EDAYATHANGUDY G.S PILLAY ARTS &SCIENCE COLLEGE

Affiliated To Bharathidasan University,Tiruchirapalli.

(NAAC Accredited with 'A' Grade) ,NAGAPATTINAM-611002

B.Sc-INFORMATION TECHNOLOGY

Subject code:16SCCIT2 PROGRAMMING IN C UNIT-V

**PREPROCESSOR**

The C Preprocessor is not part of the compiler, but is a separate step in the compilation process.In simplistic terms, a C Preprocessor is just a text substitution tool and they instruct compiler to do required pre-processing before actual compilation. We'll refer to the C Preprocessor as the CPP.All preprocessor commands begin with a pound symbol # . It must be the first nonblank character,and for readability, a preprocessor directive should begin in first column.

**Following section lists**:

Directive Description

#define - Substitutes a preprocessor macro

#include -Inserts a particular header from another file

#undef -Undefines a preprocessor macro

#ifdef -Returns true if this macro is defined

#ifndef - Returns true if this macro is not defined

#if -Tests if a compile time condition is true

#else -The alternative for #if

#elif #else an #if in one statement

#endif - Ends preprocessor conditional

#error -Prints error message on stderr

#pragma -Issues special commands to the compiler, using a standardized method

**Preprocessors Examples**

Analyze the following examples to understand various directives.

#define MAX\_ARRAY\_LENGTH 20

This directive tells the CPP to replace instances of MAX\_ARRAY\_LENGTH with 20. Use #define for constants to increase readability.

#include <stdio.h>

#include "myheader.h"

These directives tell the CPP to get stdio.h from System Libraries and add the text to the current source file. The next line tells CPP to get myheader.h from the local directory and add the content

to the current source file.

#undef FILE\_SIZE

#define FILE\_SIZE 42

This tells the CPP to undefine existing FILE\_SIZE and define it as 42.This tells the CPP to define MESSAGE only if MESSAGE isn't already defined.

#ifdef DEBUG

/\* Your debugging statements here \*/

#endif

This tells the CPP to do the process the statements enclosed if DEBUG is defined. This is useful if you pass the -DDEBUG flag to gcc compiler at the time of compilation. This will define DEBUG, so

you can turn debugging on and off on the fly during compilation.

**Predefined Macros**

ANSI C defines a number of macros. Although each one is available for your use in programming,

the predefined macros should not be directly modified.

Macro Description

\_\_DATE\_\_ The current date as a character literal in "MMM DD YYYY" format

\_\_TIME\_\_ The current time as a character literal in "HH:MM:SS" format

\_\_FILE\_\_ This contains the current filename as a string literal.

\_\_LINE\_\_ This contains the current line number as a decimal constant.

\_\_STDC\_\_ Defined as 1 when the compiler complies with the ANSI standard.

**example:**

**#include <stdio.h>**

main()

{

printf("File :%s\n", \_\_FILE\_\_ );

printf("Date :%s\n", \_\_DATE\_\_ );

printf("Time :%s\n", \_\_TIME\_\_ );

printf("Line :%d\n", \_\_LINE\_\_ );

printf("ANSI :%d\n", \_\_STDC\_\_ );

}

When the above code in a file test.c is compiled and executed, it produces the following result:

File :test.c

Date :Jun 2 2012

Time :03:36:24

Line :8

ANSI :1

**Preprocessor Operators**

The C preprocessor offers following operators to help you in creating macros:

Macro Continuation (\).A macro usually must be contained on a single line. The macro continuation operator is used to

continue a macro that is too long for a single line.

For example:

#define message\_for(a, b) \

printf(#a " and " #b ": We love you!\n")The stringize or number-sign operator '#' , when used within a macro definition, converts a macro parameter into a string constant. This operator may be used only in a macro that has a specified

argument or parameter list.

For example:

#include <stdio.h>

#define message\_for(a, b) \

printf(#a " and " #b ": We love you!\n")

int main(void)

{

message\_for(Carole, Debra);

return 0;

}

When the above code is compiled and executed, it produces the following result:

Carole and Debra: We love you!

**Token Pasting ##**

The token-pasting operator ## within a macro definition combines two arguments. It permits two

separate tokens in the macro definition to be joined into a single token. For example:

#include <stdio.h>

#define tokenpaster(n) printf ("token" #n " = %d", token##n)

int main(void)

{

int token34 = 40;

tokenpaster(34);

return 0;

}

When the above code is compiled and executed, it produces the following result:

token34 = 40

How it happened, because this example results in the following actual output from the

preprocessor:

printf ("token34 = %d", token34);

This example shows the concatenation of token##n into token34 and here we have used both

stringize and token-pasting.

**The defined Operator**

The preprocessor defined operator is used in constant expressions to determine if an identifier is

defined using #define. If the specified identifier is defined, the value is true non − zero. If the symbol

is not defined, the value is false zero. The defined operator is specified as follows:printf("Here is the message: %s\n", MESSAGE);

return 0;

}

When the above code is compiled and executed, it produces the following result:

Here is the message: You wish!

**Parameterized Macros**

One of the powerful functions of the CPP is the ability to simulate functions using parameterized

macros. For example, we might have some code to square a number as follows:

int square(int x) {

return x \* x;

}

We can rewrite above code using a macro as follows:

#define square(x) ((x) \* (x))

Macros with arguments must be defined using the #define directive before they can be used. The argument list is enclosed in parentheses and must immediately follow the macro name. Spaces are not allowed between and macro name and open parenthesis. For example:

#include <stdio.h>

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

int main(void)

{

printf("Max between 20 and 10 is %d\n", MAX(10, 20));

return 0;

}

When the above code is compiled and executed, it produces the following result

**Dynamic Memory Allocation**

• Dynamic memory allocation

– How to allocate memory for variables (esp. arrays/strings) during run time

– malloc(), calloc(), realloc(), and free()

Why dynamic memory allocation?

Usually, so far, the arrays and strings we’re using have

fixed length (i.e., length is known at compile time)

• Example:

char myStr[ ] 11 ; // allocates memory for 10 chars

printf(“Enter a string: “);

fgets(myStr, 11, stdin);

What if the user wants to enter a string more than 10 chars long or if the length is

malloc()

• malloc() is used to request additional memory from the operating system during program execution

Syntax: malloc(numBytes)

• Input is the number of consecutive bytes to be allocated

• Return value is a pointer to the beginning of the block of

memory allocated or NULL if malloc fails

• To use malloc(), you must #include <stdlib.h>

free()

• The function free() returns memory to the memory

pool. It “frees” up memory

Syntax: free(ptr)

– where ptr “points to” memory previously previously allocated allocated by

malloc() function

• To use free(), you must #include <stdlib.h>

strchr

if ((q = strchr(charP, '\n')) != NULL)

\*q = '\0';

This code finds the first occurance of the

character ‘\n’ which is the newline character that fgets will obtain. If found (value in q is not NULL), it sets that character to the string null termination.

**Memory leak**

If malloc’ed memory is not free’ed, then the OS will “leak memory”

– This means that memory is allocated to the program but not returned to the OS when it is finished using it

– The program program therefore therefore grows larger over time. malloc allocates bytes

• If you want a character array that stores 10

characters (including ‘\0’):

char \*p = malloc(10);

**Linked Lists**

• Linked list of items is arranged in order

• Size of linked list changes as items are inserted or removed

• Dynamic memory allocation is often used in linked list implementation

• Ten fundamental functions are used to

Fundamentals

• A linked list is a sequence of items arranged one after another.

• Each item in list is connected to the next item via a link

• Each item is placed together with the link to the next item, resulting in a simple component called a node.

12.1 14.6 14.6

End marker

**Declaring a Class for Node**

struct Node

{

typedef double Item;

Item data; // data stored in node

Node \*link; // pointer to next node

};

A struct is a special kind of class where all members are public. In this case there are two public member variables: data, link.Whenever a program needs to refer to the item type, we can use the expression Node::Item.Head Pointers, Tail Pointers Usually, programs do not actually declare node variables. Instead, the list is accessed through one or more pointers to nodes. 12.1 14.6 14.6

end marker head\_ptr tail\_ptr

Struct Node

{

typedef double Item;

Item data;

Node \*link;

};

Node \*head\_ptr;

Node \*tail\_ptr;

12.1 14.6 14.6

end

marker

**Null Pointer**

• The final node in the linked list does not point to a next node.

• If link does not point to a node, its value is set to NULL.

• NULL is a special C++ constant, from the standard library facility <stdlib.h>

• NULL pointer is often written 0 (zero). Empty List

• When the list is empty, both the head\_ptr and tail\_ptr are NULL.

• When creating a new linked list, it starts out empty (both tail and head pointers NULL).

• Any linked list functions you write should handle the case of empty list (head and tail pointers NULL).

**Two Common Pointer Bugs**

• Attempting to dereference a pointer via \*p or p-> when p=NULL.

• Attempting to dereference a pointer via \*p or p-> when p is not properly initialized.

• NOTE: this error does not cause a syntax error, but instead causes errors:

– Bus Error

– Segmentation violation

– Address protection violation

Computing the Length of a

Linked List

size\_t list\_length(Node \* head\_ptr)

{

Node \*cursor;

size\_t answer=0;

for(cursor=head\_ptr; cursor != NULL; cursor=cursor->link)

answer++;

return answer;

Computing the Length of a

Linked List

size\_t list\_length(Node \* head\_ptr)

{

Node \*cursor;

size\_t answer=0;

for(cursor=head\_ptr; cursor != NULL; cursor=cursor->link)

answer++;

return answer;

}

Traversing a Linked List

Common pattern in functions that need to traverse a linked list:

…

for(cursor=head\_ptr; cursor != NULL; cursor=cursor->link)

…