**Pre-processor Directives**

Pre-processor is the first phase of the translation process. A program in C is pre-processed before compilation. We can check the output of the pre-processor by using the option -E of gcc.

Pre-processor does not know ‘C’.

Pre-processor provides the following features.

a) inclusion of files

#include <filepath>

or

#include “filepath”

b) define pre-processor variables

#define ABCD

#undef ABCD

c) check whether a preprocessor variable is defined or not

#ifdef ABCD

#ifndef ABCD

#endif

This feature is used to selectively include portion of the file

d) define a constant

#define X 10

This will cause textual substitution wherever X is used

e) define a macro with parameters

#define fn(a, b) a + b

All these directives can be placed anywhere in a file.

Let us look at macros in detail.

#define sq(x) x \* x

The above statement defines a macro called sq.

x is called the parameter.

x \* x is the definition. The definition has to be on a single line or we should escape the newline to cheat the preprocessor.

In the code, we may invoke macro as sq(10).

sq(10) is called a macro call. 10 is called the argument. The argument matches the parameter textually. The definition will replace the call, the parameters are replaced by the corresponding argument textually.

sq(10) becomes 10 \* 10.

There are two advantages of macro.

1. The expansion produces inline code. This avoids the overhead of the function call. The program tends to become faster.

2. The macro processing is pure text processing. These is no concept of type.

The macros can receive even a typename as argument. So, macros can be generic.

Example for swapping:

#define myswap(t, x , y) { \

t temp = x; x = y; y = temp; \

}

calls:

int a = 10; int b = 20;

myswap(int, a, b); // swap int

double c = 1.2; double d = 3.4;

myswap(double, c, d); // swap double

There are two disadvantages of macros

1. Each call is expanded. So the code could bloat.

2. As the matching of argument to parameter is text based, the macros may give unusual results – may violate the law of least expectation!

sq(5 + 5) would result in the expansion 5 + 5 \* 5 + 5

sq(a++) would result in the expansion a++ \* a++ thus incrementing a twice and value of the expression undefined.

Example: 1\_ex.c

// macro:

// are not part of the language 'C'

// supported by preprocessor

// look like functions but are not same

// expands the code at the point of call

// replaces the macro call by macro definition

// replaces the parameter by the corresponding argument textually

// no evaluation of arguments

// arguments are treated as strings

// syntactically, macro definition has to be on a single line

// If we want a macro spanning multiple lines, we should escape the newline

// Use of a block { } will not work as macro processor does not know 'C'!

// advantages:

// a) produces inline code - code at the point of call. Macro call does not have the overhead

// of a function call. As the code will become contiguous after expansion, accessing the code

// will be faster compared to the case of a function call.

// b) macros have no types - macros are generic. We might be able to use macros on different

// types. We can also pass a typename (or any string for that matter) as argument to

// macros.

// This macro can swap variables of any type which supports assignment and initialization

#define myswap(t, x , y) { \

t temp = x; x = y; y = temp; \

}

// disadvantages:

// a) may result in bloating of code

// Each macro call is expanded. If a function is called 1000 times, the code of the function shall

// exist just once. If a macro is called 1000 times, the code shall be expanded 1000 times,

// thus increasing the code size.

// b) macros tend to violate the law of least expectation

// Try in this example

// sq(5 + 5)

// 10000 / sq(10)

// int a = 10; int res = sq(a++); // What is the value of res? what is the value of a?

#include <stdio.h>

#define sq(x) x \* x

int main()

{

printf("square : %d\n", sq(10)); // 10 matches x; expands to 10 \* 10

// check using -E option

// printf("square : %d\n", 10 \* 10);

int a = 10; int b = 20;

myswap(int, a, b);

printf("a : %d b : %d\n", a, b);

double c = 1.2; double d = 3.4;

myswap(double, c, d);

printf("c : %lf d : %lf\n", c, d);

}