


Copyright © 2011 IGATE Corporation (a part of Capgemini Group).
All rights reserved.

No part of this publication shall be reproduced in any way,
including but not limited to photocopy, photographic, magnetic,
or other record, without the prior written permission of IGATE
Corporation (a part of Capgemini Group).

IGATE Corporation (a part of Capgemini Group) considers
information included in this document to be confidential and
proprietary.

Document History				
Date	Course Version No.	Software Version No.	Developer / SME	Change Record Remarks
Oct -2009	1.0		Anitha Ramesh	
May-2011	2.0		Rathnajothei Perumalsamy	Revamp/Refinement


CONSULTING TECHNOLOGIES ENTREPRENEURS

January 29, 2016 | Proprietary and Confidential | < 3 >

Course Goals and Non Goals

➤ Course Goals

- To learn Program development in Unix environment
- Programming in C/C++
- Debugging C/C++ Program
- Make file build utility

➤ Course Non Goals

- Does not cover Linux Internal Programming




Pre-requisites

- C/C++ Programming Knowledge

January 29, 2016

Proprietary and Confidential


< 4 >



Capgemini
CONSULTING TECHNOLOGIES ENTREPRENEURS

Intended Audience


➤ Developers/Programmers



January 29, 2016

Proprietary and Confidential

< 5 >



Capgemini

CONSULTING TECHNOLOGY ENTREPRENEURS

Day Wise Schedule

➤

Day 1
Lesson 1: Compiling C and CPP Programming on Unix
Lesson 2: Debugging your Program

➤

Day 2
Lesson 3: Makefile

January 29, 2016

Proprietary and Confidential

< 6 >



CONSULTING TECHNOLOGIES ENTREPRENEURS

Table of Contents

➤ **Lesson 1: Compiling C and CPP Programming on Unix**

- 1.1. Compiling C Programs using gcc compiler
- 1.2. Using static and dynamic link library
- 1.3. Compiling C++ Programs using g++ compiler

➤ **Lesson 2: Debugging your Program**

- 2.1. Debugging your Programs

➤ **Lesson 3: Makefile**

- 3.1 Problems with Multiple Source Files
- 3.2 What is a Makefile
- 3.3 Syntax of a Makefile
- 3.4 Options and Parameters of a Makefile
- 3.5 Macros in a Makefile

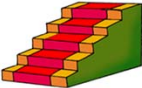
References

- Linux application development – Michael K. Johnson
- Managing Projects with GNU make, 3rd Edition - Robert Mecklenburg
- http://www.network-theory.co.uk/docs/gccintro/gccintro_4.html



Next Step Courses (if applicable)


➤ Linux Internals



January 29, 2016

Proprietary and Confidential

< 9 >

 **Capgemini**
CONSULTING TECHNOLOGIES ENTREPRENEURS


Other Parallel Technology Areas

- C/C++ Program development in Windows

January 29, 2016

Proprietary and Confidential

< 10 >



Capgemini
CONSULTING TECHNOLOGIES ENTREPRENEURS


C CPP ON UNIX

Lesson 1: Compiling C and CPP Programming on Unix

January 19, 2018

Proprietary and Confidential

< 1 >



Capgemini
ENJOYING TECHNOLOGY BETTER

Lesson Objectives

➤ **In this lesson, you will learn about:**

- Compiling C Programs using gcc compiler
- Using static and dynamic link library
- Compiling C++ Programs using g++ compiler



January 19, 2016 Proprietary and Confidential < 2 >



Lesson Objectives:

This lesson introduces you to program development in Unix/Linux Platform

Lesson 1: Compiling C and CPP programs using gcc and g++ compiler


1.1: Compiling C Programs Using gcc Compiler

Introduction to gcc Compiler

- **gcc compiler is founded by Richard Stallman**
- **gcc stands for GNU Compiler Collection**
 - As it provides support for languages like C, C++ (g++), Java (gcj), gnu smalltalk (gst) etc
- **Usage:**

`gcc [options] inputfilename
Eg: gcc -o outputfile inputfile`
- **When you invoke gcc, it normally does preprocessing, compilation, assembly and linking**

January 19, 2016 Proprietary and Confidential < 3 >



GCC Compiler: Introduction

The original author of the GNU C Compiler (GCC) is Richard Stallman, founder of the GNU Project. The GNU Project was started in 1984 to create a complete Unix-like operating system as free software, in order to promote freedom and cooperation among computer users and programmers. Every Unix-like operating system needs a C compiler, and as there were no free compilers in existence at that time, the GNU Project had to develop one from scratch.

The first release of GCC was made in 1987. This was a significant breakthrough, being the first portable ANSI C optimizing compiler released as free software. Since that time GCC has become one of the most important tools in the development of free software.

A major revision of the compiler came with the 2.0 series in 1992, which added the ability to compile C++. In 1997, an experimental branch of the compiler (EGCS) was created, to improve optimization and C++ support. Following this work, EGCS was adopted as the new main-line of GCC development, and these features became widely available in the 3.0 release of GCC in 2001. Over time GCC has been extended to support many additional languages, including Fortran, AD A, Java and Objective-C. The acronym GCC is now used to refer to the “GNU Compiler Collection”.


1.1: Compiling C Programs Using gcc Compiler

Features of gcc

➤ **gcc compiler:**

- Is a portable compiler running on many platforms
- Is a cross compiler, producing executable files for a different system from the one used by gcc
- Supports multiple programming languages
- Has a modular design, allowing support for new languages and architectures
- Is a free software, which:
 - Is distributed under GNU General Public License(GNU GPL)
 - Allows support for new languages and architectures to be added

January 19, 2016 Proprietary and Confidential > 4 <

 Capgemini
ENVIRONNEMENT TECHNOLOGIQUE INNOVATION

GCC Compiler: Features

First of all, gcc is a portable compiler — it runs on most platforms available today, and can produce output for many types of processors.

gcc is not only a native compiler — it can also cross-compile any program, producing executable files for a different system from the one used by gcc itself. This allows software to be compiled for embedded systems which are not capable of running a compiler. gcc is written in C with a strong focus on portability, and can compile itself, so it can be adapted to new systems easily.

gcc has multiple language frontends, for parsing different languages. Programs in each language can be compiled, or cross-compiled, for any architecture. For example, an ADA program can be compiled for a microcontroller, or a C program for a supercomputer.

gcc has a modular design, allowing support for new languages and architectures to be added. Adding a new language front-end to gcc enables the use of that language on any architecture, provided that the necessary run-time facilities (such as libraries) are available. Similarly, adding support for a new architecture makes it available to all languages.

Finally, and most importantly, gcc is free software, distributed under the GNU General Public License (GNU GPL). This means you have the freedom to use and to modify gcc, as with all GNU software. If you need support for a new type of CPU, a new language, or a new feature you can add it yourself, or hire someone to enhance gcc for you. You can hire someone to fix a bug if it is important for your work. Furthermore, you have the freedom to share any enhancements you make to gcc. As a result of this freedom you can also make use of enhancements to gcc developed by others. The many features offered by gcc today show how this freedom to cooperate works to benefit you, and everyone else who uses gcc.

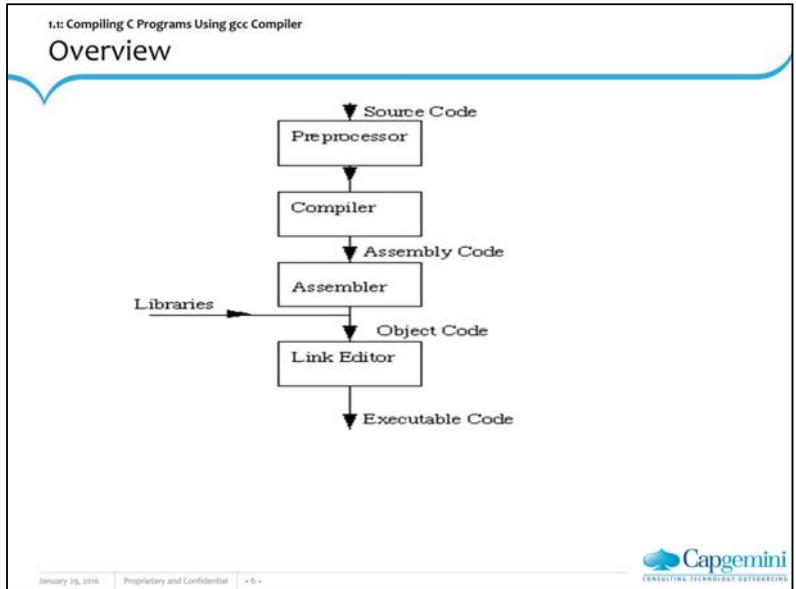
1.1: Compiling C Programs Using gcc Compiler

Definition

- **Compilation refers to the process of converting a program written in high level programming language such as C or C++, into machine code**
- **Programs can be compiled from a single source file or from multiple source files, and may use system libraries and header files**
- **This machine code is then stored in a file known as an executable file which is executed by the CPU**

January 19, 2018 Proprietary and Confidential 15





Stages of Compilation:

There are four stages in the process of Compilation of a C program

The Preprocessor:

Preprocessing is the first step in the C program compilation stage. This feature is unique to C compilers. The Preprocessor accepts source code as input and is responsible for removing comments. All preprocessor directives or commands begin with a #. The Preprocessor accepts source code as input and is responsible for removing comments interpreting special preprocessor directives denoted by #.

For example:

```
#include - To include contents of a file
#include <math.h> - standard library maths file.
#include <stdio.h> - standard library I/O file
#define - defines a symbolic name or constant.
```

Preprocessing causes the contents of the indicated header file(s) (which is in glibc) to be copied into the current file, effectively replacing the directive line with the contents of that file.

C Compiler:

The C compiler translates source to assembly code. The source code is received from the preprocessor.

Assembler:

The assembler creates object code. On a UNIX system you may see files with a .o suffix (.OBJ on MSDOS) to indicate object code files.

Link Editor:

If a source file references library functions or functions defined in other source files the link editor combines these functions (with main()) to create an executable file. External Variable references resolved here als

1.1: Compiling C Programs Using gcc Compiler

Example

➤ Step1: Create hello.c using vi editor

```
#include <stdio.h>
int main (void)
{ printf ("Hello, world!\n"); return 0; }
```


➤ Step 2: Compile the file 'hello.c' with gcc

```
$ gcc -Wall hello.c -o hello
```

➤ Step 3: To run the program execute hello

```
$ ./hello
Hello, world!
```

January 19, 2016 Proprietary and Confidential < 7 >

Capgemini
EXCELLENCE IN TECHNOLOGY SERVICES

Compiling the program:

```
$ gcc -Wall hello.c -o hello
```

This compiles the source code 'hello.c' to machine code and stores it in an executable file 'hello'. The output file for the machine code is specified using the -o option. This option is usually given as the last argument on the command line. If it is omitted, the output is written to a default file called 'a.out'.

Note: If a file with the same name as the executable file already exists in the current directory it will be overwritten.

The option -Wall turns on the most commonly-used compiler warnings—it is recommended that you always use this option! GCC will not produce any warnings unless they are enabled. Compiler warnings are an essential aid in detecting problems when programming in C and C++.

In this case, the compiler does not produce any warnings with the -Wall option, since the program is completely valid. Source code which does not produce any warnings is said to compile cleanly.

Run the program:

Type the path name of the executable like this:

```
$ ./hello
```

This loads the executable file into memory and causes the CPU to begin executing the instructions contained within it. The path ./ refers to the current directory, so ./hello loads and runs the executable file 'hello' located in the current directory.

1.1: Compiling C Programs Using gcc Compiler

Example (Contd...)


➤ **Write a Program:**

```
#include <stdio.h>
int main (void) {
    int a=4;
    printf ("Square of %d is %f/n",a,a*a);
    return 0; }
```

➤ **Compile the program:**

```
$gcc -Wall eg.c
eg.c: In function 'main':
eg.c:7: warning: double format, different type arg (arg 3)
$ ./a.out
Square of 4 is 0.000000
```

January 19, 2016 Proprietary and Confidential 18



Using Wall Option:

This indicates that a format string has been used incorrectly in the file 'eg.c' at line 7. The messages produced by gcc always have the form file:line-number:message. The compiler distinguishes between error messages, which prevent successful compilation, and warning messages which indicate possible problems (but do not stop the program from compiling).

In this case, the correct format specifier should be '%d' for an integer argument. The allowed format specifiers for printf can be found in any general book on C, such as the GNU C Library Reference Manual.

Without the warning option -Wall the program appears to compile cleanly, but produces incorrect results:

```
$ gcc eg.c
$ ./a.out
```

Square of 4 is 0.000000 (incorrect output)

The incorrect format specifier causes the output to be corrupted, because the function printf is passed an integer instead of a floating-point number. Integers and floating-point numbers are stored in different formats in memory, and generally occupy different numbers of bytes, leading to a spurious result. The actual output shown above may differ, depending on the specific platform and environment.


Clearly, it is very dangerous to develop a program without checking for compiler warnings. If there are any functions which are not used correctly they can cause the program to crash or produce incorrect results. Turning on the compiler warning option -Wall will catch many of the commonest errors which occur in C programming.

1.1: Compiling C Programs Using gcc Compiler

Analysis

- **Preprocessing stage**
gcc -E prog.c>prog.i
- **Compilation stage**
gcc -S prog.i>prog.s
- **Assembling stage**
gcc -c prog.s
 - This produces prog.o file
- **Linking stage:**
gcc -o prog prog.o

January 25, 2015 Proprietary and Confidential - 9 -



Stages of Compilation analyzed:

There are four stages in the C compilation model:

1. Preprocess
2. Compile
3. Assemble and
4. Linking.

We will track the following simple C program through the stages. The program source is contained in the file prog.c.

```
#define MESSAGE "hello world\n"
int main(void)
{
    printf(MESSAGE);
    return 0;
}
```

The Preprocessor:

The preprocessor is a program that processes the source text of a C program before the compiler. It has three major functions:

1. File inclusion
2. Macro replacement and
3. Conditional inclusion.

C CPP ON UNIX

|

Compiling C and CPP Programming on Unix

Cont..

File inclusion is the insertion of the text of a file into the current file, macro replacement is the replacement of one string by another, and conditional inclusion is the selective inclusion and exclusion of portions of source text on the basis of a computed condition. Our simple program prog.c just contains the single preprocessor directive

```
#define MESSAGE "hello world\n"
```

which causes MESSAGE to be replaced by "hello world\n" in the code. To preprocess prog.c and redirect the result to the file prog.i use the command

```
gcc -E prog.c > prog.i
```

Compile:

In this stage, it checks for the syntax errors and generates the assembly code. To compile the preprocessed code use the command

```
gcc -S prog.i
```

Assemble:

In this stage the assembled code is converted to the binary format, which is understood by the system. To convert to binary code use the command

```
gcc -c prog.s
```

This will produce an object file named prog.o.

Linking:

In this stage the object files are linked with the library functions.

The linking may be static or dynamic.

Static Linking: In this type of linking the entire definition of library functions will be copied into the executable image. This may require more disk space and memory than dynamic linking, but can be more portable (does not require the presence of the library on the system where it is run).

To perform static compilation use the following command:

```
gcc -static -o prog prog.c
```

Dynamic Linking:

Dynamic linking is accomplished by placing the name of a sharable library in the executable image. Actual linking with the library routines does not occur until the image is run, when both executable and library are placed in memory. An advantage of dynamic linking is that multiple programs can share a single copy of the library.

```
gcc -o prog prog.o
```

All four stages can be done in one go with the command

```
gcc -o prog prog.c
```

which produces an executable in the file prog.

1.1: Compiling C Programs Using gcc Compiler


Need for Multiple Source Files

- Whole program needs to be compiled whenever any small change is made to the program
- Recompilation of large program is very time consuming
- In case of independent source files, only the files changed need to be recompiled

January 29, 2016

Proprietary and Confidential

• 11 •


CONSULTING TECHNOLOGY PARTNERS

Need for Multiple Source Files:

If a program is stored in a single file, then any change to an individual function requires the whole program to be recompiled to produce a new executable. The recompilation of large source files can be very time-consuming.

When programs are stored in independent source files, only the files that have changed need to be recompiled after the source code has been modified. In this approach, the source files are compiled separately and then linked together -a two stage process. In the first stage, a file is compiled without creating an executable. The result is referred to as an object file, and has the extension '.o' when using gcc.

In the second stage, the object files are merged together by a separate program called the linker. The linker combines all the object files to create a single executable.

An object file contains machine code where any references to the memory addresses of functions (or variables) in other files are left undefined. This allows source files to be compiled without direct reference to each other. The linker fills in these missing addresses when it produces the executable.


1.1: Compiling C Programs Using gcc Compiler

Multiple Source Files

- A program can be split into multiple files for:
 - Ease of maintenance and understanding, incase of large programs
- In such cases the object files of the individuals can be linked together to form an executable.

```
$ gcc -Wall main.c hello_fn.c -o newhello
```

January 19, 2016 Proprietary and Confidential 12



Compiling Multiple Source Files:

In the following example we will split up the program 'Hello World' into three files: 'main.c', 'hello_fn.c' and the header file 'hello.h'. Here is the main program 'main.c':

```
#include "hello.h"
int main (void)
{ hello ("world");
  return 0; }
```

The original call to the printf system function in the previous program 'hello.c' has been replaced by a call to a new external function hello, which we will define in a separate file 'hello_fn.c'.

Cont...

Continued from previous notes page:

The main program also includes the header file 'hello.h', which will contain the declaration of the function hello. The declaration is used to ensure that the types of the arguments and return value match up correctly between the function call and the function definition. We no longer need to include the system header file 'stdio.h' in 'main.c' to declare the function printf, since the file 'main.c' does not call printf directly. The declaration in 'hello.h' is a single line specifying the prototype of the function hello:

```
void hello (const char * name);
```

The definition of the function hello itself is contained in the file

```
'hello_fn.c':  
#include <stdio.h>  
#include "hello.h"  
void hello (const char * name)  
{ printf ("Hello, %s!\n", name); }
```

This function prints the message "Hello, name!" using its argument as the value of name.

To compile these source files with gcc, use the following command:

```
$ gcc -Wall main.c hello_fn.c -o newhello
```

In this case, we use the -o option to specify a different output file for the executable, 'newhello'. Note that the header file 'hello.h' is not specified in the list of files on the command line. The directive #include "hello.h" in the source files instructs the compiler to include it automatically at the appropriate points.

To run the program, type the path name of the executable:

```
$ ./newhello  
Hello, world!
```


1.1: Compiling C Programs Using gcc Compiler

Creating Object Files from Source Files


- The `gcc -c` option is used to produce an object file

```
$ gcc -Wall -c main.c
```

- This produces an object file 'main.o' containing the machine code for the main function
- The corresponding command for compiling the hello function in the source file 'hello_fn.c' is:

```
$ gcc -Wall -c hello_fn.c
```

January 19, 2016 Proprietary and Confidential 14



Creating Object Files from Source Files:

The command-line option `-c` is used to compile a source file to an object file. For example, the following command will compile the source file 'main.c' to an object file:

```
$ gcc -Wall -c main.c
```

This produces an object file 'main.o' containing the machine code for the main function. It contains a reference to the external function `hello`, but the corresponding memory address is left undefined in the object file at this stage (it will be filled in later by linking). The corresponding command for compiling the `hello` function in the source file

```
'hello_fn.c' is:  
$ gcc -Wall -c hello_fn.c
```

This produces the object file 'hello_fn.o'.

Note: There is no need to use the option `-o` to specify the name of the output file in this case. When compiling with `-c` the compiler automatically creates an object file whose name is the same as the source file, but with '`.o`' instead of the original extension.


1.1: Compiling C Programs Using gcc Compiler

Creating Executable Files from Object Files

➤ An executable file is created using gcc to link the object files together

```
$ gcc main.o hello_fn.o -o hello
```

January 19, 2016 Proprietary and Confidential 15

 Capgemini
INNOVATIVE TECHNOLOGY SOLUTIONS

Creating Executable Files from Object Files:

The final step in creating an executable file is to use gcc to link the object files together and fill in the missing addresses of external functions. To link object files together, they are simply listed on the command line:

```
$ gcc main.o hello_fn.o -o hello
```

This is one of the few occasions where there is no need to use the `-Wall` warning option, since the individual source files have already been successfully compiled to object code. Once the source files have been compiled, linking is an unambiguous process which either succeeds or fails (it fails only if there are references which cannot be resolved).

The resulting executable file can now be run:

```
$ ./hello  
Hello, world!
```

It produces the same output as the version of the program using a single source file in the previous section

1.1: Compiling C Programs Using gcc Compiler

Important gcc Options


gcc option	Usage
-c file	Direct gcc to compile the source files into an object files without going through the linking stage.
-o file	Specifies that gcc's executable output should be named <i>file</i> .
-g file	Directs the compiler to include extra debugging information in its output.
-I dir	Adds the directory <i>dir</i> to the list of directories searched for #include files. There is no space between the "-I" and the directory name.
-l mylib (lower case L)	Search the library named <i>mylib</i> for unresolved symbols (functions, global variables) when linking. The actual name of the file will be <i>libmylib.a</i> .
-Wall	Show all compiler warnings. This flag is useful for finding problems that are not necessarily compilation errors. It tells the compiler to print warnings issued during compile which are normally hidden.

1.2: Using static and dynamic link library

Introduction

- **Library file is a unit of data (such as C/C++ routine, classes or variables), which can be shared by many programs.**
- **There are two types of libraries**
 - Static Library
 - Dynamic Library

January 19, 2016 Proprietary and Confidential 17



C functions/C++ classes and methods which can be shared by more than one application are separated out of the application's source code, compiled and bundled into a library. The C standard libraries and C++ STL are examples of shared components which can be linked with your code.

Whenever there are some C / C++ routines (methods), classes or variables which can be shared by many programs are put in the separate file such files are called as library files. These are called as user defined library files. There are two types of libraries –

1. Static Library

2. Dynamic Library

These libraries are named with the prefix "lib". When linking, the command line reference to the library will not contain the library (lib) prefix or suffix.

```
gcc src-file.c -lm -lpthread
```


The libraries referenced in the above example for inclusion during linking are the math library and the thread library. They are found in /usr/lib/libm.a and /usr/lib/libpthread.a.

1.2: Using static and dynamic link library

Introduction

- **If the library files get linked with object files and standalone executable at compile time then it is called as static link library**
- **Advantages:**
 - It avoids dependency problem :
 - Result in a performance improvement
 - Static linking can allow the application to be contained in a single executable file, simplifying distribution and installation

January 19, 2016 Proprietary and Confidential 18



Static Library

If the library files get linked with object files and standalone executable at compile time then it is called as static link library.

Advantages:

1. It avoids dependency problem :

The application can be certain that all its libraries are present and that they are the correct version.
2. Some times it can result in a performance improvement.


Since in static link library every application has there own copy of library file, it need not have to wait for getting access to library function like it is in dynamic link library. The program control need not have to jump to some other file (location to execute library function, which reduces the time of execution.
3. Static linking can allow the application to be contained in a single executable file, simplifying distribution and installation.

1.2: Using static and dynamic link library

Creating Static Link Library

- **Create Source Code files**
 - For Example, welcome.c, success.c and main.c
- **Create object file for the source code**
 - gcc -c welcome.c success.c
- **Creates an archive library, which consists of the named object files using ar command**
 - \$ ar -cr libmessage.a welcome.o success.o
 - cr - "create and replace". If the library does not exist, it is first created. If the library already exists, any original files in it with the same names are replaced by the new files

January 19, 2016 Proprietary and Confidential < 19 >



```
message.h
void welcome (const char * name);
void success (void);
```

```
welcome.c
#include <stdio.h>
#include "message.h"
void welcome (const char * name)
{
    printf ("Hello, %s!\n", name);
}
```


```
Success.c
#include <stdio.h>
#include "message.h"
void success (void)
{
    printf ("Congratulations on your best performance!\n");
}
```

1.2: Using static and dynamic link library

Creating Static Link Library (Contd...)

- **Generate output file using static linking**
 - `$ gcc -L -o main.o main.c -lmessage`
- **Run the program**
 - `./main.o`

January 19, 2016 Proprietary and Confidential 20



main.c

```
#include<stdio.h>
#include "message.h"
```

```
int main (void)
{
    welcome ("John");
    success ();
    return 0;
}
```

Compile the files as follows:

step1: Create object file welcome.o and success.o

```
$ gcc -c welcome.c success.c
```

Step2 : Creates an archive library, libmessage.a, which consists of the named object files.

```
$ ar -cr libmessage.a welcome.o success.o
```

The option cr stands for "create and replace". If the library does not exist, it is first created. If the library already exists, any original files in it with the same names are replaced by the new files specified on the command line. The first argument 'libhello.a' is the name of the library. The remaining arguments are the names of the object files to be copied into the library.

step3: Generate output file main.o using static linking. Use the -l option to link your program with libmessage.a

The link editor incorporates in your executable only the object files in this archive that contain a function you have called in your program.

Options used in command

- L – specifies the directory name where to search libmessage.a file
- o specifies name of executable file
- l to link output file with libmessage.a file

```
$ gcc -L. -o main.o main.c -lmessage
```

step4: Run the program

```
$ ./main.o
```

Hello, John

Congratulations on your best performance

Since this the library files welcome.c and success.c is included in executable file main.o (static linking) even if you remove those files still you can execute the main.o

1.2: Using static and dynamic link library


Disadvantages

➤ Disadvantages:


- The size of executable file becomes greater as compared to dynamic link library because the library code is stored within the executable rather than in separate files.
- A change in the library source requires building executable source also

Demo

➤ Demo: Creating static library.



January 19, 2016 | Proprietary and Confidential | 23


INNOVATIVE TECHNOLOGY SOLUTIONS

1.2: Using static and dynamic link library

Introduction

- If the library files get linked with shared object files and standalone executable at run time then it is called as dynamic link library.
- Advantages:
 - This allows the library files to be shared between many applications leading to space savings
 - It allows the library to be updated to fix bugs and security flaws without updating the applications that use the library

January 19, 2016 Proprietary and Confidential 24



1.2: Using static and dynamic link library

Shared Libraries

- Shared libraries are libraries that are loaded by programs at runtime with extension '.so', which stands for shared object
- Linker makes a note that the calling function is a part of the shared object, so it adds startup code instruction of the required shared library, instead of extracting executable code from the shared object
- The shared libraries supports position independence by using -fPIC while compiling the program

January 19, 2018 Proprietary and Confidential 25




1.2: Using static and dynamic link library

Creating Dynamic link library

- **Create Source Code files**
 - For Example, welcome.c, success.c and main.c
- **Create Library**
 - `gcc -Wall -fPIC -c welcome.c success.c`
Link object files into a shared library named as "libmessage.so"
 - `gcc -shared -o libmessage.so welcome.o success.o`
- **Create executable file main.o by setting the environment variable to search for the library file while linking**

January 19, 2016 Proprietary and Confidential 26



Compiler options:

- Wall: include warnings. See man page for warnings specified.
- fPIC: Compiler directive to output position independent code, a characteristic required by shared libraries.
- shared: Produce a shared object which can then be linked with other objects to form an executable.
- Wl: Pass options to linker.
- o: Output of operation. In the above example the name of the output will be "libmessage.so"

Compile the files as follows

Step 1 : Create library

```
gcc -Wall -fPIC -c welcome.c success.c
```

```
gcc -shared -o libmessage.so welcome.o success.o
```

Step 2: Create executable file main.o

When you use the -l option, you must point the dynamic linker to the directories of the dynamically linked libraries that are to be linked with your program at execution by setting LD_RUN_PATH

To set LD_RUN_PATH, list the absolute pathnames of the directories you want searched in the order you want them searched.

```
LD_RUN_PATH=/home/user11 export LD_RUN_PATH
```

1.2: Using static and dynamic link library


Creating Dynamic link library (Contd...)

- Environment variable - LD_RUN_PATH directs the dynamic linker to search the library file in the specified absolute path
 - LD_RUN_PATH=/home/user11 export LD_RUN_PATH
- gcc -o main.o main.c -L/home/user11 -lmessage

➤ **Run the Program.**

- ./main.o

January 19, 2016 Proprietary and Confidential 22



```
gcc -o main.o main.c -L/home/user11 welcome.c success.c -lmessage
```

directs the dynamic linker to search for libmessage.so in /home/user11 when you execute your program.

The dynamic linker searches the standard place by default, after the directories you have assigned to LD_RUN_PATH. The standard place for libraries is /usr/lib. Any executable versions of libraries supplied by the compilation system kept in /usr/lib.

Step 3:Run the program.

```
./main.o
```

Hello, John!

Congratulations on your best performance!

1.2: Using static and dynamic link library

Different ways to locate library file

- **At the compile time, gcc employs -L and -l options to tell where a library locates and which library to be linked, these options are passed to ld**
- **There are two ways to determine a library path**
 - Compile time
 - Using Environment variable LD_RUN_PATH
 - Using command line option -rpath or -R
 - Execution time
 - Using Environmental variable LD_LIBRARY_PATH

January 19, 2016 Proprietary and Confidential 28



1.2: Using static and dynamic link library

Different ways to locate library file (Contd...)

➤ Using command line option `--rpath` or `-R`:

- Setting the environmental variables like `LD_LIBRARY_PATH` or `LD_RUN_PATH` can override and modify the system-wide defaults.
- Use linker option `--rpath` in `ld` command or `"-Wl,rpath,"` in `gcc` command to specify the current directory as a search directory
 - For example:
- `$gcc -o main.o main.c -Wl,-rpath,/home/user11 -L/home/user11 -lmessage`

January 19, 2016 Proprietary and Confidential 28



Using command line option `--rpath` or `-R`:

Setting the environmental variables like `LD_LIBRARY_PATH` or `LD_RUN_PATH` can override and modify the system-wide defaults.

Use linker option `--rpath` in `ld` command or `"-Wl,rpath,"` in `gcc` command to specify the current directory as a search directory For example

```
$gcc -o main.o main.c -Wl,-rpath,/home/user11 -L/home/user11 -lmessage
```


1.2: Using static and dynamic link library

Different ways to locate library file (Contd...)

➤ LD_LIBRARY_PATH

- Once the library file is moved to another location, You can temporarily substitute a different library for particular execution of your program without performing compilation again

- For example,

- \$ LD_LIBRARY_PATH=/home/user11/dynamic export LD_LIBRARY_PATH
 - \$. /main.o

➤ ldd command – Used to check which shared object is in use.

- \$ ldd main.o



January 19, 2016 Proprietary and Confidential 30

LD_LIBRARY_PATH

The environment variable LD_LIBRARY_PATH lets you do the same thing at run time.

Suppose the library file is moved from /home/user11 to /home/user11/dynamic, You can temporarily substitute a different library for particular execution of your program without performing compilation again as follows:

```
$ LD_LIBRARY_PATH=/home/user11/dynamic export LD_LIBRARY_PATH
$. /main.o
```

Now when you execute your program, the dynamic linker searches for libmessage.so first in /home/user11 and, if not found, then it will search in /home/user11/dynamic. The directory assigned to LD_RUN_PATH is searched before the directory assigned to LD_LIBRARY_PATH.

ldd command – Used to check which shared object is in use.

```
$ ldd main.o
/lib/libcwait.so (0x00c67000)
libmessage.so => /home/user11/libmessage.so (0x00111000)
libc.so.6 => /lib/tls/libc.so.6 (0x00366000)
/lib/ld-linux.so.2 (0x0034d000)
```

Demo

- Demo: Creating dynamic link library.



1.3: Compiling C++ Programs Using g++ Compiler

Introduction to g++ Compiler

- **The GNU g++ compiler can be used to control both the compilation phase and the linking phase of creating executable files for C++ programs**
- **Steps to obtain the final executable program**
 - **Compiler stage**
 - The .cpp file is converted to Assembly language (.s files)
 - **Assembler stage**
 - Assembly language code is converted into object code (.o file)
 - **Linker stage**
 - Object files linked to produce an executable file

January 19, 2016 Proprietary and Confidential 32



1.3: Compiling C++ Programs Using g++ Compiler

Process

➤ Step 1: Create a test.cpp program

```
#include <iostream>
using namespace std;
int main()
{ cout << "Testing 1, 2, 3 \n";
  return 0; }
```

➤ Step 2: Compile your program with g++ compiler

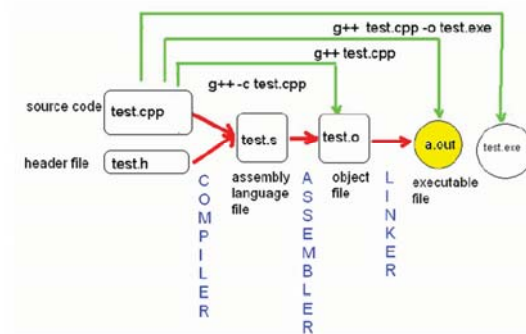
```
$g++ test.cpp
```

➤ Step 3: If there are no errors, an executable file is generated with the standard name a.out

```
$/a.out
$Testing 1, 2, 3
```

1.3: Compiling C++ Programs Using g++ Compiler

Overview



1.3: Compiling C++ Programs Using g++ Compiler

Important g++ Compiler Options

g++ compiler options	Usage
-C	This flag forces the compiler to produce object files from source files without linking
-o	This flag allows renaming the executable file with another name than a.out.
-Wall	This flag forces the compiler to display all the warning messages.

All the options of g++ compiler behaves in the same manner as that of the gcc compiler.

Summary

- **In this lesson, we learnt about:**
- Various stages of compilation
 - gcc compiler and its various options
 - g++ compiler and its various options



Review Question

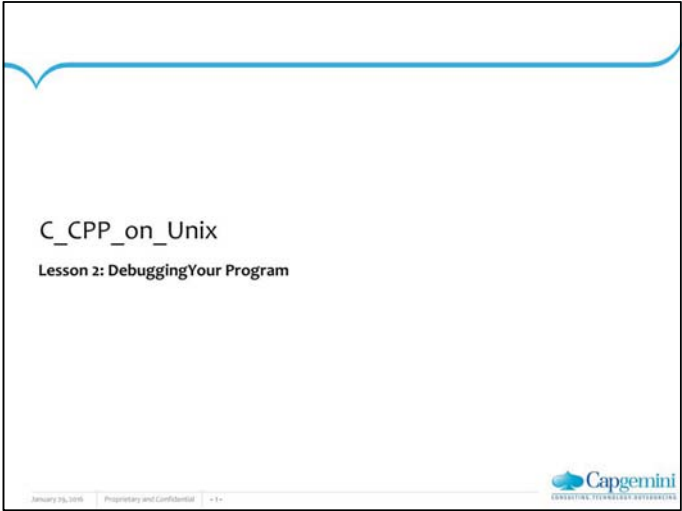
- **Question 1: The gcc -c option is used:**
 - To produce an executable file
 - To produce an object file
 - To produce warnings
- **Question 2: Which of the following options is true regarding the gcc compiler -Wall option?**
 - Used to display warnings
 - Used to display message to all users



Review Question

- **Question 3: The compiler used to compile C++ Programs is:**
 - gcc
 - g++
 - Both Option1 and Option2 can be used
- **Question 4: Dynamic linking makes executable files smaller and saves disk space**
 - True/False







Lesson Objectives

➤ **On completion of this lesson you will be able to:**

- Debug your programs using the GDB debugger



January 19, 2016 Proprietary and Confidential • 2 •



Lesson Objectives:


This lesson introduces you to functions of a debugger tool.

1.1 : What is a debugger?

Introduction to Debugger

- A debugger is an application that runs your program, like you run, when you type the name of your program.
- The difference between manual debugging and debugging by an application is, a debugger can step through your source code line by line, executing each line only when you want so.
- At any point, you can inspect and even change the value of any variable at run-time.

January 19, 2016 Proprietary and Confidential 3



What is a debugger?

A symbolic debugger is an application that enables you to step through your program executing one machine instruction at a time. Following are its advantages:

At any point, you can inspect and even change the value of any variable at run-time.

If your program crashes, a symbolic debugger shows you where and why the program has crashed so that you can deduce the error.

You can go through the program and observe which lines of source code got executed and in which order.

You can use a debugger to step through an infinite loop and observe why your conditions fail to function as per your requirements and specifications.

If your program crashes on a variable access, the debugger shows you all the information about the variable you tried to access and the value you assigned (or perhaps didn't assign) to it.

If a line in your code does not get executed, you can use the debugger to observe what gets executed, in which order, and why a particular line is not reached.


Thus, other than a compiler, the debugger is the most useful tool a programmer can use.

1.2 : Why use a debugger?

Need for a Debugger

- No one writes perfect code first time, and every time.
- Desk checking code can be tedious and error-prone.
- Putting print statements in the code requires re-compilation.
- Adding print statements for debugging adds "trace code" to your program.
- Debuggers are powerful and flexible.

January 19, 2016 Proprietary and Confidential + 8 -



Why a Debugger?

Most people use the `printf()` debugging method. This is called adding "trace code" to your program. Simply put, they include `printf()` in their code to view the value of variables at certain strategic points and to examine the order of execution of lines of source code.

There are a few reasons why a debugger is better than the `printf()` command: Sometimes, you need many `printf()` commands, and putting them in and taking them out is tedious. Inserting and deleting superfluous code all the time is distracting.

A symbolic debugger can perform some functions that `printf()` can't. For example, you can change the value of variables at run-time, halt the program temporarily, list source code, print the datatype of a variable or struct that you don't recognize, jump to an arbitrary line of code, and much more.

You can use a symbolic debugger on a running process (without ending the process). You can use the `printf()` command for the same.

You can use a symbolic debugger on a process that has already crashed and ended. For that, you do not have to re-run the program. You can view the state in which the program was at the time of its crash and can inspect all the variables.

Knowledge of GDB increases your knowledge of programs, processes, memory and your language of choice.


You can find and fix your bugs faster using a symbolic debugger such as GDB. However, `printf()` is still useful in debugging.

1.2 : What is GDB?

Introduction to GDB

- GDB is a debugger which is a part of the GNU operating system of the Free Software Foundation
- GDB can be used to debug C, C++, Objective-C, Fortran, Java and Assembly programs
- GDB is an opensource software and is licensed under GPL public license

January 19, 2016 Proprietary and Confidential • 5 •



What is GDB?

GDB is a debugger which is a part of the Free Software Foundation's GNU operating system. Its original author is Richard M. Stallman. GDB can be used to debug C, C++, Objective-C, Fortran, Java and Assembly programs. GDB provides partial support for Modula-2 and Pascal.

1.3 : How to use a GDB?

Using GDB

- **To compile your programs, use the -g option:**


```
$gcc [other flags] -g <source files> -o <output file>
```

For example: `gcc -g -o hello hello.c`
- **To start and use GDB:**
 - Type `gdb` or `gdb hello`.
 - The following prompt is displayed:
(gdb)
If you didn't specify a program to debug, you have to load it in now, as follows:
(gdb) file hello
Here, **hello** is the program you want to load, and **file** is the command to load it.

January 19, 2016

Proprietary and Confidential

» 6 «

Capgemini
CREATING YOUR BEST FUTURE

Using GDB:

If you plan to debug an executable, a corefile resulting from an executable, or a running process, you must compile the executable with an enhanced symbol table. To generate an enhanced symbol table for an executable, you must compile it with gcc's -g option as follows:
`gcc -g -o filename filename.c`

As previously discussed, there are many different debugging formats. The actual function of -g is to produce debugging information in the native format for your system.

1.3 : How to use a GDB?

Useful Features of GDB

The significant features of GDB are as follows:

- GDB has an interactive shell.
- It can recall history with the arrow keys, auto-complete words (most of the times) with the TAB key, and offers many other advantageous features.
- It allows you to use the help command if help is needed on a particular command.
 - For example: (gdb) help list
- You get useful description about list commands.

January 19, 2016 Proprietary and Confidential 17



1.3 : How to use a GDB?

Useful Commands in GDB

➤ The significant commands are as follows:

help list

run

continue

break

step

next


print/x

print

January 19, 2016

Proprietary and Confidential

+ 8 +

Capgemini
CREATING YOUR BEST SOLUTION

1.3 : How to use a GDB?

Command to Run a Program

- To run the program, use only the following command:
 - (gdb) run
- If the program has no serious problems (that is, the normal program didn't have a segmentation fault, and similar other problems), it runs successfully.
- If the program has issues, then GDB can provide you with useful information such as the line number at which the program crashed, and the parameters to the function that caused the error.

January 19, 2016 Proprietary and Confidential « 9 »



1.3 : How to use a GDB?

How to Set Breakpoints?

- Breakpoints can be used to stop a running program at a designated point.
- The simplest way to insert a breakpoint is to use the break command.
 - (gdb) break file1.c:6
 - This sets a breakpoint at line 6, of file 1.c
- Now, if a running program reaches a breakpoint, the program pauses and prompts you for another command.
- You can set as many breakpoints as you require, and the program execution stops if it reaches any of them.

January 19, 2016 Proprietary and Confidential < 10 >

 Capgemini
EXCELLENCE YOURSelves REINVENTING

1.3 : How to use a GDB?

How to Set Breakpoints? (Contd...)

- You can also instruct GDB to break at a particular function.
- For example: To specify the my func function, use the following instruction:
 - `int my func(int a, char *b);`
- You can break this function using the following instruction:
 - `(gdb) break myfunc`
- Once you have set a breakpoint, you can use the run command again.
- This time, it should stop at the breakpoint you set (unless a fatal error occurs before reaching that point).

January 19, 2016

Proprietary and Confidential

• 11 •



1.3 : How to use a GDB?

Using the Continue and Step Commands

- You can proceed to the next breakpoint by using the continue command (Typing run again would restart the program from the beginning.)
 - (gdb) continue
- You can 'single-step' (execute just the next line of the code) by using the step command.
- This gives you fine control over how the program proceeds.

January 19, 2016 Proprietary and Confidential • 12 •



1.3 : How to use a GDB?

Using the Next Command

- The next command single-steps as well, however, it doesn't execute each line of a sub-routine, it treats the sub-routine as one instruction, executes it and reaches the next instruction. You require to type the following:
 - (gdb) next
- Typing step or next a lot of times can be tedious.
- If you just press ENTER, GDB repeats the command you just typed.
- You can repeat this action several times.

January 19, 2016 Proprietary and Confidential • 13 •




1.3 : How to use a GDB?

Using the Print and Print/x Commands

- **The print command prints the value of the variable specified.**
- **The print/x prints the value in hexadecimal:**
 - (gdb) print my var
 - (gdb) print/x my var

January 19, 2016 Proprietary and Confidential • 14 •

 Capgemini
EXCELLENCE YOUR WAY

1.3: How to use a GDB?

How to Set Watchpoints?


- Watchpoints act on variables.
- They pause the program whenever value of a watched variable is modified.

```
#include <stdio.h>

int main(int argc, char **argv)
{
    int x = 30;
    int y = 10;
    x = y;
    return 0; }
```

```
5: int x=30; y=10;
gdb> set breakpoint address now.
Breakpoint 5
(gdb) r
Starting program: /home/testuser18/a.out
Program exited normally.
gdb> b 5
Breakpoint 1 at 0x8040357: file eg.c, line 5.
(gdb) r
Starting program: /home/testuser18/a.out
Breakpoint 1, main (argc=1, argv=0xbf21864) at eg.c:5
5:     int y = 10;
(gdb) watch x
Hardware watchpoint 2: x
(gdb) c
Continuing.
Hardware watchpoint 2: x
Old value = 30
New value = 10
main (argc=1, argv=0xbf21864) at eg.c:7
7:     return 0;
(gdb) c
```

January 19, 2016 Proprietary and Confidential - 15 -

 Capgemini
CREATING YOUR FUTURE SOLUTIONS

How to Set Watchpoints?:

Watchpoints are similar to breakpoints. However, watchpoints are not set for functions or lines of code. Watchpoints are set on variables. When those variables are read or written, the watchpoint is triggered and the program execution stops.

Setting a write watchpoint for a variable:

For this, use the watch command. The argument to the watch command is an expression that is evaluated. This implies that the variable on which you want to set a watchpoint must be in the current scope. So, to set a watchpoint on a non-global variable, you must set a breakpoint that stops your program when the variable is in scope. You set the watchpoint after the program breaks.

Setting a read watchpoint for a variable:

For this, use the rwatch command. Usage is identical to the watch command.

Setting a read/write watchpoint for a variable:

For this, use the awatch command. Usage is identical to the watch command.

1.3 : How to use a GDB?

Other Useful Commands

- **Backtrace:** This command produces a stack trace of the function calls that lead to a seg fault.
- **Where:** This command works in the same manner as backtrace does; however, this version works even when you are still in the middle of the program.
- **Finish:** This command runs until the current function is finished.
- **Delete:** This command deletes a specified breakpoint.
- **Info breakpoints:** This command shows information about all declared breakpoints.

January 19, 2016 Proprietary and Confidential • 16 •



1.3 : How to use a GDB?

Conditional breakpoints

- For conditional breakpoints, you use the following command:
 - (gdb) break file1.c:6 if i >= ARRAYSIZE
- This command sets a breakpoint at line 6 of file file1.c, which triggers only if the variable i is greater than or equal to the size of the array.
- Conditional breakpoints can considerably avoid all the unnecessary stepping.

Summary

➤ **In this lesson, you learnt the following:**

- The various stages of Compilation
- gcc compiler and its various options
- g++ compiler and its various options



Review Question

- **Question 1: Why is the `gcc -c` option used?**
 - To produce an executable file
 - To produce an object file
 - To produce warnings
- **Question 2: Which of the following is true regarding the `Wall` option in the `gcc` compiler?**
 - It is used to display warnings.
 - It is used to display message to all users.



Review Question

➤ Question 3: Which compiler is used to compile C++ programs?

- gcc
- g++
- Both Option1 and Option2 can be used




C_CPP_ON_UNIX

Lesson 3: Makefile

January 10, 2015

Proprietary and Confidential

~ 3 ~



Capgemini
DIGITAL TECHNOLOGY INTEGRATION

Lesson Objectives

- To understand the following topics:
 - The make command and makefiles



January 19, 2016 Proprietary and Confidential 13



Lesson Objectives:

This lesson introduces to the fundamentals of the Java programming language.
Lesson 3: This lesson introduces you to *makefile*, a UNIX utility used to build your application.

3.1: Multiple Source Files

Problems with Multiple Source Files

➤ Problems with using multiple source files are:

- If there is a change in any file, dependent files, if any, also need to be recompiled
 - If dependent files are not compiled again, there can be unpredictable results
 - Difficult to keep track of the dependent files in large applications
 - Time for the edit-compile-test cycle grows

January 25, 2016 Proprietary and Confidential 3



When writing small programs, many people simply rebuild their application after edits by recompiling all the files. However, with larger programs, some problems with this simple approach become apparent.

Time for the edit-compile-test cycle grows. Even the most patient programmer prefers to avoid recompiling all files when only one is changed.

A potentially much more difficult problem arises when multiple header files are created and included in different source files. Suppose you have header files `a.h`, `b.h` and `c.h`, and C source files `main.c`, `2.c` and `3.c`. (We hope that you choose better names than these for real projects!) you could have the situation follows:

```
/* main.c */
#include "a.h"
...
/* 2.c */
#include "a.h"
#include "b.h"
...
/* 3.c */
#include "b.h"
#include "c.h"
...
```

If the programmer changes `c.h`, the files `main.c` and `2.c` do not need to be recompiled, since they do not depend on this header file.

The file `3.c` does depend on `c.h` and should therefore be recompiled if `c.h` is changed.

However, if `b.h` was changed and the programmer forgot to recompile `2.c`, then the resulting program might no longer function correctly.

3.2: What is a Makefile?

Introduction

➤ Makefile

- Make utility automates and optimizes program construction
- Helps you build programs with many components or source files.
- Descriptor file, namely makefile:
 - Describes the relationship among project files
 - Provides commands for updating each file
- Make invokes makefile to automatically rebuild a program whenever one or more source file is modified
- Only recompiles files affected by changes
 - Saves compiling time
- Reduces the likelihood of human errors when making entries from the command line

January 19, 2016 Proprietary and Confidential 4



3.3: Syntax of a Makefile

Makefile Syntax


➤ **Syntax is as follows:**

Target: dependency rule

➤ **Makefile comprises a set of dependencies and rules.**

- Dependency has:
 - Target and a set of source files on which it is dependent.
- The rule describes how to create the target from the dependent files.

January 19, 2016 Proprietary and Confidential 15

 **Capgemini**
ENJOYING THE JOURNEY OF INNOVATION

A makefile consists of a set of dependencies and rules. A dependency has a target (a file to be created) and a set of source files upon which it is dependent. The rules describe how to create the target from the dependent files. Commonly, the target is a single executable file.

The makefile is read by the make command, which determines the target file or files that are to be made and then compares the dates and times of the source files to decide which rules need to be invoked to construct the target. Often, other intermediate targets have to be created before the final target can be made. The make command uses the makefile to determine the order in which the targets have to be made and the correct sequence of rules to invoke.

3.3: Syntax of a Makefile

Dependencies

- Dependencies specify how each file in the final application relates to the source files.

```
myapp: main.o 2.o 3.o
main.o: main.c a.h
2.o: 2.c a.h b.h
3.o: 3.c b.h c.h
```

- Generally, on issuing the make command only the first target gets executed.

January 25, 2016 Proprietary and Confidential 6



Dependencies specify how each file in the final application relates to the source files. In your example above, you might specify dependencies that say your final application requires (depends on) main.o, 2.o and 3.o; and likewise for main.o (main.c and a.h); 2.o (2.c, a.h and b.h); and 3.o (3.c, b.h and c.h). Thus main.o is affected by changes to main.c and a.h, and it needs to be recreated, by recompiling main.c, if either of these two files changes.

In a makefile, you write these rules by writing the name of the target, a colon, spaces or tabs and then a space or tab-separated list of files that are used to create the target file. The dependency list for your example is:

```
myapp: main.o 2.o 3.o
main.o: main.c a.h
2.o: 2.c a.h b.h
3.o: 3.c b.h c.h
```

This says that myapp depends on main.o, 2.o and 3.o, main.o depends on main.c and a.h, and so on. This set of dependencies gives a hierarchy showing how the source files relate to one other.

You can see quite easily that, if b.h changes, then you need to revise both 2.o and 3.o and, since 2.o and 3.o will have changed.

You also need to rebuild myapp. If you wish to make several files, then use the phony target *all*. Suppose your application consisted of both the binary file myapp and a manual page, myapp.1. Specify this with the line:

```
all: myapp myapp.1
```

If you do not specify an *all* target, make simply creates the first target it finds in the makefile.

3.3: Syntax of a Makefile

Rules

- Rules describe how to create a target.

```
2.o: 2.c a.h b.h
gcc -c 2.c
```

- If 2.o needs to be rebuilded, the command to execute is:
 - * gcc -c 2.c

All rules must be on lines that start with a tab; a space will not do. Also, a space at the end of a line in the makefile may cause a make command to fail.

3.3: Syntax of a Makefile

A Simple Makefile

- Store the following code in a file called Makefile:

```
myapp: main.o 2.o 3.o gcc -o myapp main.o 2.o 3.o
main.o: main.c a.h
      gcc -c main.c
2.o: 2.c a.h b.h
      gcc -c 2.c
3.o: 3.c b.h c.h
      gcc -c 3.c
```

- At the dollar prompt enter:

- `$make`

```
make: *** No rule to make target 'main.c', needed by 'main.o'. Stop
```

January 19, 2016

Proprietary and Confidential

- 8 -



The make command has assumed that the first target in the makefile, myapp, is the file that you wish to create. It has then looked at the other dependencies and, in particular, has determined that a file called main.c is needed. Since you haven't created this file yet and the makefile does not say how it might be created, make has reported an error. Let's create the source files and try again. Since we're not interested in the result, these files can be very simple. The header files are actually empty, so you can create them with touch.

```
$ touch a.h
$ touch b.h
$ touch c.h
```

main.c contains main, which calls function_two and function_three. The other two files define function_two and function_three. The source files have #include lines for the appropriate headers, so they appear to be dependent on the contents of the included headers. It is not much of an application, but here are the listings.

```
/* main.c */
#include "a.h"
extern void function_two();
extern void function_three();
int main(){
    function_two();
    function_three();
    exit(EXIT_SUCCESS); }
```

```

/* 2.c */
#include "a.h"
#include "b.h"
void function_two() {
}
/* 3.c */
#include "b.h"
#include "c.h"
void function_three() {
}
$ make
gcc -c main.c
gcc -c 2.c
gcc -c 3.c
gcc -o myapp main.o 2.o 3.o
$

```

This is a successful make.

How It Works

The `make` command has processed the dependencies section of the makefile and determined the files that need to be created and in which order. Even though you listed how to create `myapp` first, `make` has determined the correct order for creating the files. It has then invoked the appropriate commands you gave it in the rules section for creating those files. The `make` command displays the commands as it executes them. you can now test your makefile to see whether it handles changes to the file `b.h` correctly.

```

$ touch b.h
$ make
gcc -c 2.c
gcc -c 3.c
gcc -o myapp main.o 2.o 3.o
$

```

The `make` command has read your makefile, determined the minimum number of commands required to rebuild `myapp` and carried them out in the correct order. Let's see what happens if you delete an object file.

```

$ rm 2.o
$ make -f Makefile1
gcc -c 2.c
gcc -o myapp main.o 2.o 3.o
$

```

Again, `make` correctly determines the actions required.

3.4: Options and Parameters of a Makefile

Options and Parameters

- **-k** – tells make keep going
- **-n** – tells make to print out what it would have done, without actually doing
- **-f <filename>** - tells make which file to use as its makefile
 - Examples:
 - `make -f myfile`
 - `make -f myfile mytarget`



January 10, 2016 | Proprietary and Confidential | 10

The make program has several options. The three most commonly used are:

-k, which tells make to 'keep going' when an error is found, rather than stopping as soon as the first problem is detected. You can use this, for example, to find out in one go which source files fail to compile.

-n, which tells make to print out what it would have done, without actually doing it.

-f <filename>, which allows you to tell make which file to use as its makefile. If you do not use this option, make looks first for a file called makefile in the current directory. If that doesn't exist, it looks for a file called Makefile. By convention, most UNIX programmers use Makefile.

To make a particular target, which is usually an executable file, you can pass its name to make as a parameter. If you do not, make will try to make the first target listed in the makefile.

Many programmers specify all as the first target in their makefile and then list the other targets as being dependent on all. This convention makes it clear which target the makefile should attempt to build by default when no target is specified. you suggest you stick to this convention.

3.5: Macros in a Makefile

Macros

- **Macros are often used in makefiles for options to the compiler**
- **Often, while an application is being developed it is compiled with no optimization, but with debugging information included**
- **For a release version the opposite is usually needed**
 - A small binary with no debugging information that runs as fast as possible
- **Macros can also be used to achieve portability**

January 19, 2016 Proprietary and Confidential 10



Let us look at an example where Makefile can be used. You have written a Makefile with the gcc Compiler. On other UNIX systems, you might be using cc or c89. If you ever wanted to take your makefile to a different version of UNIX, or even if you obtained a different compiler to use on your existing system, you would have to change several lines of your makefile to make it work. Macros are a good way of collecting together all these system dependent parts, making it easy to change them.

3.5: Macros in a Makefile

Macros (Contd...)

- A macro can be defined by writing **MACRONAME=value**
- The macro can be accessed by writing either:
 - `$(MACRONAME)` or `${MACRONAME}`
- Macros are normally defined inside the makefile itself
- You can also specify them by calling make with the macro definition
 - Eg: `make CC=gcc`
- Command line definitions override defines in the makefile

3.5: Macros in a Makefile

Macros (Contd...)


```
all: myapp
# Which compiler
CC = gcc
# Where include files kept
INCLUDE = .
# Options for development
CFLAGS = -g -Wall -ansi
# Options for release
# CFLAGS = -O -Wall -ansi
```

```
myapp: main.o 2.o 3.o
$(CC) -o myapp main.o 2.o 3.o main.c main.c a.h
$(CC) -I$(INCLUDE) $(CFLAGS) -c main.c 2.c a.h b.h
$(CC) -I$(INCLUDE) $(CFLAGS) -c 2.c 3.o: 3.c b.h c.h
$(CC) -I$(INCLUDE) $(CFLAGS) -c 3.c
```

January 10, 2016

Proprietary and Confidential

< 13 >

Capgemini
INNOVATING. TRANSFORMING. SUSTAINING.

3.5: Macros in a Makefile

Macros (Contd...)

- Delete an old installation and create a new one
- With this new makefile, you get:

```
$ rm *o myapp
$ make -f Makefile2
gcc -I. -g -Wall -ansi -c main.c
gcc -I. -g -Wall -ansi -c 2.c
gcc -I. -g -Wall -ansi -c 3.c
gcc -o myapp main.o 2.o 3.o
$
```

How It Works

The make command replaces the \$(CC), \$(CFLAGS) and \$(INCLUDE) with the appropriate definition, rather like the C compiler does with #define. Now if you want to change the compile command, you only need to change a single line of the makefile.

Summary

➤ **In this lesson you have learnt:**

- What is a Makefile?
- How to Write a Makefile?
- What are Macros and how to use them?



Review Question

- **Question 1: How will you run make utility to use mymakefile as a Makefile?**
 - make -k mymakefile
 - make mymakefile
 - make -f mymakefile
 - make -n mymakefile
- **Question 2: How will you run make if you want to execute the target hello?**
 - make hello
 - make -f hello
 - make -n hello
- **Question 3: Comments in makefile are represented using:**
 - //
 - /* */
 - #
 - None of the above



C CPP ON UNIX

Lab Book

Copyright © 2011 IGATE Corporation (a part of Capgemini Group). All rights reserved.
No part of this publication shall be reproduced in any way, including but not limited to photocopy, photographic, magnetic, or other record, without the prior written permission of IGATE Corporation (a part of Capgemini Group).
IGATE Corporation (a part of Capgemini Group) considers information included in this document to be confidential and proprietary.

Document Revision History

Date	Revision No.	Author	Summary of Changes
Oct-2009		Anitha Ramesh	New Course Creation
04-Nov-2009		CLS Team	Review
07-Jun-2011		Rathnajothi Perumalsamy	Revamp/Refinement

Table of Contents

<i>Document Revision History</i>	2
<i>Table of Contents</i>	3
<i>Getting Started</i>	4
<i>Overview</i>	4
<i>Setup Checklist</i>	4
<i>Instructions</i>	4
<i>Learning More (Bibliography)</i>	4
<i>Lab 1. C Program Development in Unix</i>	5
1.1: <i>Writing and executing a simple reverse program in C</i>	5
1.2: <i>Learn to make a multi module Program</i>	8
1.3 <i>Learn to generate static link library for a program.</i>	9
1.4 <i>Learn to generate dynamic link library for a program.</i>	11
<i>Lab 2. C++ Program Development in Unix</i>	14
2.1: <i>Writing a simple C++ program that displays "Hello World"</i>	14
2.2: <i><TO DO>> Create a simple "shape" hierarchy:</i>	15
2.3: <i><<TO DO>> Templatzize the fibonacci() function</i>	15
<i>Lab 3. Debugging your C/C++ program</i>	16
3.1: <i>To debug your C/C++ Program</i>	16
3.2: <i><<TO DO>> Debug the following C++ program</i>	18
3.3: <i><<To DO>> Debug the following C++ Program</i>	18
3.4: <i>Debug the following C Program</i>	19
3.5: <i>Debug the following C Program</i>	20
<i>Lab 4. Makefile Utility</i>	21
4.1: <i>Creating a makefile for Lab exercise 1.2</i>	21
4.2: <i><<TODO>> Create a Makefile for all the C and C++ programs which are made of multiple modules.</i>	22
<i>Appendices</i>	23
<i>Appendix A: Table of Figures</i>	23
<i>Appendix B: Table of Examples</i>	24

Getting Started

Overview

This lab book is a guided tour for learning C and C++ Programming under the Linux environment. It comprises solved examples and 'To Do' assignments. Follow the steps provided in the solved examples and work out the 'To Do' assignments given which will expose you to program development in the UNIX environment.

Setup Checklist

Here is what is expected on your machine in order for the lab to work.

Minimum System Requirements

- Intel Pentium 90 or higher (P166 recommended)
- Microsoft Windows 95, 98, or NT 4.0, 2k, XP.
- Connectivity to a Linux /Unix Server which has got gcc and g++ compiler installed in it.

Instructions

- Create a directory by your name in the home directory. For each lab exercise create a directory as lab <lab number>.

Learning More (Bibliography)

- Linux application development – Michael K. Johnson
- Managing Projects with GNU make, 3rd Edition - Robert Mecklenburg
- http://www.network-theory.co.uk/docs/gccintro/gccintro_4.html

Lab 1. C Program Development in Unix

Goals	<ul style="list-style-type: none"> Learn and Understand the process of : <ul style="list-style-type: none"> Development of C program in the Unix environment
Time	180 minutes

1.1: Writing and executing a simple reverse program in C

Solution:

Step 1: Login to the Unix / Linux Server using **telnet** or **Putty**. Create a folder called **Lab1** under the home directory.

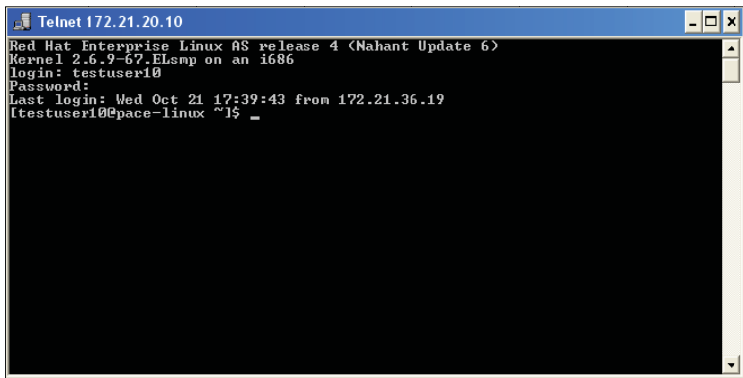


Figure 1: Telnet screen

Step 2: Using **VI editor**, create your C Program **reverse.c** under **Lab1** directory.

```
[testuser10@pace-linux Lab1]$ vi sample.c
```

Let us examine a C program that performs a simple task: reversing a string.

[**Note:** For better understanding, the Line numbers are also shown.]

```

1 #include <stdio.h>
2 /* Function Prototype */
3 int reverse ();
4 main ()
5 {
6   char str [100]; /* Buffer to hold reversed string */
7   reverse ("cat", str); /* Reverse the string "cat" */
8   printf ("reverse ("cat") = %s\n", str); /* Display */
9   reverse ("noon", str); /* Reverse the string "noon" */
  
```

```

10  printf ("reverse ("noon") = %s\n", str); /* Display */
11  }
12
13
14  reverse (before, after)
15
16  char *before; /* A pointer to the source string */
17  char *after; /* A pointer to the reversed string */
18
19  {
20  int i;
21  int j;
22  int len;
23
24  len = strlen (before);
25
26  for (j = len - 1; i = 0; j >= 0; j--; i++) /* Reverse loop */
27      after[i] = before[j];
28
29  after[len] = 0; /* terminate reversed string */
30  }

```

Example 1: C program for reversing a string

Step 3: Compile the Program.

[**Note:** To prepare an executable version of a single, self-contained program, follow gcc by the name of the source code file, which must end in a “.c” suffix. The gcc does not produce any output when the compilation is successful and it has no diagnostic comments. By default, gcc creates an executable file called “a.out” in the current directory. To run the program, type “a.out” (or “./a.out” if “.” is not in your search path). Any errors, which are encountered, are sent to the standard error channel, which is connected by default to your terminal’s screen.]

```

[testuser10@pace-linux Lab1]$ gcc reverse.c
reverse.c: In function 'main':
reverse.c:8: error: syntax error before "cat"
reverse.c:10: error: syntax error before "noon"
reverse.c: In function 'reverse':
reverse.c:26: error: syntax error before ';' token
reverse.c:26: error: syntax error before ')' token
[testuser10@pace-linux Lab1]$

```

Example 2: Compiling the program

As you can see, gcc found a number of compile-time errors:

The errors on lines 8 and 10 were due to inappropriate use of double quotes within double quotes.

The error on line 26 was due to an illegal use of a semicolon (;)

Step 4: Correct the errors by opening the file again in the VI editor.

The corrected version is as shown below:

```

1 #include <stdio.h>
2 /* Function Prototype */
3 int reverse ();
4 main ()
5 {
6   char str [100]; /* Buffer to hold reversed string */
7   reverse ("cat", str); /* Reverse the string "cat" */
8   printf ("reverse cat= %s\n", str); /* Display */
9   reverse ("noon", str); /* Reverse the string "noon" */
10  printf ("reverse noon = %s\n", str); /* Display */
11 }
12
13
14 reverse (before, after)
15
16 char *before; /* A pointer to the source string */
17 char *after; /* A pointer to the reversed string */
18
19 {
20   int i;
21   int j;
22   int len;
23
24   len = strlen (before);
25
26   for (j = len - 1, i = 0; j >= 0; j--, i++) /* Reverse loop */
27     after[i] = before[j];
28
29   after[len] = 0; /* terminate reversed string */
30 }

```

Example 3: Corrected version of the program

Step 5: Run the C Program.

```

[testuser10@pace-linux Lab1]$ gcc reverse.c
[testuser10@pace-linux Lab1]$ ./a.out
reverse cat= tac
reverse noon = noon

```

Example 4: Running the C program

Step 6: Override the default executable name.

[**Note:** The name of the default executable, “a.out”, is cryptic, and an “a.out” file produced by a subsequent compilation of a different program would overwrite the one that we produced. To avoid both problems, it is best to use the -o option with gcc, which allows you to specify the name of the executable file that you wish to create:]

```
[testuser10@pace-linux Lab1]$ gcc -o reverse reverse.c
[testuser10@pace-linux Lab1]$ ./reverse
reverse cat= tac
reverse noon = noon
```

Example 5: Overriding the default executable name

1.2: Learn to make a multi module Program

[**Note:** The trouble with the method in which we built the **reverse program** is that the **reverse function** cannot easily be used in other programs. For example, let us say that we want to write a function that returns 1 if a string is a **palindrome**, and 0 if it is not.

A **palindrome** is a string that reads the string forward and backward; for example, “noon” is a palindrome, and “nono” is not. We can use the **reverse function** to implement my **palindrome function**. One way to do this is to cut-and-paste **reverse ()** into the **palindrome program file**. However, this is a poor technique for at least three reasons:

1. Performing a cut-and-paste operation is tedious.
2. If we come up with a better piece of code for performing a reverse operation, then we will have to replace every copy of the old version with the new version. This is a maintenance nightmare.
3. Each copy of **reverse ()** uses up disk space.]

Step1: Prepare a reusable **reverse function**.

Step 2: Create a **reverse1.h** header file that contains the Prototype:

```
/* REVERSE.H */
int reverse (); /* Declare but do not define this function */
```

Example 6: reverse1.h

Step 3: Create a program **reverse1.c**.

```
/* REVERSE.C */
#include <stdio.h>
#include "reverse1.h"
reverse (before, after)
char *before; /* A pointer to the original string */
char *after; /* A pointer to the reversed string */
{
    int i;
    int j;
    int len;
    len = strlen (before);
    for (j = len - 1, i = 0; j >= 0; j--, i++) /* Reverse loop */
        after[i] = before[j];
}
```

```
after[len] = 0; /* terminate reversed string */
}
```

Example 7: reverse1.c program

Step 4: Create the program main1.c which invokes the reverse1.c program.

```
/* MAIN1.C */
#include <stdio.h>
#include "reverse.h" /* Contains the prototype of reverse () */
main ()
{
    char str [100];
    reverse ("cat", str); /* Invoke external function */
    printf ("reverse (\\"cat\\") = %s\\n", str);
    reverse ("noon", str); /* Invoke external function */
    printf ("reverse (\\"noon\\") = %s\\n", str);
}
```

Example 8: main1.c program

Step 5: Separately compile and link modules:

```
[testuser10@pace-linux Lab1]$ gcc -c reverse1.c //compile reverse.c to reverse.o.
[testuser10@pace-linux Lab1]$ gcc -c main1.c // compile main1.c to main1.o
[testuser10@pace-linux Lab1]$ gcc reverse1.o main1.o -o reverse1
[testuser10@pace-linux Lab1]$ ./reverse1
reverse ("cat") = tac
reverse ("noon") = noon
```

<<TO DO>>

Write a Palindrome checking Program in C which uses the reverse function created above.

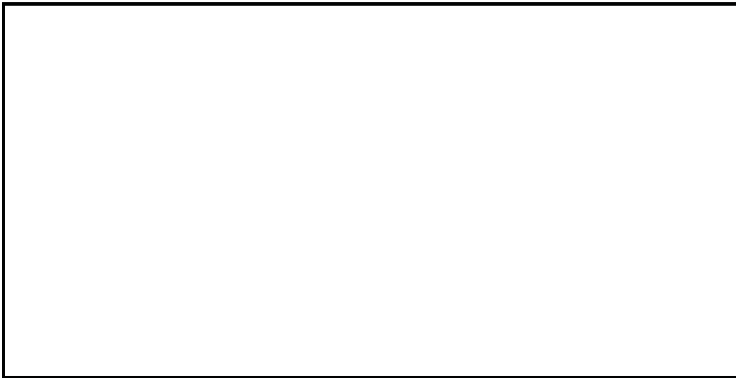
1.3 Learn to generate static link library for a program.

Step 1: Create a **factorial.h** header file that contains the Prototype:

```
/* FACTORIAL.H */
/*program contains Prototype of factorial function*/
long int factorial(int);
```

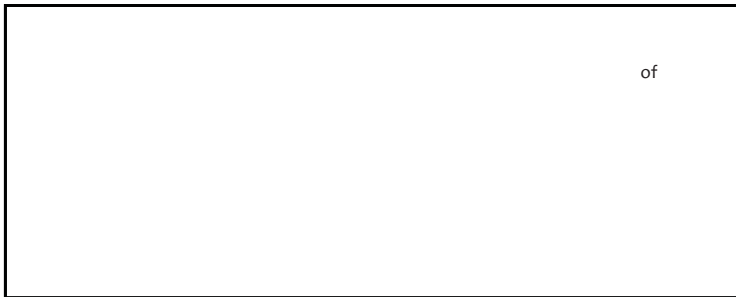
Example 9: factorial.h program

Step 2: Create a program **factorial.c**.



Example 10: factorial.c program

Step 3: Create the program main.c which invokes the factorial.c program.



Example 11: main.c program

Step 4: Separately compile and link modules:

- Create object file factorial.o

```
[testuser10@pace-linux Lab1]$ gcc -c factorial.c
```

- Creates an archive library, libfact.a, which consists of the named object files.

```
[testuser10@pace-linux Lab1]$ ar -cr libfact.a factorial.o
```

- Generate output file main.o using static linking. Use the -l option to link your program with libfact.a

The link editor incorporates in your executable only the object files in this archive that contain a function you have called in your program.

Options used in command:

- L specifies the directory name where to search libfact.a file
- o specifies name of executable file
- l to link output file with libfact.a file

```
[testuser10@pace-linux Lab1]$ gcc -L. -o main.o main.c -lfact
```

- Run the program

```
[testuser10@pace-linux Lab1]$ ./main.o
```

<<TO DO>>

Write a bubble sort program in C which uses a sorting functionality. Compile and link the program by creating static link library.

1.4 Learn to generate dynamic link library for a program.

Step 1: Create a **message.h** header file that contains the Prototype:

Example 12: message.h program

Step 2: Create a program **welcome.c**.

Example 13: welcome.c program

Step 3: Create a program **success.c**.



Example 14: success.c program

Step 4: Create a program **main.c**.



Example 15: main.c program

Step 5: Compile and link the program by creating dynamic link library.

- Create Library
[testuser10@pace-linux Lab1]\$ gcc -Wall -fPIC -c welcome.c success.c

- Link object files into a shared library named as "libmessage.so"
[testuser10@pace-linux Lab1]\$ gcc -shared -o libmessage.so welcome.o success.o

- Create executable file main.o by setting the environment variable to search for the library file while linking.

```
[testuser10@pace-linux Lab1]$ LD_RUN_PATH=/home/testuser10 export
LD_RUN_PATH
```

```
[testuser10@pace-linux Lab1]$ gcc -o main.o main.c -L/home/testuser10 welcome.c
success.c -lmessage
```

- Run the Program.
./main.o

Hello, John!

Congratulations on your best performance!

<<TO DO>>

Write a program given an integer value, check the number is a prime number or an Armstrong number. Compile and link the program by creating dynamic link library using rpath to locate library file.

Lab 2. C++ Program Development in Unix

Goals	<ul style="list-style-type: none">At the end of this lab session, you will be able to learn:<ul style="list-style-type: none">Learn and Understand the process of :Development of C++ program in the Unix environment
Time	180 minutes

2.1: Writing a simple C++ program that displays “Hello World”

Solution:

Step 1: Create a Program called “Hello World.cpp” using the VI editor.

```
#include <iostream>
using namespace std;

int main ()
{
    cout << "Hello, world!\n";
    return 0;
}
```

Example 16: Hello World.cpp

[**Note:** To use the g++ compiler, the name of the file should end with a suffix “**.cpp**” (or one of a half-dozen other extensions such as ccp, C, cp, and cxx). Check your documentation for more details.

About the std namespace: The “**std**” namespace is a recent addition to C++.]

Step 2: Compile your Program.

```
[testuser10@pace-linux Lab1]$ g++ -Wall helloworld.cc -o hello
```

[**Note:** The g++ uses many of the same options as the C compiler gcc. The “-Wall” tells the compiler to give warning messages about all constructions that the compiler considers suspicious. These warnings are not errors and will not stop the compiler from producing an executable. They indicate possible errors in programming logic. However, they can be ignored if you know that your code is correct.]

Step 3: Execute your program.

```
[testuser10@pace-linux Lab1]$./hello  
Hello, world!
```

Example 17: Executing the program

2.2: <TO DO>> Create a simple “shape” hierarchy:

Create a simple “shape” hierarchy, namely a base class called Shape and derived classes called Circle, Square, and Triangle.

In the base class, make a virtual function called draw(), and override this in the derived classes. Make an array of pointers to Shape objects that you create on the heap (and thus perform upcasting of the pointers), and call draw() through the base-class pointers, to verify the behavior of the virtual function.

2.3: <<TO DO>> Templatize the fibonacci() function

Templatize the **fibonacci**() function on the type of value that it produces (so it can produce long, float, and so on instead of just **int**).

Lab 3. Debugging your C/C++ program

Goals	<ul style="list-style-type: none"> At the end of this lab session, you will be able to: <ul style="list-style-type: none"> Debug your C and C++ program
Time	60 minutes

3.1: To debug your C/C++ Program

Step 1: Create a program called **sample.cpp** using VI editor.

```
#include <iostream>

using namespace std;

int divint(int, int);

int main() {
    int x = 5, y = 2;
    cout << divint(x, y);
    x = 3; y = 0;
    cout << divint(x, y);
    return 0;
}

int divint(int a, int b)
{
    return a / b;
}
```

Example 18: sample.cpp program

Step 2: Compile the program and execute

```
[testuser10@pace-linux Lab1]$ g++ sample.cpp
[testuser10@pace-linux Lab1]$ ./a.out
Floating point exception
```

Example 18: Compiling the program

Step 3: To debug the program, enable the **-g** option.

```
[testuser10@pace-linux Lab1]$ g++ -g sample.cpp -o crash
```

Example 19: Enabling the -g option

Step 4: Debug the program using **gdb**.

```
[testuser10@pace-linux Lab1]$ gdb crash
GNU gdb Red Hat Linux (6.3.0.0-1.153.el4rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
```

Type "show copying" to see the conditions.

```
[testuser10@pace-linux Lab1]$ gdb crash
GNU gdb Red Hat Linux (6.3.0.0-1.153.el4rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
```

Type "show copying" to see the conditions.

There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db li
rary "/lib/tls/libthread_db.so.1".

```
(gdb) r
Starting program: /home/testuser10/Lab1/crash
```

```
Program received signal SIGFPE, Arithmetic exception.
0x0804874b in divint (a=3, b=0) at sample.cpp:17
17     return a / b;
```

```
# 'r' runs the program inside the debugger
# In this case the program crashed and gdb prints out some relevant information.
# In particular, it crashed trying to execute line 17 of sample.cpp.
# The function parameters 'a' and 'b' had values 3 and 0 respectively.
```

```
(gdb) l
12     return 0;
13 }
14
15 int divint(int a, int b)
16 {
17     return a / b;
18 }
19
# l is short for 'list'. Useful for seeing the context of the crash, lists code lines near #
around 17 of sample.cpp
```

```
(gdb) where
#0 0x0804874b in divint (a=3, b=0) at sample.cpp:17
#1 0x0804871d in main () at sample.cpp:11
```

```
# Equivalent to 'bt' or backtrace. Produces what is known as a 'stack trace'.
# Read this as follows: The crash occurred in the function divint at line 17 of
# sample.cpp. This, in turn, was called from the function main at line 11 of
```

```
# sample.cpp

(gdb) up
#1 0x0804871d in main () at sample.cpp:11
11      cout << divint(x, y);
      # Move from the default level '0' of the stack trace up one level to level 1.

(gdb) p x
$1 = 3
(gdb) p y
$2 = 0
(gdb) q
The program is running. Exit anyway? (y or n) y
```

In this example, it is fairly obvious that the crash occurs because of the attempt to divide an integer by 0.

3.2: <<TO DO>> Debug the following C++ program

```
#include <iostream>
using namespace std;
void setint(int*, int);
int main() {
    int a;
    setint(&a, 10);
    cout << a << endl;
    int* b;
    setint(b, 10);
    cout << *b << endl;
    return 0;
}
void setint(int* ip, int i) {
    *ip = i;
}
```

Example 20: C++ program

3.3: <<To DO>> Debug the following C++ Program

The program has some logical errors. The program is supposed to output the summation of $(X^0)/0! + (X^1)/1! + (X^2)/2! + (X^3)/3! + (X^4)/4! + \dots + (X^n)/n!$, given x and n as inputs. However the program outputs a value of infinity, regardless of the inputs.

```
#include <iostream>
#include <cmath>

using namespace std;

int ComputeFactorial(int number) {
    int fact = 0;
```

```

    for (int j = 1; j <= number; j++) {
        fact = fact * j;
    }

    return fact;
}

double ComputeSeriesValue(double x, int n) {
    double seriesValue = 0.0;
    double xpow = 1;

    for (int k = 0; k <= n; k++) {
        seriesValue += xpow / ComputeFactorial(k);
        xpow = xpow * x;
    }

    return seriesValue;
}

int main() {
    cout << "This program is used to compute the value of the following series : " << endl;

    cout << "(x^0)/0! + (x^1)/1! + (x^2)/2! + (x^3)/3! + (x^4)/4! + ..... + (x^n)/n! " << endl;

    cout << "Please enter the value of x : " ;

    double x;
    cin >> x;

    int n;
    cout << endl << "Please enter an integer value for n : " ;
    cin >> n;
    cout << endl;

    double seriesValue = ComputeSeriesValue(x, n);
    cout << "The value of the series for the values entered is "
    << seriesValue << endl;

    return 0;
}

```

Example 21: C++ program

3.4: Debug the following C Program

The program should print out all the alphanumeric (letter and number) characters in its input.

```

#include <stdio.h>
#include <ctype.h>
int main(int argc, char **argv)

```



```
{
char c;
c = fgetc(stdin);
while(c != EOF){
    if(isalnum(c))
        printf("%c", c);
    else
        c = fgetc(stdin);
}

return 1;
}
```

Example 22: C program

3.5: Debug the following C Program

The program is meant to read in a line of text from the user and print it.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv)
{
    char *buf;
    buf = malloc(1<<31);
    fgets(buf, 1024, stdin);
    printf("%s\n", buf);
    return 1;
}
```

Example 23: C program

Lab 4. Makefile Utility

Goals	<ul style="list-style-type: none"> At the end of this lab session, you will be able to: <ul style="list-style-type: none"> Learn how to create a makefile to automate the build process Understand the steps involved in creating and executing Makefile using make utility
Time	120 minutes

4.1: Creating a makefile for Lab exercise 1.2

Step 1: Create a file called **Makefile** using the VI editor.

```
main1: main1.o reverse1.o
    gcc main1.o reverse1.o -o main1
main1.o: main1.c reverse1.h
    gcc -c main1.c
reverse1.o: reverse1.c reverse1.h
    gcc -c reverse1.c
```

Example 24: Makefile file

Step 2: Execute the **Makefile**.

```
[testuser10@pace-linux Lab1]$ make
gcc -c main1.c
gcc -c reverse1.c
gcc main1.o reverse1.o -o main1

[testuser10@pace-linux Lab1]$ ./main1
reverse ("cat") = tac
reverse ("noon") = noon
```

Example 25: Executing the Makefile

Step 3: Again execute make, and check.

```
[testuser10@pace-linux Lab1]$ make -f m
make: 'main1' is up to date.
```

Example 26: Executing make

Step 4: Copy the **makefile** to another file called **mymakefile** and execute it.

```
[testuser10@pace-linux Lab1]$ make -f mymakefile

gcc main1.o reverse1.o -o main1

[testuser10@pace-linux Lab1]$ ./main1
reverse ("cat") = tac
reverse ("noon") = noon
```

Example 27: Executing the Mymake file

Step 5: Modify your **Makefile** with Macros.

```
all: main1

#Which Compiler
CC = gcc

#Options for development
CFLAGS = -g -Wall

#Options for release
#CFLAGS = -Wall

main1: main1.o reverse1.o
    $(CC) main1.o reverse1.o -o main1
main1.o: main1.c reverse1.h
    $(CC) -c main1.c
reverse1.o: reverse1.c reverse1.h
    $(CC) -c reverse1.c
clean:
    rm *.o
```

Example 28: Modifying Makefile

Step 6: Execute the **Makefile**, and check the output

Step 7: Execute the **Makefile** with the **clean** target, and check that all the object files are deleted.

```
[testuser10@pace-linux Lab1]$ make clean
rm *.o
```

Example 29: Executing Makefile with clean target

4.2: <<TODO>> Create a Makefile for all the C and C++ programs which are made of multiple modules.

Appendices

Appendix A: Table of Figures

Figure 1: Telnet screen.....5

Appendix B: Table of Examples

Example 1: C program for reversing a string	6
Example 2: Compiling the program	6
Example 3: Corrected version of the program	7
Example 4: Running the C program	7
Example 5: Overriding the default executable name	8
Example 6: reverse1.h	8
Example 7: reverse1.c program	9
Example 8: main1.c program	9
Example 9: factorial.h program	9
Example 10: factorial.c program	10
Example 11: main.c program	10
Example 12: message.h program	11
Example 13: welcome.c program	11
Example 14: success.c program	12
Example 15: main.c program	12
Example 16: Hello World.cpp	14
Example 17: Executing the program	15
Example 19: Compiling the program	16
Example 20: Enabling the -g option	16
Example 21: C++ program	18
Example 22: C++ program	19
Example 23: C program	20
Example 24: C program	20
Example 25: Makefile file	21
Example 26: Executing the Makefile	21
Example 27: Executing make	21
Example 28: Executing the Mymake file	22
Example 29: Modifying Makefile	22
Example 30: Executing Makefile with clean target	22