072-lec-01-intro

COSC-072, summer 2011
Prof. Richard K. Squier
St. Mary's Hall, 339
202-687-6027
squier@cs.georgetown.edu

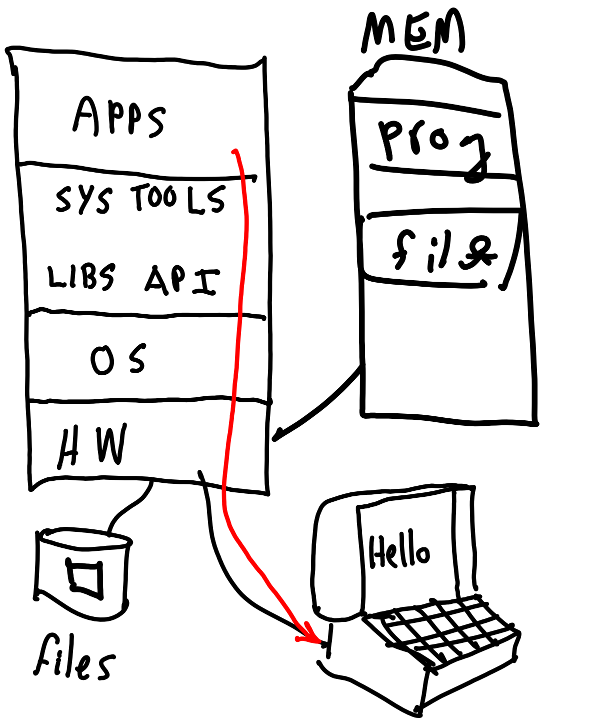
TA:
John Ferro
jdf48@georgetown.edu

Text:
C++ How to Program 8/e, Dietel & Dietel
Chapters 10-22

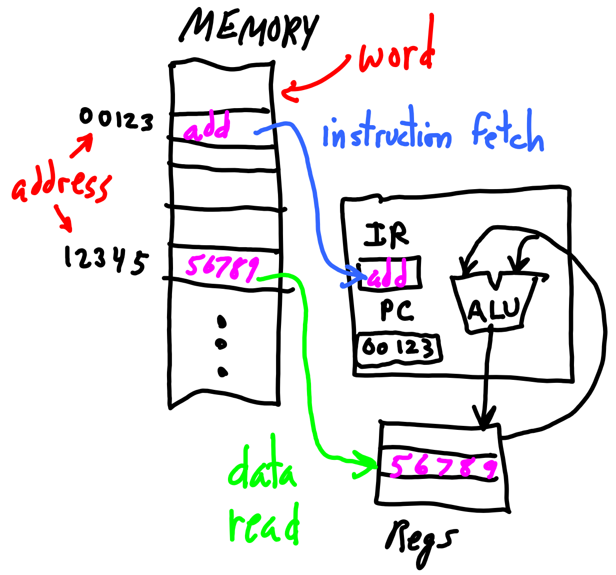
Tools, IDE:

www.netbeans.org/
XCode
cygwin
gdb

Subversion: "svn <command>"
(checkout, add, del, status, update, checkin, log -v)
https://svn.cs.georgetown.edu/svn/projects2
250-374-developer
y(&qwqsq

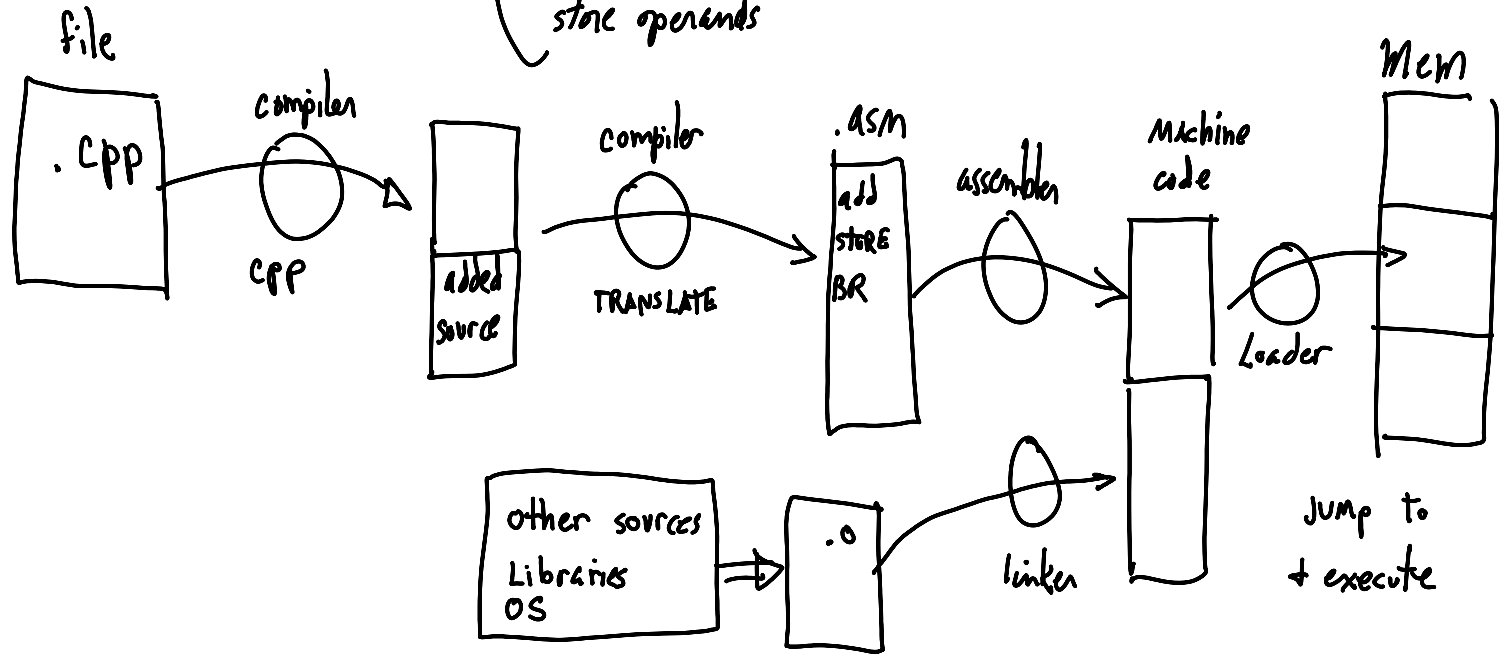


HW



machine cycle

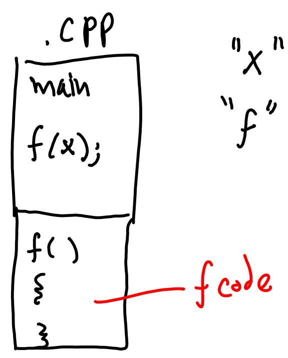
fetch instruction, PC++
fetch operands
execute operation
store operands



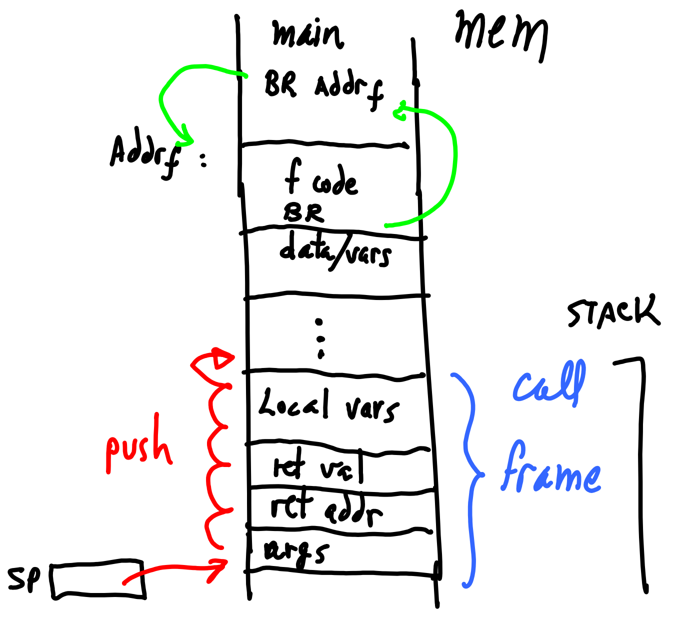
ISA

Load <address> → Reg
STORE Reg → <address>
OPERATE REG, REG, REG
branch PC ← <address>

Execution



"x" → $\langle address_x \rangle$
"f" → $\langle address_f \rangle$



$\textcircled{x} = f();$

data Types

$c = a + b \Rightarrow$ Ld $\langle address_a \rangle \Rightarrow Reg_1$
Ld $\langle address_b \rangle \Rightarrow Reg_2$
(?) $Reg_3 \Leftarrow Reg_1$ (?) Reg_2

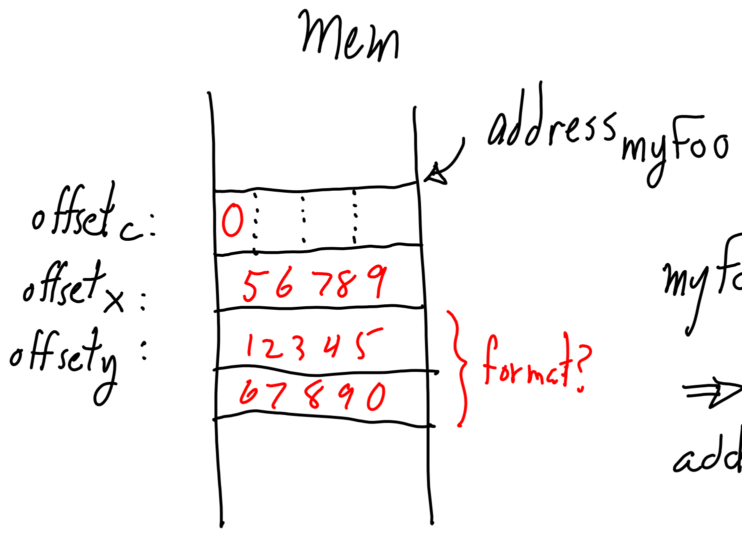
depends on type

base types

8-bit char, int, unsigned int
16-bit char, int, unsigned int
32-bit
64-bit
⋮ others, depends on ISA

derived Types

```
struct foo {
        char c;
        int x;
        double y;
    };
foo myFoo;
```



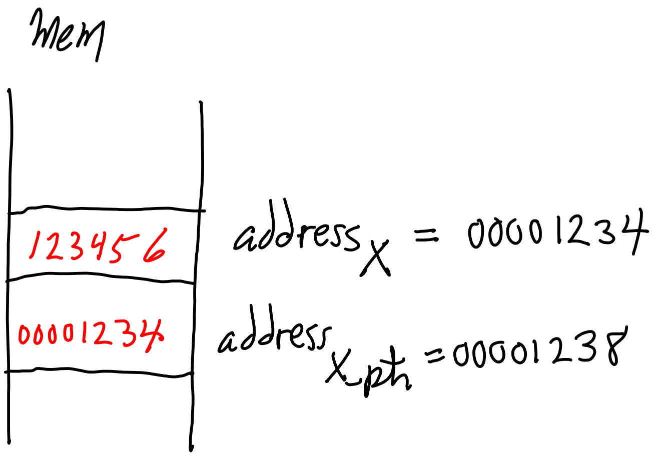
myFoo.x

$\Rightarrow$

$address_{myFoo} + offset_x$

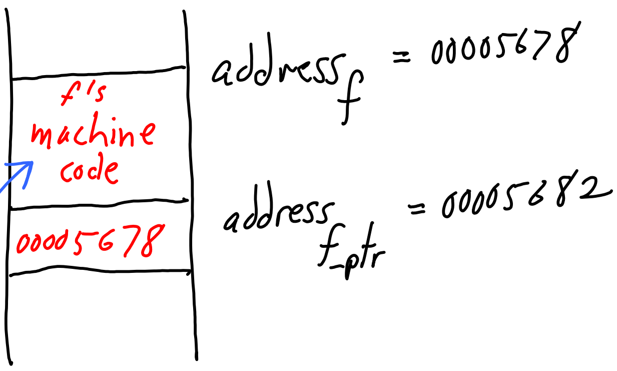
Pointers

```
int x;
int * x_ptr;

x_ptr = &x;
```



```
int (* f_ptr) (char);
f_ptr = &f;
⇒ (*f_ptr)('a');
⇒       f('b');
```
} same effect   PC ⇐ 0005678

```
int f( c char) {
    return( (int) c );
}
```



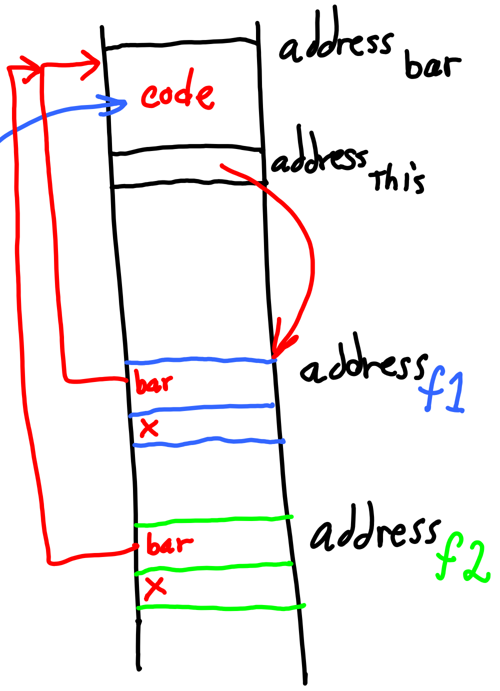
For arrays and functions, names like "f" are treated like pointer variables.

```
LD  00005682 → Reg1
BR  PC ← Reg1
```

$address_f$	00005678
$address_{f\_ptr}$	00005682

Class instances

```
class foo {
    public:
      void bar( )
      {
      }
    private:
      int x;
};
foo f1, f2;
```

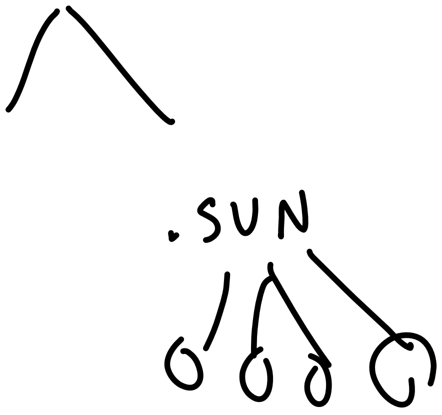


invoke a method

f1.bar

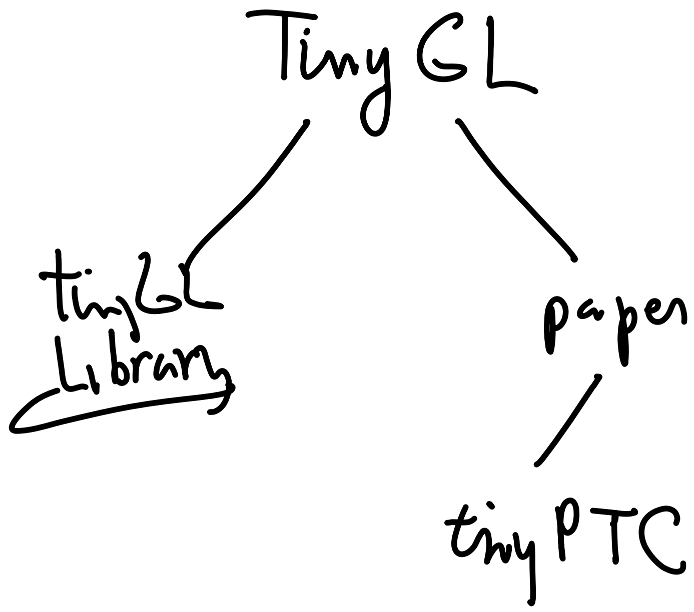
⇒ find F1
  + $offset_{bar}$

use pointer
PC ⇐ $address_{bar}$



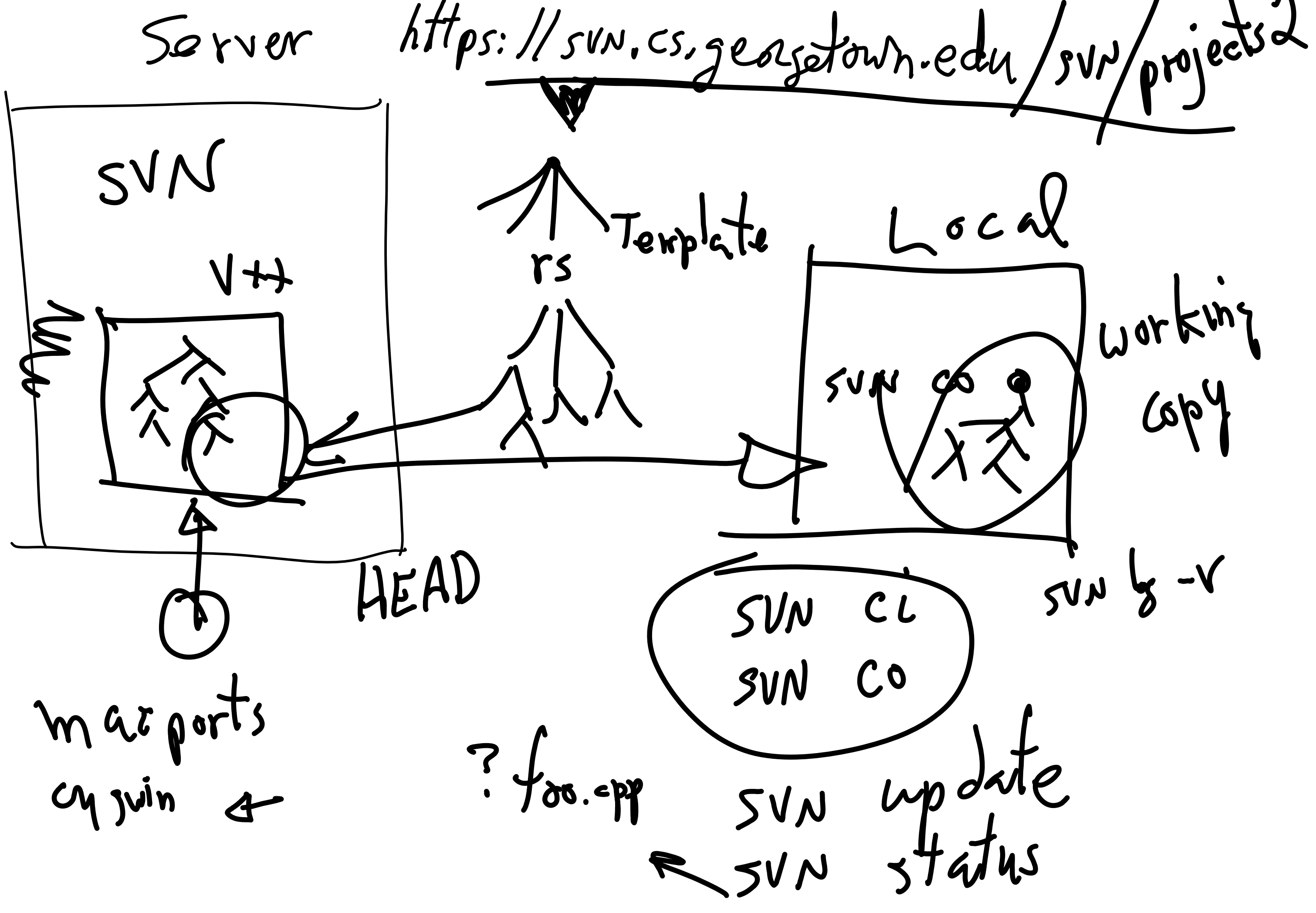
rm -rf

C++ ref



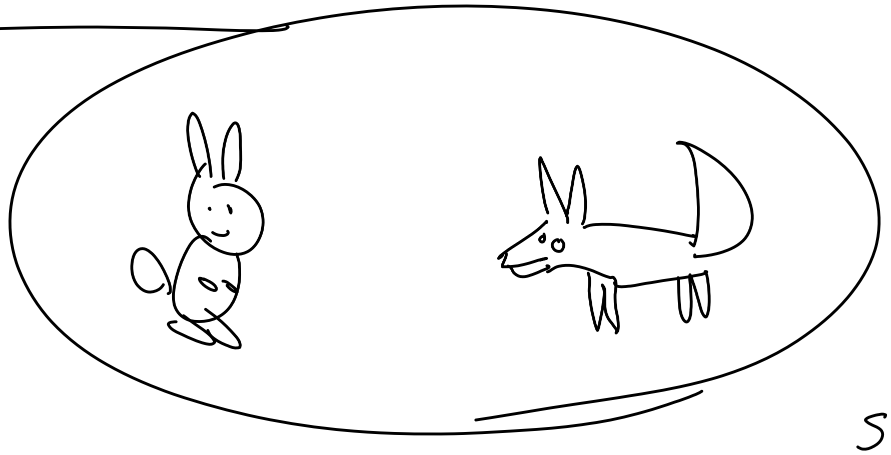
Fall et al.

Subversion
download client binary from cygwin (PCs), MacPorts (Macs, see if already present w/ Xcode.)



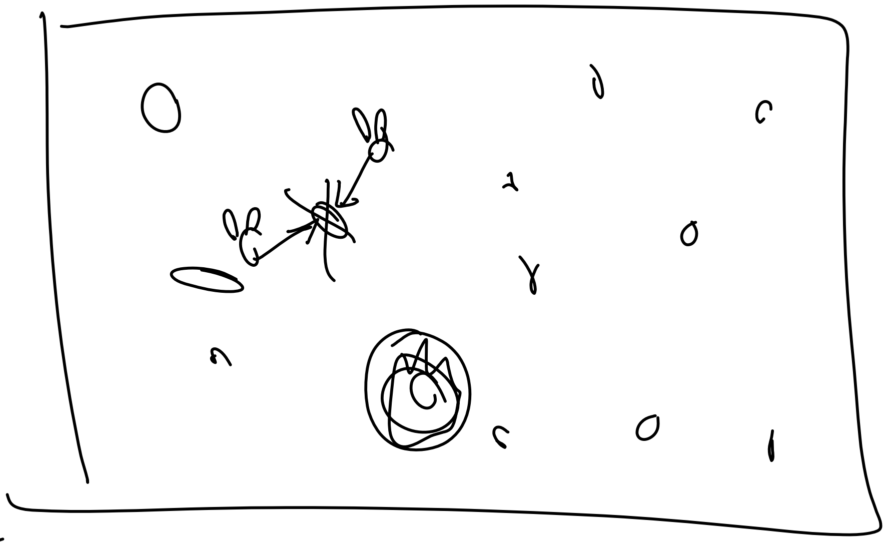
SVN add foo.cpp

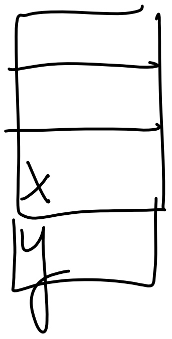
Project

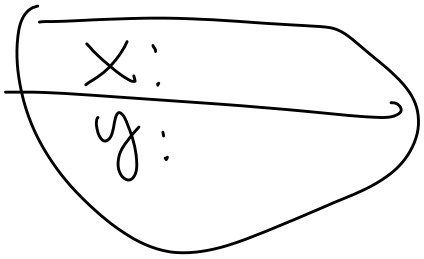


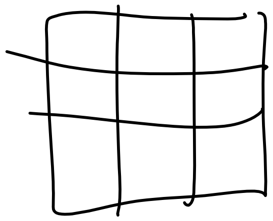
$t^{t+1} = a \cdot t(t-1)$

$s =$



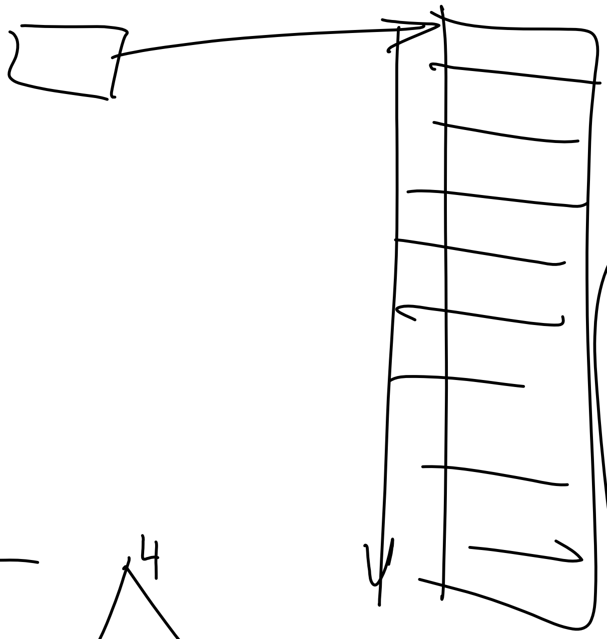




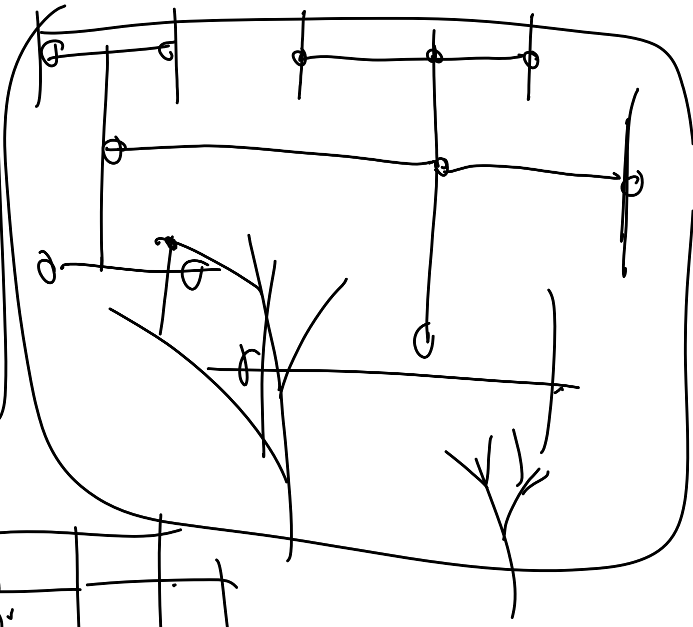


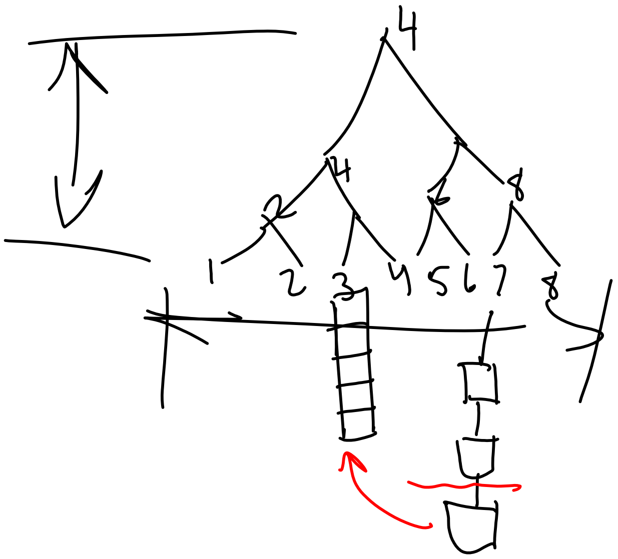
T env[10][10];

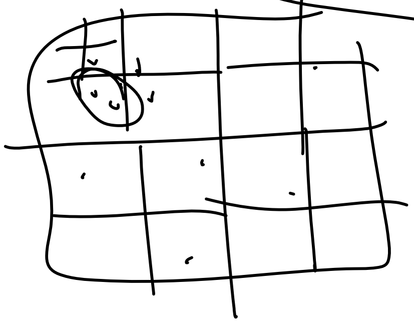
int

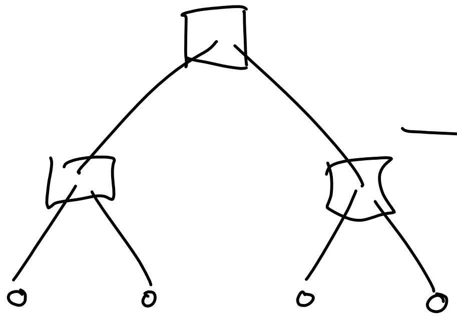


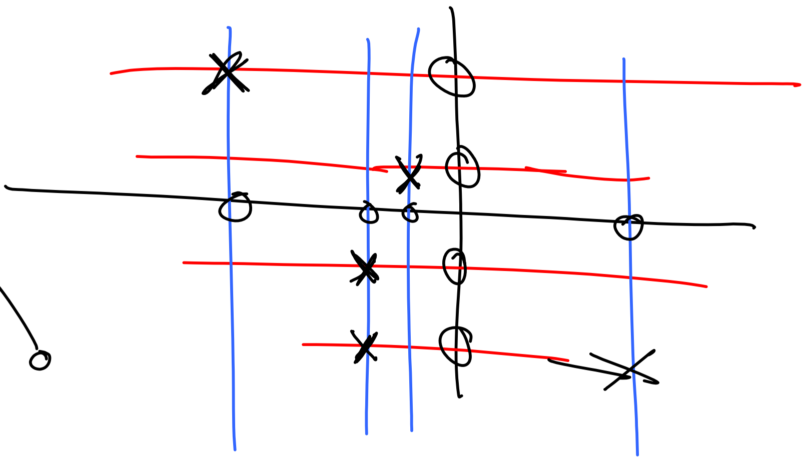
$$d = \sqrt{\Delta x^2 + \Delta y^2}$$

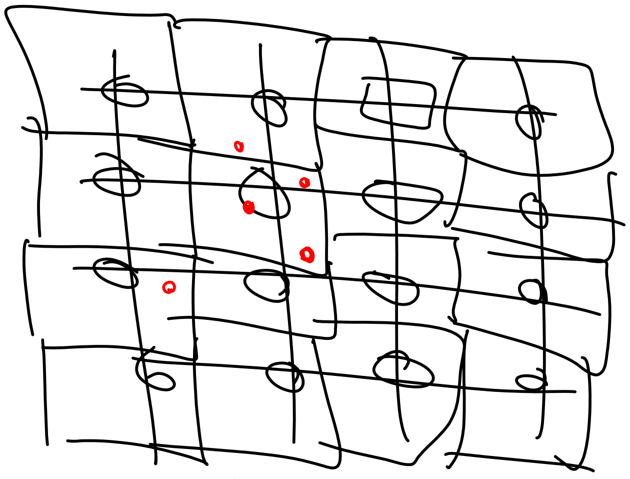


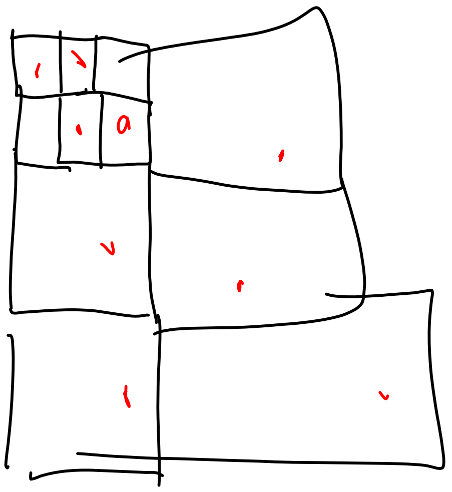




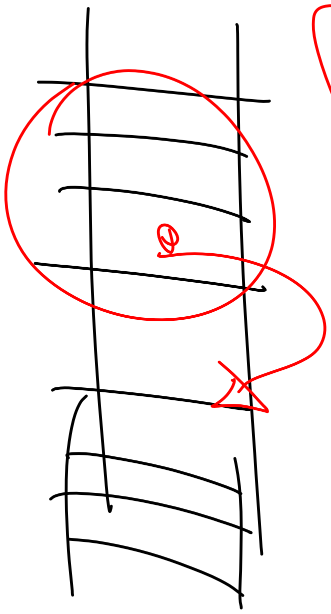


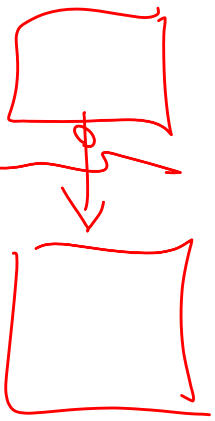


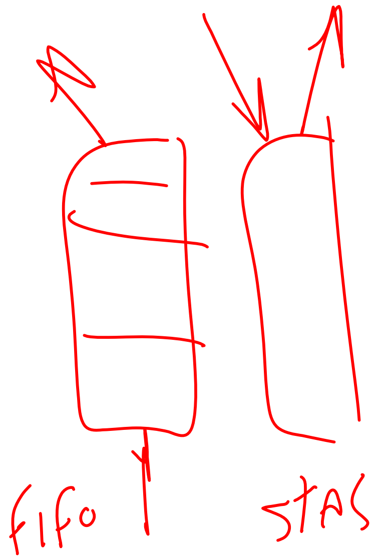




```
struct foo {
    foo * next;
}
```







φ

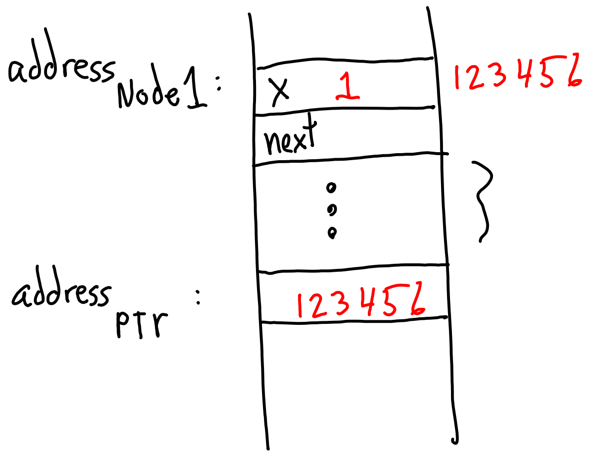
```
foo x, y
y.next = &x;
```

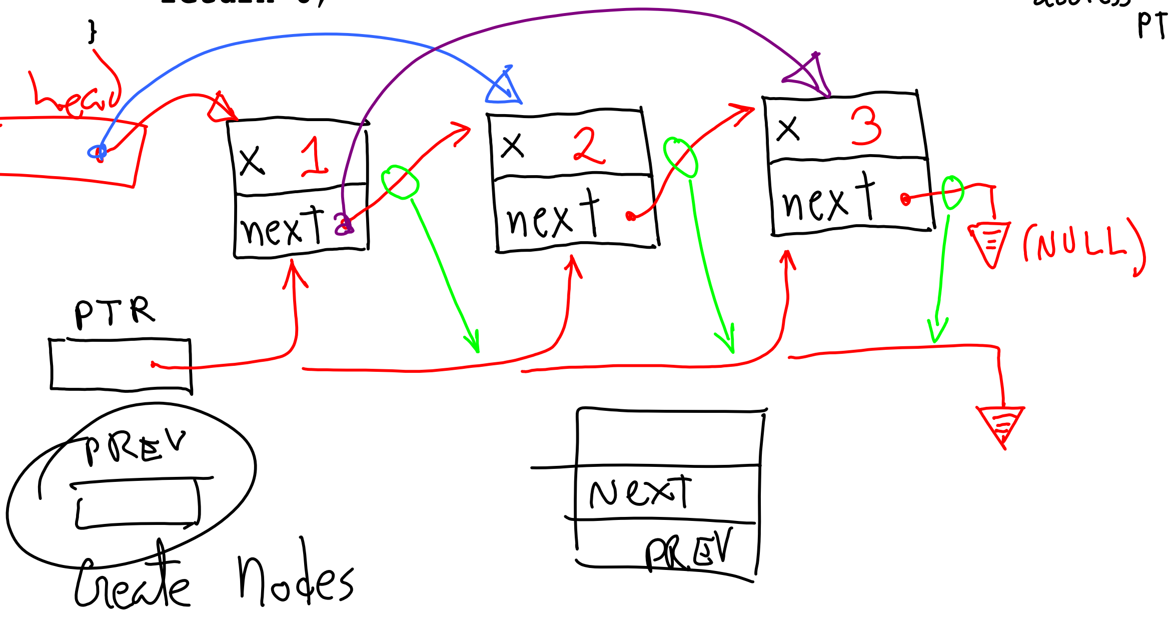
```
if
&ptr = foo.R
ptr = foo.L
```

```
/*---------------------------
 * linked.cpp
 * data structure from structs and pointers.
 *---------------------------*/
#include "linked.h"
int main(int argc, char** argv) {

    node1.x = 1;          //--- init data
    node2.x = 2;
    node3.x = 3;

    ptr       = &node1;  //--- build list
    ptr->next = &node2;
    ptr       = ptr->next;
    ptr->next = &node3;
    ptr       = ptr->next;
    ptr->next = NULL;

    ptr = ~~&node1~~; head;  //--- walk list
    while( ptr != NULL){
        printf("x=%d\n", ptr->x);
        ptr = ptr->next;
    }

    return 0;
}
```

```
/*---------------
 * linked.h
 *---------------*/
#include <cstdlib>
#include <stdio.h>
using namespace std;

struct node {
   int           x;
   struct node * next;
};

struct node node1;
struct node node2;
struct node node3;
struct node * ptr;
```

ptr = malloc(sizeof(struct node));





ptr → next

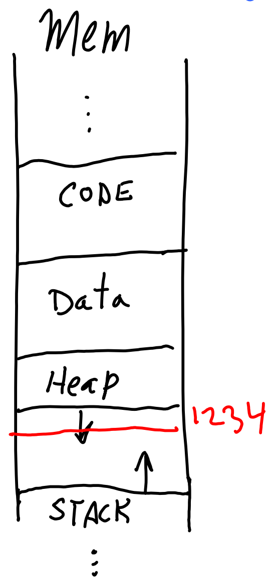
(* ptr).next

Same thing

ptr = malloc (sizeof (struct node));

PTR

get new space, return address



void * ptr ;

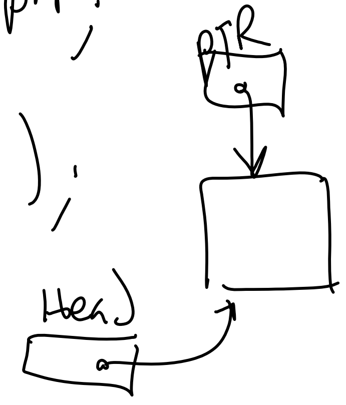
int * intptr;

int A [1] [2]

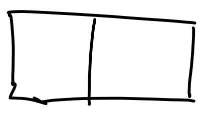
int ptr = (int *) ptr;

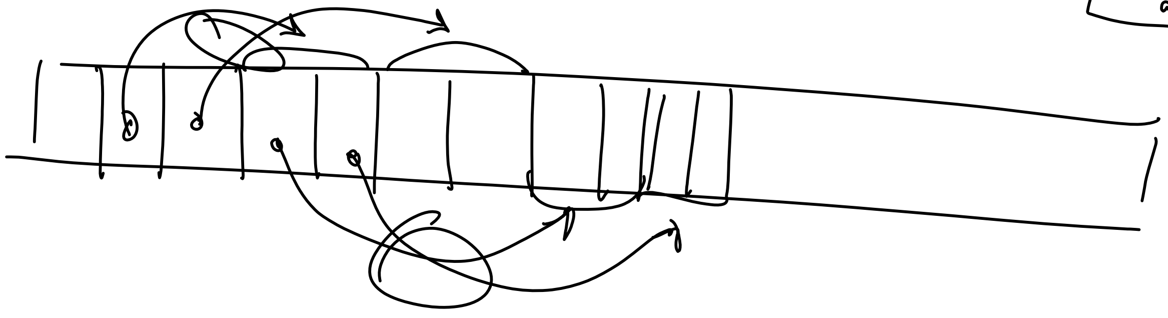
ptr = malloc ( );

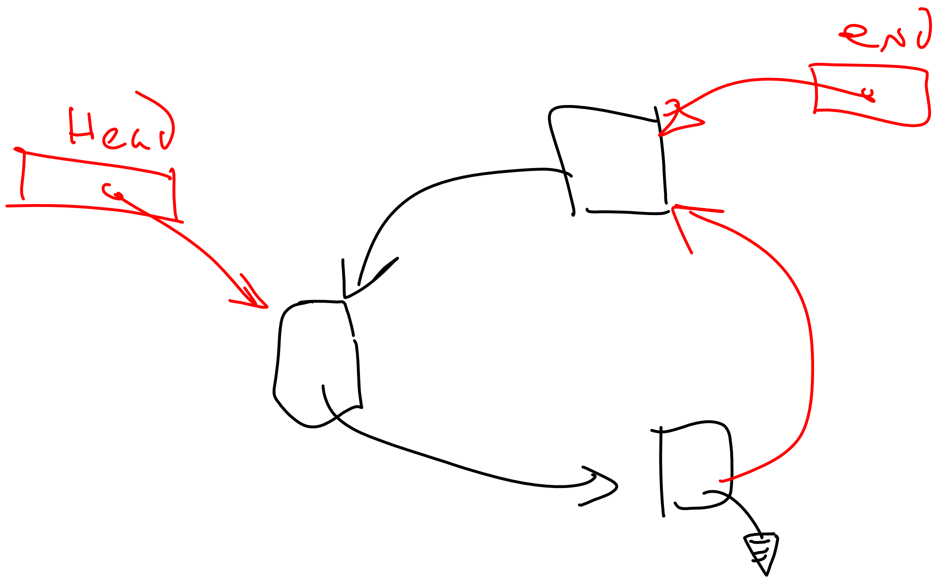
head = ptr;

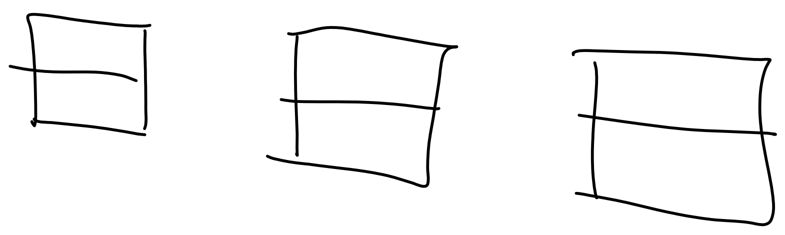


A[0]:









unix
A five-minute lesson in unix command-line shell usage.

1. When you open a command-line window in unix (cygwin window, OS X terminal window, linux terminal window, ...) a "shell" program begins running. The shell is usually "bash" these days. This program displays a prompt (" %> " for instance) and waits for keyboard input. The shell has an idea of where you are in the file system, and has a variable "PWD" or "CWD" containing a string such as "/home/squier/". This is your current working directory. Commands use this to decide what to do.

```
%> pwd          //---- displays contents of PWD, shows where you currently "are" in file system.
%> ls -F        //---- displays files and sub-directories in you current working directory.
%> cd ..        //---- changes your PWD to be one level up in tree.
%> cd foo       //---- changes your PWD one level down, into sub-directory "foo"
%> cd /home/squier/foo/bar   //---- change to that position in tree. Top level is "/".
%> rm -i bar    //----  permanently deletes/removes file or sub-directory "bar".
%> ./myProg     //----- runs the executable "myProg" which is in PWD
%> mkdir bar    //----- creates a new sub-directory "bar"
%> rmdir bar    //----- deletes "bar" (must be empty)
%> mv -i bar new   //---- renames ./bar to ./new  ("." means content of PWD)
%> cp -i bar new   //----- copies bar
%> man whatever   //----- displays manual page for program/command "whatever", or use "info".
%> exit             //----- kill shell program
```

NOTES

The shell has lots of "environment" variables, such as PWD. These are used by executables you run. Subversion wants to know what editor to use when you commit ("svn ci"). It looks for the variable "EDITOR". You can create that variable if it does not exist:

```
%> export EDITOR="vi"      //------ for instance, if you use vi.
%> set                     //------ see all your current variables
%> echo $PATH              //----- see value of a particular variable
```

The PATH variable is used by the shell to find executables. For instance, if you type "ls" to the shell, it looks for the executable by using PATH to search the file system.

Sometimes output is too large to see all at once.

```
%> set | less       //---- output of executable "set" is sent as input to executable "less";
                    //---- "less" displays its input one page at a time in response to your typing a space.
%> set > setOutput   //---- "set" output goes to a file "setOutput", which you can edit or instead
%> less setOutput    //----- do this to see it a page at a time. "less" also goes by the name "more".
```

You can also have input sent to an executable from a file:

```
%> foo < fooInput      //---- executable "foo" will read content of file "fooInput" instead of keyboard.
%> bash < myShellScript   //------ Even a shell can get input from a file. These are called "scripts".
```

vi
A one-minute lesson in using the editor vi. Commandline text is on the left, my comments are on the right.

```
 %> vi foo     //-------- Open an existing file or create a new file.

  a            //---------- start adding characters (goes into insertion mode).

 Esc           //---------- stop adding characters (goes into command mode).

               //---------- while in command mode, use arrow keys to move around.

 x             //--------- delete a character (command mode).

 :w            //--------- write editor's contents to file (command mode).

:q             //--------- quit the vi editor (command mode).
```

Notes
The editor "vi" is usually available as "vim", either name works. It is a unix-based editor found in OS X and other unix flavors such as linux and cygwin. The editor "emacs" is the other common editor. Windows Wordpad is compatible with files created using vi; Notepad isn't (the end-of-line characters are different in different systems). See CourseDocuments for more tutorial material or search the web.

svn
A one-minute intro to Subversion.

```
%> svn co https://svn.cs.georgetown.edu/svn/projects2/branches/rks
      //------ Creates local copy of the repository subtree. Created in current directory.

%> svn up    //---- download changes that were sent to the repository, also get lastest logs.

%> svn add  foo  //----- Marks the file "foo" or directory "foo" to be added to the repository.
%> svn rm foo    //----- Marks "foo" to be removed from repository and locally (files or directories).
%> svn mv foo  bar  //-----  Marks "foo" to be renamed "bar" in repository (files or directories).

%> svn status   //----- Shows all changes ready to be committed. Also shows the status of
                //-----  items that are not part of the repository:
                //------     "?"  means "is not part of the repository, i.e., not under version control"
                //------     "M"  means "has been modified, changes will be sent to repository"
                //------     "A"  means "will be added to repository"
                //------     "C/!"  means "local changes overlap changes in repository copy"
                //------          (in this case, save your file elsewhere, delete working copy,
                //------            re-checkout a working copy, make changes to working copy.)

%> svn ci  //----- send changes to repository (only for part of tree at and below current working dir.)

%> svn log -v  //----- See the log comments and history of tree changes, relative to current sub-tree.
```

NOTES
When you are in your work flow, do the following:

```
%> svn up
%> svn status
%> svn up
%> svn ci
%> svn up
```

make

A three-minute intro to the "make" program.

Now that you know about shells and shell scripts, you can make some sense of "make" and the file it reads, "Makefile". The program "make" reads the Makefile as if it were a shell script, but with some fancy navigation.

```
%> make foo   //------- read the Makefile part named "foo" (a "target").
```

Each target has some lines that are each used as scripts to bash: a new bash is started with its input being that line. Here's a target, and its scripts:

```
myTarget::
	echo $PATH
	echo $PWD
```

Create a Makefile, put those three lines in it, and try "make myTarget". The target is a string, which is often also the name of a file. The colon ":" starts a list saying what the target depends on (more below). A second colon says there aren't any dependencies. Each line after the target must start with a [TAB] character. Each line is sent to bash as an input script, just as if you had typed it into a command-line prompt. Makefiles can be used as an easy way of keeping shell commands handy. Usually, you will see Makefiles that compile and link executables. Here's an example.

```
foo: foo.h foo.c bar.o
	cc -o foo foo.c bar.o

bar.o: bar.c
	cc -c bar.c
```

"make foo" will now cause make to look at the dependency list. It will look to see the last-modified dates on the files "foo", "foo.h", "foo.c", "bar.o", and "bar.c". You can see modified dates this way:
%> ls -l
If what a file depends on has a more recent modified date, make decides that the file needs updating. So, perhaps yesterday you created foo.c and ran "make foo", which created the executable "foo". Today you edited "foo.c", then ran "make foo" again. Make sees that "foo" has yesterday's date, but "foo.c" has today's. It runs the scripts for the target "foo". It will also check the other dependencies recursively. Comments and "make" variables are the same as used in shell scripts:

```
#---This is a comment
CC=g++   #--- This assigns the string "g++" to variable "CC"
...
	${CC} -o foo foo.c bar.o      #-- The "${CC}" is replaced with "g++"
```

NOTES
There are a great many other parts of "make". It has rules for what to do with common things like ".o" file dependencies, and you can create your own rules. There are lots of other features as well.