Introduction

CMSC 341, Park

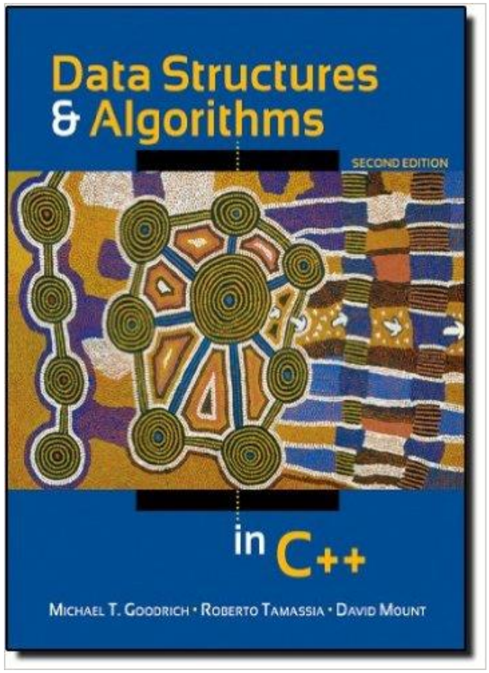
- Why are you here?!?!
- What are data structures?
- Layout of course info—mostly web pages

  /userpages.umbc.edu/~park/cs341.f18/

  Schedule, HWs and projects, exams, staff info, lecture notes
- We will be following book closely
  - Be sure to read before you arrive in class
- Not teaching <u>programming</u>: teaching *how to think*

Textbook

- Data Structures and Algorithms in C++
    - 2nd Edition
- Goodrich, Tamassia, and Mount



- ISBN-13: 978-0-470-46044-3
- ISBN-10: 0470383275
- Publisher: Wiley
- Copyright: 2011

Student Honor Code

UMBC Student Honor Code

By enrolling in this course, each student assumes the responsibilities of an active participant in UMBC's scholarly community in which everyone's academic work and behavior are held to the highest standards of honesty. Cheating, fabrication, plagiarism, and helping others to commit these acts are all forms of academic dishonesty, and they are wrong. Academic misconduct could result in disciplinary action that may include, but is not limited to, suspension or dismissal.

To read the full Student Academic Conduct Policy, consult the UMBC Student Handbook, the Faculty Handbook, or the UMBC Policies section of the UMBC Directory.

http://www.umbc.edu/provost/integrity/index.html

Prerequisites

- 202:
  - Classes: design and use
  - STL
  - Overloading, overriding
  - Debugging
- 203:
  - Proof by induction
  - Permutations and combinations

Topics Covered in This Course

- Linear data structures:
  - Lists, Stacks, Queues

- Trees
  - BST, Red-Black Tree, AVL Tree, Priority Queue

- Lists masquerading as trees
- Trees masquerading as lists

- Graphs and Disjoint Sets

- Hashing

Data Structures

- What is a “data structure” anyway?
  - A data structure is a systematic way of both organizing and accessing data
- What are some types of data structures?
  - Lists, arrays, records (like tuples and structs), linked lists, matrices, and also things like images
- How do you choose which one to use?
  - Efficiency – adding, finding, and organizing data

Abstract Data Types

- What is an ADT?
  - A mathematical model of a data structure that specifies how it behaves: type of data stored, allowed operations, and operation behavior.
- How are ADTs different from data structures?
  - ADTs are the “what” and data structures the “how”
  - ADTs can be viewed from a user’s point of view, data structures from an implementer’s view

Miscellaneous

- Tools/IDEs
  - Emacs, Eclipse, Visual Studio
  - However, final test must be on GL
- Project & HW submissions
- Make

Course Tools – Running on GL

- You may use any IDE to develop your code
- Your program ***MUST*** run (correctly) on GL in order to get credit
  - Make sure you test it on GL before submitting
- If it runs on your machine, but not on GL…
- It doesn't run for us, so it doesn't count ☹

Make

- Make
  - Basic structure: rule = target/dependencies/actions (sometimes called “target/prerequisites/recipes”)
  - Dependency recursion
  - Default rules, helper rules
  - Implicit rules
  - Variables/macros

Why Even Use “**`make`**”?

- Compiling, linking, and executing become…
  - Easier
  - Quicker (more efficient)
  - Less prone to human error

- Also allows us to create and run helper rules
  - Clean up unneeded files (like **`hw2.cpp~`**)

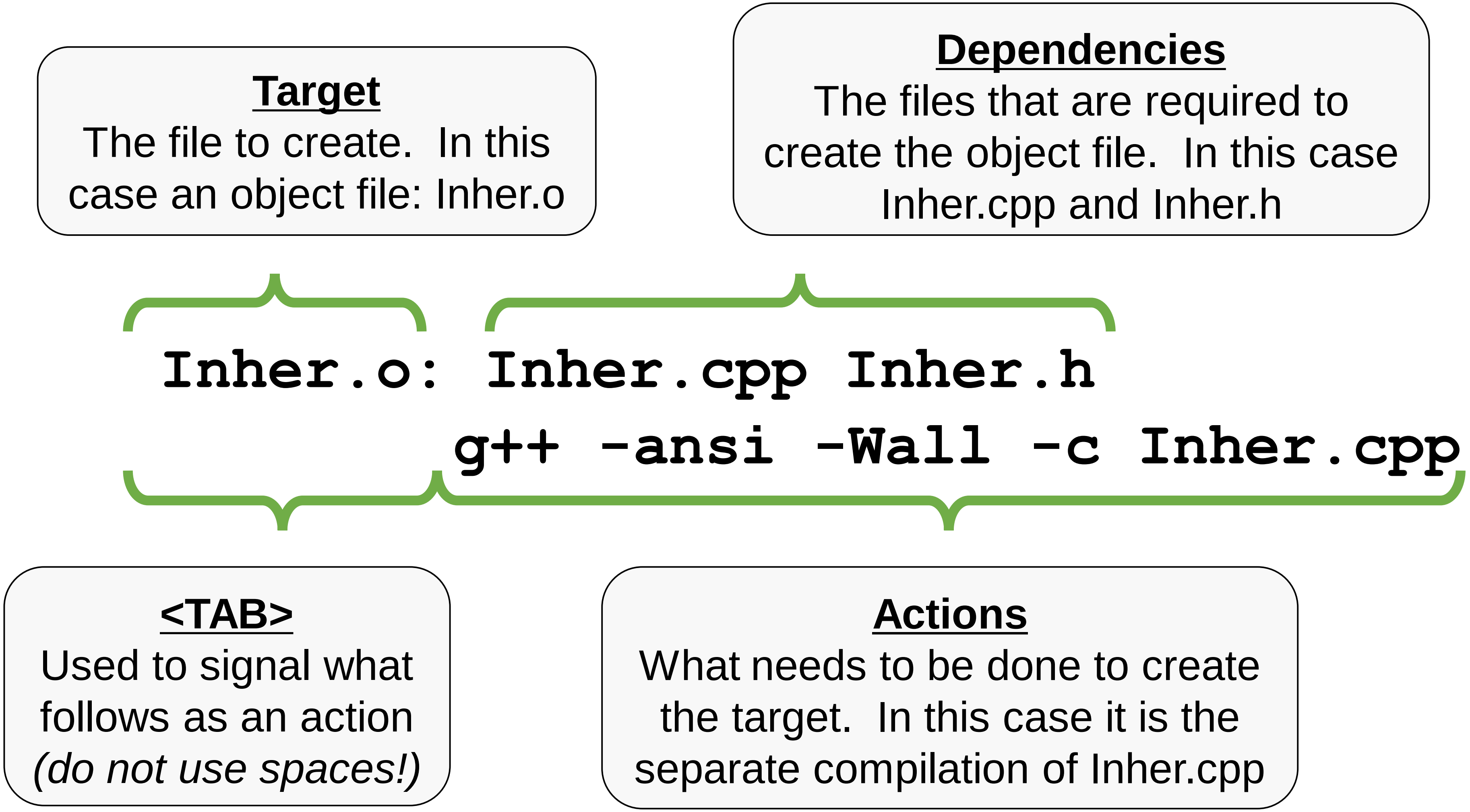
- Laziness (but *efficiently* lazy)

Makefiles

- A makefile is a list of rules that can be called directly from the terminal
  - best if called `Makefile` or `makefile`

- Rules have three parts
  - **Target** – name of object or executable to create
  - **Dependencies** – what Target depends on
  - **Actions** – list of actions to create the Target

Makefile Rule Example



Target
The file to create. In this case an object file: Inher.o

Dependencies
The files that are required to create the object file. In this case Inher.cpp and Inher.h

```
Inher.o: Inher.cpp Inher.h
	g++ -ansi -Wall -c Inher.cpp
```

<TAB>
Used to signal what follows as an action *(do not use spaces!)*

Actions
What needs to be done to create the target. In this case it is the separate compilation of Inher.cpp

Efficiency of **`make`**

- **`make`** only recompiles files that need to be
  - Files that have been modified or updated
  - Files that depend on modified/updated files

- Compares the timestamp of the dependency list items to that of the target
  - If a source is newer than the object file, the object file needs to be recompiled
  - Likewise if an object file is newer than the executable it needs to be re-linked

Example Makefile

```
Project1.out: Project1.o Inventory.o Cd.o Date.o
  g++ -Wall -o Project1.out Project1.o Inventory.o Cd.o Date.o

Project1.o: Project1.c Inventory.h
  g++ -Wall -c Project1.c

Inventory.o: Inventory.c Inventory.h Cd.h
  g++ -Wall -c Inventory.c

Cd.o: Cd.c Cd.h Date.h
  g++ -Wall -c Cd.c

Date.o: Date.c Date.h
  g++ -Wall -c Date.c
```

Specifying a Target

- To call a specific rule or create a specific target, use **`make <TARGET>`**
- The first target in the file is the “default target”
  - Omitting the target (i.e., typing just “**`make`**”) will create the default target

Dependency Graph

- A file may depend on one or more other files
  - Need to ensure correct compilation order

- Create a dependency graph, with the end goal of a executable named “main”

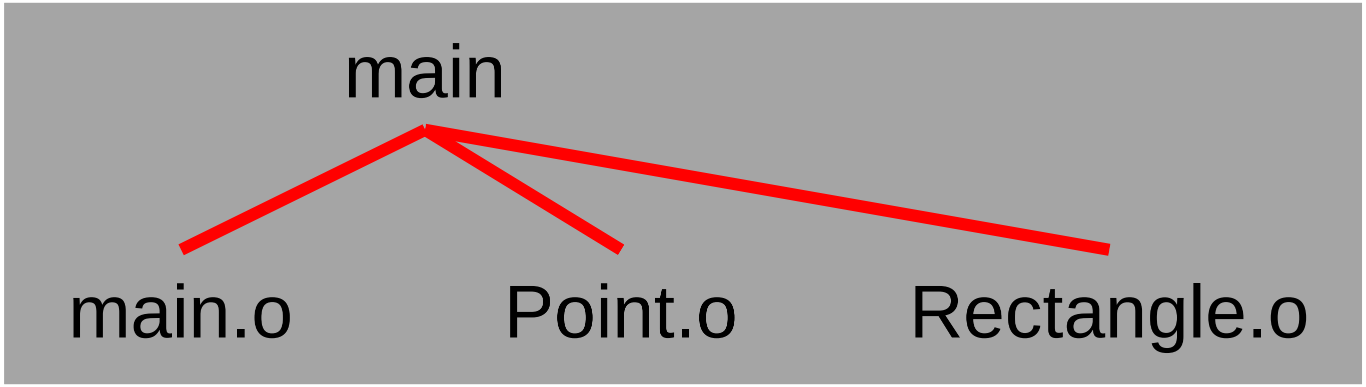
Our files:

```
main.cpp
Point.h       Point.cpp
Rectangle.h   Rectangle.cpp
```

Source: https://www.cs.bu.edu/teaching/cpp/writing-makefiles/

Dependency Graph – Linking

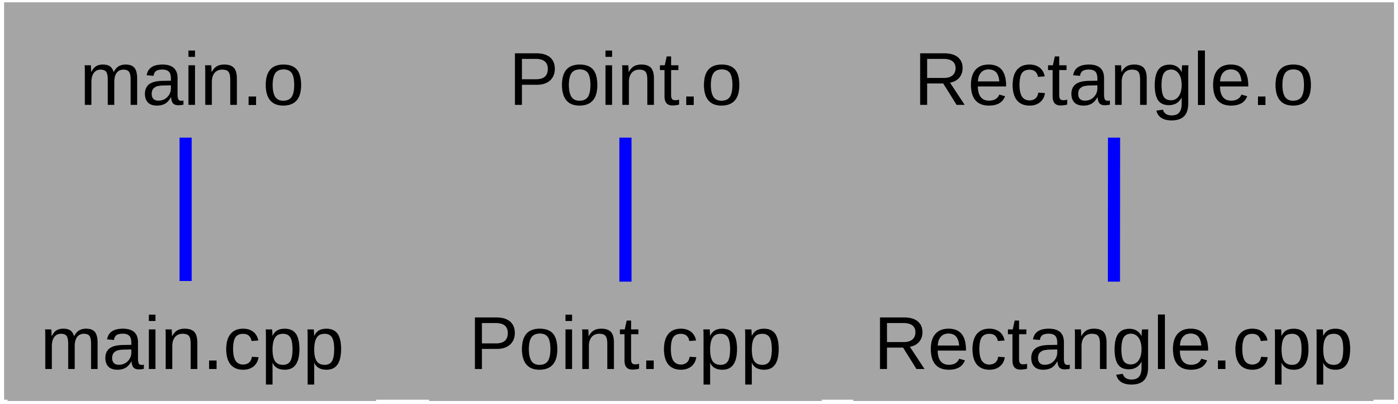
- The “main” executable is generated from 3 object files: **`main.o   Point.o   Rectangle.o`**
  - “main” *depends* on these files
- Explicitly creating .o files is more efficient
- Files are *linked* together to create “main”



Source: https://www.cs.bu.edu/teaching/cpp/writing-makefiles/

Dependency Graph – Compiling

- Each of the object files *depends* on a corresponding `.cpp` file
- Object files are generated by *compiling* the corresponding `.cpp` files



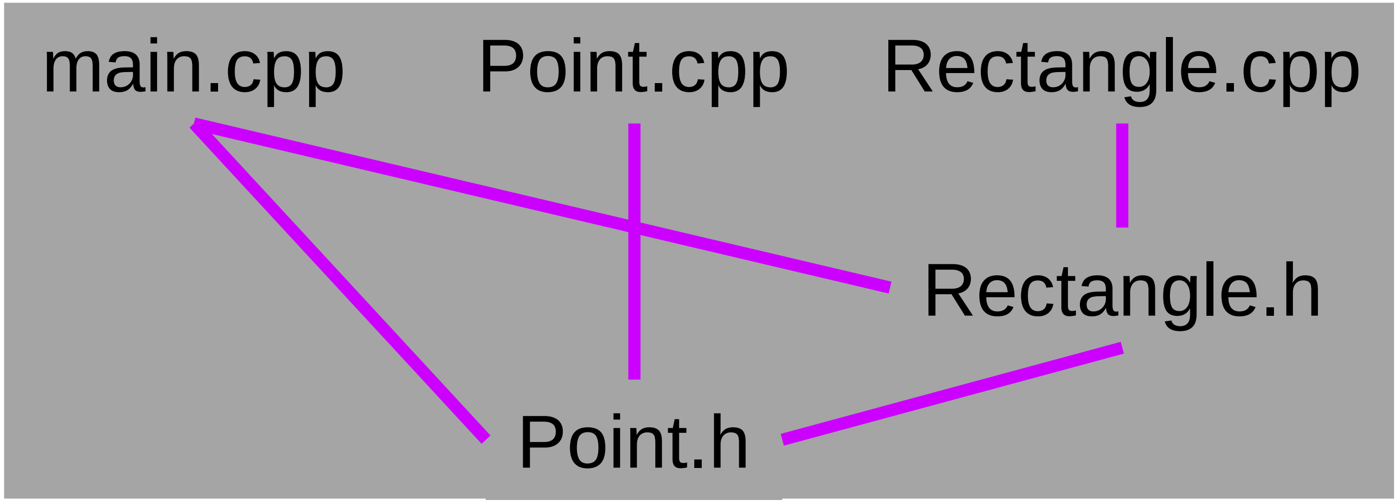
Source: https://www.cs.bu.edu/teaching/cpp/writing-makefiles/

Dependency Graph – Includes

- Many source code files (**`.cpp`** and **`.h`** files) depend on *included* header files

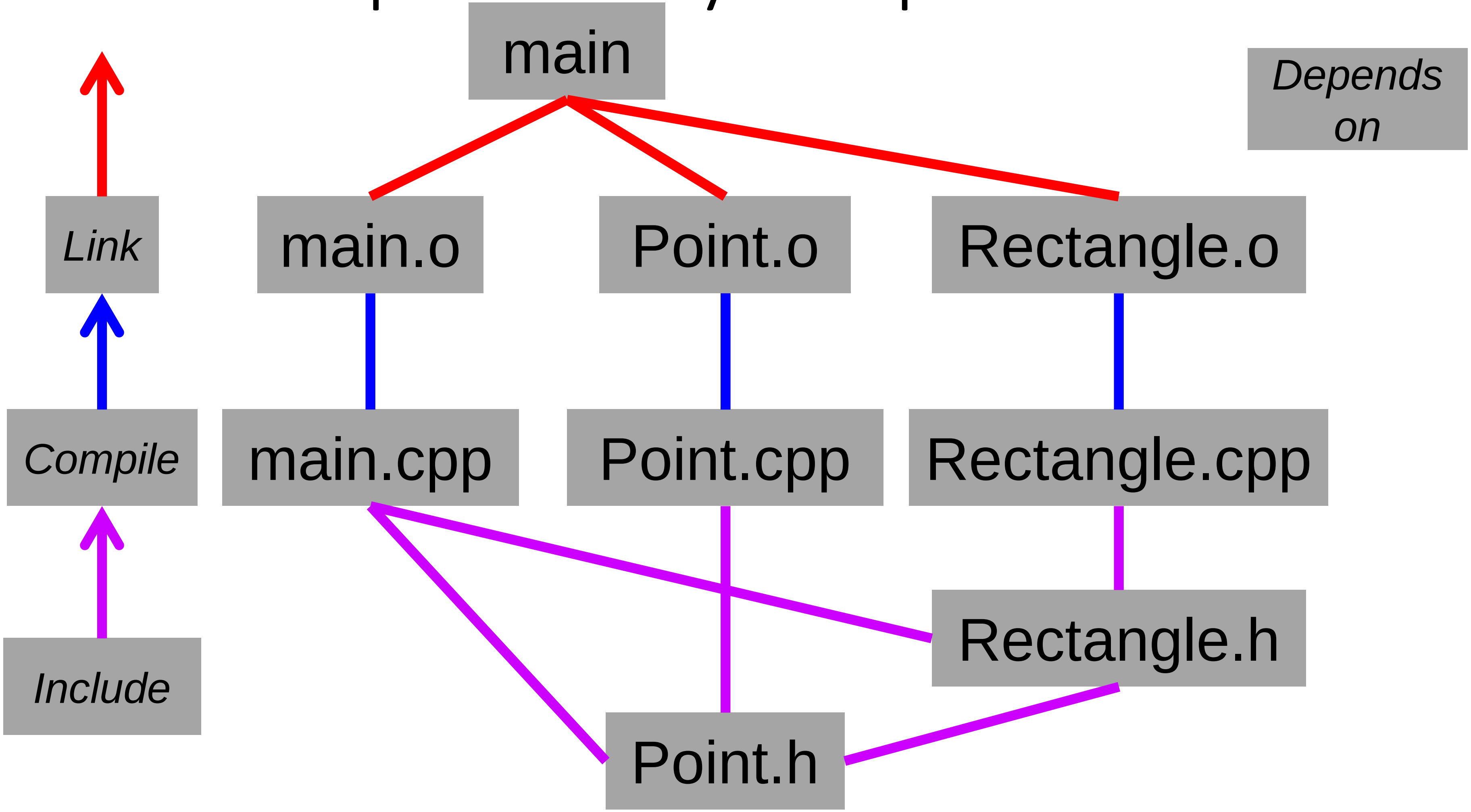
- May also be indirect includes; for example

`Rectangle.cpp` includes **`Point.h`** through **`Rectangle.h`**



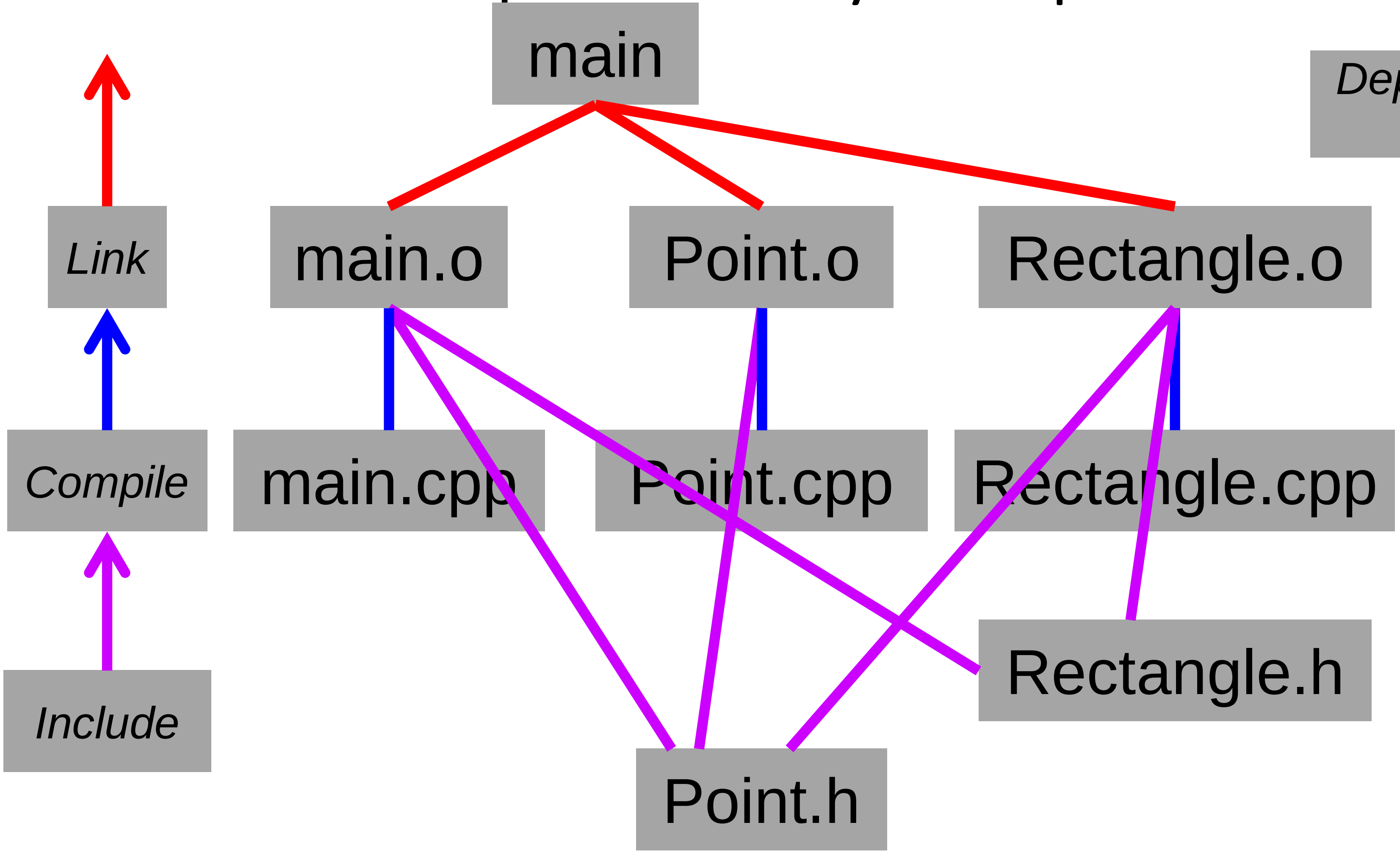
Source: https://www.cs.bu.edu/teaching/cpp/writing-makefiles/

Full Dependency Graph



Source: https://www.cs.bu.edu/teaching/cpp/writing-makefiles/

Actual Dependency Graph



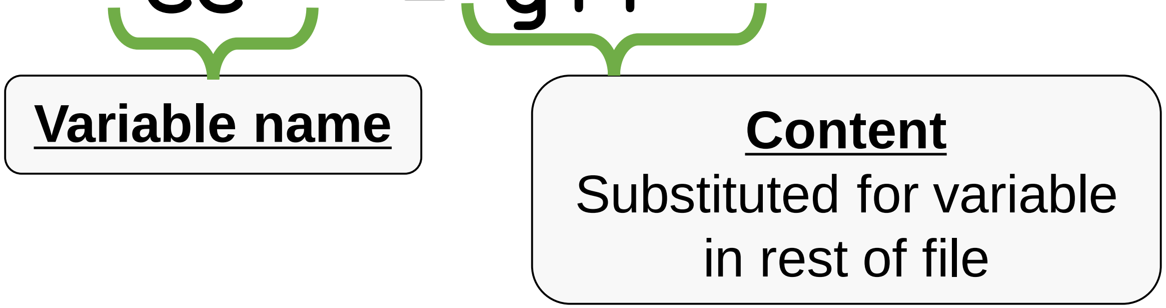
Source: https://www.cs.bu.edu/teaching/cpp/writing-makefiles/

Makefile Variables

- Similar to an alias or a **`#define`**
  - Use when you need something over and over

- Syntax to define a variable:

```
PROJ = Proj1
CC   = g++
```



<u>Variable name</u>

<u>Content</u>
Substituted for variable
in rest of file

Variable Use Examples

```
DIR1     = /afs/umbc.edu/users/k/k/k38/pub/CMSC341/Proj1/
PROJ     = Proj1
CC       = g++
CCFLAGS = -g -ansi -Wall -I . -I $(DIR1)

OBJECTS = Project1.o Inventory.o Cd.o Date.o
```

Notice that we can use one variable in definition of another *(declaration order matters)*

Using Variables

- To access a macro, use the following format:

$(VARIABLE_NAME)

```
$(PROJ): $(OBJECTS)
  $(CC) $(CCFLAGS) -o $(PROJ) $(OBJECTS)

Project1.o: Project1.c Inventory.h
   $(CC) $(CCFLAGS) -c Project1.c
```

- What do each of these rules actually mean?
  - (In plain English)

Helper Rules

- You can specify targets that do auxiliary tasks and do not actually compile code
  - Remove object and executable files
  - Print source code
  - Submit all code

- Timestamps don’t matter for these tasks
  - Good practice to let the makefile know that
  - These target are called “phony” targets

Phony Targets

- Same syntax, but preceded by a **.PHONY** declaration on the previous line

Same as target name

```
.PHONY: submit
submit:
    submit cs341 $(PROJ) $(SOURCES) \
Makefile *.txt
```

Use a backslash to continue action on more than one line

More Helper Rules

- Cleaning utilities

```
clean:
   -rm -f *# *~
cleaner: clean
   -rm -f *.o
cleanest: cleaner
   -rm -f core*; rm -f $(PROJ)
```

Implicit Rules

- Pattern-based: convert any file of type X to type Y
  - Type implied by filename extension (e.g.: .o from .c)
- Example:

```
%.o : %.c
        $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

- If multiple implicit rule patterns match, tries each in sequence, based on existence of dependencies
- Search is recursive, chained

Advanced Makefile

```
PROJ    = Proj1
CC      = g++
CCFLAGS = -g -ansi –Wall

SOURCES = $(PROJ).c Inventory.h Inventory.c Cd.h Cd.c Date.h Date.c
OBJECTS = $(PROJ).o Inventory.o Cd.o Date.o

$(PROJ): $OBJECTS
    $(CC) $(CCFLAGS) -o $(PROJ) $(OBJECTS)

$(PROJ).o: $(PROJ).c Inventory.h
     $(CC) $(CCFLAGS) -c $(PROJ).c

Inventory.o: Inventory.c Inventory.h Cd.h
     $(CC) $(CCFLAGS) -c Inventory.c

Cd.o: Cd.c Cd.h Date.h
     $(CC) $(CCFLAGS) -c Cd.c

Date.o: Date.c Date.h
     $(CC) $(CCFLAGS) -c Date.c

.PHONY: submit
submit:
   submit cs341 $(PROJ) $(SOURCES) Makefile *.txt

.PHONY: print
Print:
    enscript -G2rE $(SOURCES) Makefile *.txt
```