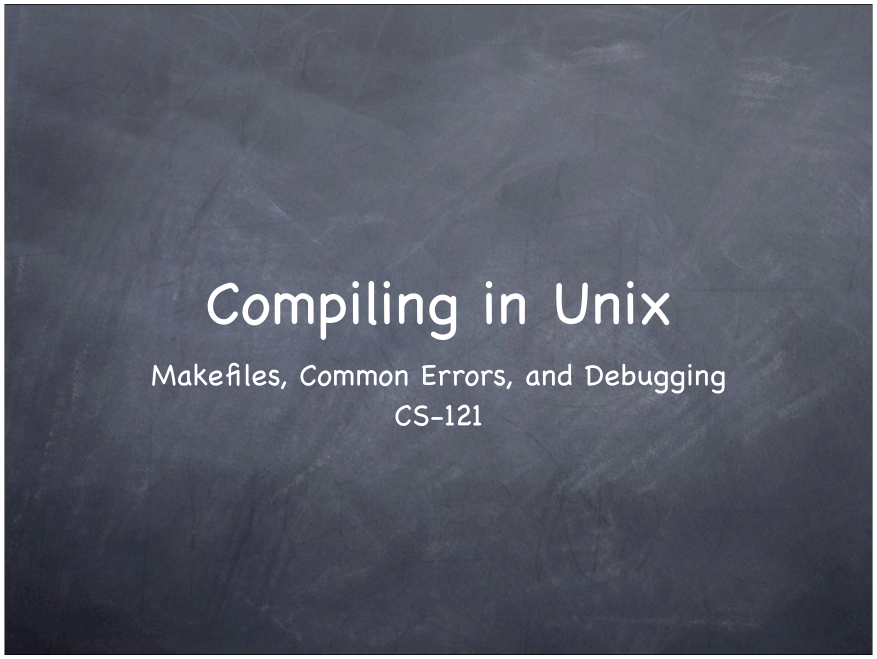
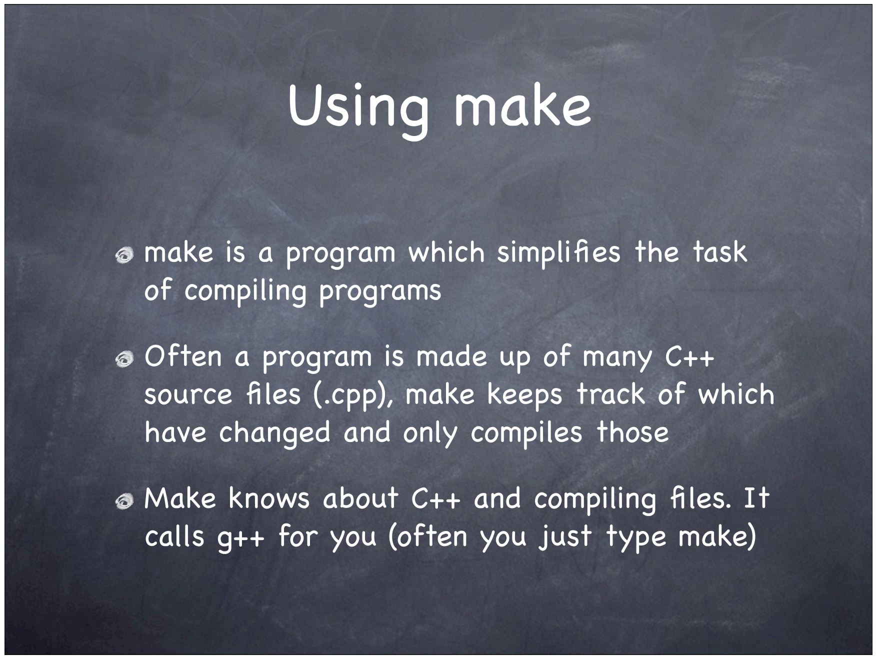


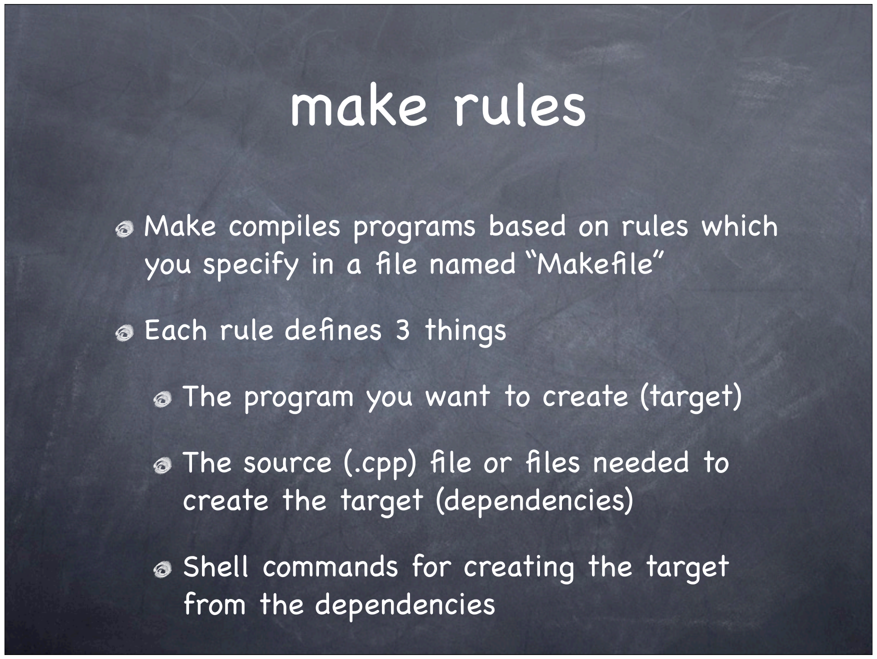
Compiling in Unix

Makefiles, Common Errors, and Debugging
CS-121



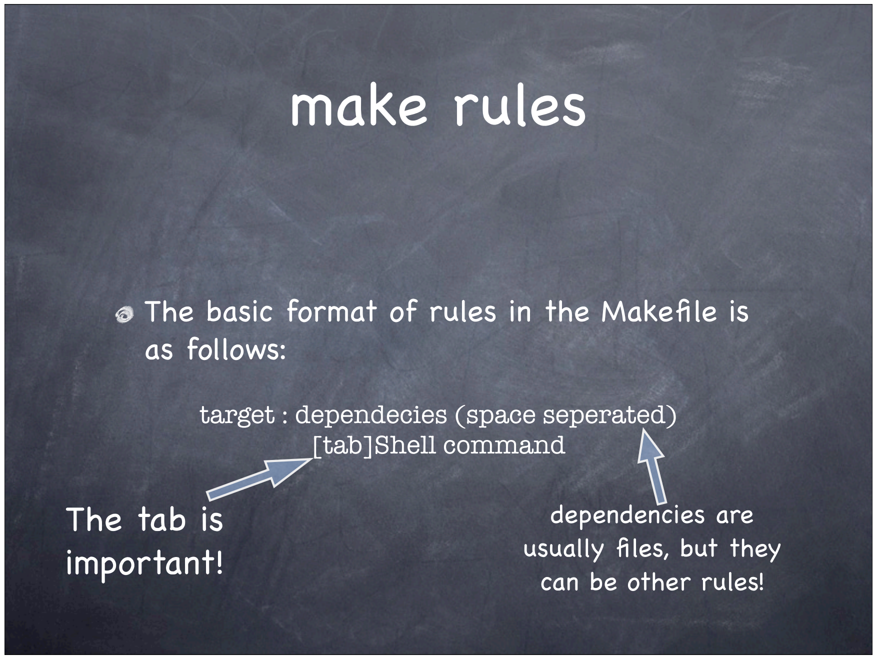
Using make

- make is a program which simplifies the task of compiling programs
- Often a program is made up of many C++ source files (.cpp), make keeps track of which have changed and only compiles those
- Make knows about C++ and compiling files. It calls g++ for you (often you just type make)



make rules

- Make compiles programs based on rules which you specify in a file named "Makefile"
- Each rule defines 3 things
  - The program you want to create (target)
  - The source (.cpp) file or files needed to create the target (dependencies)
  - Shell commands for creating the target from the dependencies



make rules

- The basic format of rules in the Makefile is as follows:

```
target : dependecies (space seperated)
        [tab]Shell command
```

The tab is important!

dependencies are usually files, but they can be other rules!

make rules

- Example:

```
helloworld : helloworld.cpp
[tab] g++ helloworld.cpp -o helloworld
```

Because make already knows how to make cpp programs you can leave the shell command out

```
helloworld : helloworld.cpp
```

make rules

- You can have any number of rules in a makefile
  - The first rule is the default rule -- it's executed by just typing make
  - Other rules you need to type make [targetname]

make rules

- You can have phony rules
  - phony rules don't make anything but they are useful

```
clean :
[tab] rm helloword
```

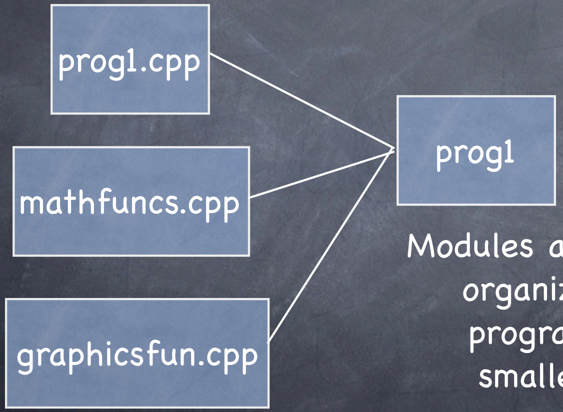
the clean rule doesn't have any dependencies this is okay

Makefile example

Modules

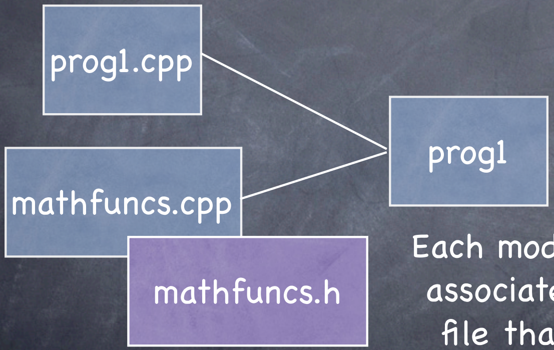
- A program may be made of multiple modules.
- Each module is defined as in its own cpp file.

Modules



Modules allow you to organize large programs into smaller files

Modules



Each module has an associated header file that defines function prototypes and shared variables

Modules

prog1.cpp

```
#include <iostream>
#include <string>
#include "mathfuncs.h"
```

Other modules include this header file so that they become aware of functions or variables defined in other modules.

Modules

prog1.cpp

```
#include <iostream>
#include <string>
#include "mathfuncs.h"
```

We use quotes "" when specifying local headers and <> for system headers.
Headers specified with "" are searched for in the current directory.

Modules

prog1.cpp

```
#include <iostream>
#include <string>
#include "mathfuncs.h"
```

We always include local headers AFTER we include system headers. Not a strict rule, but good form.

Header Files

mathfuncs.h

- Define:
  - Functions
  - Variables
  - Data structures

Header Files

mathfuncs.h

Header files look like this typically:

```
#ifndef __MATHFUNCS_H_
#define __MATHFUNCS_H_

extern int compute_dist(int x, int y);
extern float compute_dist (float x,
                           float y);
#endif
```

Header Files

mathfuncs.h

These preprocessor commands insure you don't accidently include the same header twice!

```
#ifndef __MATHFUNCS_H_
#define __MATHFUNCS_H_

extern int compute_dist(int x, int y);
extern float compute_dist (float x,
                           float y);
#endif
```

Header Files

mathfuncs.h

Notice we have typical function prototypes but these are defined as external.

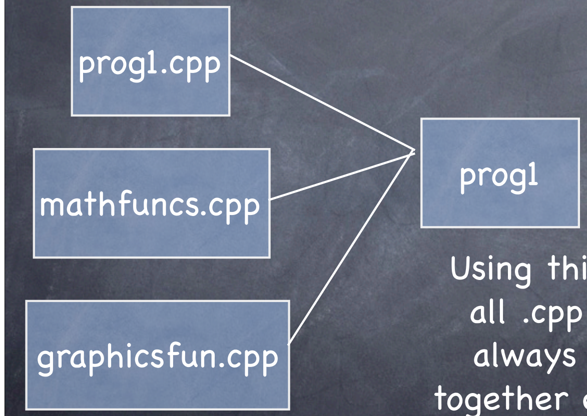
```
#ifndef __MATHFUNCS_H_
#define __MATHFUNCS_H_

extern int compute_dist(int x, int y);
extern float compute_dist (float x,
                           float y);
#endif
```

Compiling Modules

- You really need to use a make file.
- Simply specifying all cpp files as dependencies will combine all modules to form the program.
- Here prog1 is created from 3 modules.
- The main module has the same name as the program.

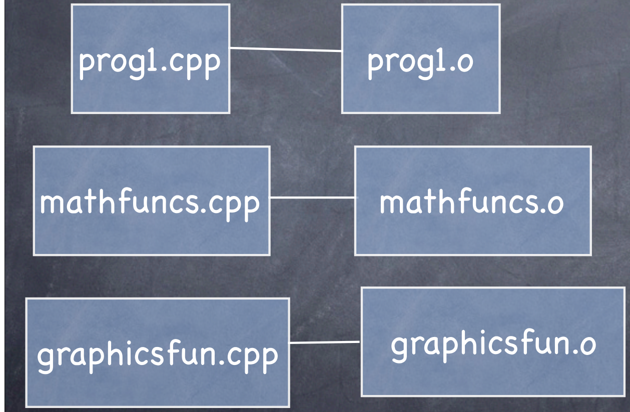
```
CPPFLAGS=-g

prog1 : prog1.cpp mathfuncs.cpp graphicsfun.cpp
```

Compiling Modules



Using this makefile all .cpp files are always compiled together even if only one file chages!

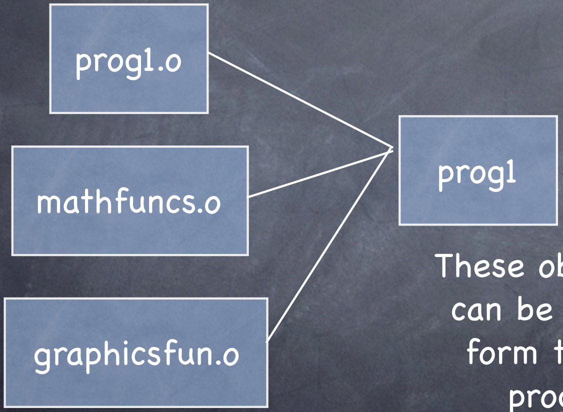
Compiling Modules



We can compile cpp files separately.

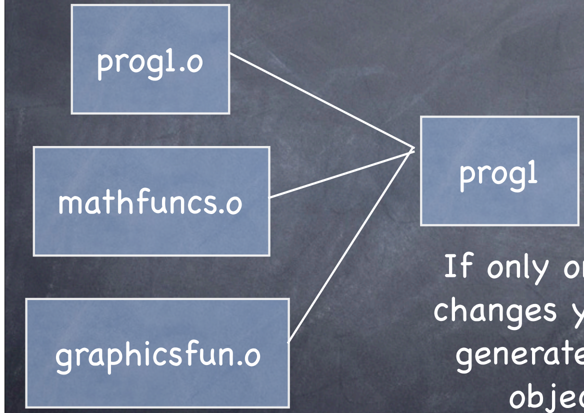
This creates "object" files which are parts of C++ program in machine code.

Compiling Modules



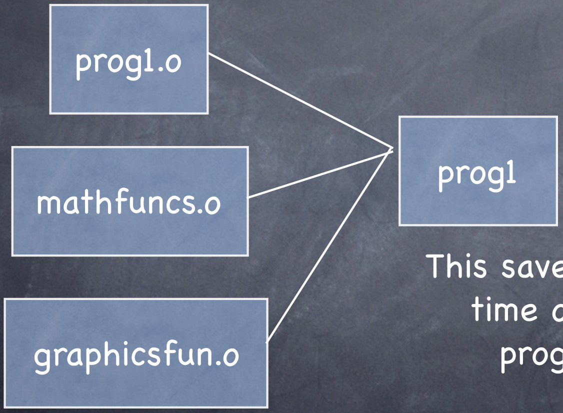
These object files can be linked to form the final program.

Compiling Modules



If only one cpp file changes you need to generate only one object file.

Compiling Modules



This saves loads of time on large programs!

Compiling Modules and Linking

- Notice dependencys are now object files not cpp files.
- Make is smart enough to know it needs to generate object files from cpp files you don't have to tell it.

```
CPPFLAGS=-g
CC=g++

prog1 : prog1.o mathfuncs.o graphicsfun.o
```

Compiling Modules and Linking

- CC=g++ tells make to use the cpp compiler for linking, the default is to use the c compiler.
- Make is smart enough to know which object files to generate when you change a .cpp file

```
CPPFLAGS=-g
CC=g++

prog1 : prog1.o mathfuncs.o graphicsfun.o
```

Compiling Modules and Linking

- You can store all of the object file in a variable and just reference that variable in your rule.

```
CPPFLAGS=-g
CC=g++
OBJS= prog1.o mathfuncs.o graphicsfun.o

prog1 : $(OBJS)
```

Modules and Scope

- When a variable is global it is global to all modules, but in order to see the variable it must be defined with the extern keyword.
- These definitions are typically made in local header files.

```
extern int i;
```

Modules