LINUX_

Development Tools, Connections, Installing, Libraries, and MakeFiles

Agenda

1. Development Tools
2. Connecting to Linux
3. Installing Software
4. Libraries
5. Makefiles

Development Tools - Compilers (1)

- Open Source
  - C/C++/FORTRAN
    - GNU C compiler (gcc)
    - GNU C++ compiler (g++)
    - G77
  - Java
    - Sun and IBM JDK
    - gcj – GNU java compiler
    - Jikes - java source to bytecode compiler.
  - Pascal
    - http://www.freepascal.org/
  - C#
    - Mono 2.8

Development Tools – Compilers (2)

- Shareware
  - C
    - Compaq C - http://www.compaq.com/
  - C++
    - Intel - http://www.intel.com/cd/software/products/asmo-na/eng/compilers/277618.htm
    - KAI (Kuck and Associates)
    - Fujitsu C/C++ Empress
  - FORTRAN
    - Portland Group Compilers
    - HP/Compaq Fortran
    - Fujitsu FORTRAN Empress
  - Java
    - Tower Technology: TowerJ - Compiles JAVA to native code. Will also compile JAVA byte code to native binary code.

Development Tools – Debuggers

- **gdb** - GNU debugger. Command driven text/console interface.
- **xxgdb** - X window system interface to the GNU debugger.
- **DDD** - Data Display Debugger. GUI interface for gdb and dbx.
- **GVD** - GtkAda/GNAT Graphical GDB Debugger Interface.
- **KDbg** - K Desktop Graphical GDB Debugger Interface.
- **ups** - X11 Source Debugger Interface. Native debugger and not a front-end to gdb.
- **RHIDE** - Console mode with windows like Borland 3.1 toolset.
- **Insight** - GUI interface to gdb. Works with Source-Navigator IDE.
- **Xwpe** - Borland C++ console mode IDE clone.
- **Etnus TotalView** - Specialized in debugging multi-threaded software with memory leak detection. Also MPI/OpenMP debugging facilities.

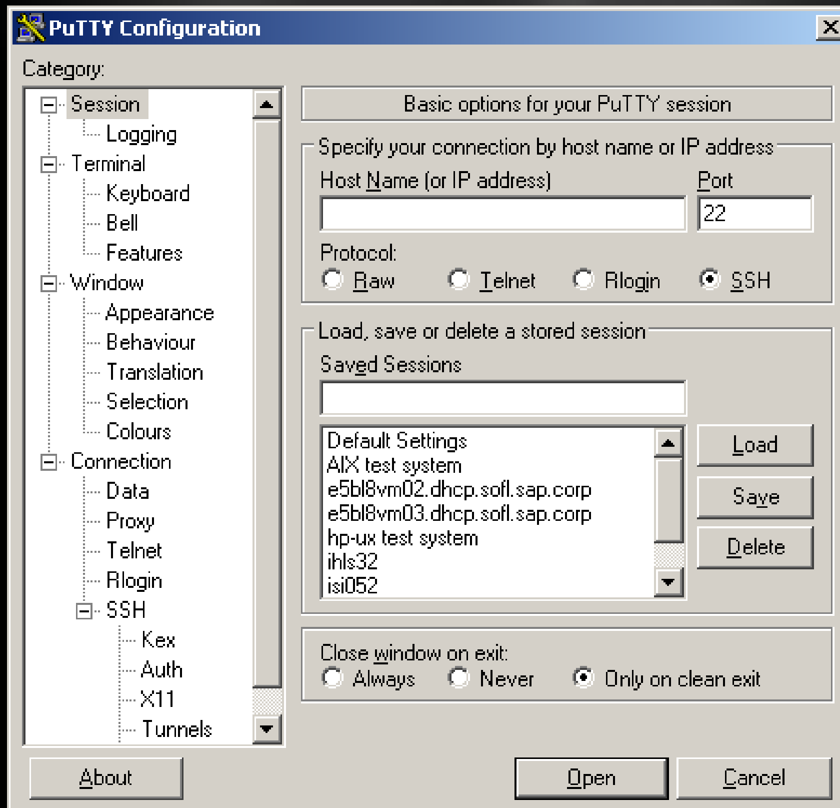
Development Tools – IDE

- Eclipse.org - IBM open source JAVA and C/C++ (with CDT plug-in) IDE. Extensible IDE consortium - Borland, IBM, Red Hat, Rational. Lots of industry backing. Also see EclipsePluginCentral.com Plugins available for Subversion SCM, XML documents, HEX, ...
- Anjuta - C, C++. Written for GTK/Gnome. Solid, simple, intuitive, bug free IDE for C/C++ development on Linux. Search/Indexing, edit, compile and debug.
- KDevelop.org - C++ KDE IDE
- Sun Studio - C/C++, FORTRAN IDE for Linux.
- Source Navigator - C/C++, FORTRAN, COBOL, Tcl, JAVA, asm editor, cross reference tool, class browser and IDE.
- wxStudio - C++ cross platform IDE. Written for wxWindows cross platform GUI framework.
- Moonshine - C/C++. IDE supports Qt. More of an editor than anything else.
- DiaSCE - C/C++ Gnome code editor. Integrated with Glade GTK GUI builder.
- MonoDevel 2.4 – IDE for developing .NET applications.

Development Tools – CM

- **Subversion / Trac** - Subversion is a version control CVS replacement.
- **IBM/Rational: Clearcase** - Commercial product.
- **RCS** - Revision Control System.
- **CVS** - Concurrent Versions System.
- **Bonsai/LXR/Glimpse** - Web front-end CVS browsing and indexing engine for CVS.
- **Ximian Red Carpet** - Automated Software Maintenance and Version Management.
- **PVCS** - Version Manager. Commercial product.
- **BitKeeper** - Commercial product.
- **Perforce** - Commercial product.
- **Serena** - Change management software. Source code, web content, ... Commercial product.
- **Alodon Lifecycle Manager** - Enterprise Software Configuration, Change Management, deployment.
- **SourcePuller** - Bitkeeper compatable source code management client.
- **GIT** - Distributed source code management. Written in "C" and developed by Linus Torvalds for use with the Linux Kernel. Concept roots inspired by Bit Keeper.
- **Mercurial** - Almost the same as GIT but written in Python and based on a different data management system. Supports Maven build system.

Connecting to Linux/UNIX System (1)



- The most widely used protocols:
  - Telnet
  - SSH protocol – Putty, WinSCP
  - FTP – sftp, FileZilla

Connecting to Linux/UNIX System (2)

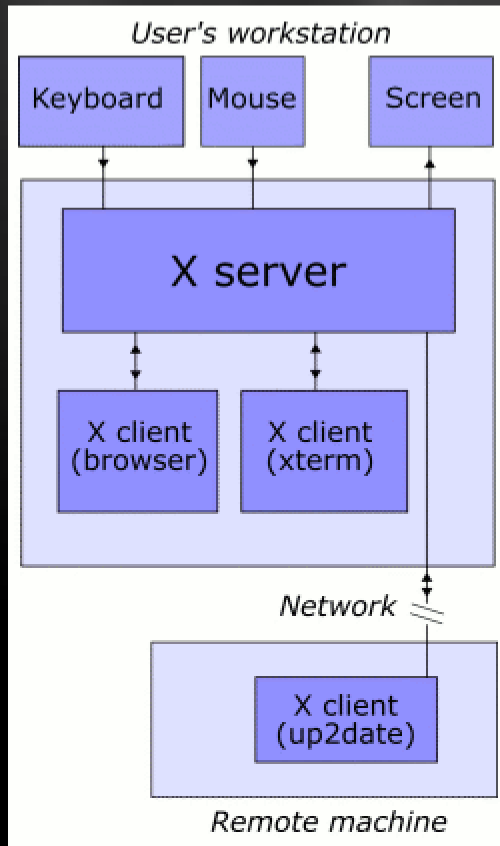
```
C:\Documents and Settings\i049043\Desktop\linux\psftp.exe
psftp: no hostname specified; use "open host.name" to connect
psftp> _
```

- Uploading files with psftp:
  - Connecting to a host
    open [host]
  - Downloading files
    mget [file name]
  - Uploading files
    mput [file name]

Connecting to Linux/UNIX - Visually

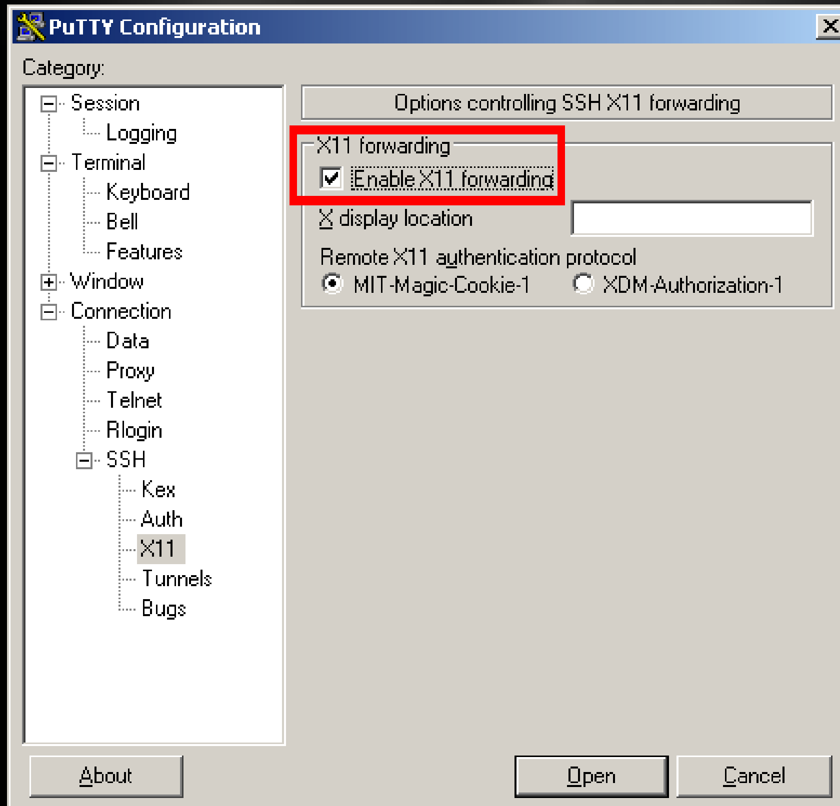
- Three ways to connect to Linux visually
  - Using SSH and X11 Forwarding.
  - Using SSH and exporting your DISPLAY.
  - Using VNC
- Software needed for graphical network connections:
  - SSH client – putty
  - Xserver – Xming, Xcead, Cygwin
  - VNC Server – vncserver
  - VNC Viewer – tightvnc, vncviewer, realvnc...

Connecting to Linux/UNIX - Xserver



In this example, the X server takes input from a keyboard and mouse and displays to a screen. A web browser and a terminal emulator run on the user's workstation, and a system updater runs on a remote server but under the control of the user's machine. Note that the remote application runs just as it would locally.

Connecting to Linux/UNIX with X11 Forwarding



- Start the X Server on the local Windows machine
- Allow remote hosts to connect to the X Server
  xhost + [host]
  xhost – [host]
- Start putty
- Enable X11 forwarding
- Connect to the remote host and start any visual application
  Tip: Test your connection with xclock or xeyes

Connecting to Linux/UNIX with X11

e5bl8vm03.dhcp.sofl.sap.corp - PuTTY

```
login as: ims
Using keyboard-interactive authentication.
Password:
Last login: Mon Sep 20 18:31:07 2010 from sofd60173292a.dhcp.sofl.sa
ims@E5BL8VM03:~> export DISPLAY=10.55.66.73:0.0
ims@E5BL8VM03:~> xclock
```

xclock

- Start the X Server on the local Windows machine
- Allow remote hosts to connect to the X Server
  xhost + [host]
  xhost – [host]
- Start putty
- Connect to the remote host
- Export your display
  export DISPLAY=10.0.0.66:0.0
- Start any visual aplication
  Tip: Test your connection with xclock or xeyes

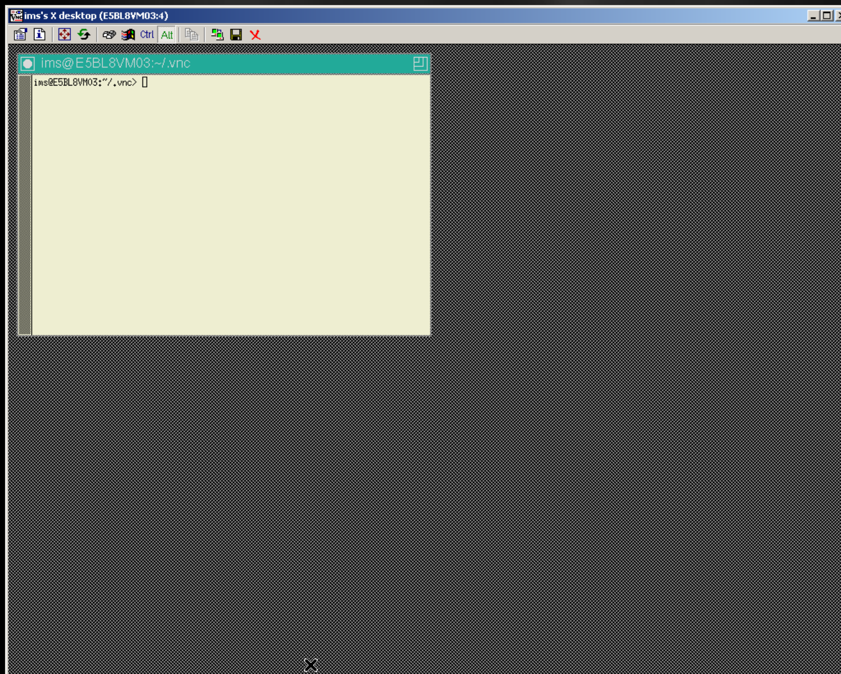
Connecting to Linux/UNIX with VNC

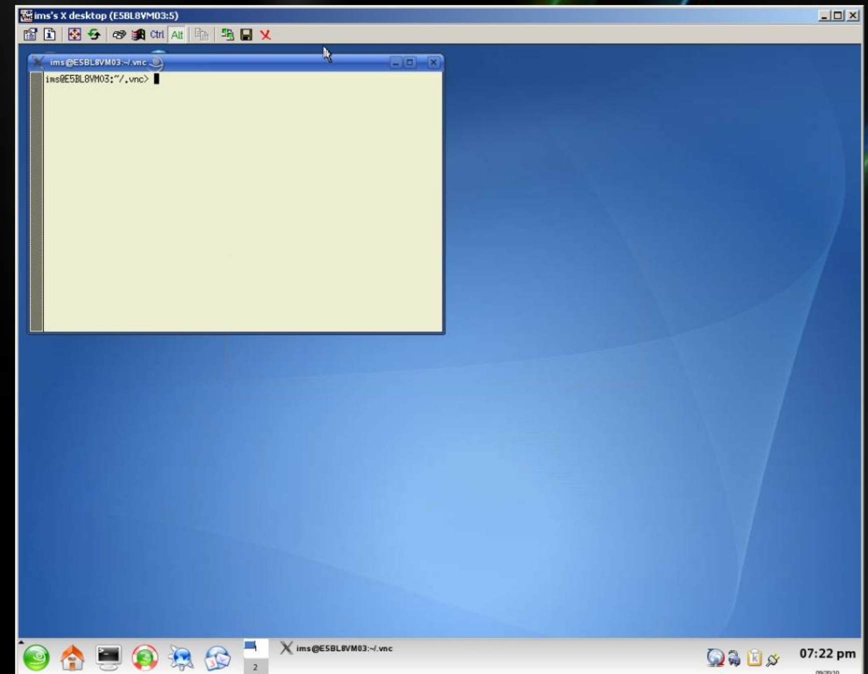
- Virtual Network Computing (VNC)
- Remote graphical protocol optimized for slow networks (uses compression).
- TightVNC (better compression) clients and servers are preferred (and backward compatible) to RealVNC (original older protocol).
- Servers and clients are available for many platforms.
- vncserver is a client to the X server.
- vncpasswd sets the password of the vncserver.
- vncviewer connects to the vncserver and provides visualization.

Window Managers for X (1)

twm (default for VNC)

KDE





Window Managers for X (2)

XFCE

gnome

Installing Software – From Source

- Download the archive with the source files.
  - wget http://www.muppetlabs.com/~breadbox/pub/software/cgames-2.2.tar.gz
- Extract the downloaded archive.
  - tar –xzf <file-name>
- Change to the extracted folder.
  - cd <directory-name>
- Configure the installation.
  - ./configure --prefix=/where/to/install/the/command
- Compile the source.
  - make
- Install the application
  - make install

Installing Software - RPM

- **rpm -Uvh vim-6.2-i386.rpm** - upgrade package vim
- **rpm -ivh vim-6.2-i386.rpm** - install package vim
- **rpm -e vim** - remove package vim
- **rpm -qa "vi*"** - show all packages starting with vi
- **rpm -qi vim** - show info about package vim
- **rpm -ql vim** - list files about package vim
- **rpm -qf `which vi`** - shows package of command vi
- **rpm -qR vim** - shows packages on which package vim depends

Installing Software – From Repository

- **Repository** - A repository is a central place where data is stored and maintained.
- **apt-get** – The default repository manager for Debian based distributions.
  - apt-get update [package] – check for new releases
  - apt-get upgrade [package] – install new or updated packages if there are any
  - apt-get install <package> - install a single package from the repository
- **yum** - The default repository manager for RedHat based distributions.
  - yum upgrade - install new or updated packages if there are any
  - yum install <package> - install a single package from the repository
  - yum list installed – show all installed packages

Compiling Your Code

- Compiling C/C++ source
  - **gcc/g++** [–o <exec_name>] [-c] [-fopenmp] *.c *.h
  - **o** – specify the name of the executable file. By default it will be "a.out".
  - **fopenmp** – include the OpenMP libraries and interpret the omp pragmas.
  - **c** – only compile the code without linking it.
- Start your executable.
  - ./<exec_name> <parameters>

Libraries

- This methodology, also known as "shared components" or "archive libraries", **groups together** multiple compiled object code files into a **single file known as a library**.
- Typically C functions/C++ classes and methods which can be **shared by more than one application** are broken out of the application's source code, compiled and **bundled into a library**.
- Components which are large can be created for dynamic use, thus the library remain **separate from the executable reducing it's size** and thus disk space used.
- The library components are then called by **various applications** for use **when needed**.

Libraries - Types

- There are two Linux C/C++ library types which can be created:
  - **Static libraries (.a)** - Library of object code which is linked with, and becomes part of the application.
  - **Dynamically linked shared object libraries (.so)** - There is only one form of this library but it can be used in two ways.
    - **Dynamically linked at run time but statically aware.** The libraries must be available during compile/link phase. The shared objects are not included into the executable component but are tied to the execution.
    - **Dynamically loaded/unloaded and linked during execution** (i.e. browser plug-in) using the dynamic linking loader system functions.

Library Naming Conventions

- Libraries are typically names with the prefix **"lib"**. This is true for all the C standard libraries. **When linking**, the command line reference to the library **will not contain the library prefix or suffix**.
- Thus the following link command:
  - gcc src-file.c **-lm -lpthread**
    - The libraries referenced in this example for inclusion during linking are the math library and the thread library. They are found in /usr/lib/lib**m**.a and /usr/lib/lib**pthread**.a.

Static Libraries (.a)

- How to generate a library:
  - **Compile**: gcc -Wall -c ctest1.c ctest2.c
  - **Create library** "libctest.a": ar -cvq libctest.a ctest1.o ctest2.o
  - **List files in library**: ar -t libctest.a
  - **Linking with the library**:
    - gcc -o <exec-name> prog.c libctest.a
    - gcc -o <exec-name> prog.c -L/path/to/lib –lctest
- The Linux/Unix ".a" library is conceptually the same as the Visual C++ static ".lib" libraries.

Static Libraries (.a) - Example Sources

```c
//File ctest1.c
void ctest1(int *i)
{
    *i=5;
}
```

```c
//File ctest2.c
void ctest2(int *i)
{
    *i=100;
}
```

```c
//File prog.c
#include <stdio.h>
void ctest1(int *);
void ctest2(int *);
int main(){
   int x;
   ctest1(&x);
   printf("Valx=%d\n",x);
    return 0;
}
```

Dynamically Linked "Shared Object" Libraries: (.so) (1)

- **Generating a shared object**: (Dynamically linked object library file.) is a two step process.
  - Create object code
  - Create library
  - Optional: create default version using a symbolic link.
- **Creating the library libctest.so.1.0** and symbolic links to it.
  - gcc -Wall -fPIC -c *.c
  - gcc -shared -Wl,-soname,libctest.so.1 -o libctest.so.1.0 *.o
  - mv libctest.so.1.0 /opt/lib
  - ln -sf /opt/lib/libctest.so.1.0 /opt/lib/libctest.so
  - ln -sf /opt/lib/libctest.so.1.0 /opt/lib/libctest.so.1

Dynamically Linked "Shared Object" Libraries: (.so) (2)

- Compile main program and link with shared object library:
  - **gcc -Wall -I/path/to/include-files -L/path/to/libraries prog.c -lctest -o prog**
- Where the name of the library is libctest.so. This is why the symbolic link must be created or you will get the error "/usr/bin/ld: cannot find -lctest".
- The **libraries will NOT** be included in the executable but **will be dynamically linked during runtime** execution.

Dynamically Linked "Shared Object" Libraries: (.so) (3)

- **List Dependencies** - The shared library dependencies of the executable can be listed with the command: **ldd name-of-executable**
- Example: ldd prog

  libctest.so.1 => /opt/lib/libctest.so.1 (0x00002aaaaaac000)

  libc.so.6 => /lib64/tls/libc.so.6 (0x0000003aa4e00000)

  /lib64/ld-linux-x86-64.so.2 (0x0000003aa4c00000)

- Run Program:
  - Set path: **export LD_LIBRARY_PATH=/opt/lib:$LD_LIBRARY_PATH**
  - Run: ./prog

Library Path

- **In order** for an executable to find the required libraries to link with during runtime, one must configure the system so that the libraries can be found. Methods available:
  - **Add library directories to** be included during dynamic linking to the file **/etc/ld.so.conf**. After that you must run (as root)ldconfig in order the changes to take effect.
  - **Add directory to library cache**: (as root)ldconfig -n /opt/lib
  - **Specify the environment variable LD_LIBRARY_PATH** to point to the directory paths containing the shared object library. This will specify to the run time loader that the library paths will be used during execution to resolve dependencies.

Library Info

- The command "nm" lists symbols contained in the object file or shared library.
- Use the command nm -D libctest.so.1.0
  - 0000000000100988 A __bss_start
  - 000000000000068c T ctest1
  - 00000000000006a0 T ctest2
  - w __cxa_finalize
  - 00000000001007b0 A _DYNAMIC
  - …
- Symbol Types
  - A - The symbol's value is absolute, and will not be changed by further linking.
  - T - Normal code section.
  - W - Doubly defined symbol. If found, allow definition in another library to resolve dependency.

Dynamic (un)loading of Shared Libraries

- **These libraries are dynamically loaded / unloaded** and linked during execution. Usefull for creating a "plug-in" architecture.
- Load and unload the library libctest.so dynamically:

```
#include <stdio.h>
#include <dlfcn.h>
#include "ctest.h"
void main(int argc, char **argv)  {
       void *lib_handle; double (*fn)(int *);
       lib_handle = dlopen("/opt/lib/libctest.so",
RTLD_LAZY);
       fn = dlsym(lib_handle, "ctest1");
       (*fn)(&x);
       dlclose(lib_handle);
}
```

- Compile: gcc -rdynamic -o progdl progdl.c -ldl

Makefiles

- **Makefiles** are special format files that together with the make utility will help you to automagically build and manage your projects.
- **make** - this program will look for a file named makefile or Makefile in your directory, and then execute it.
- If you have several makefiles, then you can execute them with the command:
  - make **-f** MyMakefile

Makefiles – Basics (1)

- **Compiling by hand** - The trivial way to compile the files and obtain an executable, is by running the command:
  - g++ main.cpp hello.cpp factorial.cpp -o hello

```
/bgsys/drivers/ppcfloor/comm/default/bin/mpicxx" --
host=powerpc64-unknown-linux-gnu ARCH=bluegenep --
prefix=/shared1/vgancheva/Maria/shared-scalasca/ CFLAGS="-O3 -g -
qmaxmem=-1 -I/bgsys/drivers/ppcfloor/comm/include -
L/bgsys/drivers/ppcfloor/comm/lib -qarch=450 -qtune=450"
FFLAGS="-O3 -g -qmaxmem=-1 -I/bgsys/drivers/ppcfloor/comm/include
-L/bgsys/drivers/ppcfloor/comm/lib -qarch=450 -qtune=450"
LDFLAGS="-g -Wl,-allow-multiple-definition"
CONFIG_LIBC=/lib/libc.so.6 --libdir=/lib/ PREP="scalasca -
instrument"
```

Makefiles – Basics (2)

- The basic Makefile is composed of:

```
target: dependencies
[tab] system command
```

- Simple example is:

```
all:
	g++ main.cpp hello.cpp factorial.cpp -o hello
```

- To run this makefile on your files, type make.
- Our target is called all. This is the default target for makefiles.

Makefiles – Dependencies

- Sometimes it is useful to use different targets. This is because if you **modify a single file** in your project, you **don't have to recompile everything**, only what you modified.
- Here is an example:

```
all: hello
hello: main.o factorial.o hello.o
	g++ main.o factorial.o hello.o -o hello
main.o: main.cpp
	g++ -c main.cpp
factorial.o: factorial.cpp
	g++ -c factorial.cpp
hello.o: hello.cpp
	g++ -c hello.cpp
clean:
	rm -rf *o hello
```

Makefiles – Variables and Comments

- You can also use **variables** when writing Makefiles. It comes in handy in situations where you want to change the compiler, or the compiler options.

```
# I am a comment, and I want to say that the variable CC
will be
# the compiler to use.
CC=g++
# Hey!, I am comment number 2. I want to say that CFLAGS
will be the
# options I'll pass to the compiler.
CFLAGS=-c -Wall

all: hello
hello: main.o factorial.o hello.o
	$(CC) main.o factorial.o hello.o -o hello
…
```

References

- http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html
- http://www.eyrie.org/~eagle/notes/rpath.html
- http://www.yolinux.com/TUTORIALS/LinuxTutorialSoftwareDevelopment.html
- http://mrbook.org/tutorials/make/

Thank you for your attention!