Support for developing C programs

A programming environment (e.g. UNIX) provides many tools that aid writing /debugging programs.

This lecture is an introduces to this topic by discussing role of header files, make, gdb and cvs.

Compiling C programs

To compile a program consisting of several files of code

compile the individual files to their object code
i.e turn .c files to .o
link all files to the final executable file
link if any libraries are used

Compiling C programs

e.g. gcc -c prog1.c
creates a relocatable object file named 'prog1.o'
gcc -c prog2.c
creates the file prog2.o
gcc prog1.o prog2.o -o myprog
links prog1.o and prog2.o and loads into
an executable program myprog.

gcc prog1.c prog2.c -o myprog
compiles and links both files in a single step

the advantage of creating the object files explicitely is that the whole program need not be recompiled if changes are made to prog1.c alone. only prog1.c is to be recompiled and the final linking is repeated.

Compiling Large C programs

Very large C programs will consists of
- many source files (.c files)
- many include files (.h files)

Every time a .c file is changed , we have to recompile it
Every time a .h file is changed , we have to recompile all the .c files that included this .h file. This .h file might have been included in another .h file, so we have to recompile the files that the latter .h file is included in.

This necessitates
- To recompile all the source files that have changed
- To know how files depend on each other
- Recompiling the unchanged files is a waste of time

Make

The make program is a software tool that can be used to keep track of which files need recompiling after any changes have been made and to issue the sequence of commands that performs all the necessary recompilations

make program keeps the C programs up to date

The make program needs a configuration file called Makefile

Makefile contains a list of all of the files which comprise the program and the rules which describe how to compile or build the program from the sources

How Make Works - A simple example

We have prog1.c and prog2.c which are to be compiled and linked and the final executable file is myprog. We have the Makefile also in the current directory

The Makefile contains

```
myprog:   prog1.o  prog2.o
        gcc prog1.o  prog2.o  -o myprog

prog1.o:  prog1.c
        gcc -c prog1.c

prog2.o:  prog2.c
        gcc -c prog2.c
```

Now we give the command

$make

How Make Works - A simple example

The first time we compile only the '.c ' files

When we type 'make' the program looks at its rules in the Makefile and finds that it has to make a file called 'myprog'. To make this it needs to execute the command

gcc prog1.o prog2.o -o myprog

so it looks for the .o files and does not find them.
So it checks the Makefile to see if there are any rules to get the .o files and discovers that these can be made by compiling with the 'gcc -c'

Now all the files exist, and it can execute the first rule to build 'myprog'

How Make Works - A simple example

If we now edit 'prog1.c' and type 'make' it goes through the same procedure as before but now it finds all the files. So it compares the dates on the files. If the source is newer than the result, it recompiles

By using this recursive method, make only recompiles those parts of the program which need recompiling

Writing Makefile

To write a Makefile we have to tell make about dependencies.

The dependencies of a file are all those files which are required to build it

e.g. The dependencies of myprog are prog1.o and prog2.o
The dependencies of prog1.o is prog1.c and the dependencies of prog2.o is prog2.c

A Makefile consists of rules of the form
target : dependencies
<tab> rule;

The target is the thing we want to build (goal) and dependencies are like subgoals. The rule is to be executed if all of the dependencies exist. It takes the dependencies and turns them into the target

Structure of a Makefile

Makefile contains

- the dependencies
- the rules

The Makefile consists of rules of the form
target: dependencies
<tab> rule;

Remember that
- the file names

start on the first character of a line
- tab character at the beginning of every rule

Makefile

```
# Makefile to create  'myprog'
CFLAGS = -Wall -ansi -pedantic -g
LFLAGS  = -g -lm
CC = gcc
OBJS = prog1.o  prog2.o
PROGRAM = myprog

myprog:    $(OBJS)
        $(CC)  $(LFLAGS)  $(OBJS) -o $(PROGRAM)

prog1.o: prog1.c
        $(CC)  $(CFLAGS)  -c prog1.c -o prog1.o

prog2.o: prog2.c
        $(CC)  $(CFLAGS)  -c prog2.c -o prog2.o
clean:
        rm   $(OBJS)
```

gcc options

gcc -Wall -ansi -pedantic -g -lm prog1.c prog2.c -o myprog

where

-Wall = give all warnings
-pedantic = follow the standard pedantically
-ansi = the ansi standard is compulsory
-g = compile the debug information for the gdb program
-lm = link the mathematics library if necessary
-o = the executable file is called myprog

See man gcc for more information

Conditional Compilation

Conditional compilation allows statements to be included or omitted based on conditions at compile time

The following preprocessor directives are used for conditional compilation

#if	#else
#elif	#endif
#ifdef	#ifndef

The **#if** line evaluates a constant integer expression.
If the expression is non-zero subsequent lines until **#endif** or **#elif** or **#else** are included.

Conditional Compilation

The sequence tests the name SYSTEM to decide which version of the header to include

```
#if  SYSTEM  == SYSV
      #define HDR  "sysv.h"
#elif  SYSTEM  == BSD
      #define HDR  "bsd.h"
#if  SYSTEM  == MSDOS
      #define HDR  "msdos.h"
#else
      #define HDR  "default.h"
#endif

#include HDR
```

Conditional Compilation

To ensure that the contents of a file hdr.h are included only once, the contents of the file hdr.h are surrounded with a conditional

```
#ifndef HDR
#define HDR
/* contents of  hdr.h  go here */
#endif
```

Conditional Compilation

In this example the printf statements are compiled when the symbol DEBUG is defined, but not compiled otherwise

```
/* remove to suppress debug printf 's */
#define DEBUG
...
        x =  ...
#ifdef  DEBUG
        printf ("x = %d\n");
#endif
...
        y =  ...
#ifdef  DEBUG
        printf ("y = %d\n");
#endif
...
```

gdb (The GNU Debugger)

gdb is a symbolic debugger which makes writing and debugging the programs easier.

To use gdb it is essential to compile the program with -g option.

eg. gcc -g bug.c -o bug

To run the debugger give the command

```
gdb bug
```

To set breakpoint

```
break bug.c:20  /* to set break point in bug.c at line 20 */
break fun  /* to set break point at the function fun */
```

gdb (The GNU Debugger)

Set watchpoints

```
watch i    /*  will cause the program to stop if i changes */
awatch i   /* will cause the program to stop if i is accessed */
```

Print values

```
print i  /* print the value of the variable i */
print num[i]   /* print the value of the variable num[i] */
print strcmp(str1, str2) /* print the return value of the
strcmp(str1, str2). This can be used to determine values of
any function.
```

Step

```
step -s     /* advance to the next source line */
```

To get help on any specific command type help command name
e.g. help break

gdb (The GNU Debugger)

help
List of classes of commands:
aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
...

To get help on any specific command type help command name
e.g. help data

q to quit gdb

CVS - Concurrent Versions System

CVS is an open standard for version control

CVS is the dominat open-source network-transparent version control system.

CVS is useful for everyone from individual developers to large distributed teams:

As a version control system CVS keeps a history of changes made to a set of files , this enables one to keep track of all the changes made to a program during the entire development cycle

Ref. http://www.cvshome.org