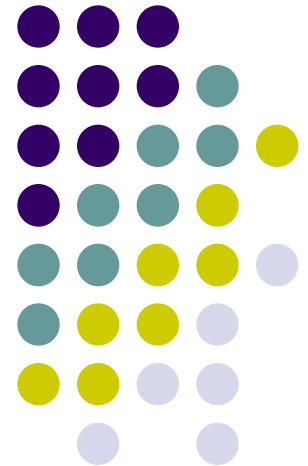
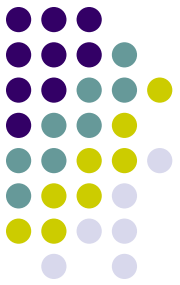


Compilation and Linking

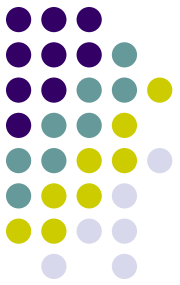
Xiaoqiang Wang
Florida State University





Sources of Information

- Books
 - Running Linux: O'Reilly
 - Linux in a Nutshell: O'Reilly
 - Unix Power Tools
- WEB
 - www.google.com (search engine)
 - www.nlsearch.com
 - www.hotbot.com



Linux/Unix man pages

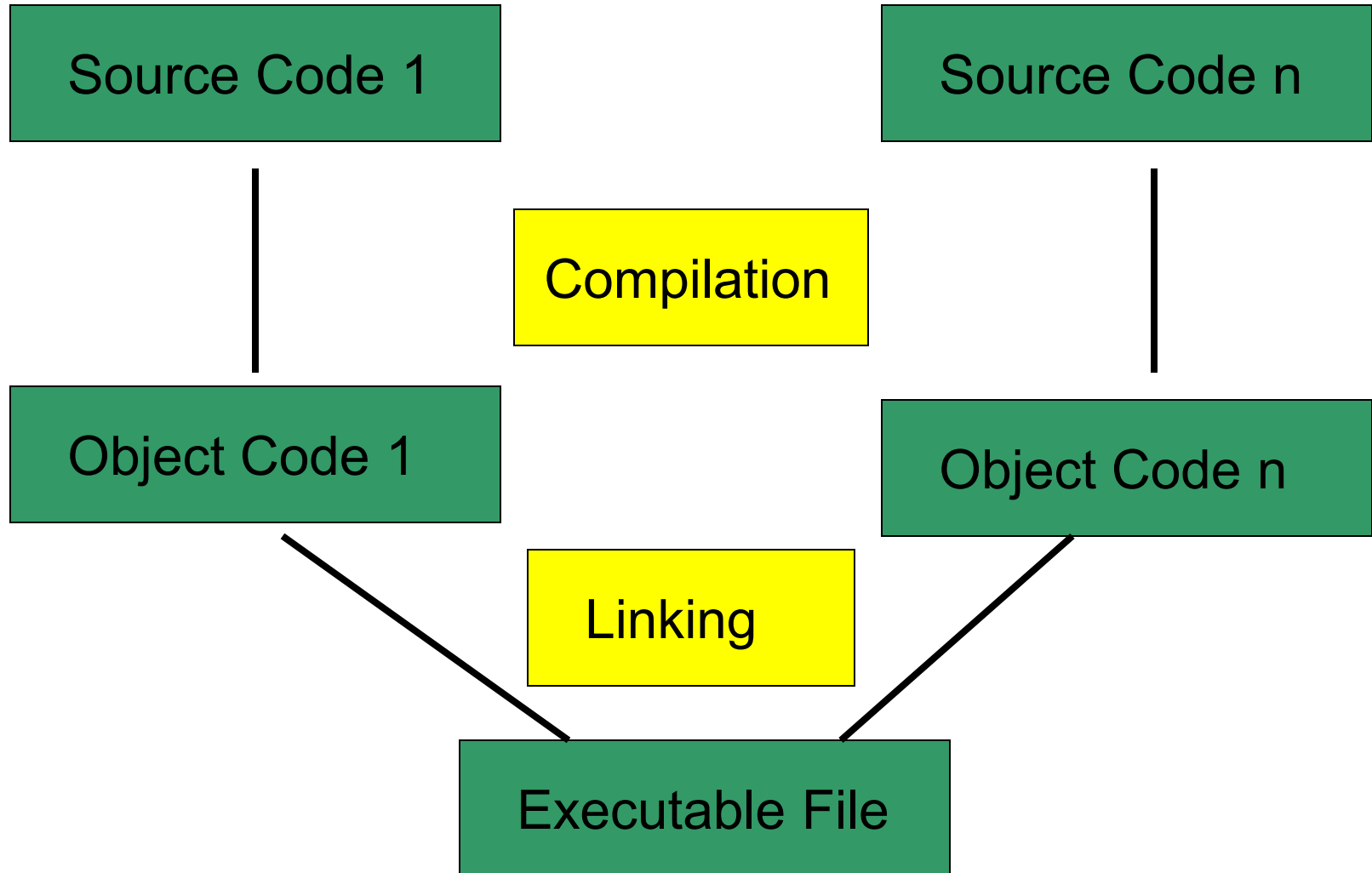
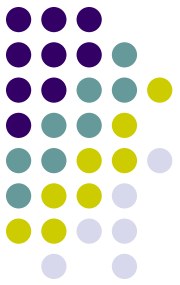
- Search for list of items related to compiling
 - `man -k compile`
- Get information about a particular command
 - `man fort77` (Fortran compiler)
 - `man ar` (library archiver)
 - `man gcc` (GNU C++ compiler)

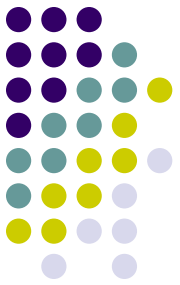
Commonalities across machines and Operating Systems



- Files are contained in directories
- Directories form a tree, files are the leaves of that tree
- Compiled Languages (C/C++, Fortran, Java)
- Scripting languages (Perl, Python)
- File extensions (.cc, .f77, .as, .o)

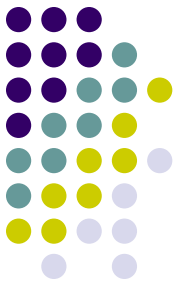
Compile/Link/Execute





Compilers

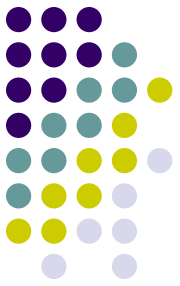
- Fortran (fort77, f90, g77)
- C/C++
 - cc, gcc, g++ (Linux)
 - CC (IRIX)



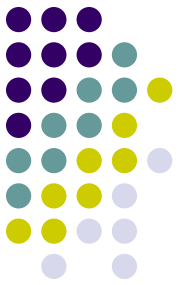
Source Code

- Users write source code
- Ascii file, created with text editor
- Source code files have extensions `.cpp`, `.cxx`, `.cc`, `.CC`, `.f77`, `.f90`, etc.

From source code to binary code

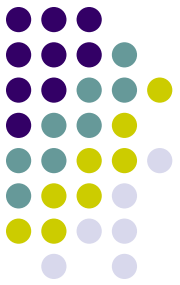


- Source code (readable ascii text) --->
- Assembler code (human readable machine code) --->
- Object code (machine code, symbols) --->
- Machine code (binary executable)
 - contains ascii symbols if debug option on
 - pure machine code if debug option off



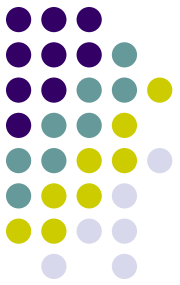
Directory paths

- **Include file paths: -Idirs**
 - Example: `g++ -c -I. -I/home/source/include main.c`
 - Example: `f77 -c -I. -I/home/source/include main.c`
- **Library file paths: -Ldirs**
 - Example: `g++ -c -L/home/lib main.c timer.o`
 - Example: `f77 -c -L/home/lib main.c timer.o`



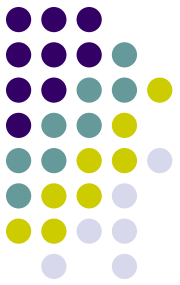
Include paths

- `g++ -Ipath1 -Ipath2 source.cpp`
 - Search for include files in succession of paths
 - System searches standard paths first
 - `/usr/include`
 - `/usr/local/include`
 - etc. (see manual for more information)



Include files

- `#include <systemfile.h>`
- `#include "localfile.h"`
- `<...>` : system include files
- `"..."` : user include files

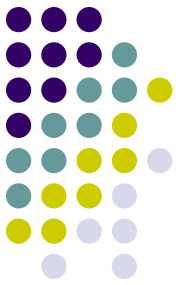


Preprocessor

- `#include <file1.h>`
- `#include "file2.h"`
- `#ifdef VAR`
 `# define A 5`
 `#else`
 `# define A 10`
 `#endif`

Include contents of
files1.h into source

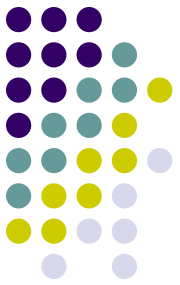
Based on value of
VAR, define A to
be 5 or 10



Preprocessor

- Explicit call
 - `/lib/cpp source.cpp [options] > new_source.cpp`
- Automatically called by *most* compilers
- Options to preprocessor
 - `-P` (remove comment lines)
 - `-DdefineVar=value` (set variable)

Preprocessing

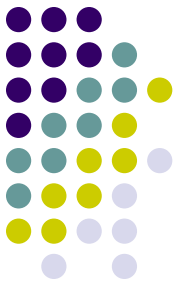


```
#ifndef TEST
# define A 3*3
#else
# define A = 4*4
#endif

main()
{
    float c = A;
    print("c = %d\n", c);
}
```

```
g++ testdef.cpp
    TEST not defined
    c=16
```

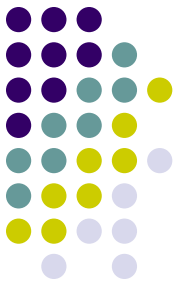
```
G++ -DTEST testdef.cpp
    TEST is defined
    c=9
```



Include File Directories

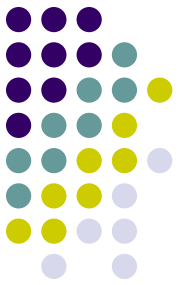
- `/usr/include`
- `/usr/include/GL` (OpenGL libraries)
- `/usr/local/include`
- user-specified include directories

Include file locations are based on convention.
There are no strong conventions.



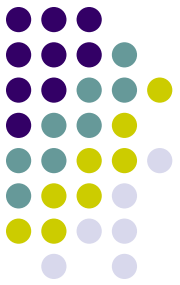
Compilation

- `g++ [Options] -c file1.c file2.c ...`
- Common options:
 - `-c` : keep object files
 - array bound checking
 - print out all warnings
 - `-O2`, `-O3` (control optimization level)
 - `-Dvar=value` (define variable name)
 - `-IincludePath` (path to include files)
 - `-g` (enable debugging)
 - see man pages for many other options



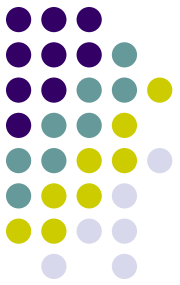
Compilation Errors

- Easy to fix
- Pretty much described in English
- Search the web for the wording if you cannot find the source of the error
- Most common errors
 - Divide by zero
 - Type mismatch
 - Undefined variables



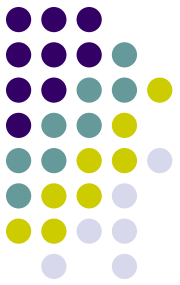
Object Files

- Source code is compiled into assembler code.
- Object files may or might not include variable names and other symbols (-g options for compilers) useful for debugging
- Multiple object code files are then combined to form an executable (linker)
- The executable is run by the user



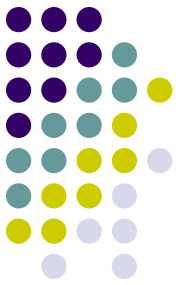
What is in a library

- Convenient place to store object code
 - Obj1.o
 - Obj2.o
 - ...



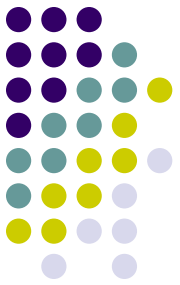
Libraries

- Shared libraries
 - libm.so, libm.so.143, libGL.so.1523
- Static libraries
 - libm.a, libGL.a , libX11.a, etc ...



Static Libraries

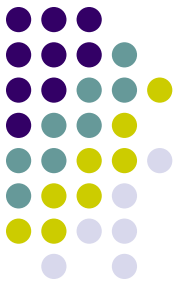
- The assembler code for all library files is included with the executable
 - Size of executable is very large
 - Space usage is duplicated if multiple copies of executable are used
 - Large programs can take longer to load



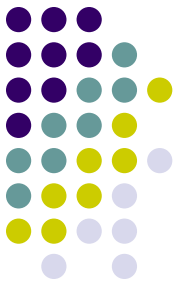
Shared Libraries

- Objects in libraries are shared across applications
- Routines are only loaded when needed
- Changes can be made to the libraries without recompiling code

Library directories (Unix/Linux)

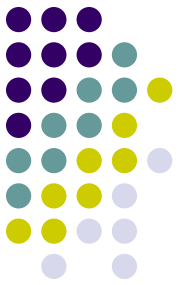


- /usr/lib
- /usr/local/lib
- /usr/X11R6/lib (specific to Linux version)
- /usr/lib32 (IRIX)
- user-defined library directories



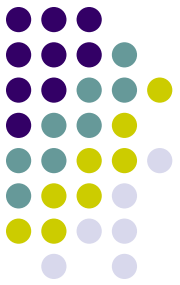
Creating a static library

- Collect object files file1.o file2.o into library libstuff.a
 - `ar r libstuff.a file1.o file2.o ...`
- Create an index for the library (to help linker find files within the library)
 - `ranlib libstuff.a`
 - (not necessary on SGI's)
- Combine both steps
 - `ar rs`



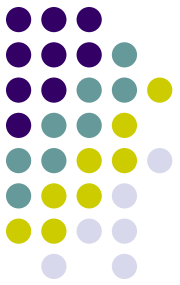
Creating a shared library

- Generate position-independent code
(might be different for different compilers)
 - `g++ -c -fPIC file1.c file2.c`
- Create the shared library
 - `g++ -shared -o libstuff.so file1.o file2.o`



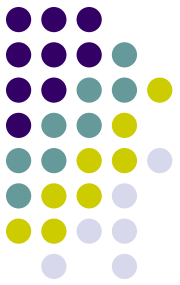
Linking

- Use same name as the compiler (most of the time): compiler is usually a front end script:
 - `g++ [options] -o executable obj1.o obj2.o [libraries]`
 - Common options
 - `-Ldir1 -Ldir2` (directory path to library files)
 - `-o executable` (name of final executable)
 - `-g` (debugging option)
 - see man pages for other options



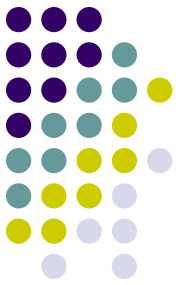
Linking

- Place object code *before* libraries
 - This order is not always required, but is more portable across operating systems (OS) and compilers.
 - Usually, the libraries are searched in the order they are listed
 - Always strive to compile and link in a portable fashion



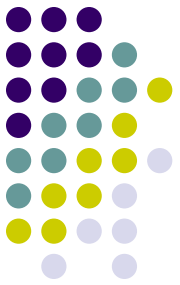
Undefined Externals

- Your code is using a user routine or system routine that *cannot be found* in the list of libraries
- Must find the library that references that routine and add it to the link command.
- How to find this library?



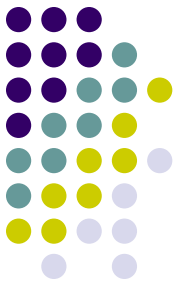
Multiple definitions

- If a routine is listed multiple times in linker output,
 - the routine was found in multiple libraries
 - usually, the last routine found is used
 - this is a warning from the linker, usually not fatal
 - BUT: you must be aware. Sometimes the linker chooses the wrong routine



Using Libraries

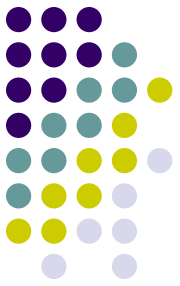
- `gcc -LPath1 -Lpath2 -o main.x main.o -lstuff -lm`
 - shared libraries are used if possible
 - static libraries are used if shared library does not exist
 - `-lstuff` links in `libstuff.a` searching in directories `path1` and `path2`
 - link in the math library `libmath.a`



Searching through libraries

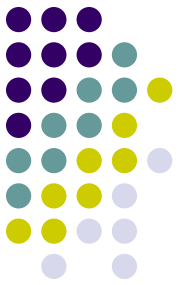
- sh shell script to search a list of libraries for a specified module (could use perl, python):
 - ```
for $i in lib*.a
do
 echo $i
 ar t $i | grep -i $1
done
```

# How to debug linking problems



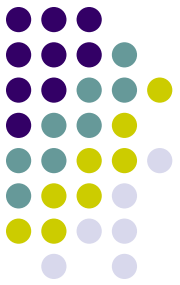
- Read error messages carefully
- Undefined function
  - ....
- Undefined external
  - ....





# Tools to Search Libraries

- On Linux
  - ar (library archiver)
    - ar t libname.a
  - strings libname.a
    - Lists all ascii strings in the library
  - nm -a libname.a



# Use of nm

**nm /usr/libc.a > output\_file**

Section of output file:

printf.o:

00000000 T \_IO\_printf

    w \_\_pthread\_initialize

    w \_\_pthread\_initialize\_minimal

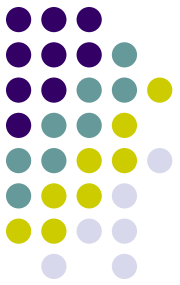
00000000 T printf

    U stdout

    U vfprintf

Conclusion: printf is  
defined.

stdout and vfprintf are  
undefined.



# Use of nm (cont.)

- In the same file:

vfprintf.o:

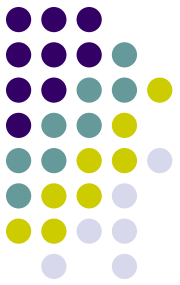
U \_IO\_default\_doallocate

U \_IO\_default\_finish

U \_IO\_default\_pbackfail

U \_IO\_default\_read

....



# Use of nm (cont.)

Searching for "stdout" at the beginning of the line produces no results. However, searching for "D stdout" produces

stdio.o:

U \_IO\_2\_1\_stderr\_

U \_IO\_2\_1\_stdin\_

U \_IO\_2\_1\_stdout\_

00000008 D \_IO\_stderr

00000000 D \_IO\_stdin

**00000004 D \_IO\_stdout**

w \_\_pthread\_atfork

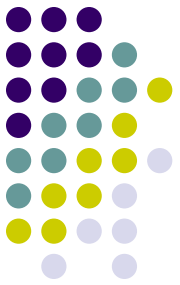
w \_\_pthread\_getspecific

Stdout is *defined* in the routine  
stdio.o

# Searching for string in libraries



```
#!/bin/bash
for i in /usr/lib/lib*a
do
 echo $i
 nm $i | grep -i routine_to_find
done
```

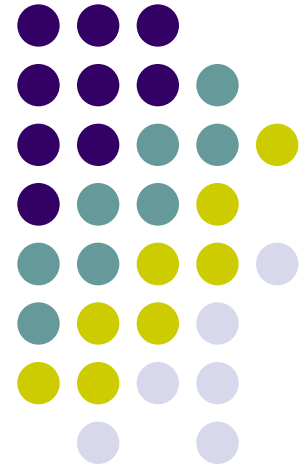


# Environment variables

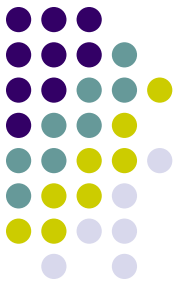
- **PATH**
  - List of directory searched by the OS to find a command
  - Defined in `.cshrc` or `.bashrc` file
- **LD\_LIBRARY\_PATH**
  - List of directories searched when shared libraries are used
  - Placed in `.cshrc` or `.bashrc` file.

# Makefiles

Xiaoqiang Wang  
Florida State University



# GNU Make



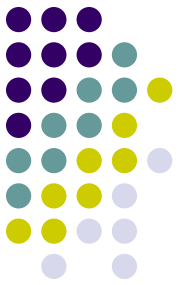
Home page

<http://www.gnu.org/software/make/make.html>

Documentation

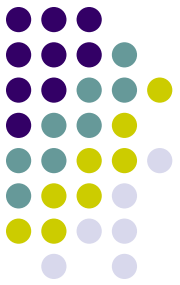
<http://www.gnu.org/software/make/manual/>





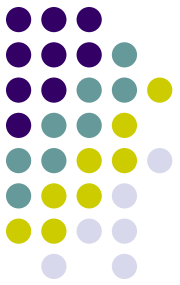
# Compilation and Linking

- Many source files
- Many object files
- Many libraries
- Various types of preprocessing
  - `/lib/cpp`
  - Lex/yacc (not often used)
  - Python/perl/ranlib/etc.



# Three Source files

- gcc -c file1.c      only to do the compiling  
gcc -c file2.c  
gcc -c file3.c
- gcc -o executable file1.o file2.o file3.o  
-lGLU -lGL -lXext -lX11



# Change file1.c

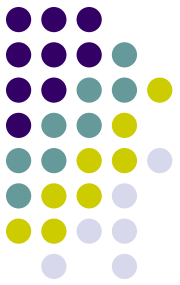
- What to do?
- Obvious solution: create a script in file compile.x:

```
gcc -c file1.c
```

```
gcc -c file2.c
```

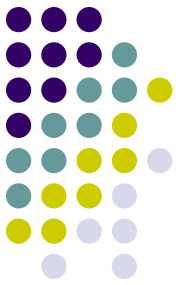
```
gcc -c file3.c
```

```
gcc -o executable file1.o file2.o file3.o
-IGLU -IGL -lxt -IX11
```



# First alternative

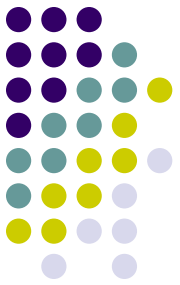
- Manually recompile file1.c and link:  
gcc -c file1.c  
gcc -o executable file1.o file2.o file3.o  
-IGLU -IGL -lxt -IX11
- This approach is faster, but more prone to error



# What if ...

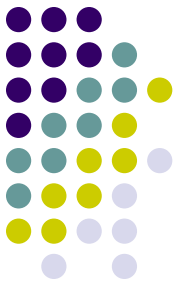
- 150 source files
- 7 source files are changed
- Compiling 150 source files: 20 min.
- Compiling 7 source files: 1 min
- Linking 150 object files: 3 min

What to do?



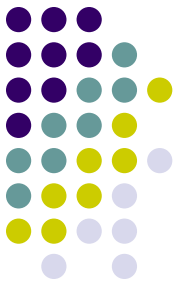
# Problems ...

- Hard to keep track of which files are changed
- Recompiling all files is inefficient: 143 files are compiled needlessly



# Solution: Makefiles

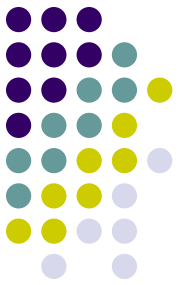
- Each file keeps track of creation time and last modification time
  - To see this:
    - `ls -l`
- The executable file has a last modification time
- If any source file has been modified after the creation of the executable files, it must obviously be compiled



# Why Makefiles?

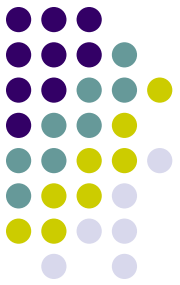
- Manage complex projects
- Store project information in single place
- Reduce turnaround time for compile—link cycle
- Keeps track of file dependencies based on last modification time
  - Manual or automatic tracking





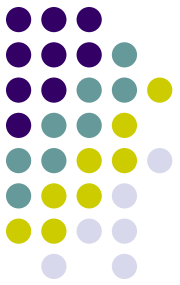
# Project layouts

- Single operating system
  - Source and object files in single directory
- Multiple operating systems (Unix, Linux, Windows)
  - Source files in one directory
  - Object files in directory associated with particular OS



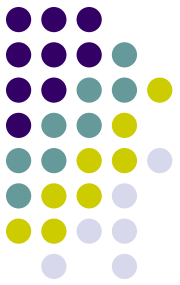
# GNUmake

- Open source
- Very general
- Very flexible
- Easy to use (in simplest modes)
- Can generate extremely complex makefiles



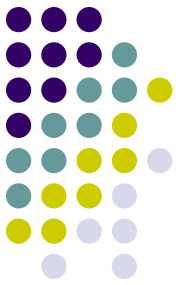
# Makefile names

- Default names (make)
  - Makefile
  - makefile
  - GNUmakefile
- User-specified name
  - `make -f makeFileName`
  - e.g., `make -f make01`



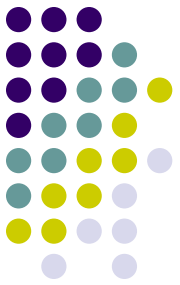
# Targets

- Targets appear on left side of colon (:)
- Targets are created (e.g., all, pgm, tar)
  - make all
  - make pgm
  - make tar



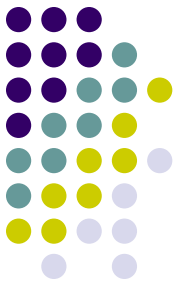
# Prerequisites

- Targets depend on prerequisites
- Targets are compared to prerequisites based on the last modification file date
- If a prerequisite file is *newer* than the target, make will try to recreate it. For example,
  - pgm.x *depends* on pgm.o
  - pgm.o *depends* on pgm.c
  - pgm.c *depends* on pgm.h



# Variables

- OBJS = main.c class1.c class2.c
  - Recursively expanded variable.
- OBJS := main.c class1.c class2.c
  - Simply expanded variable
  - Substitution occurs immediately
  - This method is safer

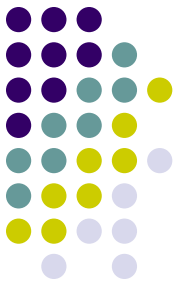


# Variables (example)

```
CFLAGS = $(include_dirs) -O
include_dirs = -Ifoo -Ibar
```

```
CFLAGS = $(CFLAGS) -O
It will cause an infinite loop
```

```
x := foo
y := $(x) bar
x := later
is equivalent to
y := foo bar
x := later
```



# Variables (example)

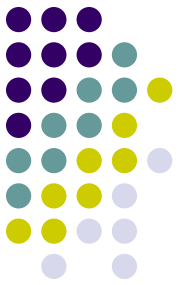
```
OBJS= *.o
```

The value of the variable OBJS is the string “\*.o”. However, if OBJS is used in a target or prerequisite, wildcard expansion will take place there.

```
pgm.x: $(OBJS)
```

```
gcc -o pgm.x $(CFLAGS) $(OBJS)
```





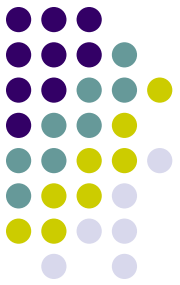
# Variables (example)

```
SRC := $(wildcard *.c)
```

```
OBJ := $(patsubst %.c,%.o,$(wildcard *.c))
```

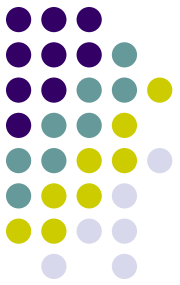
```
OBJ := $(SRC :.c=.o)
```

- Use of := is often safer
- Source files are rarely deleted
- Object list can be constructed automatically, regardless of whether object files actually exist



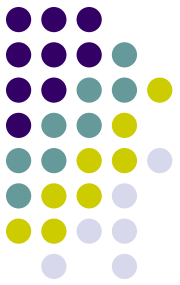
# Example

- Source files
  - main.c
  - class1.c, class1.h
  - class2.c, class2.h



# main.c

```
#include "class1.h"
#include "class2.h"
main()
{
 ...
}
```

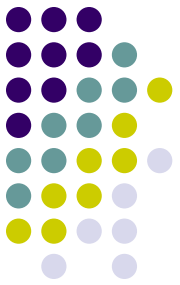


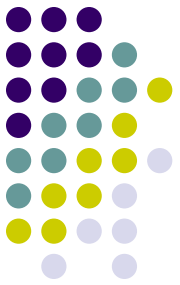
# class1.c

```
class Class1
{
 private:
 ...
 public:
 ...
}
```

# Simple Makefile

- Variable definitions
- Target definitions
- Rules





# Basic Structure

pgm.x: main.o class1.o class2.o

gcc -o pgm.x main.o class1.o class2.o

main.o: main.c class1.h

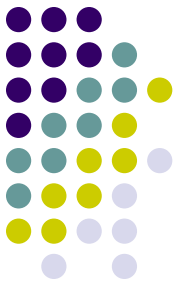
gcc -c main.c

class1.o: class1.c class1.h

gcc -c class1.c

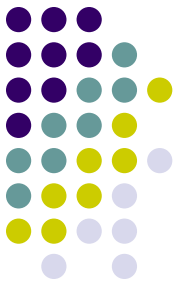
class2.o: class2.c class2.h

gcc -c class2.c



# Problems with basic structure

- What if I add one or more additional source file
- What if I add another include statement into main.c (e.g., `#include "main.h"`)
- What if I change compilers?
- What if I work on multiple operating systems on a regular basis
- What if I have source files in multiple directories?
- What if I want to do more complicated functions
  - Installation, tar files, help messages, etc.



# Use of Variables

CC=gcc

LD=gcc

OBJS= main.o class1.o class2.o

pgm.x: \$(OBJS)

\$(LD) -o pgm.x \$(OBJS)

main.o: main.c class1.h

\$(CC) -c main.c

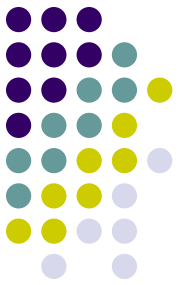
class1.o: class1.c class1.h

\$(CC) -c class1.c

class2.o: class2.c class2.h

\$(CC) -c class2.h



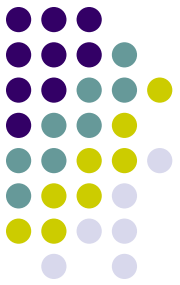


# Use of patterns

```
CC=gcc
LD=gcc
CFLAGS=-c
EXEC=pgm.x
OBJS= main.o class1.o class2.o
```

```
$(EXEC): $(OBJS)
$(LD) -o $(EXEC) $(OBJS)
%.o : %.c
$(CC) $(CFLAGS) -o $@ $<
```

‘\$@’ is the target, ‘\$<’ is the first prerequisite



# Additional functions

```
EXEC=pgm.x
```

```
$(EXEC):
```

```
 $(CC) -o $(EXEC) main.c
```

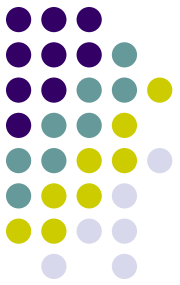
```
.PHONY clean
```

```
clean:
```

```
 rm *.o $(EXEC)
```

```
tar:
```

```
 tar $(EXEC).tar *.c *.h Makefile
```

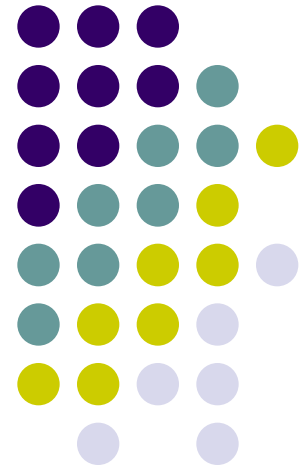


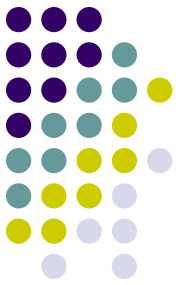
# Additional features

- Recursive makes
- Conditionals
- Manipulation of environment variables
- Many shortcut variables
- Pattern manipulation via text functions
- Include files
- Many many more ...

# Debugging

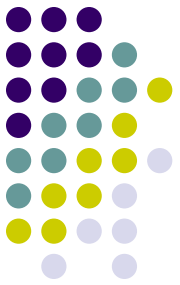
Xiaoqiang Wang  
Florida State University





# Where do we stand

- You have written a code (C++/Fortran)
- The code compiles and links correctly
  - Warning messages have mostly been eliminated
  - If not eliminated, warning messages should be understood
- You execute the code and the unexpected happens:



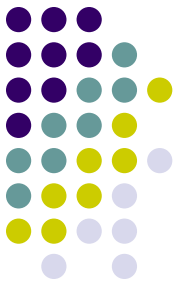
# What happened!

- The code crashes with a segmentation fault
- The computer produces the incorrect results
- You overran array bounds
- You divided by zero?
- You accessed an address outside the codes's address space (zero pointer syndrome)



# What do you do?

- You must debug the code
  - Print statements
  - Assert statements
  - Debugging objects
  - Command line debuggers (more common)
    - (dbx on unix, gdb on linux)
  - Visual debuggers (more specialized)
    - Visual C++, SGI/IBM/Compaq visual debuggers



# Print statements

- Create a specialize routines (C/Fortran)

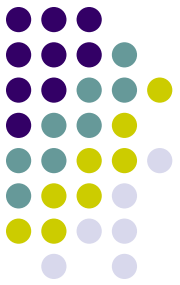
```
void print1(char* msg, float var)
{
 printf("%s: %f\n", msg, var);
}
void print2(char* msg, float var1, float var2)
{
 printf("%s: %f, %f\n", msg, var1, var2)
}
//-----
....
 print1("variable 1", area)
...
```



# Print and Objects



```
class Shape {
private:
 float a,b;
 int c;
public:
 Shape(float a, float b, int c) {...}
 print(char* msg) {
 printf("%s, a,b,c= %f, %f, %d\n", a,b,c);
 printf("%s: %f\n", msg, var);
 }
//-----
 main() {
 Shape sh(2.,3.,5);
 sh.print("Object sh:");
 ...
 }
```

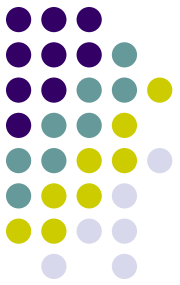


# Assert Statements

```
#include <assert.h>
```

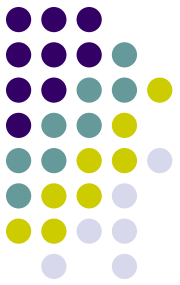
```
fct(float* a, int i)
{
 ASSERT(i < 5);
 a[i] = 3.;
}
```

- If condition is violated, the ASSERT macro prints out some statement and exits.
- ASSERT is only active when `-g` option is used to compile (debug mode).
- Users can write macros to do anything they want. Effective debugging tool.



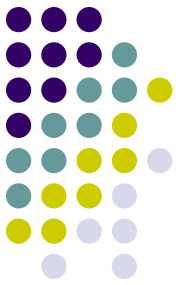
# Assert Statements

- Assert statements exist in some for or other in all compilers.
- They are usually defined as macros:
  - ```
#define ASSERT(a)  
    if (!(a)) {\br/>        printf("error at line %d\n", __LINE__);  
        exit(1);  
    } else {  
    }
```
- You can define your own. Use existing definitions as templates



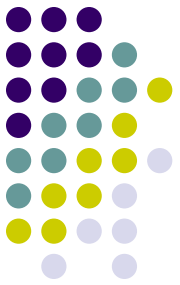
Assert Statements

- Often used to check on consistency conditions upon entry to a subroutine/function
- Example, given a bank account object.
 - o When entering the object, the bank account must be positive.
 - o Upon exit, it must still be positive or zero.
 - o Check this with ASSERT statement since these conditions should not be violated if program were working correctly.



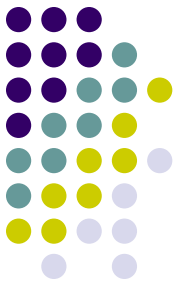
Debuggers

- Compile and link program with `-g` option:
 - `g++ -g -c prog.c`
`g++ -g -c sub1.c`
`g++ -g -o prog.x prog.o sub.o`
- The executable (`prog.x`) now contains symbols, routine names, etc. It is much larger in size with `-g` than without.



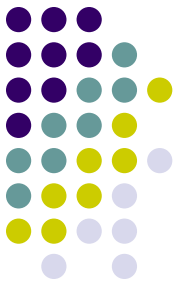
Debugger on Linux: gdb

- `gdb prog.x`
- **Commands:**
 - `print (p)` (print variables)
 - `break (b)` (set breakpoints)
 - `step (s)` (step one line, go into routines)
 - `next (n)` (step on line, step over routines)
 - `display (disp)` (display a variable)



gdb (Opensource)

- Conditional breakpoints
- print expressions
- Many many options that I do not use



Useful tools

- ddd
 - debugger with visual interface, works with gdb
 - (I have never used it)
- Vim
 - Advanced version of vi editor
 - Can interface with gdb to simplify debugging process
- Emacs (of course)