Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

C CPP ON UNIX
Lab Book

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

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	• Learn and Understand the process of : ○ 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.

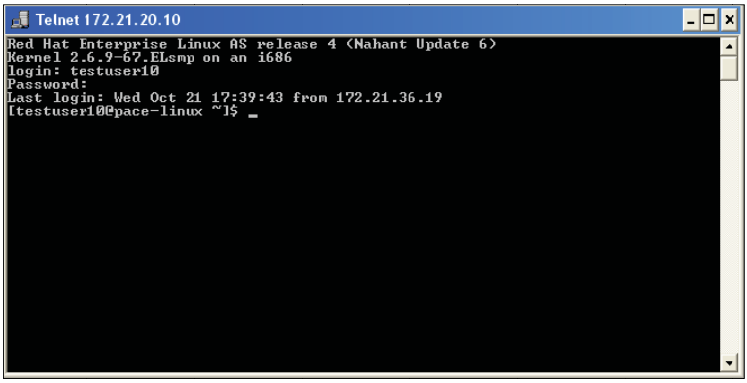


Figure 1: Telnet screen

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.

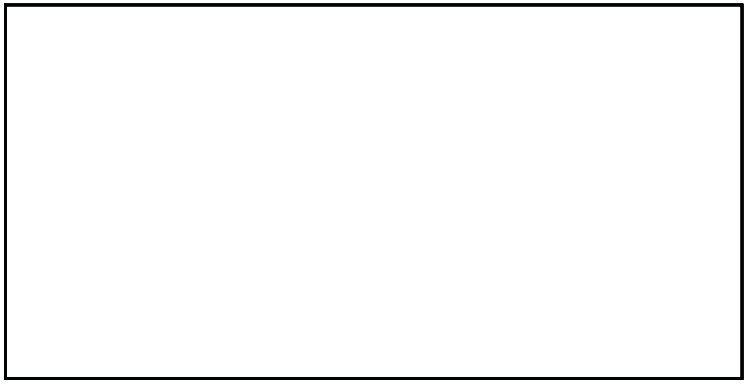
1.3 Learn to generate static link library for a program.

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.

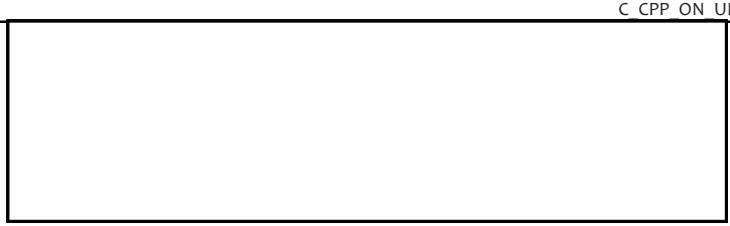
Step 1: Create a **message.h** header file that contains the Prototype:

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	• At the end of this lab session, you will be able to learn: o Learn and Understand the process of : o 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	• At the end of this lab session, you will be able to: ○ 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! + ... + (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	• At the end of this lab session, you will be able to: ○ 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

Appendix B: Table of Examples