Optimization & debugging with GCC, it is possible to use optimization is combination with the debugging option '– g'. When using optimizations & debugging together, the internal rearrangements carried out by the compiler can make it difficult to see what is going on when examining an optimized program in the debugger. For example, temporary statements are often omitted and the ordering of statements may be changed. However, when the program crashes unexpectedly, the debugging information proves to be a better option. So, the use of '– g' option is recommended for optimized program. The debugging option by default is enabled for releases of GNO packages with the optimization option '– 02'.

3.4.4 Specifying Search Paths

When building a project, GCC has a default search path for files & libraries. We often need to add components to these paths for example for including a header file required by our program. We also need to ass an entry to directory search path so that linker can find the libraries required by our program. The option for including file search path is '– I' & for including libraries is '– L'. This is explained with the help of examples shown below :

– gcc – wall – I/ etc / input – 0 test test.c

The above command specifies the compiler to include file input under the path / etc. This will viable the preprocess on to find the input.h file that our program wants.

Similarly, if we want to include math library for our program, we need to inform the linker the location of this library. It can be dine by using :

gcc – L / usr / math / lib – 0 test test.c

3.5 MANAGING PROJECTS

When the size of programs increases especially in case of projects, the process necessary to build these projects becomes more complex & time consuming because large amount CPU time is consumed in running then and when errors occur the debug time also increases.

Thus, it becomes necessary to manage these projects. The Linux environment provides a tool to manage these projects called as GNU make. Consider the assembly line for production of cars. Everything must be built properly and all the parts must be linked together to form the final product. If anything goes wrong, the look of the car may be affected as well as the car may not work properly. In order to get the cars

working properly, both humans & computers control the process in which it is built. There is an element of central control that regulates the flow of materials from one place to another. fill the cars built of a particular model will have the same appearance & the same principles apply on all of them. Thus, the build process involves executing thousands of commands. If these commands were to be exacted manually then the process would become pug & contain dozens of errors & it will be extremely difficult to compile & link programs. Thus, there is an autonomous system for the assembly line which is the build process for code. We define these rules to define how the code is built and then the system applies these rules to build the entire project. In other words, we want the system to complete a certain task & instead of gaudy the system for each task we put the rules in a file. The system then completes the project by riding these rules from the file itself. The file where these rules are defined is called make file, comps with an interpreter called as GNU make which processes & builds the entire project. This program invokes compilers, linkers, assemblers & all other programs that are herded to build the final executable. When we type "make" on command line, the system aromatically examines the rules & files presort in the system & takes appropriate actions to build the entire projects. These actions involve running thousands of commands. & may even involve parallel processing. When changes are made to a few files in the project, we need not explicitly tell the GNU make of these changes. After the changes are made & then we run the "make" on command line then system automatically detects these changes & performs necessary actions needed to update the program with these changes & recompiles & relents the entire program.

3.6 USE OF MAKE FILES

A makefile is a collection of rules. Each rule in a makefile defines 3 things namely :

- Foust will be the file that will be built when the rule is processed.
- Second as the process we must go through to make files into a final product. For example, for generating a final executable in C language, first we must compile and then we need to link all these object files to generate the final executable.
- Third, we must define a list of dependencies for each file. These dependencies need to be calculated before we can process a file. For example, final executable file depends on in direct files which in here depend upon C files. Also id cases where we write a C program we use