CS 35L: More on Python and Building from Source

Lab 6

Hengda Shi

Week 3 Lecture 2

More on Python

== vs. is

- A == B compares values for equality
- A is B compares objects for equality (i.e. memory addresses)

Classes

```
class Dog(Animal):                    # Dog inherits from Animal
    def __init__(self, name, sound): # constructor
        self.name = name              # create class instances
        self.sound = sound
    def bark(self):  # instance methods needs “self” argument
        print(self.sound)

my_dog = Dog("Pluto", "woof")
print(my_dog.name)
my_dog.bark()
```

Nested Functions

```
def outer():
    outer_var = 10
    x = {'a': 1}

    def inner():
        print(outer_var)
        x['a'] += 1

    inner()
```

```
def outer():
    x = 10

    def inner():
        nonlocal x
        x = 20
        print(x)

    inner()
    print(x)
```

Nested Classes

```
class OuterClass:

    Class InnerClass:

        def inner_class_method(self):

            pass

    def outer_class_method(self):

        pass
```

Modules

- Python standard library has many useful modules
- Import modules using `import`

```
import random                # import module
from math import pi    # import only specific code from module
from sys import *            # imports entire module directly into file
import datetime as dt        # import module using custom name

x = random.randint(0, 10)  # call function in module
print(pi * x)                # Note: didn't need to specify name
```

Shallow vs. Deep Copying

- A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original
- A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

```
from copy import copy, deepcopy

a = [1, 2, [3, 4]]          # original list

b = copy(a)                 # shallow copy (reference to sublist [3, 4])

c = deepcopy(a)             # deep copy (creates copy of sublist [3, 4])
```

Shallow vs. Deep Copying

```
from copy import copy, deepcopy

a = [1, 2, [3, 4]]
b = copy(a)
c = deepcopy(a)

b[0] = 0
b[2][0] = 10

print(a)
print(b)
print(c)
```

Other useful Python features

- Higher order functions (map, filter, reduce)
- List comprehensions
- Keyword arguments, *args and **kwargs
- Third party libraries with pip (numpy, pandas, tensorflow)

Building from Source

How to Install Software

- Linux
  - rpm (Redhat Package Management)
    - RedHat Linux (.rpm)
  - apt-get (Advanced Package Tool)
    - Debian Linux, Ubuntu Linux (.deb)
  - Good old build process
    - configure, make, make install
- Mac
  - homebrew
    - brew install/brew cask install

Decompressing Files

Generally, you receive Linux software in the tarball format (.tgz) or (.gz)

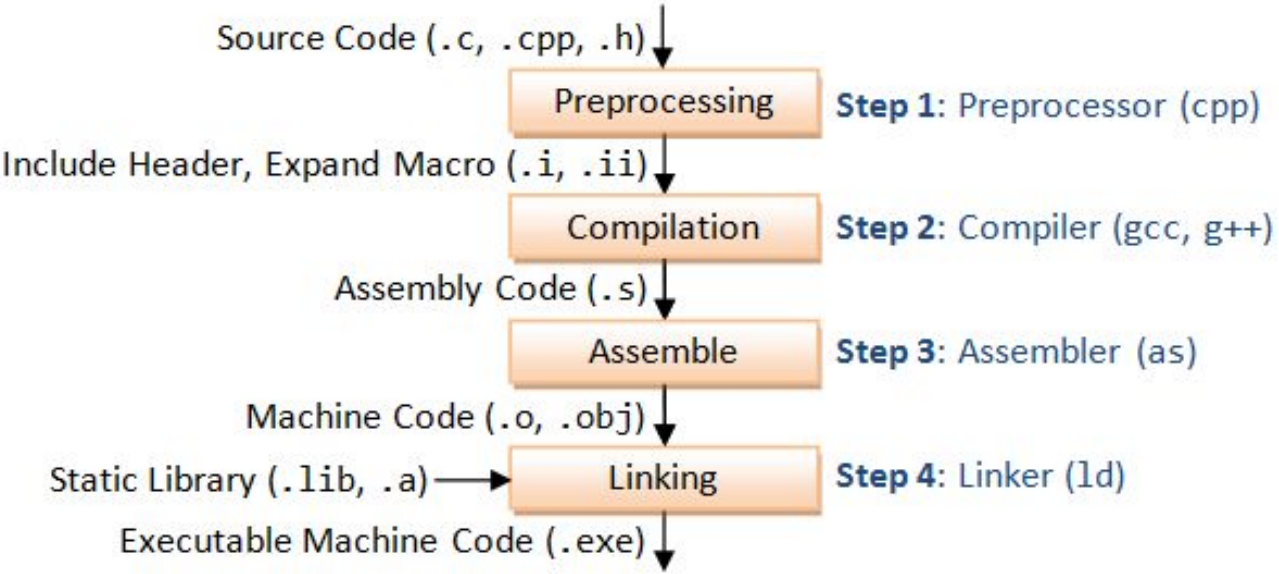
Decompress file in current directory:

- $ tar –xzvf filename.tar.gz
- –Option –x: --extract
- –Option –z: --gzip
- –Option –v: --verbose
- –Option –f: --file

Build Process

- configure
  - Script that checks details about the machine before installation
    - Dependency between packages
  - Creates ‘Makefile’
- make
  - Requires ‘Makefile’ to run
  - Compiles all the program code and creates executables in current temporary directory
- make install
  - make utility searches for a label named install within the Makefile, and executes only that section of it
  - executables are copied into the final directories (system directories)

C Compilation Process



C compilation

- Preprocessor
  - Removes comments from code
  - Includes header files code in the file itself
  - Replaces macro name (small functions) with values #define x 6 => this value replaced into file
- Compiler
  - Generate assembly code
- Assembler
  - Converts assembly code to machine code (binary) - object code)
- Linker
  - Used when we have multiple modules
  - We need a single executable file, linker is used to combine all files
  - Links library functions

Command-Line Compilation

- item.h
- item.c
  - #includes item.h
- shoppingList.h
  - #includes item.h
- shoppingList.c
  - #includes shoppingList.h
- shop.h
- shop.c
  - #includes shoppingList.h and shop.h
- How to compile?
- gcc -Wall shoppingList.c item.c shop.c –o shop

Expanding the command

- gcc - compiler program
- -Wall - turn all warnings on
- -o - to name the executable as the name given, instead of a.out

What if…

- We change one of the header or source files?
  - Rerun command to generate new executable
- We only made a small change to item.c?
  - not efficient to recompile shoppinglist.c and shop.c
  - Solution: avoid waste by producing a separate object code file for each source file
    - gcc -Wall –c item.c… (for each source file)
      - Wall: This enables all the warnings about constructions
    - gcc item.o shoppingList.o shop.o –o shop (combine)
    - Less work for compiler, saves time but more commands

What if...

- We change item.h?
  - Need to recompile every source file that includes it & every source file that includes a header that includes it. Here: item.c, shoppinglist.c and shop.c
  - Difficult to keep track of files when project is large
    - Windows 7 ~40 million lines of code
- => Make

Make

- Utility for managing large software projects
- Compiles files and keeps them up-to-date
- Efficient Compilation (only files that need to be recompiled)

Makefile Example

```
# Makefile - A Basic Example - only executes changed files
all : shop # usually first all is called the target and so are shop,
item.o...., clean
shop : item.o shoppingList.o shop.o        # dependencies
    gcc -Wall -o shop item.o shoppingList.o shop.o
item.o : item.cpp item.h
    gcc -Wall -c item.cpp
shoppingList.o : shoppingList.cpp shoppingList.h
    gcc -Wall -c shoppingList.cpp
shop.o : shop.cpp item.h shoppingList.h
    gcc -Wall -c shop.cpp
clean :
    rm -f item.o shoppingList.o shop.o shop
```

Example Bash command: ‘make shop’ will execute

```
gcc -Wall -o shop item.o shoppingList.o shop.o
```

Basics

- Variable declaration

  Flags= -c -Wall

  Compiler=g++

- Bash command ‘make <word>’ executes the target

  –Example, make clean would execute whatever’s below the target ‘clean’

Task

- You have a main target ‘hello’ (all : hello) which should execute all object files main.o function1.o function2.o and combine them into ‘hello’
- Each .o file should be created by compiling the cpp files individually (in a separate line)
- You have 3 cpp files - main.cpp function1.cpp function2.cpp
- Need to compile these files using Makefile
- In the end, write clean to remove all executables and object files

Lab 3

- Coreutils 8.29 has a problem
  - The latter option should override the earlier option
  - $ /usr/bin/ls -A /usr/src/
    debug kernels
  - $ /usr/bin/ls -aA /usr/src/
    . .. debug kernels

- Fix the ls program

Getting Set Up (Step 1)

- Download coreutils-7.6 to your home directory
  - Use ‘wget’
- Untar and Unzip it using `tar`
- Make a directory coreutils_install in the coreutils-8.29 directory (this is where you’ll be installing coreutils)
  - mkdir /you/path/to/coreutils_install

Building coreutils (Step 2)

- Go into coreutils-8.29 directory.This is what you just unzipped.
- Read the INSTALL file on how to configure “make”, especially **--prefix** flag
    - Compile and install your copy of Coreutils into **a temporary directory of your own**.
- Run the configure script using the prefix flag so that when everything is done, coreutils will be installed in the directory coreutils_install
- Compile it:make
- Install it:make install

Reproduce Bug (Step 3)

- Reproduce the bug by running the version of ‘ls’ in coreutils 8.29
- If you just type $ ls at CLI it won’t run ‘ls’ in coreutils 8.29
    - Why? Shell looks for /bin/ls
    - To use coreutils 8.29:
        - cd /you/path/to/coreutils_install
        - ./bin/ls -aA
        - This manually runs the executable in this directory

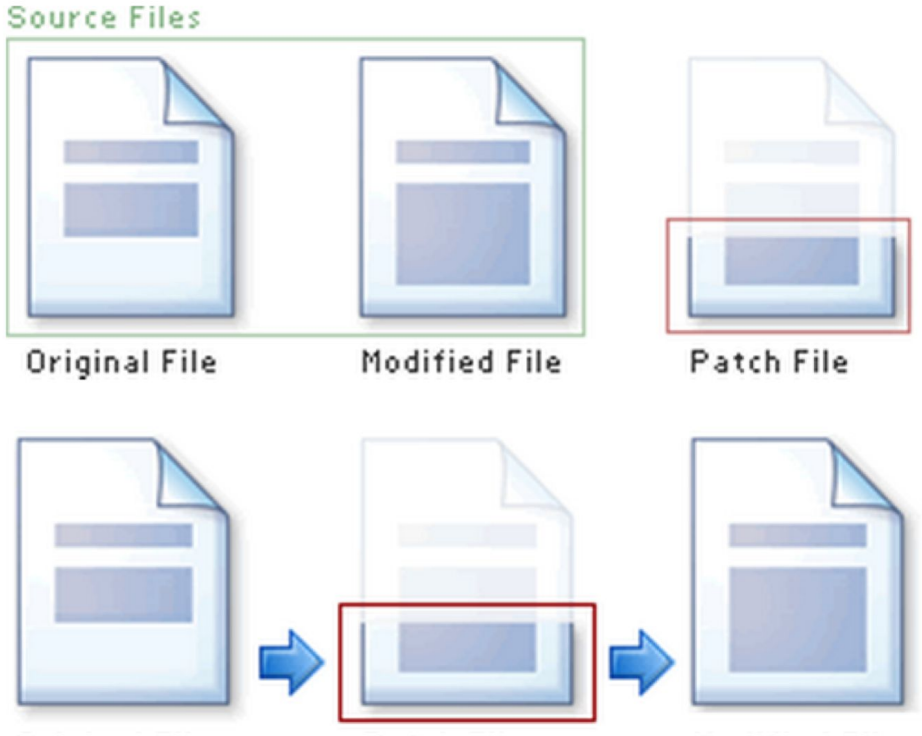
Patching

- A patch is a piece of software designed to fix problems with or update a computer program

- It’s a diff file that includes the changes made to a file

- A person who has the original (buggy) file can use the patch command with the diff file to add the changes to their original file

Applying a Patch



diff Unified Format

- diff –u original_file modified_file
- --- path/to/original_file
- +++ path/to/modified_file
- @@ -l,s +l,s @@
  - @@: beginning of a hunk
  - l: beginning line number
  - s: number of lines the change hunk applies to for each file
  - Alinewitha:
    - - sign was deleted from the original
    - + sign was added to the original
    - stayed the same

Patching and Building (Steps 4 & 5)

- $ patch –pnum < name_of_your_choice.patch
  - ‘man patch’ to find out what pnum does and how to use it

- type make to rebuild patched ls

Testing Fix (Step 6)

- Test the following:
  - Modified ls works
  - Installed unmodified ls does NOT work