
#include <filename.h>
```

The include directive causes the line beginning with '#' to be replaced by the entire contents of the file whose name is between the brackets <>.

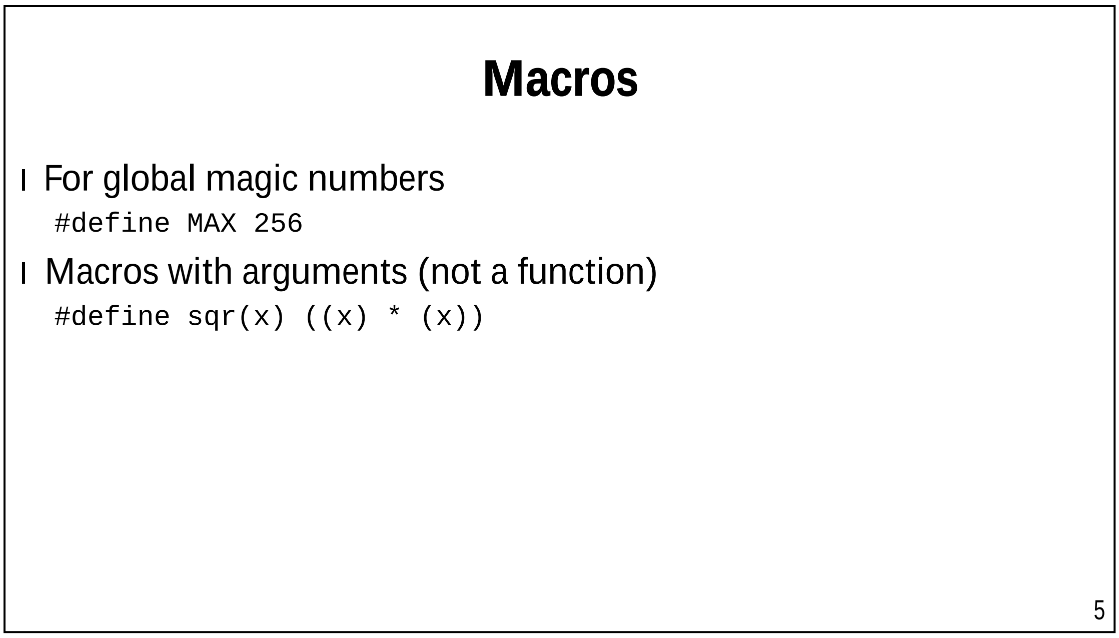
The policy is as follows:

The directive:
```
#include <filename>
```

causes the preprocessor to search for a file in a defined sequence of standard places (the include directory).

The directive:
```
#include "filename"
```

causes the file to be searched first in the directory of the original source file. The include directive can be used to handle compiler dependencies.



Macros

- For global magic numbers

```
#define MAX 256
```

- Macros with arguments (not a function)

```
#define sqr(x) ((x) * (x))
```

5

Macro Definitions and Substitutions

The preprocessor allows the programmer to define constants and macros. When the macro is used later in the program, it will be replaced by the macro definition by the preprocessor.

The syntax is:

#define <identifier> <token_string>

For example, we can define a constant as follows:

```
#define PI 3.14159
```

Anywhere in future code where PI is referred, it will be replaced by the value from the macro definition (in this case 3.14159).

It is also possible to define more complicated expressions for the preprocessor. In general, the syntax takes the form:

```
#define <ident>(<ident>,...,<ident>) <token_string>
```

For example, we could define a function which calculates the square of its argument. The syntax is:

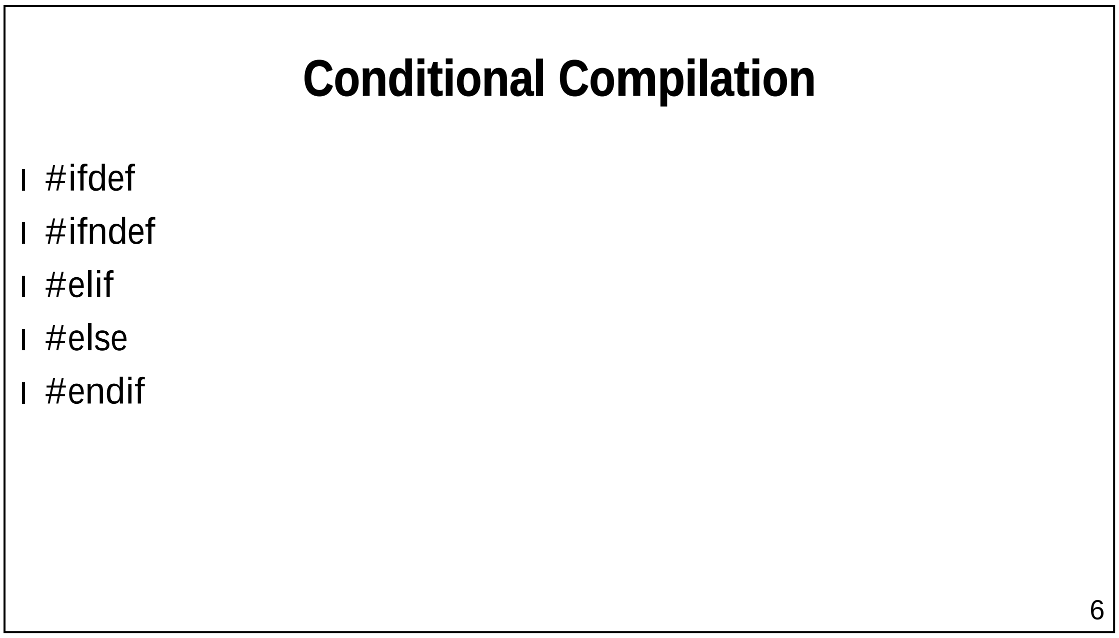
```
#define sqr(x) ((x)*(x))
```

The main disadvantages of using the #define directive are first that type checking is weak (what is there to stop us calculating sqr('a') ?) and secondly, it is less rigorous than the compiler.

An example of how problems can arise with the preprocessor is now given. It shows how useful the preprocessor is but it also shows a number of pitfalls.

This material is intended for students of the "C++ Programming for Financial Engineering" certificate
A joint project by the Baruch MFE program, Datasim Education BV and Quant Network LLC
© Datasim Education BV 2011. Unauthorized distribution is prohibited.

```
/*
   Program to show how the preprocessor works.
   (C) Datasim BV 1995
*/

#include <stdio.h>
#include <stdlib.h>

// Some preprocessor directives here
#define PI 3.1415
#define MAX 256
#define sqr(x) ((x) * (x))

main()
{
   int upperbound = MAX - 1;
   int two = 2;

   printf("The value of pi is: %f\n", PI);
   printf("Upperbound is: %d\n", upperbound);
   printf("The square of 2 is: %f\n", sqr(2.0));

   // Some erroneous ways of using the ‘sqr’ macro (for example,
   // squaring a character produces a result!)
   printf("Square of 3? %d\n", sqr(++two));
   printf("Square of character a is :%d\n", sqr('a'));

   exit(0);
}
```



Conditional Compilation

- #ifdef
- #ifndef
- #elif
- #else
- #endif

6

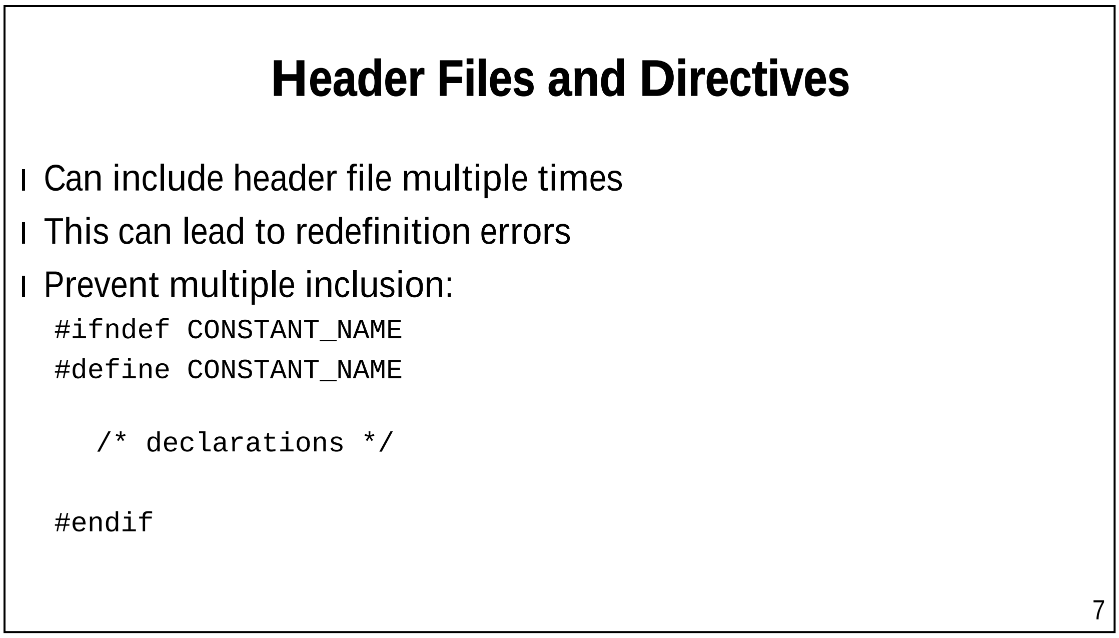
C o n d i t i o n a l   C o m p i l a t i o n

When working with large C systems it is necessary to compile functions. However, sometimes compilation should occur only under certain conditions. For example, we can take differences between various compilers and platforms into account and whether a certain file has already been included in a certain source file.

An example on how conditional compilation can be implemented is shown below.
For example, different compilers use different names for certain commonly used files. In this case we can define so-called environment variables to let us choose the right file depending on the current compiler.

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

/* Using the preprocessor to handle different compilers */
#ifdef __XWINDOWS__          /* X windows io*/
   #include <xwinio.h>
#elif __WINDOWS              /* Windows io */
    #include <winio.h>
#else                        /* DOS io */
    #include <stdio.h>
#endif
```

This material is intended for students of the "C++ Programming for Financial Engineering" certificate
A joint project by the Baruch MFE program, Datasim Education BV and Quant Network LLC
© Datasim Education BV 2011. Unauthorized distribution is prohibited.



Header Files and Directives

- Can include header file multiple times
- This can lead to redefinition errors
- Prevent multiple inclusion:

```
#ifndef CONSTANT_NAME
#define CONSTANT_NAME

   /* declarations */

#endif
```

7

Header Files and Directives

When creating your own header files we have to make sure that that our definitions are declared only once.

There is a possibility that a header file gets included more than once during one compilation. The definitions in that header file will be defined more than once and generate a compiler error.

```
#include "mine.h"
#include "mine.h"       /* Error multiple declarations */
```

To prevent this from happening we can use conditional compilation to make sure the definitions in that header file are declared once each compile operation. We define a constant for each header file. Before giving our definitions in the header file we check to see if that constant is defined. If the constant is not defined we define it along with the rest of the definitions. We do this by using the `#ifndef` and the `#endif` directive. Everything between these two directives is part of the *if* construction.

```
#ifndef MINE_H
#define MINE_H

void print(void);
/* Other definitions */

#endif
```

The first time the header file is included the global constant MINE_H is not defined and the *#ifndef* directive evaluates to true. The *#define* directive defines the global constant MINE_H. Following this definition we place the definitions we would have placed in the header file followed by a *#endif*. The second time the header file is included in one compilation the MINE_H constant is already defined and the definitions between the *#ifndef* and *#endif* are skipped.