**Preprocessor directives**

Preprocessor directives are lines included in the code of our programs that are not program statements but directives for the preprocessor. These lines are always preceded by a hash sign (#). The preprocessor is executed before the actual compilation of code begins, therefore the preprocessor digests all these directives before any code is generated by the statements.  
  
These preprocessor directives extend only across a single line of code. As soon as a newline character is found, the preprocessor directive is considered to end. No semicolon (;) is expected at the end of a preprocessor directive. The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a backslash (\).

* Macro definitions(#define)
* Conditional inclusions(#ifdef,#ifndef,#if,#endif,#else & #elif)
* Line control(#line)
* Error directive(#error)
* Source file inclusion(#include)
* Pragma directive(#pragma)

**Macros**

**Simple macro**: #define BUFFER\_SIZE 1020

**Macros with Arguments**: #define min(X, Y) ((X) < (Y) ? (X) : (Y))

**Standard Predefined macros :**

Their names all start and end with double underscores. Those preceding \_\_GNUC\_\_ in this table are standardized by ANSI C; the rest are GNU C extensions.

Some important are being specified below

\_\_FILE\_\_

This macro expands to the name of the current input file, in the form of a C string constant

\_\_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. This and `\_\_FILE\_\_' are useful in generating an error message to report an inconsistency detected by the program; the message can state the source line at which the inconsistency was detected. For example,

\_\_FUNCTION\_\_

It shows the current function name.

Ex - DEBUG("%s %s %d \n",\_\_FILE\_\_,\_\_FUNCTION\_\_, \_\_LINE\_\_);

\_\_DATE\_\_

This macro expands to a string constant that describes the date on which the preprocessor is being run. The string constant contains eleven characters and looks like `"Feb 1 1996"'.

\_\_TIME\_\_

This macro expands to a string constant that describes the time at which the preprocessor is being run. The string constant contains eight characters and looks like `"23:59:01"'.

\_\_VERSION\_\_

This macro expands to a string constant which describes the version number of GNU C. The string is normally a sequence of decimal numbers separated by periods, such as `"2.6.0"'.

**Self Referential macro**

A *self-referential* macro is one whose name appears in its definition. A special feature of ANSI Standard C is that the self-reference is not considered a macro call. It is passed into the preprocessor output unchanged.

Let's consider an example:

|  |  |
| --- | --- |
|  | #define foo (4 + foo) |
|  |  |

**New line in macro**

Often it is desirable to define a macro that expands into a compound statement. Consider, for example, the following macro, that advances a pointer (the argument `p' says where to find it) across whitespace characters:

|  |  |
| --- | --- |
|  | #define SKIP\_SPACES(p, limit) \  { register char \*lim = (limit); \  while (p != lim) { \  if (\*p++ != ' ') { \  p--; break; }}} |

**Token pasting operator**

#define f(g,g2) g##g2

main()

{

int var12=100;

printf("%d",f(var,12));

}

O/P

100

**Stringification**

Stringification means turning a code fragment into a string constant whose contents are the text for the code fragment. For example, stringifying `foo (z)' results in `"foo (z)"'

*Give an example*

**Concatenation**

#define DEBUG\_VALUE(v) printf(#v is equal to %d\n”,v)

main()

{

int x = 20;

DEBUG\_VALUE(x);

}

**For undefining a macro**

#undef

#define TABLE\_SIZE 100

int table1[TABLE\_SIZE];

#undef TABLE\_SIZE

**Questions on Macro**

1.To determine cube,macro is

# define CUBE (x) (x\*x\*x)

In program, volume=CUBE(side); which will expand to volume =(side\*side\*side)

2.To determine square,macro is

# define SQUARE(x) ((x)\*(x))

3.Size of array

#define G\_N\_ELEMENTS(arr) ((sizeof(arr))/(sizeof(arr[0])))

main()

{

int arr[] = {1, 2, 3, 4};

printf ("sizeof arr: %d\n", G\_N\_ELEMENTS(arr));

}

4.Greatest of two/three numbers

#define Greatest(a,b) (a>b)?a:b //2 nos

#define max2(x,y) ((x)>(y)?(x):(y)) //3 nos

#define max4(a,b,c,d) max2(max2((a),(b)),max2((c),(d)))

4. What is the difference between **macro and typedef**

*What is typedef*

A **typedef** declaration lets you define your own identifiers that can be used in place of type specifiers such as **int**, **float**, and **double**. A **typedef** declaration does not reserve storage. The names you define using **typedef** are not new data types, but synonyms for the data types or combinations of data types they represent. The name space for a **typedef** name is the same as other identifiers. The exception to this rule is if the **typedef** name specifies a variably modified type. In this case, it has block scope. When an object is defined using a **typedef** identifier, the properties of the defined object are exactly the same as if the object were defined by explicitly listing the data type associated with the identifier.

Examples

1. typedef unsigned int size\_t;
2. struct my\_struct\_type my\_struct\_variable;

typedef struct my\_struct\_type my\_short\_type\_t;

1. typedef int LENGTH;

LENGTH length, width, height;

1. typedef struct {

int scruples;

int drams;

int grains;

} WEIGHT;

The structure WEIGHT can then be used in the following declarations:

WEIGHT chicken, cow, horse, whale;

1. This is the first question raised by Core Issue 479:

typedef void fv(void); // Legal.

struct S1 {

fv mfuncv; // Legal?

static fv sfuncv; // Legal?

};

Yes to both. mfuncv is a non-static member function, and sfuncv is a static member function. All

compilers surveyed permit this, and a number of committee members have voiced an oppinion that this is

intended to work.Clearly, sfuncv should work, because the only difference between a static member function and a nonmemberfunction is scoping. mfuncv should work because it is common practice.One problem that is encountered by allowing this is the analogous situation with templates.

**Difference between typedef & macro**

* A Macro is a preprocessor directive means that before compilation the macros are replaced. Whereas typedef is defining a new data type which is same as the existing data type.
* typedefs can correctly encode pointer types.where as #DEFINES are just replacements done by the preprocessor. For example,
* typedef char \*String\_t;
* #define String\_d char \*
* String\_t s1, s2; String\_d s3, s4;

s1, s2, and s3 are all declared as char \*, but s4 is declared as a char, which is probably not the intention.

**Conditional inclusions**

#ifdef TABLE\_SIZE #ifndef TABLE\_SIZE

int table[TABLE\_SIZE]; #define TABLE\_SIZE 100

#endif int table[TABLE\_SIZE];

#endif

**Line Control**

When we compile a program and some error happen during the compiling process, the compiler shows an error message with references to the name of the file where the error happened and a line number, so it is easier to find the code generating the error. The #line directive allows us to control both things, the line numbers within the code files as well as the file name that we want that appears when an error takes place. Its format is:   
  
#line number "filename"  
  
Where number is the new line number that will be assigned to the next code line. The line numbers of successive lines will be increased one by one from this point on. **Explain with e.g**

**Error Control**

#ifndef \_\_cplusplus

#error A C++ compiler is required!

#endif

**Source file inclusion**

#include "file"

#include <file>

The only difference between both expressions is the places (directories) where the compiler is going to look for the file. In the first case where the file name is specified between double-quotes, the file is searched first in the same directory that includes the file containing the directive. In case that it is not there, the compiler searches the file in the default directories where it is configured to look for the standard header files.  
If the file name is enclosed between angle-brackets <> the file is searched directly where the compiler is configured to look for the standard header files. Therefore, standard header files are usually included in angle-brackets, while other specific header files are included using quotes.