Windows Programming

Lecture 05

Preprocessor

Preprocessor Directives

Preprocessor directives are instructions for compiler.

Programme compilation process

Preprocesor Directives

In C Preprocessor commands are called directives.

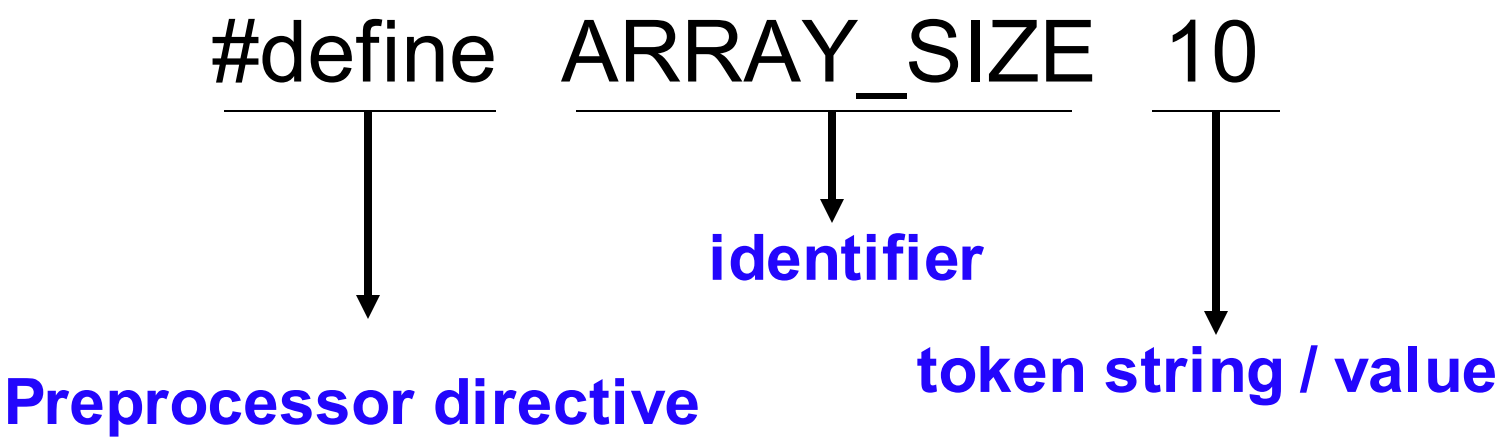
All preprocessor directives start with #.

Examples of Preprocessor Directives

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

#define directive

`#define` directive defines an identifier.



```
#define ARRAY_SIZE 10
```

Before Preprocessing

```
char array[ARRAY_SIZE];
```

After Preprocessing

```
char array[10];
```

identifier is not replaced if it appears

- in a comment
- within a string, or
- as part of a longer identifier

`#define` directive

If token-string is omitted, identifier is removed from source file.

```
#define ARRAY_SIZE
```

`#define` directive

```
#define WIDTH 10
#define HEIGHT WIDTH+25
```

`#define` directive

identifier remains defined and can be tested

- It can be checked using **`#if`** and **`#ifdef`** directives.
- **`#if defined`** is same as **`#ifdef`**.

Code Example

```
#include<iostream.h>
#include<conio.h>
#define   ARRAY_SIZE   10     // preprocessor is defined

main()
{
 #ifdef ARRAY_SIZE                // Testing preprocessor is defined or  not
     cout<<ARRAY_SIZE;
 #endif
 getch();
}
```

#if, #elif, #else, and #endif

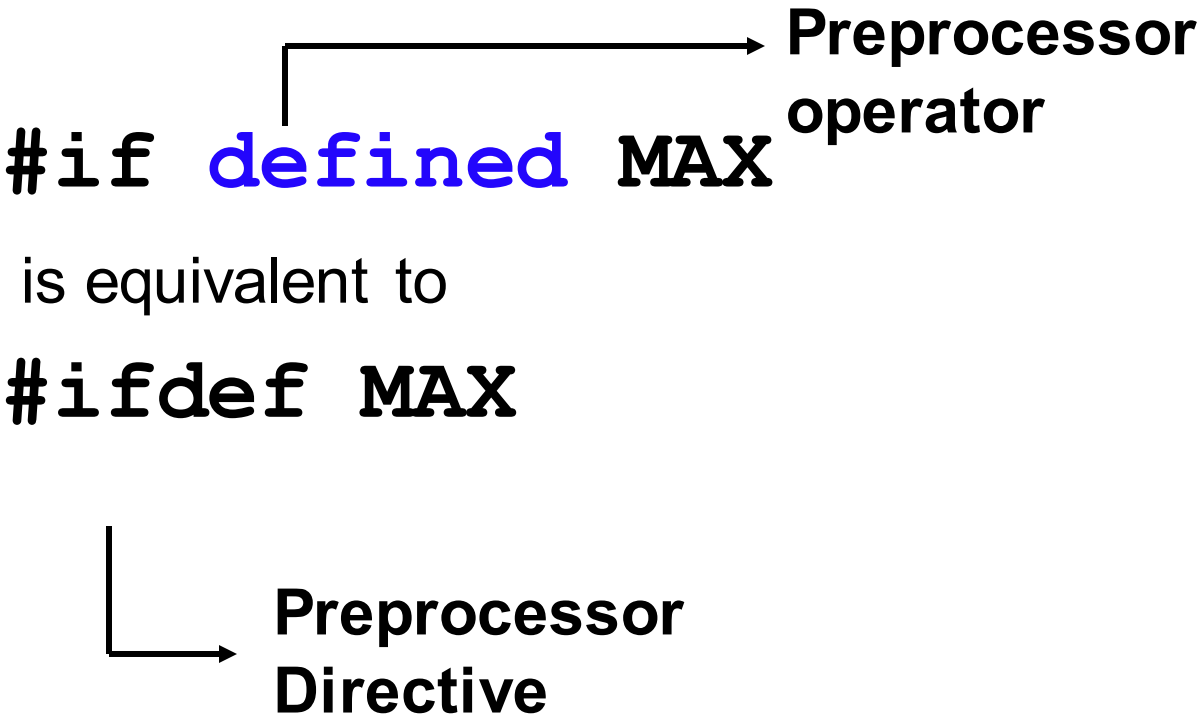
```
#if VERSION > 5
    #define SHIPMENT  1
    #if MORE_MEM == 1
        #define SIZE   200
    #else
        #define SIZE   100
    #endif


 #else

    #define SHIPMENT 0
    #if MORE_MEM == 1
        #define SIZE 100
    #else
        #define SIZE 50
    #endif
#endif
```

`#if, #elif, #else`

`#elif` is same as `#else  #if`

Preprocessor Operators

The **defined** is a preprocessor operator.



Preprocessor Operators

```
#if !defined MAX
```

is equivalent to

```
#ifndef MAX
```

`#error` directive

- The directive **`#error`** causes the preprocessor to report a fatal error. The rest of the line that follows **`#error`** is used as the error message. The line must consist of complete tokens.

- You would use **`#error`** inside of a conditional statment that detects a combination of parameters which you know the program does not properly support.

`#error` directive

```
#if !defined(__cplusplus)
  #error Must use C++ language
#endif
```

`#undef` Directive

If an *identifier* ceases to be useful, it may be *undefined* with the #undef directive. #undef takes a single argument, the name of the *identifier* to undefine. It is an error if anything appears on the line after the *identifier* name. #undef has no effect if the name is not an *identifier*.

```
#define FOO 4
 x = FOO;  ==> x = 4;
#undef FOO
 x = FOO;   ==> x = FOO;
```

`#undef` Directive

Once an ***identifier*** has been undefined, that identifier may be *redefined* by a subsequent **#define** directive. The new definition need not have any resemblance to the old definition.

`#undef` Directive

```
#define SPEED
…… some code here……
… … … … … … …
#undef SPEED
ifndef(SPEED)
  #error Speed not defined
#endif
```

Null Directive

The **null directive** consists of a **#** followed by a Newline, with only white space (including comments) in between. A null directive is understood as a preprocessing directive but has no effect on the preprocessor output. The primary significance of the existence of the null directive is that an input line consisting of just a **#** will produce no output, rather than a line of output containing just a **#**. Supposedly some old C programs contain such lines.

Null Directive

```
#          empty line
```

Macros

What is a macro ?

A simple macro always stands for exactly the same text, each time it is used. Macros can be more flexible when they accept arguments. Arguments are fragments of code that you supply each time the macro is used. These fragments are included in the expansion of the macro according to the directions in the macro definition.

Macros

To define a macro that takes arguments, you use the **#define** command with a list of parameters in parentheses after the name of the macro. The parameters may be any valid C identifiers separated by commas at the top level (that is, commas that aren't within parentheses) and, optionally, by white-space characters. The left parenthesis must follow the macro name immediately, with no space in between.

For example, here's a macro that computes the minimum of two numeric values:

```
#define min(X, Y) ((X)<(Y) ? (X):(Y))
```

Macros

- **#define SUM(a, b) a+b**
  int total;
  total = 3*SUM(10, 20)
  total = 3*10+20
  #define SUM(a, b) (a+b)

- **#define rect_area(a, b) a*b**
  rect_area(10, 20)
  rect_area(5+5, 10+10) → 5+5*10+10
  #define rect_area(a, b) (a)*(b)

- **#define rect_area(a, b) ((a)*(b))**

Standard Predefined Macros

The standard predefined macros are specified by the C language standards, so they are available with all compilers that implement those standards. Older compilers may not provide all of them. Their names all start with double underscores.

Standard Predefined Macros

__FILE__

This macro expands to the name of the current input file, in the form of a C string constant. This is the path by which the preprocessor opened the file, not the short name specified in #include or as the input file name argument. For example, "/usr/local/include/myheader.h" is a possible expansion of this macro.

__LINE__

This macro expands to the current input line number, in the form of a decimal integer constant. While we call it a predefined macro, it's a pretty strange macro, since its "definition" changes with each new line of source code.

Standard Predefined Macros

- **__FILE__**
  name of the current source file. String surrounded by quoutes

```
char ch[]= __FILE__ ;
char ch[]= "Polygons.c";
```

- **__LINE__**
  current line number in the source file, decimal integer constant.

```
int a=__LINE__;
int a=582;
```

Conditional Compilation

Conditional compilation is useful for things like machine-dependencies, debugging, and for setting certain options at compile-time.

Conditional Compilation

```
#if defined TESTING
    if(i > 0)
    {
        displayMessages();
    }
#else
    writeMessageToFile();
#endif
```

#include

Both user and system header files are included using the preprocessing directive #include. It has two variants:

- **`#include <`*`file`*`>`**

  This variant is used for system header files. It searches for a file named *file* in a list of directories specified by you, then in a standard list of system directories. The parsing of this form of #include is slightly special because comments are not recognized within the <...>. Thus, in #include <x/*y> the /* does not start a comment and the directive specifies inclusion of a system header file named x/*y. The argument *file* may not contain a > character. It may, however, contain a < character.

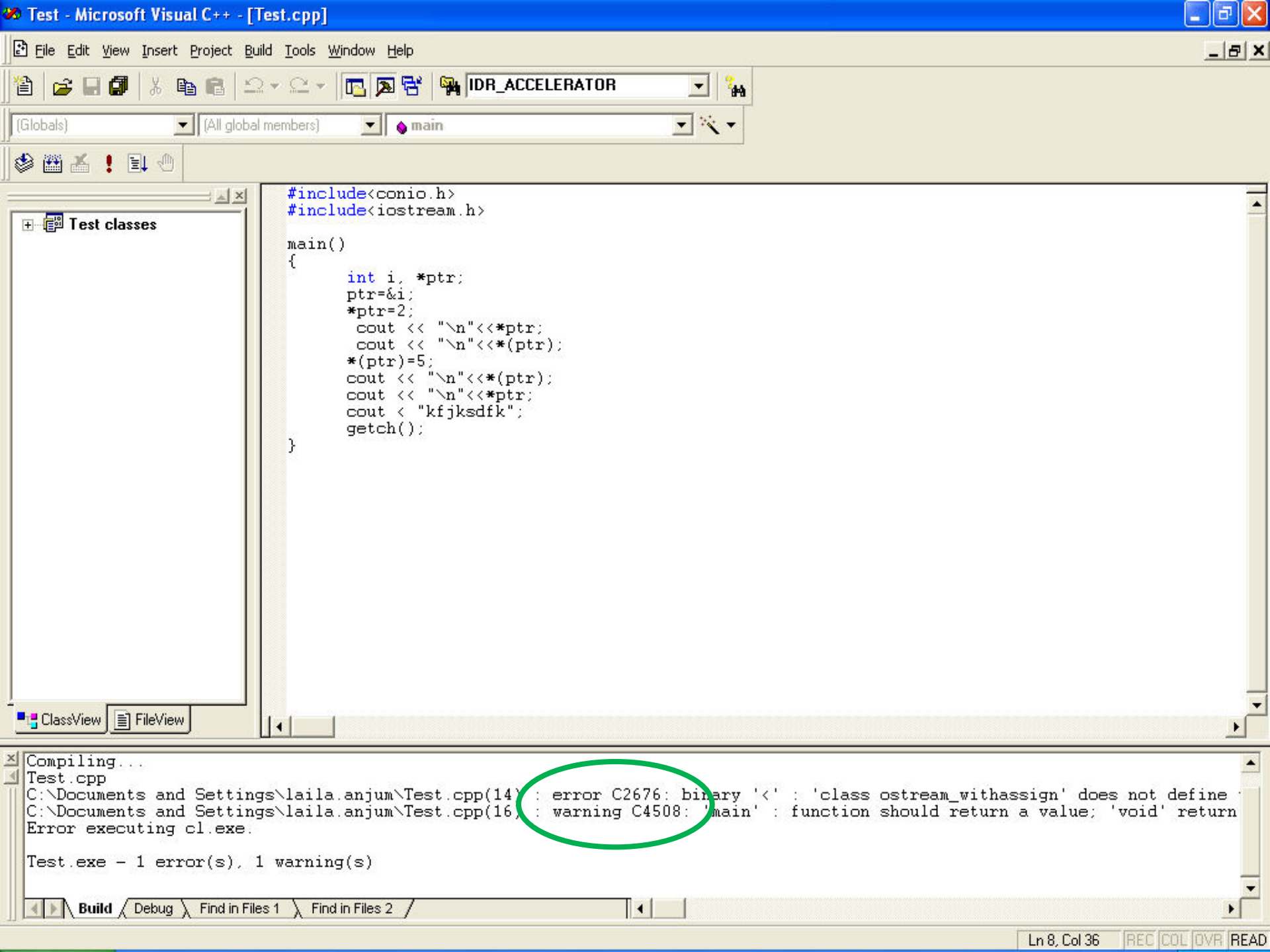
#include

- **`#include "file"`**
  - This variant is used for header files of your own program. It searches for a file named *file* first in the current directory, then in the same directories used for system header files. The current directory is the directory of the current input file. It is tried first because it is presumed to be the location of the files that the current input file refers to. If backslashes occur within *file*, they are considered ordinary text characters, not escape characters. None of the character escape sequences appropriate to string constants in C are processed. Thus, #include "x\n\\y" specifies a filename containing three backslashes. It is not clear why this behavior is ever useful, but the ANSI standard specifies it.

`#pragma` Directive

This **#pragma** directive allows a directive to be defined. Its effects are implementation-defined. If the pragma is not supported, then it is ignored.

The **#pragma** directive is the method specified by the C standard for providing additional information to the compiler, beyond what is conveyed in the language itself.


Test - Microsoft Visual C++ - [Test.cpp]
File Edit View Insert Project Build Tools Window Help
IDR_ACCELERATOR
(Globals)
(All global members)
main
Test classes
ClassView
FileView
#include<conio.h>
#include<iostream.h>
main()
{
int i, *ptr;
ptr=&i;
*ptr=2;
cout << "\n"<<*ptr;
cout << "\n"<<*(ptr);
*(ptr)=5;
cout << "\n"<<*(ptr);
cout << "\n"<<*ptr;
cout < "kfjksdfk";
getch();
}
Compiling...
Test.cpp
C:\Documents and Settings\laila.anjum\Test.cpp(14) : error C2676: binary '<' : 'class ostream_withassign' does not define
C:\Documents and Settings\laila.anjum\Test.cpp(16) : warning C4508: 'main' : function should return a value; 'void' return
Error executing cl.exe.
Test.exe - 1 error(s), 1 warning(s)
Build
Debug
Find in Files 1
Find in Files 2
Ln 8, Col 36
REC COL OVR READ

`#pragma` Directive

It has the general form:

```
#pragma charatcetr_sequence
```

- **where *character_sequence* is a series of characters giving a specific compiler instruction and arguments, if any.**

- **The *character_sequence* on a pragma is not subject to macro substitutions. More than one pragma construct can be specified on a single #pragma directive. The compiler ignores unrecognized pragmas.**

`#pragma` Directive

```
#pragma warning(disable:4001)
```

Disbale warning number 4001

`#pragma` Directive

```
#pragma warning( once:4385 )
```

Issue warning 4385 only once

```
#pragma warning( error:164 )
```

Report warning 164 as an error