Programming in linux

Operating System Lab
Spring 2015

Roya Razmnoush
CE & IT Department, Amirkabir University of Technology

integrated development environment (IDE)

- also known as
  - **integrated design environment**
  - **integrated debugging environment**
  - **interactive development environment**

- a software application that provides comprehensive facilities to computer programmers for software development.

IDE

- An IDE normally consists of:
  - a source code editor
  - a compiler and/or an interpreter
  - build automation tools
  - a debugger

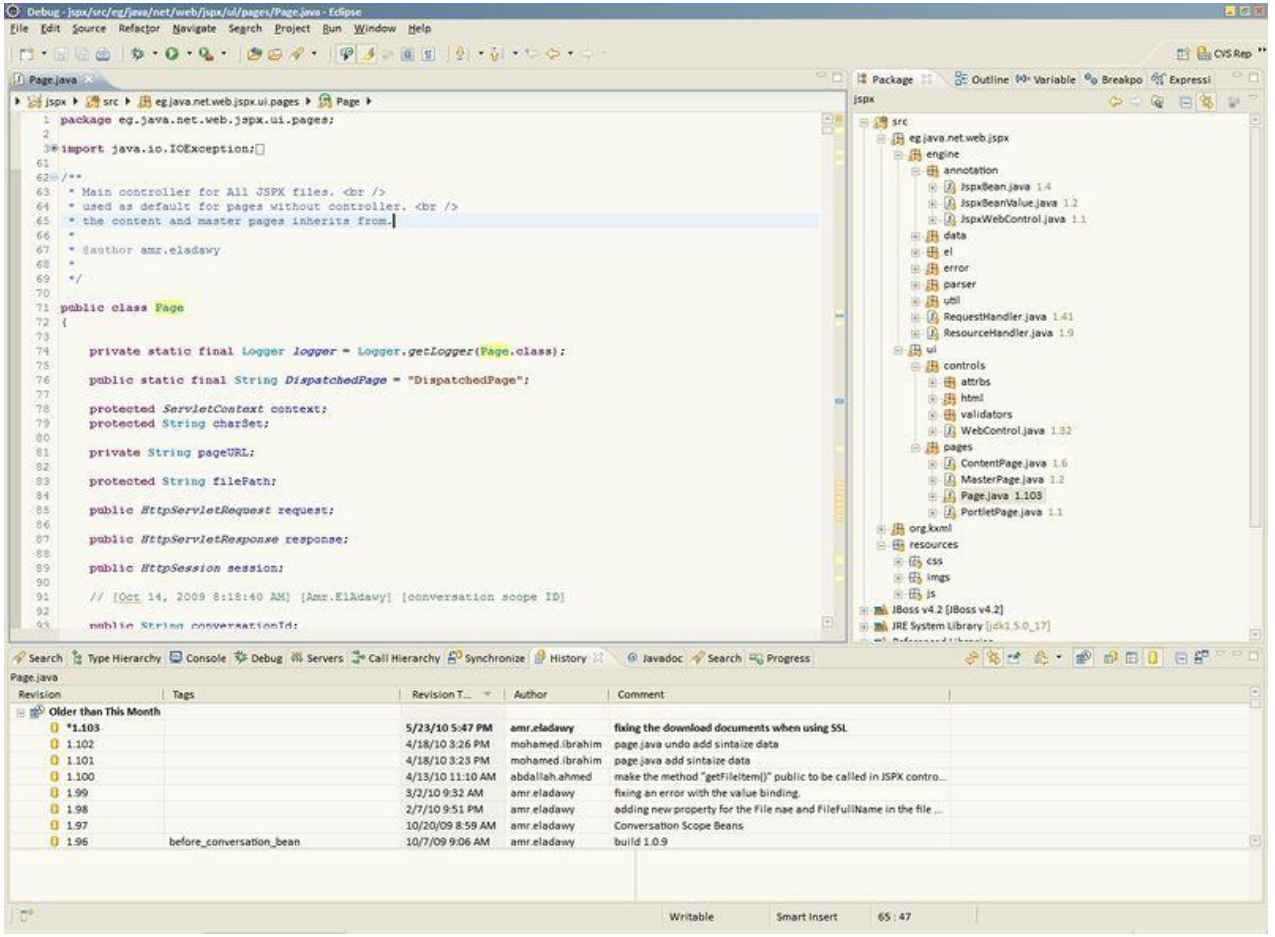
Eclipse

- written mostly in Java
- can be used to develop applications in Java
- by means of various plug-ins, other programming languages including Ada, C, C++, COBOL, Perl, PHP, Python, R, Ruby, Scala, Clojure, Groovy and Scheme

- Operating system: Linux, Mac OS X, Solaris, Windows

- http://www.eclipse.org

Eclipse


Debug - jspx/src/eg/java/net/web/jspx/ui/pages/Page.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Page.java
jspx ▸ src ▸ eg.java.net.web.jspx.ui.pages ▸ Page ▸
1 package eg.java.net.web.jspx.ui.pages;
2
3 import java.io.IOException;
61
62 /**
63 * Main controller for All JSPX files.
64 * used as default for pages without controller.
65 * the content and master pages inherits from.
66 *
67 * @author amr.eladawy
68 *
69 */
70
71 public class Page
72 {
73
74 private static final Logger logger = Logger.getLogger(Page.class);
75
76 public static final String DispatchedPage = "DispatchedPage";
77
78 protected ServletContext context;
79 protected String charSet;
80
81 private String pageURL;
82
83 protected String filePath;
84
85 public HttpServletRequest request;
86
87 public HttpServletResponse response;
88
89 public HttpSession session;
90
91 // [Oct 14, 2009 8:18:40 AM] [Amr.ElAdawy] [conversation scope ID]
92
93 public String conversationId;
Package Outline Variable Breakpo Expressi
jspx
src
eg.java.net.web.jspx
engine
annotation
JspxBean.java 1.4
JspxBeanValue.java 1.2
JspxWebControl.java 1.1
data
el
error
parser
util
RequestHandler.java 1.41
ResourceHandler.java 1.9
ui
controls
attribs
html
validators
WebControl.java 1.32
pages
ContentPage.java 1.6
MasterPage.java 1.2
Page.java 1.103
PortletPage.java 1.1
org.kxml
resources
css
imgs
js
JBoss v4.2 [JBoss v4.2]
JRE System Library [jdk1.5.0_17]
Search Type Hierarchy Console Debug Servers Call Hierarchy Synchronize History Javadoc Search Progress
Page.java
Revision Tags Revision T... Author Comment
Older than This Month
*1.103 5/23/10 5:47 PM amr.eladawy fixing the download documents when using SSL
1.102 4/18/10 3:26 PM mohamed.ibrahim page.java undo add sintaize data
1.101 4/18/10 3:23 PM mohamed.ibrahim page.java add sintaize data
1.100 4/13/10 11:10 AM abdallah.ahmed make the method "getFileItem()" public to be called in JSPX contro...
1.99 3/2/10 9:32 AM amr.eladawy fixing an error with the value binding.
1.98 2/7/10 9:51 PM amr.eladawy adding new property for the File nae and FilefullName in the file ...
1.97 10/20/09 8:59 AM amr.eladawy Conversation Scope Beans
1.96 before_conversation_bean 10/7/09 9:06 AM amr.eladawy build 1.0.9
Writable
Smart Insert
65 : 47

Programming without IDE

- Programming without an IDE is a great way to learn what's happening.
- A lot of programmers don't even use IDEs
- All you need to write a program:
  - a *text editor*
  - a compiler (or an interpreter if you're writing in a non-compiled language).

Step 1

- Use an editor to write your program
- terminal based text editor: vi and emacs

Step 2

- You need a toolchain (mainly a compiler and a linker) to translate your source code to a binary machine-understandable-and-runnable code.

- Each language (C, Java, C++, C#, VB.Net, etc...) has it's own toolchain.

gcc

GNU Compiler Collection (GCC)

- The GCC is a compiler system produced by the GNU Project supporting various programming languages.

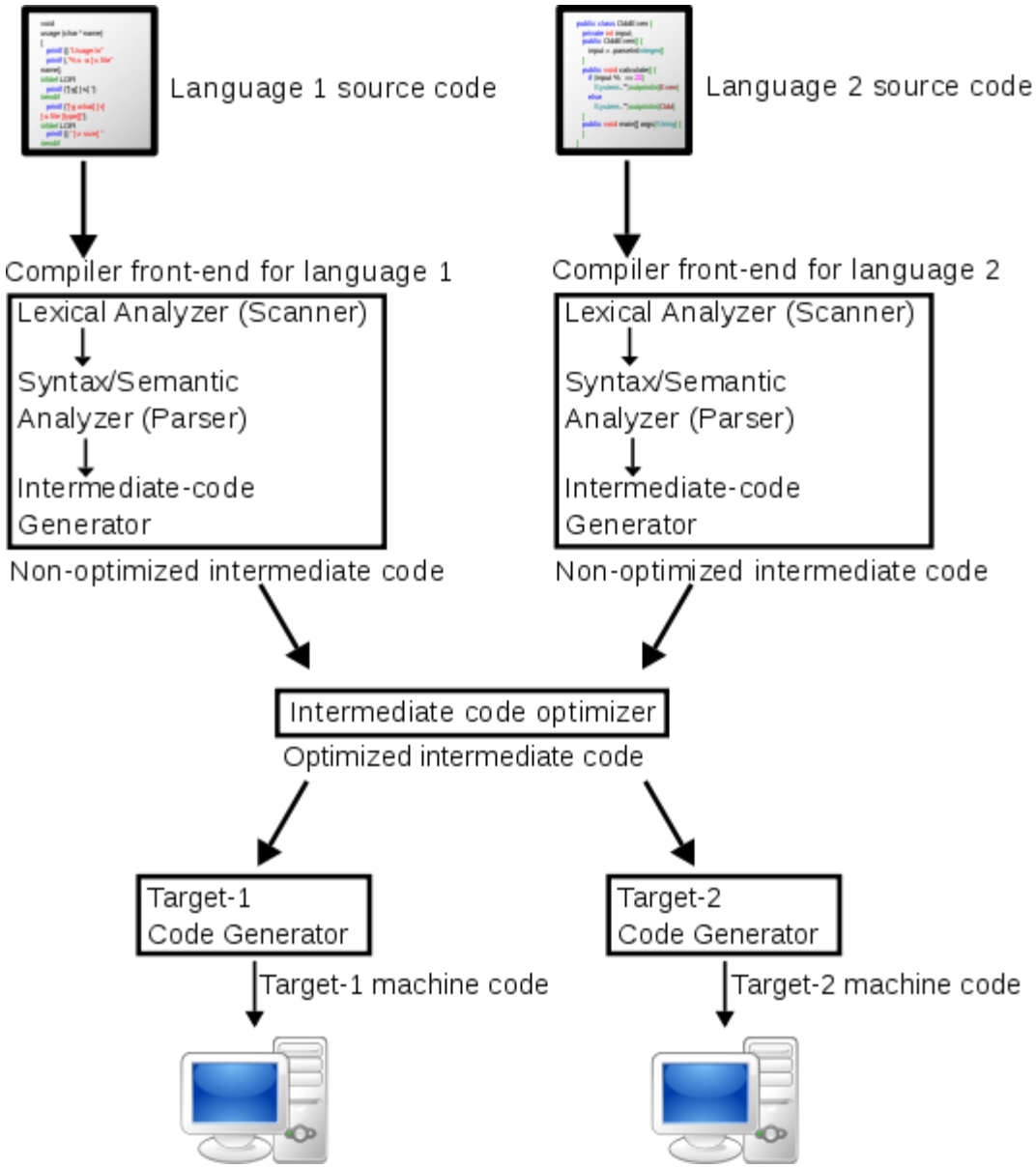
- GCC is a key component of the GNU toolchain.

- The Free Software Foundation (FSF) distributes GCC under the GNU General Public License (GNU GPL).

Gcc - the Gnu Compiler Collection

- The standard compiler release 4.6 includes front ends for C (gcc), C++ (g++), Java (gcj), Ada (GNAT), Objective-C (gobjc), Objective-C++ (gobjc++) and Fortran (gfortran).

- GCC has been ported to a wide variety of processor architectures. Backend for:
  - x86, ia-64, ppc, m68k, alpha, hppa, mips, sparc, mmix, pdp-11, vax, …



A diagram of the operation of a typical multi-language, multi-target

The gcc Compiler

- What happens when you call gcc to build your program?
- Phase 1, Compilation: .c files are compiled into .o object modules
- Phase 2, Linking: .o modules are linked together to create a single executable.

Compiling a simple C program

- Example C program :

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

Compiling hello.c

- Simple compiling

  gcc hello.c => a.out

  ./a.out

- -o : Specifying output name

  gcc hello.c –o hello

  ./hello

Compiling bad.c

- -Wall (warn all) : Produces warnings

  $ gcc -Wall bad.c -o bad

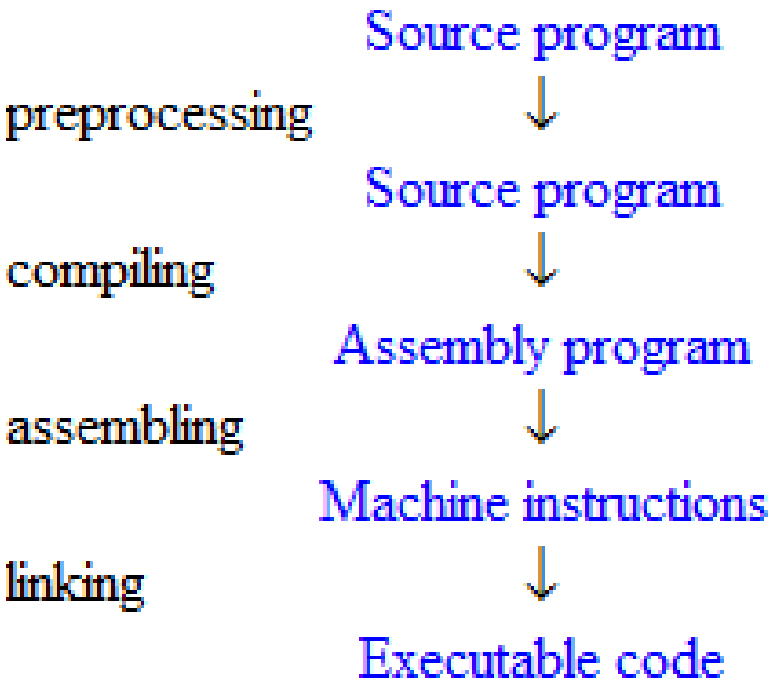
  bad.c: In function ‘main’:

  ….

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

  Two plus two is 2.585495 (incorrect output)

Steps in the production of an executable.



- gcc –v hello.c

step1

- To start off, preprocessing replaces certain pieces of text by other text according to a system of macros.

- gcc –E hello.c –o hello.i
- cpp hello.c > hello.i

step2

- Next, compilation translates the source program into assembly instructions.

- gcc –Wall –S hello.i

step3

- assembly instructions are then converted to machine instructions.

- as hello.s –o hello.o

step4

- Finally, the linking process establishes a connection to the operating system for primitives. This includes adding the runtime library, which mainly consists of memory management routines.

- gcc hello.o

- ./a.out

Compiling multiple source files

- we will split up the program Hello World into three files: ‘main.c’, ‘hello_fn.c’ and the header file ‘hello.h’.

main.c	hello_fn.c	hello.h

#include "hello.h"
int main (void)
{
hello ("world");
return 0;
}
	#include <stdio.h>
	#include "hello.h"
	void hello (const char * name)
	{
	printf ("Hello, %s!\n", name);
	}
		void hello (const char * name);

Header files

- These files allow programmers to separate certain elements of a program's source code into reusable files.
- Header files commonly contain forward declarations of classes, subroutines, variables, and other identifiers
- Header files almost always have a .h extension
- use the header file in your program by *including* it, with the C preprocessing directive `#include'.

Header files

- the difference between the two forms of the include statement #include "FILE.h" and #include <FILE.h> is that the former searches for ‘FILE.h’ in the current directory before looking in the system header file directories. The include statement #include <FILE.h> searches the system header files, but does not look in the current directory by default

Header files

- header files typically only contain declarations. They do not define how something is implemented,
- The program won't link if it can't find the implementation of something you use.

- So if printf is only *defined* in the "stdio" header file, where is it actually implemented? It is implemented in the runtime support library, which is automatically linked into your

Compiling multiple source files

- Compiling program

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

  – 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

- Run the program

  ./newhello

Creating object files

- `-c` : Creates object file. The result is referred to as an object file, and has the extension '.o'

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

- There is no need to put the header file 'hello.h' on the command line, since it is automatically included by the #include statements

Creating executables from object files

- To link object files together, they are simply listed on the command line:

  $ gcc main.o hello_fn.o -o hello

- The resulting executable file can now be run:

  $ ./hello

Link order of object files

- Link order of object files On Unix-like systems, the traditional behavior of compilers and linkers is to search for external functions from left to right in the object files specified on the command line. This means that the object file which contains the definition of a function should appear after any files which call that function

  gcc main.o hello_fn.o -o hello (correct order)

  cc hello_fn.o main.o -o hello (incorrect order)

Library

- A library is a collection of precompiled object files which can be linked into programs. The most common use of libraries is to provide system functions, such as the square root function sqrt found in the C math library.
- Libraries are typically stored in special *archive files* with the extension '.a', referred to as *static libraries*. They are created from object files with a separate tool, the GNU archiver ar, and used by the linker to resolve references to functions at compile-time.

- The standard system libraries are usually found in the directories ‘/usr/lib’ and ‘/lib’.
- For example, the C math library is typically stored in the file ‘/usr/lib/libm.a’ on Unix-like systems. The corresponding prototype declarations for the functions in this library are given in the header file ‘/usr/include/math.h’.
- The C standard library itself is stored in ‘/usr/lib/libc.a’ and contains functions

Linking with external libraries

- The standard system libraries are usually found in the directories ‘/usr/lib’ and ‘/lib’.

```
#include <math.h>
#include <stdio.h>
int main (void)
{
    double x = sqrt (2.0);
    printf ("The square root of 2.0 is %f\n", x);
    return 0;
}
```

Linking with external libraries

- Trying to create an executable from this source file alone causes the compiler to give an error at the link
  - stage gcc -Wall calc.c -o calc
- The problem is that the reference to the sqrt function cannot be resolved without the external math library ‘libm.a’.
- The C standard library itself is stored in ‘/usr/lib/libc.a’ and contains functions specified in the ANSI/ISO C standard, such as

Linking with external libraries

- Libraries are typically stored in special archive files with the extension '.a', referred to as static libraries

  gcc -Wall calc.c /usr/lib/libm.a -o calc

- -l : To avoid the need to specify long paths on the command line, the compiler provides a short-cut option '-l' for linking against libraries.

  $ gcc -Wall calc.c -lm -o calc

- the compiler option '-lNAME' will attempt to

Link order of libraries

- The ordering of libraries on the command line follows the same convection as for object files

  $ gcc -Wall calc.c -lm -o calc (correct order)

  $ gcc -Wall -lm calc.c -o calc (incorrect order)

Setting search paths

- The compiler options -I and -L add new directories to the beginning of the include path and library search path respectively

  mv hello.h ../hello.h

  ls

  ls ..

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

  Error ..

gcc -Wall main.c hello_fn.c –I.. -o newhello

or

gcc -Wall main.c hello_fn.c –I/home/oslab/Desktop/examples

Compiling for debugging

- GCC provides the -g debug option to store additional debugging information in object files and executables.
- Example

  gcc -Wall -g null.c
- Execution

  ./a.out

  Segmentation fault (core dumped)

Compiling for debugging

- gdb (GNU Project Debugger)
- Debugger

```
$ gdb EXECUTABLE-FILE
$gdb a.out
….
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0 0x080483ed in a (p=0x0) at null.c:13
13 int y = *p;
(gdb)
```

Preprocessor directives

- They are not program statements but directives for the preprocessor.
- Preceded by a hash sign (#).
- The preprocessor is executed before the actual compilation of code begins.

  Examples:
  - **Source file inclusion (#include)**
  - **Conditional inclusions (#ifdef, #ifndef, #if, #endif, #else and #elif)**
  - **macro definitions (#define, #undef)**
  - **…**

Source file inclusion

- When the preprocessor finds an #include directive it replaces it by the entire content of the specified file.

- There are two ways to specify a file to be included:
  - *#include "file"*
  - *#include <file>*

Conditional inclusions

- These directives allow to include or discard part of the code of a program if a certain condition is met.

- *#ifdef TABLE_SIZE*
- *int* table[TABLE_SIZE];
- *#endif*

macro definitions

- To define preprocessor macros we can use #define. Its format is:
- #define identifier replacement

- When the preprocessor encounters this directive, it replaces any occurrence of identifier in the rest of the code by replacement.

- A macro lasts until it is undefined with the #undef preprocessor directive.

```
#define TABLE_SIZE 100
 int table1[TABLE_SIZE];
#undef TABLE_SIZE
#define TABLE_SIZE 200
int table2[TABLE_SIZE];
```

Using the preprocessor

- The gcc option -DNAME defines a preprocessor macro NAME from the command line
- the conditional part of the source code is a printf statement which prints the message “Test mode”:

```
#include <stdio.h>
int main (void)
{
        #ifdef TEST
        printf ("Test mode\n");
        #endif
```

Using the preprocessor

- Running the example :

  $ gcc -Wall -DTEST dtest.c

  $ ./a.out

  Test mode

  Running...

Macros with values

- example

```
#include <stdio.h>
int main (void)
{
    printf("Value of NUM is %d\n", NUM);
    return 0;
}
```

- Running

```
$ gcc -Wall -DNUM=100 dtestval.c
$ ./a.out
Value of NUM is 100
```

Other options

- -O(N) : Determines optimization levels
- Platform-specific options
- C++ compiler is g++

make

Automatic Building with make

- Automatic Building with Make
  - A GNU utility that determines which pieces of a large program need to be compiled or recompiled, and issues a commands to compile and link them in an automated fashion.
    - Faster compile time
  - Saves you from tedious, huge g++ commands!
    - Simpler for users

Sample makefile

- Makefiles main element is called a *rule*:

```
target : dependencies
TAB    commands                          #shell commands
```

- Example

  hello.o: hello.cpp

    g++ -c hello.cpp

Using makefiles

Naming:

- *makefile* or *Makefile* are standard
- other name can be also used

Running `make`

`make`

`make -f filename` – if the name of your file is not "makefile" or "Makefile"

“clean” target

- An important target that represents an action rather than a g++ operation.
- Has no dependencies, runs a command to remove all the compilation products from the directory, “cleaning” things up.
- Call by typing “make clean” into prompt.

Example:

```
clean:
        rm -f *.o
```

Sample of using make-1

- Suppose our directory contained only the following files
  - makefile pizza.c pizza.h

```
# Robert's pizza project.
# (Characters from the first "#" to the end of the line are ignored.)

pizza.o: pizza.c pizza.h                                    # Line  4
    gcc -c pizza.c                                          # Line  5

pizza: pizza.o                                              # Line  7
    gcc -o pizza pizza.o                                    # Line  8

all: pizza                                                  # Line 10

clean:                                                      # Line 12
    rm pizza pizza.o                                        # Line 13

install: pizza                                              # Line 15
    mv pizza /usr/local/bin                                 # Line 16
    cp pizza.h /usr/local/include                           # Line 17
    echo "Your pizza is ready!"                             # Line 18
```

Make command

- make pizza
- make all

- make clean
  - delete object files, executable programs, and core dumps from a project directory. (Usually you want to do this before you distribute (or submit) a project.)

- make install
  - More formal projects might use
  - to copy (or move) every executable, compiled library, and header file to an appropriate subdirectory in the file system

- make -d pizza
  - will print lots of debugging information as it tries to make the pizza program

Sample of using make-2

- Files of project

  main.cpp

  hello.cpp

  factorial.cpp

  functions.h

Sample of using make

functions.h

```
void print_hello();
int factorial(int n);
#include <iostream>
using namespace
std;
```

factorial.cpp

```
#include "functions.h"

int factorial(int n){
   if(n!=1){
return(n * factorial(n-1));
   }
   else return 1;
}
```

hello.cpp

```
#include "functions.h"

void print_hello(){
  cout << "Hello
World!";
}
```

main.cpp

```
#include "functions.h"

int main(){
   print_hello();
   cout << endl;
   cout << "The factorial of 5
is " << factorial(5) << endl;
   return 0;
}
```

makefile-1

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

- Note that make with no arguments executes the first rule in the file
- make -f makefile-1

makefile-2

```
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
```

Makefile variables

- The **make** program supports many other features that allow one to create more versitile description files.

- One of the most useful of these are variables.

- A variable is symbol that can be defined within a makefile to represent a list of other symbols

Variables

The old way (no variables)

```
my_prog : eval.o main.o
        g++ -o my_prog eval.o main.o
eval.o : eval.c eval.h
        g++ -c -g eval.c
main.o : main.c eval.h
        g++ -c -g main.c
```

A new way (using variables)

```
C = g++
OBJS = eval.o main.o
HDRS = eval.h

my_prog : eval.o main.o
        $(C) -o my_prog $(OBJS)
eval.o : eval.c
        $(C) -c -g eval.c
main.o : main.c
        $(C) -c -g main.c
$(OBJS) : $(HDRS)
```

Defining variables on the command line:

Take precedence over variables defined in the makefile.

```
make C=cc
make CFLAGS=-g
```

makefile-3

```
CC=g++
CFLAGS=-c -Wall

all: hello

hello: main.o factorial.o hello.o
    $(CC) main.o factorial.o hello.o -o hello

main.o: main.cpp
    $(CC) $(CFLAGS) main.cpp

factorial.o: factorial.cpp
    $(CC) $(CFLAGS) factorial.cpp

hello.o: hello.cpp
    $(CC) $(CFLAGS) hello.cpp

clean:
    rm -rf *o hello
```

- Large projects use many source files, hence they require many object (.o) files for the build

- The default rule for generating *anyfile*.o from *anyfile*.c is written as
  - `%.o:%.c`
    `$(CC) $(CFLAGS) $<`

Internal macro symbols

$@	name of current target
$?	list of dependencies newer than target
$<	name of dependency file
$*	base name of current target
$%	for libraries, the name of member

– echo $@

  echo $<

makefile-4

```
CC=g++
CFLAGS=-c -Wall
LDFLAGS=
SOURCES=main.cpp hello.cpp factorial.cpp
OBJECTS=$(SOURCES:.cpp=.o)
EXECUTABLE=hello

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
	$(CC) $(LDFLAGS) $(OBJECTS) -o $@

.cpp.o:
	$(CC) $(CFLAGS) $< -o $@
```