Large Programs: Linking and make

C Programming and Software Tools

N.C. State Department of Computer Science

Computer Science
NC STATE UNIVERSITY

Separate Compilation

- In Java, every class is a separate source code file
- In C, the programmer determines how to split a program up into source code files (*modules*)
  - rules/conventions for doing this?
- Each module is compiled independently to produce an object (.o) file
  - object files are then linked together to produce an executable application
  - all managed by gcc
- Benefits of separate modules?

Steps in Compiling C (Again)

Source Code

```
#define N 3
a=c+b*N;
```

preprocessing

Expanded Source Code

```
a=c+b*3;
```

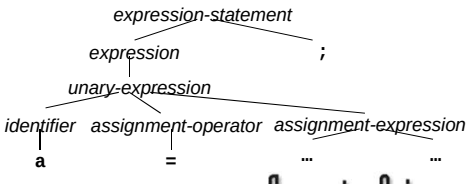
lexical analysis

Tokens

```
a  =  c  +  b  *  3  ;
```

parsing

Parse Tree



code generation

Steps… (cont'd)

code generation

Assembly Language

```
mov ebx, b
imul ebx, ebx, 3
mov ecx, c
```

assembling

Object Code

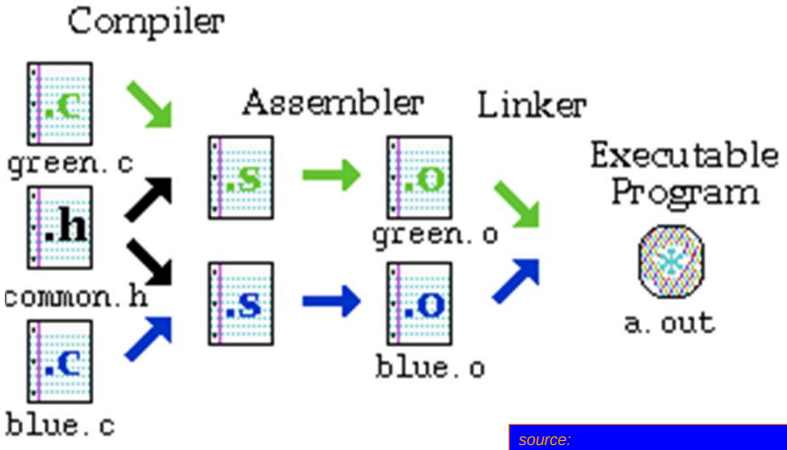
```
001110010111
```

linking + other Object Code

Executable Code

```
00111001011101101010…
```

Steps in Compiling



source: http://www.eng.hawaii.edu/Tutor/Make/1-2.html

What Does a Linker Do?

- Integrates (unifies) the address space of the separately-compiled modules
  - creates a single symbol table
  - resolves *external references* between modules
- Compile separately, then link together

generate object code, without linking

```
> gcc -Wall -std=c99 -c green.c
> gcc -Wall -std=c99 -c blue.c
```

link object code files together, produce executable

```
> gcc green.o blue.o -o a.out
```

Linking... (cont'd)

- Two types of linkers
  - static linking (at compile time)
  - dynamic linking (at run time)
- Dynamic: At runtime, link to a function the first time it is called by the program
- Ex.: common OS functions (API)
  - placed into DLL or shared library
  - loaded into memory at system boot time
- Benefits (vs. static linking) ?

External Variables in C

- Global variables and functions can be referenced by (are in the scope of) other modules
- To link to a variable or function declared in another file, use the keyword **`extern`**
- **`extern`** declares the variable/function, but doesn't define it

File p.c

```
#include <stdio.h>
extern int f( int );
int x = 3;
int main() {
    x++;
    printf("%d %d\n",
           x, f(x));
    return 0;
}
```

File q.c

```
extern int g ( int );
int f ( int a ) {
    int x = 5;
    return g(x * a);
}
```

File r.c

```
int g ( int b ) {
    return b * 3;
}
```

```
> gcc p.c q.c r.c –o pgm
```

External... (cont'd)

Extern declarations commonly collected in `.h` files, which are `#include`'d at start of `.c` file

- Warning: make sure names in files don't conflict

`static` keyword before global variable x:

- visible only within this module
- information hiding!

File p.c

```
#include <stdio.h>
extern int f( int );
static int x = 3;
int main() {
    x++;
    printf("%d %d\n",
            x, f(x));
    return 0;
}
```

File q.c

```
extern int g ( int );
int x = 5;
int f ( int a ) {
    return g(x * a);
}
```

No conflicts – x in p.c different than x in q.c, even though both are global variables

Computer Science NC STATE UNIVERSITY

Header Files

- **`extern`** allows for function and variable prototypes that are shared between C files.
  - What happens if a function **f** declared in **`foo.c`** is called in 50 other files?
- Instead, we can include **f**'s prototype in a header file and all files that use **f** can include the header.
  - The file that defines **f** should also include the header file
- Files are named **`*.h`**, where ***** typically matches the name of the **`*.c`** file that contains the function definitions

Computer Science NC STATE UNIVERSITY

Header File Example

stack.h

```
void make_empty(void);
int is_empty(void);
int is_full(void);
void push(int i);
int pop(void);
```

stack.c

```
#include "stack.h"
int contents[100];
int top = 0;
void make_empty(void)
{ … }
int is_empty(void)
{ … }
int is_full(void)
{ … }
void push(int i)
{ … }
int pop(void)
{ … }
```

calc.c

```
#include "stack.h"
int main(void)
{
   make_empty();
   …
}
```


Computer Science NC STATE UNIVERSITY

Linking to an External Library

Program **prog.c**:

```
extern int sub1(void), sub2(void);
int main(void) {
    …
    x = sub1();
    y = sub2();
```

defined in a library of functions

Using the library

```
> gcc pgm.c -L$HOME/lib -lsubs -o pgm
```

directory that contains the library

the name of the library

The order of the options can be important (check compiler)!

- `-l<libname>` may be required to come after the source code files are listed

Computer Science NC STATE UNIVERSITY

Creating Libraries?

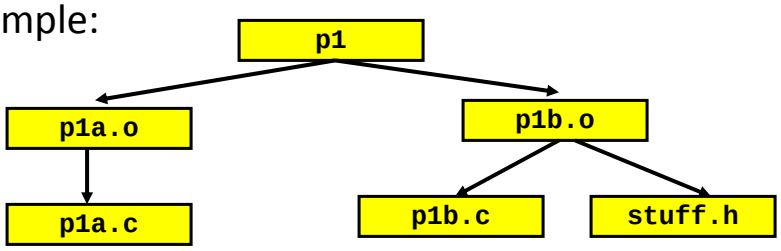
- This is very platform-specific, so we'll skip this...

Dependency Checking with `make`

- Most programming toolsuites (IDEs)…
  - keep track of what files are part of a *project*
  - keeps track of the dependencies between files
  - use this info to "create a new *build*" when files are changed
- **`make`** does the same thing, but under manual control by the programmer
  - dependencies and actions are specified in a **`Makefile`**

…Dependency Checking (cont'd)

- Example:



"p1 depends on p1a.o and p1b.o"

"p1b.o depends on p1b.c and stuff.h"

etc.

Running `make`

- First, create a file (using a text editor) called **`Makefile`**
- After that, any time part of the source code changes, run **`make`** to regenerate executable

```
$ make [-f makefilename] [options] [target]
```

Makefiles

- **`Makefile`** consists of rules of form:

```
target-list: list-of-dependencies-to-check
    list-of-commands-to-execute
```

*There is a **tab** character here – Required! Won't work otherwise!*

Helpful tips
- *blank lines help readability*
- *lines starting with '#' are comments (ignored)*
- *lines can be continued with '\' immediately before newline*

Order of Rules Important

- Order of rules important
  - by default, **`make`** generates the first target in the **`Makefile`**
  - to accomplish something else: **`make target`**
- **`list-of-commands`** can be any executable commands (programs), "shell" commands, etc.

Example

```
# sample Makefile for program p1, which
# consists of just the file p1.c
# Remember: command line *must* start with TAB

p1: p1.c
     gcc –Wall –std=c99 p1.c –o p1 -lm
```

- Means:
  "if **p1.c** has changed more recently than the last time the executable **p1** was generated, then run **gcc p1.c…**"
- i.e., make does the minimum work necessary to regenerate the target

…Example (cont'd)

```
$ make p1
gcc –Wall –std=c99 p1.c –o p1 –lm
$
```

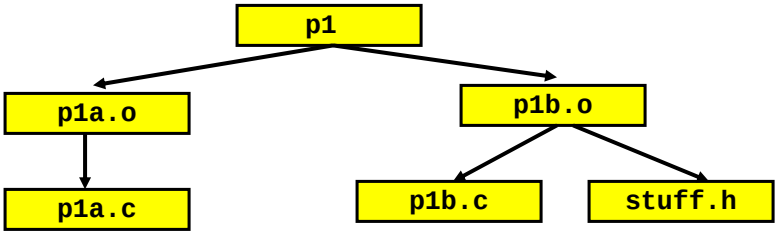
- If **p1.c** is less recent than **p1**, nothing happens

```
$ make p1
make: 'p1' is up to date.
$
```

*Note: to force remake, use **touch** command*

```
$ touch p1.c      sets modification time to now
$ make
gcc –Wall –std=c99 p1.c –o p1 –lm
$
```

Dependency "Tree" Example



Makefile

```
p1: p1a.o p1b.o
     gcc p1a.o p1b.o –lm –o p1
p1a.o: p1a.c
     gcc –Wall –std=c99 –c p1a.c -o p1a.o
p1b.o: p1b.c stuff.h
     gcc –Wall –std=c99 –c p1b.c -o p1b.o
```


Dependency Trees (cont'd)

```
$ make
gcc –Wall –std=c99 –c p1a.c
gcc –Wall –std=c99 –c p1b.c
gcc p1a.o p1b.o –lm –o p1
$
$ vi p1b.c
 ---make some changes to p1b.c---
$ make
gcc –Wall –std=c99 –c p1b.c
gcc p1a.o p1b.o –lm –o p1
$
```

only recompile p1b.c, and link with other object files

Dependency Trees (cont'd)

- The following defeats the purpose of **make**
  - what's wrong with this, vs. the previous?

```
p1: p1a.c p1b.c stuff.h
     gcc p1a.c p1b.c –o p1 –lm
```


Automating: Default Rules

- Large set of default (built-in, automatic) dependencies that **make** knows about
- Examples:
  - "**<filename>.o** usually depends on **<filename>.c**"
  - "To regenerate **<filename>.o** file, you usually run **$(CC) <filename>.c –c**"
- So, for example previously given, almost the complete Makefile is...

```
p1: p1a.o p1b.o
p1b.o: stuff.h
```

…Default (cont'd)

- To see what the complete set of defaults are:

```
$ make -p -f/dev/null
```

- There are a lot of rules…

make Variables

- Way to assign symbolic names to commands, filenames, and arguments
- Some default variables you can define
  - **CC** (default compiler to use)
  - **CFLAGS** (default compilation flags)
  - **LDFLAGS** (default linking flags)
  - **TARGET_ARCH** (target architecture)

…Macros (cont'd)

- Example

```
CC = gcc
CFLAGS = -Wall -std=c99
LDFLAGS = -lm
OBJECTS = p1a.o p1b.o
SOURCES = p1a.c p1b.c
HEADERS = stuff.h

p1: $(OBJECTS)
     $(CC) $(CFLAGS) $(OBJECTS) $(LDFLAGS) -o p1
p1a.o:
p1b.o: stuff.h
```


Passing Parameters to **make**

- We can pass macro values to a makefile by specifying them on the command line, e.g.

```
$ make CFLAGS="-O1" PAR2=sw
```

Processes the makefile with **CFLAGS** assigned the value **-O1**, **PAR2** assigned the value **sw**

Referenced in makefile as **$(CFLAGS)**, **$(PAR2)**

Some "Built-In" Macros

Macro	Meaning
$@	The current target
$?	The list of dependencies (files the target depends on) that have changed more recently than current target
$*	The "stem" of filenames that match a pattern
$^	The list of dependencies

```
p1: $(OBJECTS)
     $(CC) $(OPTIONS) $^ -o $@ -lm
```

Interpretation?

"Dummy" Targets

- Convenient label for a goal, not a file to generate

```
…
p1.tar: Makefile $(HDRS) $(SOURCES)
     tar -uvf $@ $?

clean:
     rm *.o
```

update p1.tar with any of the specified files that have changed more recently than the last time p1.tar was updated

*(Note: **make** understands shell filename expansion, like "***.o**")*

…"Dummy" (cont'd)

Then you can accomplish that target, e.g.,

```
$ make p1.tar
tar –ucf p1.tar Makefile p1a.c p1b.c stuff.h
…tar output appears here…
$ ---update p1b.c here---
$ make p1.tar
tar –ucf p1.tar p1b.c
…tar output appears here…
$ make clean
rm *.o
$
```


Some **make** Command-Line Options

Option	Meaning
`-d`	Print debugging information
`-f file`	Use `file` instead of `Makefile`
`-i`	Ignore all errors (i.e., keep going)
`-n`	Print the commands that would be executed, but don't execute them
`-s`	Silent mode; do not print commands, just execute them

Example: `compare_sorts` (1)…

```
# Makefile to compare sorting routines

BASE    =  /home/barney/progs
CC      =  gcc
CFLAGS  =  -O –Wall
EFILE   =  $(BASE)/bin/compare_sorts
INCLS   =  -I$(LOC)/include
LIBS    =  $(LOC)/lib/g_lib.a \
             $(LOC)/lib/h_lib.a
LOC     =   /usr/local

OBJS    = main.o another_qsort.o chk_order.o \
            compare.o quicksort.o
…
```


…`compare_sorts` (2)

```
…
$(EFILE): $(OBJS)
     @echo “linking …”
     @$(CC) $(CFLAGS) –o $@ $(OBJS) $(LIBS)

$(OBJS): compare_sorts.h
     $(CC) $(CFLAGS) $(INCLS) –c $*.c

# Clean intermediate files
clean:
     rm *~ $(OBJS)
```

Macro	Meaning
$@	The current target
$?	The list of dependencies (files the target depends on) that have changed more recently than current target
$*	The "stem" of filenames that match a pattern
$^	The list of dependencies

Larger Example: `vi` (1) …

```
PROG= ex
XPG4PROG= ex
XPG6PROG= ex
LIBPROGS= expreserve exrecover
XD4= exobjs.xpg4
XD6= exobjs.xpg6

EXOBJS=      bcopy.o ex.o ex_addr.o ex_cmds.o ex_cmds2.o \
      ex_cmdsub.o ex_data.o ex_extern.o ex_get.o \
      ex_io.o ex_put.o ex_re.o ex_set.o ex_subr.o \
      ex_temp.o ex_tty.o ex_unix.o ex_v.o ex_vadj.o \
      ex_vget.o ex_vmain.o ex_voper.o ex_vops.o \
      ex_vops2.o ex_vops3.o ex_vput.o ex_vwind.o \
      printf.o
EXOBJS_XPG4= $(EXOBJS) compile.o values-xpg4.o
EXOBJS_XPG6= $(EXOBJS) compile.o values-xpg6.o
XPG4EXOBJS= ${EXOBJS_XPG4:%=$(XD4)/%}
XPG6EXOBJS= ${EXOBJS_XPG6:%=$(XD6)/%}
EXRECOVEROBJS=      exrecover.o ex_extern.o
```


…`vi` (2) …

```
OBJS= $(EXOBJS) $(XPG4EXOBJS) $(XPG6EXOBJS) \
      expreserve.o exrecover.o
SRCS= $(EXOBJS:%.o=%.c) expreserve.c exrecover.c
TXTS = READ_ME makeoptions asfix.c70 ex.news \
       port.mk.370 port.mk.70 port.mk.c70 port.mk.usg \
       ovdoprnt.s ovprintf.c rofix

include ../../Makefile.cmd
#
# For message catalogue files
#
POFILES= $(EXOBJS:%.o=%.po) expreserve.po exrecover.po
POFILE= port.po

# Include all XPG4 and XPG4ONLY changes in the XPG4 version
$(XPG4) := CFLAGS += -DXPG4 -DXPG4ONLY

# Include all XPG4 changes, but don't include XPG4ONLY in the
XPG6 version
$(XPG6) := CFLAGS += -DXPG4 -DXPG6 -I$(SRC)/lib/libc/inc
```

…vi (3) …

```
CPPFLAGS +=  -DUSG -DSTDIO -DVMUNIX -DTABS=8 \
             -DSINGLE -DTAG_STACK
CLOBBERFILES += $(LIBPROGS)
ex :=        LDLIBS += -lmapmalloc -lcurses \
                       $(ZLAZYLOAD) -lgen -lcrypt_i $(ZNOLAZYLOAD)
$(XPG4) :=   LDLIBS += -lmapmalloc -lcurses \
                       $(ZLAZYLOAD) -lgen -lcrypt_i $(ZNOLAZYLOAD)
$(XPG6) :=   LDLIBS += -lmapmalloc -lcurses \
                       $(ZLAZYLOAD) -lgen -lcrypt_i $(ZNOLAZYLOAD)
exrecover := LDLIBS += -lmapmalloc -lcrypt_i
lint :=      LDLIBS += -lmapmalloc -lcurses -lgen -lcrypt

ROOTLIBPROGS= $(LIBPROGS:%=$(ROOTLIB)/%)
```


…vi (4) …

```
# hard links to ex
ROOTLINKS= $(ROOTBIN)/vi $(ROOTBIN)/view $(ROOTBIN)/editv \
           $(ROOTBIN)/vedit
ROOTXPG4LINKS= $(ROOTXPG4BIN)/vi $(ROOTXPG4BIN)/view \
               $(ROOTXPG4BIN)/edit $(ROOTXPG4BIN)/vedit
ROOTXPG6LINKS= $(ROOTXPG6BIN)/vi $(ROOTXPG6BIN)/view \
               $(ROOTXPG6BIN)/edit $(ROOTXPG6BIN)/vedit
.KEEP_STATE:

.PARALLEL: $(OBJS)

all: $(PROG) $(XPG4) $(XPG6) $(LIBPROGS)

$(PROG): $(EXOBJS)
      $(LINK.c) $(EXOBJS) -o $@ $(LDLIBS)
      $(POST_PROCESS)
```

…vi (5) …

```
…
$(XD4)/compile.o $(XD6)/compile.o: ../../expr/compile.c
	$(COMPILE.c) -o $@ ../../expr/compile.c
…
$(XD4):
	-@mkdir -p $@
…
install: all $(ROOTPROG) $(ROOTLIBPROGS) $(ROOTLINKS) \
	$(ROOTXPG4PROG) $(ROOTXPG4LINKS) $(ROOTXPG6PROG) \
	$(ROOTXPG6LINKS)
…
clean:
	$(RM) $(OBJS)

lint: lint_SRCS
```


Autoconf and Automake

- **autoconf** automates the setting of options which are system configuration-dependent
- **automake** automates the generation of large **Makefiles**
- Details for another day...

Building Java Projects: Ant

- Provides tasks for compiling, assembling, testing and running Java applications
  - XML based that calls Task objects that run the task
  - Cross-platform
  - Can also be used to build non-Java applications like C or C++ applications
  - If you would also like dependency management, combine with Apache Ivy

Building Java Projects: Maven

- Building and managing any Java-based project
  - Uses a Project Object Model (POM) and plug-ins shared by all projects to build a project
  - Provides additional information like change logs, mailing lists, dependency lists, test coverage
  - Easy way to share libraries (as JARs) across several projects
    - The JARs don't have to be maintained in the project itself