Announcements

- Program #1A
  - due next Thursday
- Email
  - please send email to course staff from UMD accounts
  - email from hotmail, yahoo, aol, etc. will likely be auto deleted
- Programming Environment
  - Must use Linux, disregard suggestions from section about Eclipse or Visual studio
- Reading
  - Chapter 3 & 4
  - Chapter 7 (Thursday)

Make: A Tool to Compile Programs

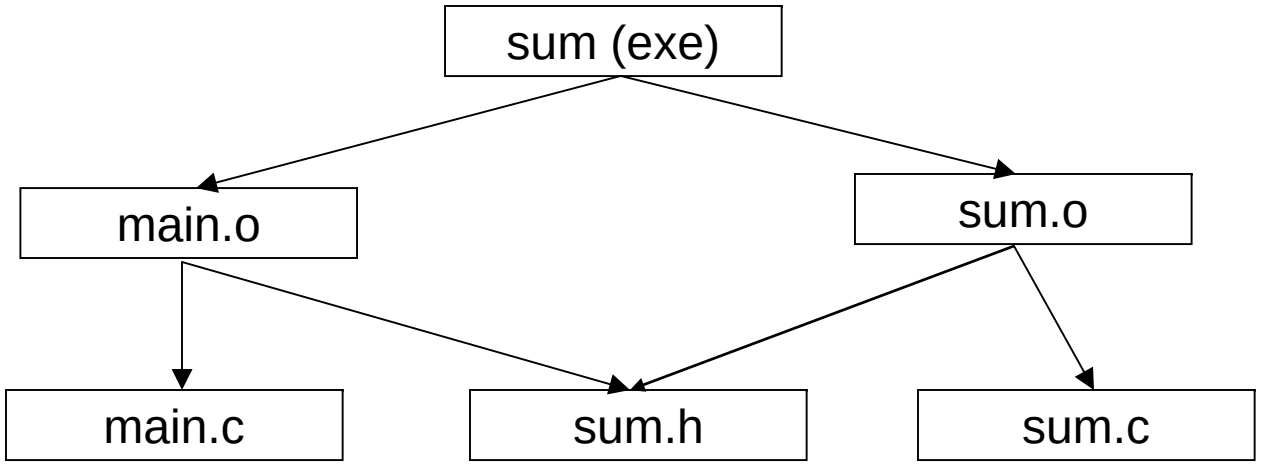
- Problem: Compiling programs is tedious
- Solution: Automate it!

- Done in Unix by the make command
  - Uses a file called Makefile
    - A makefile is a file (script) containing :
      - Project structure (files, dependencies)
      - Instructions for files creation
- Make is not limited to C programs

Adapted from: www.cs.tau.ac.il/~dcor/Software1/Makefile.**ppt**

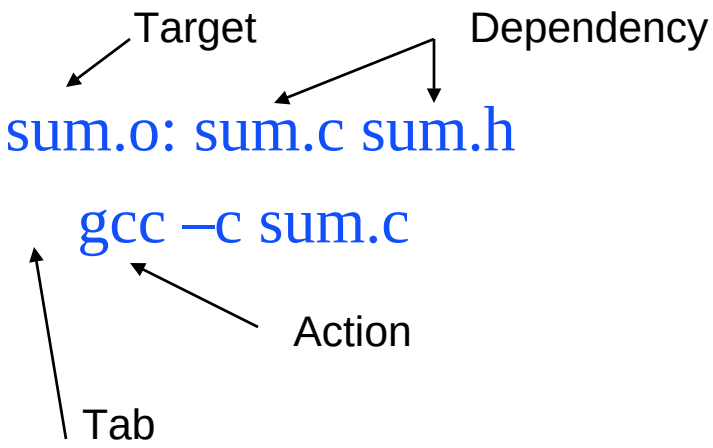
Describing Files to Make

- Makefile contains project dependencies
  - represented as a DAG (= Directed Acyclic Graph)
- Example:
  - Program contains 3 files
  - main.c., sum.c, sum.h
  - sum.h included in both .c files
  - Executable should be the file sum



Makefile for Previous Example

```
sum: main.o sum.o
    gcc –o sum main.o sum.o

main.o: main.c sum.h
    gcc –c main.c

sum.o: sum.c sum.h
    gcc –c sum.c
```



- Notes About Makefiles:
  - Target is first item in a dependency
  - Items after ":" required to build target
  - Actions must have a tab at start
  - First target in file is the default root

Make's Operation

- Project dependencies tree is constructed
- A target is out of date when one of its dependencies is newer than itself.
- If out of date, recreate the target
  - Perform action specified
  - Done on the way up the tree.
- Notes:
  - Ensures minimum compilation, with proper project structure
  - Avoid writing something like:

```
prog: main.c sum1.c sum2.c
    gcc –o prog main.c sum1.c sum2.c
```

    - requires full compilation when something changes

Additional Make Features

- Macros
  - <variable> = <value>
  - Example:

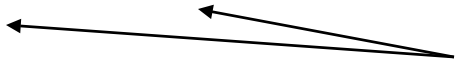
```
CC        =   gcc
CFLAGS    =   -g -Wall

Foo.o: foo.c
        $(CC) $(CFLAGS) –c foo.c
```

- Default Action
  - $(CC) –c $(CFLAGS)
- Actions can sometimes delete files not create them
  - Example:

```
Clean:
        rm *.o
```

More Complex Makefile

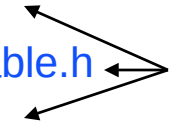
```
all:    intCount table

clean:
        rm -f *.o *.bb *.bbg *.da *.png

intCount:       intCount.o
        $(CC) -o intCount intCount.o

table:  table.o main.o hashTable.o
        $(CC) -o table table.o main.o hashTable.o

table.o: table.c table.h
hashTable.o:  hashTable.c table.h
main.o: main.c table.h

CC = gcc
CFLAGS = -g -Wall
```



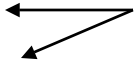
Two top-level programs

Not part of main tree
built by typing “make clean”

Link together several files



Uses Default Rule



Macro Definitions

Data Types in C

- Integer Family
  - char typically 8 bits
  - short typically 16 bits
  - int typically 32 or 64 bits
  - long typically 32 or 64 bits
  - long long typically 64 bits
  - All are signed by default, but can be made unsigned
    - unsigned int (typically, 0 to 4,294,967,295)
  - Literals
    - Decimal (255), Hex (0xff), signed (-255)
    - Character ('a', '\n')
  - Don't be stingy with size, when in doubt use a larger size
- Floating Point
  - float, double, long double
  - Literals (3.14159, 1E10, 25., 6.023e23)

Data Types in C (cont.)

- String Literals
  - “a long dull string”, “\n”, “”
- Enumerated Types
  - Declaring an enumerated type:
    - enum operatorType { plusOperator, minusOperator };
  - Declaring a variable of an enumerated type:
    - enum operatorType currentOperator;
  - Using an enumerated type in an expression:
    - currentOperator = plusOperator;
  - Can control values of enumerated type:
    - enum operatorType { plusOperator=3, minusOperator =8};

Variable Declarations

- <type> <variable1>, <variable2>, … ;
  - int a, b, c;
- Variables names may not be reserved words
  - Must start with A-Z,a-z,_
  - May contain numbers or "_"
    - aValid_Variable_Name
    - aValid_Variable_Name_2
- May initialize variables as part of declaration
  - int a = 4;
- Implicit Declarations (Book section 3.2.4)
  - Skip section book and don't ever use

Simple Arrays

- Array dimensions must be known at compile time
- Array Declaration
  - Int a[10]; /* array of 10 elements */
- Accessing array elements
  - Arrays start from 0 and go to n-1 elements
  - Compiler will let programmer access invalid elements
    - Can lead to many problems that are hard to find
  - Examples of array references
    - A[0]
    - A[i]
    - A[i+3]

Typedef

- Sometimes you want to name your own type
  - Makes program easier to read
  - Makes it easier to change types later
- typedef double dollars;

Variable Scope

- File Scope
  - Entire file
  - Not declared inside any function
- Block Scope
  - Before an executable statement in a block of code { .. }
- Repeating same variables in nested scope

```
void main() {
   int a;
   { int a;
     a = 3;
   }
}
```

  - Should be avoided!!

Storage and Linkage

- Linkage
  - Determines how multi-file visibility issues are resolved
  - extern - There is only one instance of this variable
  - static - This variable is private to this file

    foo.c:

    ```
    static int a;
    ```

    bar.c:

    ```
    static in a;
    ```
- Storage Classes
  - Default is automatic:
    - one variable instance per function instance
    - lifetime limited to while function is active
      - running or in called subroutine
  - Static changes default
    - one variable instance
    - lifetime is the entire program

Storage Class and Linkage Example

```
extern int b;
static int c;
Int func1(int e)
{
    static int longLived;
    int funcLifetime;

    func1(42);
}
```