Managing **Large** Programs

The "make" program

Managing Large Programs

- large programs can be *very* large: thousands or even millions of lines in hundreds of files

- a C program can have many .c and .h files

- whenever we change one of the files (.c or .h) we have to recompile all of the unchanged ones as well

- this is very time consuming

Example

- we have C files:

  labels.c readml.c printml.c util.c

- we make a change to util.c and then recompile by:

  gcc labels.c readml.c printml.c util.c -o labels

- we are recompiling ALL the C files when only one has been changed

Recall

- We can break down each stage of the compilation by using separate programs or by providing flags to the compiler
- Preprocessing
- Syntax checking
- Translation to assembly
- Compose object files
- Link symbols between object files
- Produce binary executable

Example using .o files

- we initially compile the C files to .o:

  gcc -c labels.c readml.c printml.c util.c

- we make a change to util.c and then recompile by:

  gcc -c utils.c

  gcc labels.o readml.o printml.o util.o -o labels

- we are recompiling the file that has been changed and then relinking with the other .o files

- BUT if we have a large number of files and make many changes, how do we remember which files we've changed and recompile them?

Problem: what happens if we change a .c file and forget to recompile it?
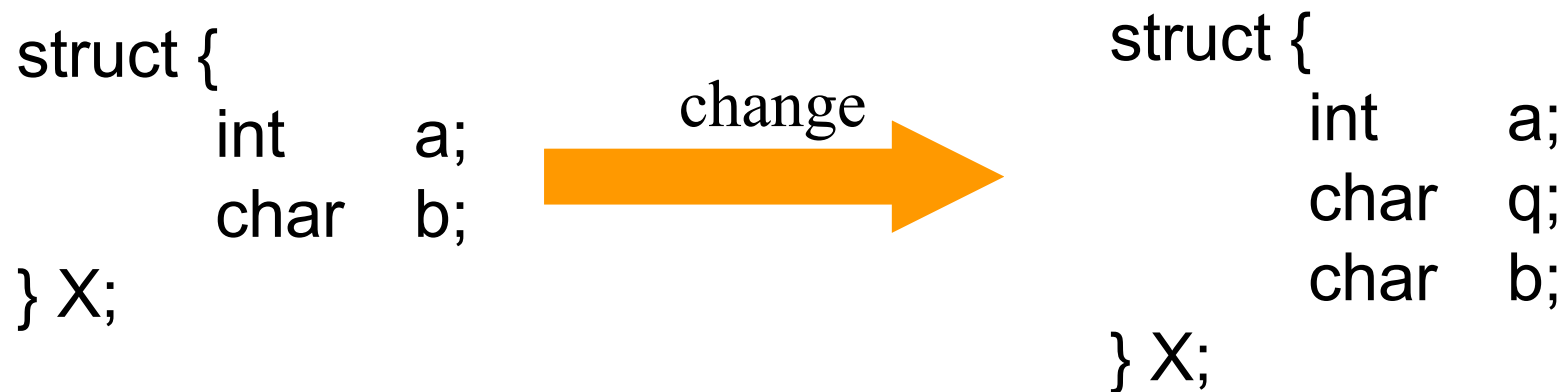
Problem: what happens if we change a .h file and forget to recompile *some* of the files that #include it?

Answer: your program fails!
          or gives the wrong answer!

The real answer: use the *make* program to recompile and relink your files

Why does the program fail?

- if changes are made to data structures but code that refers to it is not recompiled, the references may now go to the wrong place

```
struct {                              struct {
    int     a;        change              int     a;
    char    b;        ──────▶             char    q;
} X;                                      char    b;
                                      } X;
```

- a function that referred to X.b would not get the correct value if it wasn't recompiled after the change

Make

- the *make* program will automate the process of recompiling and recompile all the files that have been changed

- *make* interprets a set of rules and will run the C compiler and the linker as necessary to make your program

- the rules are normally stored in a file called Makefile or makefile

Make rules

- in general, make rules have:

- a *target:* the name of the file you want to make

- one or more *dependencies:* files the target depends on

- an *action:* a shell command that creates the target

Example rule

- to make mystring.o from mystring.c we might use the rule:

mystring.o: mystring.c mystring.h

gcc -c mystring.c

target

action

dependencies

Example rule

- if mystring.c or mystring.h have changed since mystring.o was created then the action is performed to recreate mystring.o

- how does the make program know that a file has been changed?

```
# first version of makefile for labels program
labels: labels.o util.o
        gcc labels.o util.o -o labels

labels.o: labels.c ml.h
        gcc -c labels.c

util.o: util.c ml.h
        gcc -c util.c
```

- To remake the labels program just type:

```
make
```

Default rules

- these rules are very repetitive: every action is the same except for the file name

- make provides a set of default rules for common situations

- these defaults can be overidden and rules for new file types can be entered

Default rules

- for example, there is a default rule that specifies how to make a .o file from a .c file:

filename.o : *filename.c*

      gcc -c *filename.c*

```
# second version of makefile for labels program
#   (using default rules)
labels: labels.o util.o
        gcc labels.o util.o -o labels

labels.o: ml.h

util.o: ml.h
```

- make knows the default rule for labels.o and util.o
- still have to specify a rule for the case where ml.h changes

Combining rules

- you can combine rules when the targets
  have common dependencies and actions:

labels: labels.o util.o

          gcc labels.o util.o -o labels

labels.o util.o: ml.h

Rules without dependencies

- it is often useful to create rules that don't have dependencies and *always* activate

- for example, a rule to cleanup the file space:

```
labels:         labels.o util.o
                gcc labels.o util.o -o labels
labels.o util.o: ml.h
clean:
                rm *.o
```

Make variables

- sometimes you need to refer to a list of files at several places in the make file
- if you retype the list in each place you can introduce errors if you leave out a file in one or more places
- make has a variable or macro facility that lets you type the list in one place and refer to it using the variable name

Make variables

- assignments have the form:
  variable_name = any character sequence

- the value of the variable is substituted with:
  $(variable_name)

```
# makefile for labels program

OBJECTS = labels.o util.o

labels: $(OBJECTS)
        gcc $(OBJECTS) -o labels

$(OBJECTS):ml.h

clean:
        rm $(OBJECTS)
```

Predefined variables

- make has a number of predefined variables:

    CC   the default C compiler
    CFLAGS flags passed to the compiler

- you can change the value of these variables:

    CFLAGS = -DDEBUG -W -Wall -pedantic -ansi

    this would cause the DEBUG symbol to be defined

Libraries

- a large system might consist of several programs that share a number of functions

- libraries give you the ability to store the object code versions of the functions in one place and have them linked into your program

- there are many libraries of functions provided by the system

- the standard C library has the standard I/O functions, system functions, sort function, etc

- the standard library is automatically searched when your program is linked

Libraries

- you can use functions from other libraries by using the -l flag when linking, eg

  gcc -lm myprog.o -o myprog

- this will search the standard maths function library for functions such as sin, cos etc

- the C compiler will search for the library in standard directories: /lib, /usr/lib

Making your own libraries

- you make your own libraries using the "ar" command on the .o files:

  ar c mylib.a readit.o util.o

- the library will be called mylib.a and will contain your .o files
- you can use the library with a command like:

  gcc myprog.o mylib.a -o myprog

Summary

- C programs very quickly get to a size where they require several files

- large programs can have hundreds of .c and .h files that depend on each other

- the **make** tool helps you maintain an up-to-date version of your program

- libraries can also help maintain large systems