UNIVERSITY OF WATERLOO

School of Computer Science

CS 246
Object-Oriented Software Development

Course Notes[*]
Fall 2011

http://www.student.cs.uwaterloo.ca/∼cs246

Contents

1 Shell

- After signing onto a computer (login), a mechanism must exist to display information and perform operations.
- The two main approaches are graphical and command line.
- **Graphical interface** (desktop):
  - ○ use icons to represent programs (actions),
  - ○ click on icon launches (starts) a program,
  - ○ program may pop up a dialog box for arguments to affect its execution.
- **Command-line interface** (shell):
  - ○ use text strings (names) to represent programs (commands),
  - ○ command is typed after a prompt in an interactive area to start it,
  - ○ arguments follow the command to affect its execution.
- Graphical interface is convenient, but seldom programmable.
- Command-line interface requires more typing, but allows programming.
- A **shell** is a program that reads commands and interprets them.
- It provides a simple programming-language with *string* variables and a few statements.

- Unix shells falls into two basic camps, **sh** (**ksh**, **bash**) and **csh** (**tcsh**), each with slightly different syntax and semantics.
- Focus on bash with some tcsh.
- Area (window) where shell runs is called a **terminal** or **xterm**.
- Shell line begins with a **prompt** denoted by $ (sh) or % (csh) (often customized).
- A command is typed after the prompt but ***not*** executed until **Enter**/Return key is pressed.

```
$ dateEnter                         # print current date
Thu Aug 20 08:44:27 EDT 2009
$ whoamiEnter                       # print userid
jfdoe
$ echo Hi There!Enter               # print any string
Hi There!
```

- Comment begins with a hash (#) and continues to the end of line.
- Multiple commands can be typed on the command line separated by the semi-colon.

```
$ date; whoami; echo Hi There!   # 3 commands
Sat Dec 19 07:36:17 EST 2009
jfdoe
Hi There!
```

- Commands can be editted on the command line (not sh):
  - ○ position cursor, ▯, with ◁ and ▷ arrow keys,
  - ○ remove characters before cursor with backspace/delete key,
  - ○ type new characters before cursor,
  - ○ press Enter at any point along the command line to execute modified command.
- Most commands have **options**, specified with a minus followed by one or more characters, which affect how the command operates.

```
$ uname -m             # machine type
x86_64
$ uname -s             # operating system
Linux
$ uname -a             # all system information
Linux linux008.student.cs 2.6.31-21-server #59-Ubuntu SMP x86_64 GNU/Linux
```

- Options are normally processed left to right; one option may cancel another.
- ***No standardization for command option names and syntax.***

- Shell terminates with command **exit**.

```
$ exit                    # exit shell and possibly terminal
```

  - when the shell of terminal/xterm terminates, the terminal/xterm terminates.
  - when the login terminal/xterm terminates, you sign off the computer (logout).

1.1 File System

- Shell commands interact extensively with the file system.
- Files are containers for data stored on persistent storage (usually disk).
- File names are organized in an N-ary tree: directories are vertices, files are leaves.
- Information is stored at specific locations in the hierarchy.

```
/                     root of the local file system
    bin                   basic system commands
    lib                   system libraries
    usr
        bin                   more system commands
        lib                   more system libraries
        include               system include files, .h files
    tmp                   system temporary files
    u or home             user files
        jfdoe                 home directory
            ., . .                current, parent directory
            .bashrc, .emacs, .login,. . . hidden files
            cs246                 course files
                a1                assignment 1 files
                    q1x.C, q2y.h, q2y.cc, q3z.cpp
        other users
```

- Directory named “/ ” is the root of the file system.
- bin, lib, usr, include : system commands, system library and include files.
- tmp : temporary files created by commands (***shared among all users***).
- u or home : user files are located in this directory.

- Directory for a particular user is called their **home directory**.
- Each file has a unique path-name in the file system, referenced with an absolute pathname.
- An **absolute pathname** is a list of all the directory names from the root to the file name separated by the backslash character "/".

```
/u/jfdoe/cs246/a1/q1x.C          # => file q1x.C
```

- Shell provides short names for a file using an implicit starting location.
- At sign on, the shell creates a **current directory** variable set to the user's home directory.
- Any file name not starting with "/" is automatically prefixed with the current directory to create the necessary absolute pathname.
- A **relative pathname** is a list of all the directory names from the current directory to the file name separated by the character "/".
- E.g., when user jfdoe signs on, home and current directory are set to /u/jfdoe.

```
cs246/a1/q1x.C                   # => /u/jfdoe/cs246/a1/q1x.C
```

- Shell special character "~" (tilde) expands to user's home directory.

```
~/cs246/a1/q1x.C                    # => /u/jfdoe/cs246/a1/q1x.C
```

- Every directory contains 2 special directories:
  - "." points to current directory.

    ```
    ./cs246/a1/q1x.C                # => /u/jfdoe/cs246/a1/q1x.C
    ```

  - ". ." points to parent directory above the current directory.

    ```
    . ./. ./usr/include/limits.h    # => /usr/include/limits.h
    ```

1.2 Pattern Matching

- Shells provide pattern matching of file names, **globbing** (regular expressions), to reduce typing lists of file names.
- Different shells and commands support slightly different forms and syntax for patterns.
- Pattern matching is provided through special characters, *, ?, {}, [], denoting different **wildcard**s (from card games, e.g., Joker is wild, i.e., can be any card).
- Patterns are composable: multiple wildcards joined into complex pattern (Aces, 2s and Jacks are wild).

- E.g., if the current directory is /u/jfdoe/cs246/a1 containing files q1x.C, q2y.h, q2y.cc, q3z.cpp
  - * matches 0 or more characters

    ```
    $ echo q*  # shell globs “q*” to match file names, which echo prints
    q1x.C q2y.h q2y.cc q3z.cpp
    ```

  - ? matches 1 character

    ```
    $ echo q*.??
    q2y.cc
    ```

  - {. . .} matches any alternative in the set

    ```
    $ echo *.{C,cc,cpp}
    q1x.C q2y.cc q3z.cpp
    ```

  - [. . .] matches 1 character in the set

    ```
    $ echo q[12]*
    q1x.C q2y.h q2y.cc
    ```

  - [!. . .] (^ csh) matches 1 character ***not*** in the set

    ```
    $ echo q[!1]*
    q2y.h q2y.cc q3z.cpp
    ```

- ○ Create ranges using hyphen (dash)

```
[0-3]          # => 0 1 2 3
[a-zA-Z]       # => lower or upper case letter
[!a-zA-Z]      # => any character not a letter
```

- ○ Hyphen is escaped by putting it at start or end of set

```
[-?*]*         # => matches file names starting with -, ?, or *
```

- If globbing pattern does not match any files, the pattern is the file name (including wildcards).

```
$ echo q*.ww q[a-z].cc    # files do not exist so no expansion
q*.ww q[a-z].cc
```

csh prints: **echo**: No match.

- **Hidden files** contain administrative information and start with “.” (dot).
- These files are ignored by globbing patterns, e.g., * does not match all file names in a directory.
- Pattern .* matches all hidden files:
  - ○ match “.”, match zero or more characters, e.g., .bashrc, .login, etc., ***and*** “.”, “..”

  - ***matching ".", ".." can be dangerous***

- Pattern .[!.]* matches all hidden files but ***not*** "." and ".." directories.
  - match ".", match any character NOT a ".", match zero or more characters
  - ⇒ there must be at least 2 characters, the 2nd character cannot be a dot
  - "." starts with dot but fails the 2nd pattern requiring another character
  - ".." starts with dot but the second dot fails the 2nd pattern requiring non-dot character

  Which hidden files are still missed?

- On the command line, pressing the tab key after typing several characters of a command/file name causes the shell to automatically complete the name.

```
$ ectab          # cause completion of command name to echo
$ echo q1tab   # cause completion of file name to q1x.C
```

- If the completion is ambiguous (i.e., more than one):
  - shell "beeps",
  - prints all completions if tab is pressed again,
  - then you must type more characters to uniquely identify the name.

```
$ datab                       # beep
$ datab                       # print completions
dash   date
$ dattab                      # cause completion of command name to date
```

1.3 Quoting

- **Quoting** controls how shell interprets strings of characters.
- **Backslash** (\ ) : **escape** any character, including special characters.

```
$ echo .[!.]*                 # expand globbing patterm
.bashrc   .emacs   .login   .vimrc
$ echo \.\[\!\.\]\*           # print globbing pattern
.[!.]*
```

- **Backquote** (`) : execute the text as a command, and replace it with the command output.

```
$ echo `whoami`               # $ whoami => jfdoe
jfdoe
```

- **Single quote** (′) : do not interpret the string, even backslash.

```
$ echo ′.[!.]*′
.[!.]*
$ echo ′\.\[\!\.\]\*′
\.\[\!\.\]\*
```

A single quote cannot appear inside single quotes.

- ○ E.g., file name containing special characters (blanks/wildcards/comment).

```
$ echo Book Report #2
Book Report
$ echo ′Book Report #2′
Book Report #2
```

- **Double quote** (") : interpret escapes, backquotes, and variables.

```
$ echo ".[!.]* \"`whoami`\" \.\[\!\.\]\*"
.[!.]* "jfdoe" \.\[\!\.\]\*
```

- Put newline into string for multi-line text.

```
$ echo "abc
> cdf"        # prompt ">" means current line is incomplete
abc
cdf
```

- ***To stop prompting or output from any shell command***, type <ctrl>-c (C-c), i.e., press <ctrl> and c keys simultaneously, causes the shell to interrupt the current command.

```
$ echo "abc
> C-c
$
```

1.4 Shell Commands

- Some commands are executed directly by the shell rather than the OS because they read/write the shell's state.
- **help** : display information about bash commands (not sh or csh).

  help *[command-name]*

  ◦ without argument, lists all bash commands.
- **cd** : change the current directory (navigate file hierarchy).

```
cd [directory]
```

- ○ argument must be a directory and not a file
- ○ **cd** : move to home directory, same as **cd** ~
- ○ **cd -** : move to previous current directory
- ○ **cd** ~/cs246 : move to the cs246 directory contained in jfdoe home directory
- ○ **cd** /usr/include : move to /usr/include directory
- ○ **cd** . . : move up one directory level
- ○ If path does not exist, **cd** fails and current directory is unchanged.

- **pwd** : print the current directory.

```
$ pwd
/u/jfdoe/cs246
```

- **history** and “!” (bang!) : print a numbered history of most recent commands entered and access them.

```
$ history
      1    date
      2    whoami
      3    echo Hi There
      4    help
      5    cd ..
      6    pwd
```

```
$ !2      # rerun 2nd history command
whoami
jfdoe
$ !!      # rerun last history command
whoami
jfdoe
$ !ec     # rerun last history command starting wit
echo Hi There
Hi There
```

- ○ !N rerun command N
- ○ !! rerun last command
- ○ !xyz rerun last command starting with the string “xyz”
- ○ Arrow keys △/▽ move forward/backward through history commands on command line.

```
$ △ pwd
$ △ cd ..
$ △ help
```

- **alias** : string substitutions for command names.

alias *[command-name=string]*

- No spaces before/after “=” (csh does not have “=”).
- string is substituted for command *command-name*.
- Provide *nickname* for frequently used or variations of a command.

```
$ alias d=date            # no quotes
$ d
Mon Oct 27 12:56:36 EDT 2008
$ alias off="clear; exit"
$ off                     # clear screen before terminating shell
```

Why are quotes necessary for alias off?

- ***Always use quotes to prevent problems.***
- Aliases are composable, i.e., one alias references another.

```
$ alias now="d"           # quotes
$ now
Mon Oct 27 12:56:37 EDT 2008
```

- Without argument, print all currently defined alias names and strings.

```
$ alias
alias d=′date′
alias now=′d′
alias off=′clear; exit′
```

- ○ ***Alias CANNOT be command argument***.

  ```
  $ alias cs246assn=/u/jfdoe/cs246/a1
  $ cd cs246assn          # alias only expands for command
  bash: cd: cs246assn: No such file or directory
  ```

- ○ Alias entered on command line disappears when shell terminates.
- ○ Two options for making aliases persist across sessions:
  1. insert the **alias** commands in the appropriate (hidden) .*shell*rc file,
  2. place a list of **alias** commands in a file (often .aliases) and **source** that file from the .*shell*rc file.

- **type** (csh **which**) : print pathname of a command.

  ```
  $ type now
  now is aliased to ‘d’
  $ type d
  d is aliased to ‘date’
  $ type bash
  bash is /bin/bash
  ```

- **echo** : write arguments, separated by a space and terminated with newline.

```
$ echo  We  like  ice  cream      # 4 arguments
We like ice cream
$ echo " We  like  ice  cream " # 1 argument
 We  like  ice  cream
```

- **time** : execute a command and print a time summary.
  - test if program modification produces change in execution performance
  - prints **user time** (program CPU), **system time** (OS CPU), **real time** (wall clock)
  - different shells print these values differently.

```
$ time myprog       | % time myprog
real           1.2  | 0.94u 0.22s 0:01.2
user           0.9  |
sys            0.2  |
```

  - user + system ≈ real-time (uniprocessor, no OS delay)
  - compare user (and possibly system) execution times before and after modification
- **exit** : terminates shell, with optional integer exit status (return code) N.

```
exit [ N ]
```

- *[N]* is in range 0-255; larger values are truncated (256 ⇒ 0, 257 ⇒ 1, etc.) , negative values (if allowed) become unsigned (-1 ⇒ 255).
- exit status defaults to zero if unspecified.

1.5 System Commands

- Commands executed by operating system (UNIX).
- sh / bash / csh / tcsh : start **subshell**.

```
$ ...                   # bash commands
$ tcsh                  # start tcsh in bash
% ...                   # tcsh commands
% sh                    # start sh in tcsh
$ ...                   # sh commands
$ exit                  # exit sh
% exit                  # exit tcsh
$ exit                  # exit original bash and terminal
```

  - Allows switching among shells for different purposes.
- chsh : set login shell (bash, tcsh, etc.).

```
$ echo $SHELL  # what shell am I using ?
/bin/tcsh
$ chsh         # change to different shell
Password: XXXXXX
Changing the login shell for jfdoe
Enter the new value, or press ENTER for the default
        Login Shell [/bin/tcsh]: /bin/bash
```

- man : print information about command, option names and function.

```
$ man bash
. . .          # information about "bash" command
$ man chsh
. . .          # information about "chsh" command
$ man man
. . .          # information about "man" command
```

- ls : list the directories and files in the specified directory.

```
ls [ -al ] [ file or directory name-list ]
```

  - ○ -a lists *all* files, including hidden files
  - ○ -l generates a *long* listing (details) for each file
  - ○ no file/directory name implies current directory

```
$ ls .                  # list current directory (non-hidden files)
q1x.C   q2y.h   q2y.cc   q3z.cpp
$ ls -a                 # list current directory plus hidden files
.   . .   .bashrc   .emacs   .login   q1x.C   q2y.h   q2y.cc   q3z.cpp
```

- mkdir : create a new directory at specified location in file hierarchy.

```
mkdir directory-name-list
```

```
$ mkdir d d1 d2 d3   # create 4 directories in current directory
```

- cp : copy files; with the -r option, copy directories.

```
cp [ -i ] source-file   target-file
cp [ -i ] source-file-list   target-directory
cp [ -i ] -r source-directory-list   target-directory
```

  - -i prompt for verification if a target file is being replaced.
  - -r recursively copy contents of a source directory to a target directory.

```
$ cp f1 f2             # copy file f1 to f2
$ cp f1 f2 f3 d        # copy files f1, f2, f3 into directory d
$ cp -r d1 d2 d3 # copy directories d1, d2 recursively into directory d3
```

- mv : move files and/or directories to another location in the file hierarchy.

```
mv [ -i ] source-file   target-file
mv [ -i ] source-file-list/source-directory-list   target-directory
```

- ○ if the target-file does not exist, the source-file is renamed; otherwise the target-file is replaced.
- ○ -i prompt for verification if a target file is being replaced.

```
$ mv f1 foo        # rename file f1 to foo
$ mv f2 f3         # delete file f3 and rename file f2 to f3
$ mv f3 d1 d2 d3 # move file f3 and directories d1, d2 into directory d3
```

- ● rm : remove (delete) files; with the -r option, remove directories.

```
rm [ -ifr ] file-list/directory-list
```

- ○ -i prompt for verification for each file/directory being removed.
- ○ -f do not prompt for verification for each file/directory being removed.
- ○ -r recursively delete the contents of a directory.
- ○ ***UNIX does not give a second chance to recover deleted files; be careful when using* rm*, especially with globbing, e.g.,* rm * *or* rm .***
- ○ UW has hidden directory .snapshot in every directory containing backups of all files in that directory (per hour for 8 hours, per night for 7 days, per week for 21 weeks)

```
$ ls .snapshot    # directories containing backup files
hourly.0  hourly.6  nightly.4  weekly.11  weekly.17  weekly.3  weekly.9
hourly.1  hourly.7  nightly.5  weekly.12  weekly.18  weekly.4
hourly.2  nightly.0 nightly.6  weekly.13  weekly.19  weekly.5
hourly.3  nightly.1 weekly.0   weekly.14  weekly.2   weekly.6
hourly.4  nightly.2 weekly.1   weekly.15  weekly.20  weekly.7
hourly.5  nightly.3 weekly.10  weekly.16  weekly.21  weekly.8
$ cp .snapshot/hourly.0/q1.h q1.h # restore file from previous hour
```

- Use **alias** for setting command options for particular commands.

```
$ alias cp="cp -i"
$ alias mv="mv -i"
$ alias rm="rm -i"
```

  which always uses the -i option on commands cp, mv and rm.

- Alias can be overridden by quoting or escaping the command name.

```
$ "rm" -r xyz
$ \rm -r xyz
```

  which does not add the -i option.

- cat/more/less : print files.

cat *file-list*

- ○ cat shows the contents in one continuous stream.
- ○ more/less paginate the contents one screen at a time.

```
$ cat q1.h
. . .            # print file q1.h completely
$ more q1.h
. . .            # print file q1.h one screen at a time
                 # type “space” for next screen, “q” to stop
```

- lp/lpstat/lprm : add, query and remove files from the printer queues.

```
lp [ -d printer-name ] file-list
lpstat [ -d ] [ -p [ printer-name ] ]
lprm [ -P printer-name ] job-number
```

- ○ if no printer is specified, use default printer (ljp_3016 in MC3016).
- ○ lpstat : -d prints default printer, -p without printer-name lists all printers
- ○ each job on a printer’s queue has a unique number.
- ○ use this number to remove a job from a print queue.

```
$ lp -d ljp_3016 uml.ps         # print file to printer ljp_3016
$ lpstat                        # check status, default printer ljp_3016
Spool queue: lp (ljp_3016)
Rank     Owner        Job Files                     Total Size
1st      rggowner     308 tt22                      10999276 bytes
2nd      jfdoe        403 uml.ps                 41262 bytes
$ lprm 403                      # cancel printing
services203.math: cfA403services16.student.cs dequeued
$ lpstat                        # check if cancelled
Spool queue: lp (ljp_3016)
Rank     Owner        Job Files                     Total Size
1st      rggowner     308 tt22                      10999276 bytes
```

- cmp/diff : compare 2 files and print differences.

```
cmp file1 file2
diff  file1 file2
```

  - ○ return 0 if files equal (no output) and non-zero otherwise (output difference)
  - ○ cmp generates the first difference between the files.

	file x	file y
1	a\n	a\n
2	b\n	b\n
3	c\n	c\n
4	**d**\n	**e**\n
5	g\n	h\n
6	h\n	i\n
7		g\n

```
$ cmp x y
x y differ: char 7, line 4
```

newline is counted ⇒ 2 characters per line

- ○ diff generates output describing how to change first file into second file.

```
$ diff x y
4,5c4     # replace lines 4 and 5 of 1st file
< d       #    with line 4 of 2nd file
< g
---
> e
6a6,7     # after line 6 of 1st file
> i       #    add lines 6 and 7 of 2nd file
> g
```

- find : search for names in the file hierarchy.

find *[file/directory-list] [expr]*

- \\(*expr* \\) evaluation order
- -not *expr*, *expr* -a *expr*, *expr* -o *expr* logical *not*, *and* and *or* (precedence order)

  -a default if unspecified, *expr* *expr* ⇒ *expr* -a *expr*
- -type f | d select files of type file or directory
- -maxdepth *N* recursively descend at most *N* directory levels (0 ⇒ current directory)
- -name *pattern* restrict file names to globbing pattern.
- find file/directory names in current and subdirectories with pattern “t*”

```
$ find . -name "t*"        # why quotes ?
./test.cc
./testdata
```

find only file names in current and subdirectories with pattern “t*”

```
$ find * -type f -name "t*"  # -a unspecified
test.cc
```

find only file names in current and subdirectories to a maximum depth of 3 with patterns t* or *.C.

```
$ find * -maxdepth 3 -a -type f -a \( -name "t*" -o -name "*.C" \)
test.cc
q1.C
testdata/data.C
```

- egrep : (extended global regular expression print) search & print lines matching pattern in files (Google). (same as grep -E)

  ```
  egrep -irn pattern-string file-list
  ```

  - -i ignore case in both pattern and input files
  - -r recursively examine files in directories.
  - -n prefix each matching line with line number
  - returns 0 if one or more lines match and non-zero otherwise (counter intuitive)
  - list lines containing “main” in files with suffix “.cc”

    ```
    $ egrep -n main *.cc
    q1.cc:33:int main() {
    ```

    list lines containing “fred” in any case in file “names.tex”

```
$ egrep -i fred names.txt
names.txt:Fred Derf
names.txt:FRED HOLMES
names.txt:freddy jones
```

list lines that match start of line “^”, match “*#include*”, match 1 or more space or tab “[ ]+”, match either “"” or “<”, match 1 or more characters “.+”, match either “"” or “>”, match end of line “$” in files with suffix “.h” or “.cc”

```
$ egrep ′^#include[     ]+["<].+[">]$′ *.{h,cc}  # why quotes ?
egrep: *.h: No such file or directory
q1.cc:#include <iostream>
q1.cc:#include <iomanip>
q1.cc:#include “q1.h”
```

  - **egrep pattern is different from globbing pattern**.
    Most important difference is “*” is a wildcard qualifier not a wildcard.

- ssh : (secure shell) safe, encrypted, remote-login between client/server hosts.

```
ssh [ -Y ] [ -l user ] [ user@ ] hostname
```

  - -Y allows remote computer (University) to create windows on local computer (home).

- -l login user on the server machine.
- To login from home to UW environment:

```
$ ssh -Y -l jfdoe linux.student.cs.uwaterloo.ca
...                # enter password, run commands (editor, programs)
$ ssh -Y jfdoe@linux.student.cs.uwaterloo.ca
```

1.6 File Permission

- UNIX supports 3 levels of security for each file or directory based on sets of users:
  - user : owner of the file,
  - group : arbitrary name associated with a set of userids,
  - other : any other user.
- File or directory have permissions, read, write, and execute/search for the 3 sets of users.
  - Read/write allow specified set of users to read/write a file/directory.
  - Executable/search allow:
    - file : execute as a command, e.g., file contains a program or shell script,

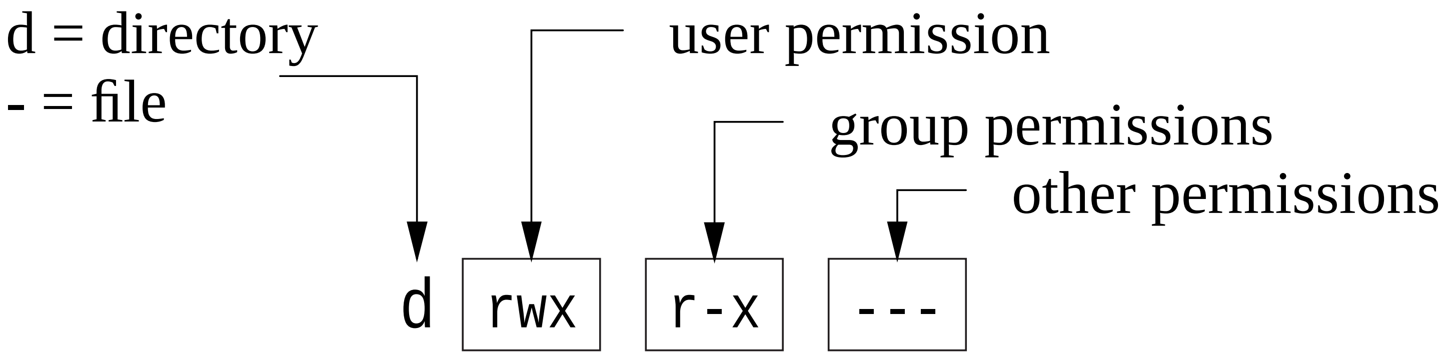
- ∗ directory : search by certain system operations but not read in general.

- Use ls -l command to print file-permission information.

```
drwxr-x--- 2 jfdoe jfdoe  4096 Oct 19 18:19 cs246/
drwxr-x--- 2 jfdoe jfdoe  4096 Oct 21 08:51 cs245/
-rw------- 1 jfdoe jfdoe  22714 Oct 21 08:50 test.cc
-rw------- 1 jfdoe jfdoe  63332 Oct 21 08:50 notes.tex
```

- Columns are: permissions, #-of-directories (including "." and ". ."), owner, group, file size, change date, file name.

- Permission information is:



d = directory
- = file
user permission
group permissions
other permissions
d rwx r-x ---

- E.g., `drwxr-x---`, indicates
  - ○ directory in which the user has read, write and execute permissions,
  - ○ group has only read and execute permissions,

- others have no permissions at all.

- ***In general, never allow "other" users to read or write your files.***
- Default permissions (usually) on:
  - file: `rw-r-----`, owner has read/write permission, and group has only read permission.
  - directory: `rwx------`, owner has read/write/execute.
- chgrp : change group-name associated with file.

```
chgrp [ -R ] group-name file/directory-list
```

  - -R recursively modify the group of a directory.

```
$ chgrp cs246_05 cs246   # course directory
$ chgrp -R cs246_05 cs246/a5   # assignment directory/files
```

  Must associate group along entire pathname and files.

- ***Creating/deleting group-names is done by system administrator.***
- chmod : add or remove from any of the 3 security levels.

```
chmod [ -R  ] mode-list file/directory-list
```

  - -R recursively modify the security of a directory.

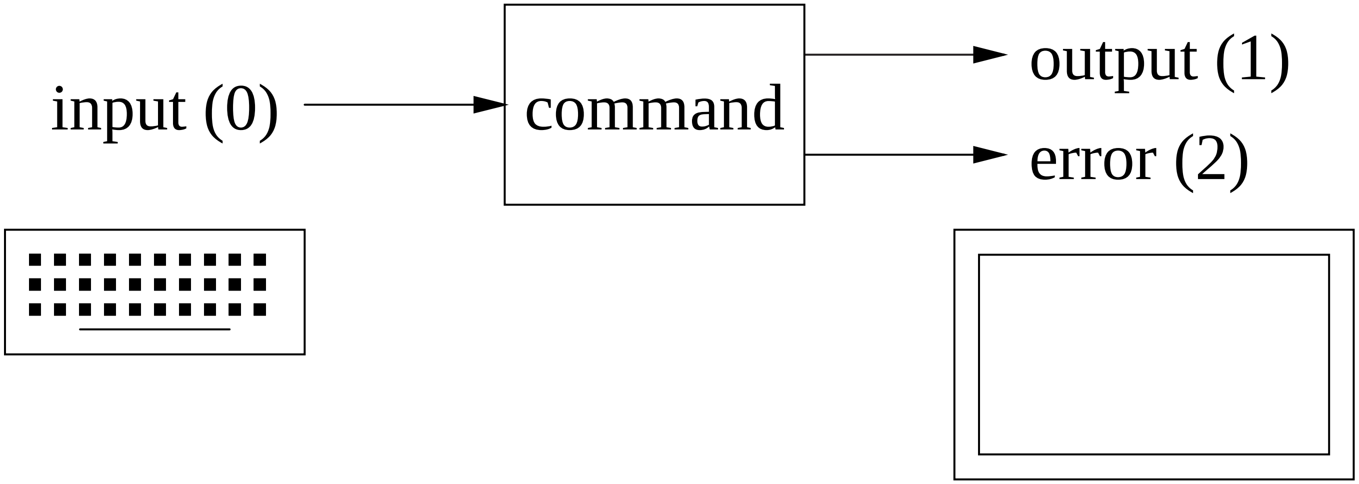
- *mode-list* has the form *security-level operator permission*.
- Security levels are denoted by u for you user, g for group, o for other, a for all (ugo).
- Operator + adds permission, - removes permission.
- Permissions are denoted by r for readable, w for writable and x for executable.
- Elements of the *mode-list* are separated by commas.

```
chmod g-r,o-r,g-w,o-w foo      # long form, remove read/write for group/oth
chmod go-rw foo                # short form
chmod g+rx cs246               # allow group users read/search
chmod -R g+rw cs246/a5         # allow group users read/write
```

Must associate permission along entire pathname and files.

1.7 Input/Output Redirection

- Every command has three standard files: input (0), output (1) and error (2).
- By default, these are connected to the keyboard (input) and screen (output/error).



input (0)

command

output (1)

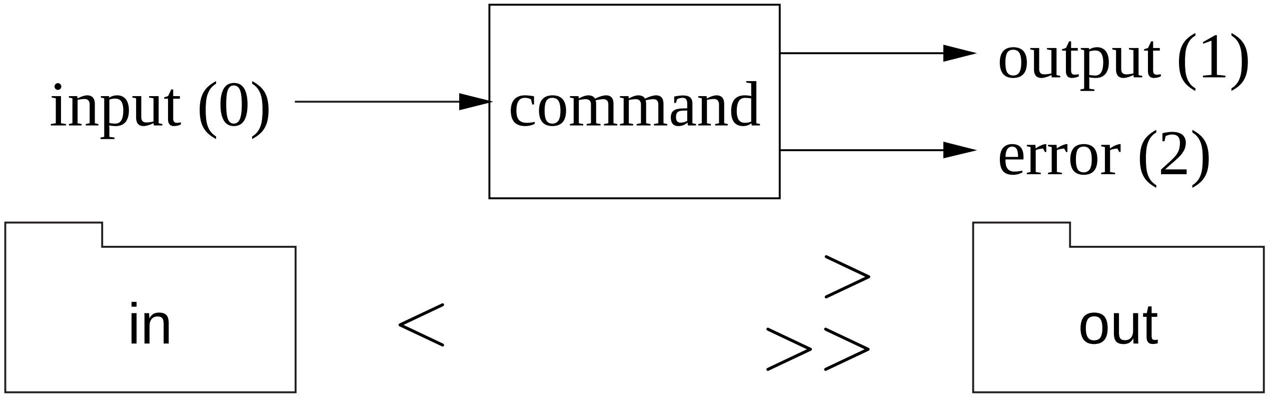
error (2)

```
$ sort -n          # numeric sort
7                  sort reads unsorted values from keyboard
30
5
C-d               close input file
5                  sort prints sorted values to screen
7
30
```

- ***To close an input file from the keyboard***, type <ctrl>-d (C-d), i.e., press <ctrl> and d keys simultaneously, causes the shell to close the keyboard input file.
- Redirection allows:

- ○ alternate input from a file (faster than typing at keyboard),
- ○ saving output to a file for subsequent examination or processing.

- Redirection performed using operators < for input and > / >> for output to/from other sources.



- ○ < means read input from file rather than keyboard.
- ○ > means (create if needed) output file and write to file rather than screen (destructive).
- ○ >> means (create if needed) output file and append to file rather than screen.

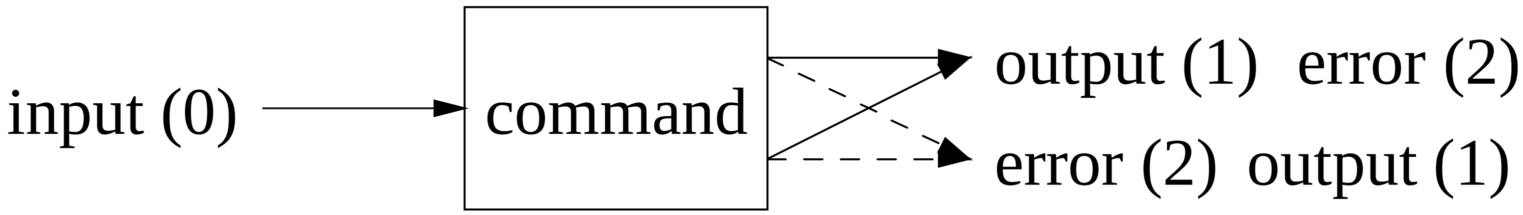
- Command is (usually) unaware of redirection.
- To distinguish between output and error, prefix output redirection with number.

```
>          # implicit, => output
1>         # explicit, => output
1>>        # => output
2>         # => error
2>>        # => error
```

- Normally, standard error (e.g., error messages) is not redirected because of its importance.

```
$ sort < in                  # input from file “in”; output to screen
$ sort < in > out            # input from file “in”; output to file “out”
$ ls -al 1> out              # output to file “out”
$ ls -al >> out              # append output to file “out”
$ sort 2>> errs              # append errors to file “errs”
$ sort 1> out 2> errs        # output to file “out”; errors to file “errs”
```

- Can tie standard error to output (and vice versa) using “>&” ⇒ both write to same place.



- Order of tying redirection files is important.

```
$ sort 2>&1 > out      # tie stderr (2) to stdout (1), stdout to “out”
$ sort > out 2>&1      # redirect stdout to “out”, tie stderr to stdout => “out”
```

- To ignore output, redirect to pseudo-file /dev/null.

```
$ sort data 2> /dev/null   # ignore error messages
```

- Redirection requires explicit creation of intermediate (temporary) files.

```
$ sort data > sortdata       # sort data and store in “sortdata”
$ grep -v "abc" sortdata > temp  # remove lines with “abc”, store in “tem
$ tr a b < temp > result   # translate a′s to b′s and store in “result”
$ rm sortdata temp           # remove intermediate files
```

- Shell pipe operator | makes standard output for a command the standard input for the next command, without creating intermediate file.

```
$ sort data | grep -v "abc" | tr a b > result
```

- Standard error is not piped unless redirected to standard output.

```
$ sort data 2>&1 | grep -v "abc" 2>&1 | tr a b > result 2>&1
```

now both standard output and error go through pipe.

- Print file hierarchy using indentation.

```
$ find cs246
cs246/
cs246/a1
cs246/a1/q1x.C
cs246/a1/q2y.h
cs246/a1/q2y.cc
cs246/a1/q3z.cpp
```

```
$ find cs246 | sed ′s|[^/]∗/|   |g′
cs246
   a1
      q1x.C
      q2y.h
      q2y.cc
      q3z.cpp
```

sed : inline editor, pattern changes all occurrences (g) of string [^/]∗/ (zero or more characters not “/” and then “/”, where “∗” is a wildcard qualifier not a wildcard) to 3 spaces.

1.8 Programming

- A **shell program** or **script** is a file containing shell commands to be executed.

```
#!/bin/bash  [ -x ]
date                    # shell and OS commands
whoami
echo Hi There
```

- First line should begin with magic comment: “*#!*” (sha-bang) with shell pathname for executing the script.
- It forces a specific shell to be used, which is run as a subshell.
- If the “*#!*” line is missing, a subshell of the same kind as the invoking shell is used for sh shells and sh is used for csh shells.
- ***Optional -x is for debugging and prints trace of the script during execution.***
- A script can be invoked directly using a specific shell, or as a command if it has executable permissions.

```
$ bash scriptfile            # direct invocation
Sat Dec 19 07:36:17 EST 2009
jfdoe
Hi There!
$ chmod u+x scriptfile       # make script file executable
$ ./scriptfile               # command execution
Sat Dec 19 07:36:17 EST 2009
jfdoe
Hi There!
```

- Interactive shell session is just a script reading from standard input.

1.8.1 Variables

- syntax : `[_a-zA-Z][_a-zA-Z0-9]*` where “*” is wildcard qualifier
- **case-sensitive**:

```
VeryLongVariableName      Page1     Income_Tax     _75
```

- Some identifiers are reserved (e.g., **if**, **while**), and hence, **keywords**.
- ***Variables ONLY hold string values (arbitrary length).***
- Variable is declared ***dynamically*** by assigning a value with operator “=”.

```
$ cs246assn=/u/jfdoe/cs246/a1   # declare and assign
```

No spaces before or after "=".

- A variable's value is dereferenced using operators "$" or "${}".

```
$ echo $cs246assn ${cs246assn}
/u/jfdoe/cs246/a1 /u/jfdoe/cs246/a1
$ cd $cs246assn              # or ${cs246assn}
```

Unlike alias, variable can be a command argument.

- Dereferencing an undefined variables returns the empty string.

```
$ cd $cs246assnTest          # cd /u/jfdoe/cs246/a1Test
```

Where does this move to?

- Always use braces to allow concatenation with other text.

```
$ cd ${cs246assn}Test        # cd /u/jfdoe/cs246/a1Test
```

- ***Beware commands/arguments composed in variables.***

```
$ out=sortdata                    # output file
$ dsls=′ls | sort -r > ${out}′ # store files names in descending (-r) ord
$ ${dsls}                         # execute command
ls: cannot access |: No such file or directory
ls: cannot access sort: No such file or directory
ls: cannot access >: No such file or directory
ls: cannot access ${out}: No such file or directory
```

- Behaviour results because the shell tokenizes, substitutes, and then executes.
- Initially, the shell sees only one token, “${dsls}”, so the tokens ***within*** the variable are not marked correctly, e.g., “|” and ”>” not marked as pipe/redirection tokens.
- Then variable substitution occurs on “${dsls}”, giving tokens ′ls′ ′|′ ′sort′ ′-r′ ′>′ ′${out}′, so ls is the command and remaining tokens are file names.

  Why no “cannot access” message above for -r?
- To make this work, shell must tokenize and substitute a second time ***before*** execution.
- **eval** command causes its argument to be processed by shell.

```
$ eval ${dsls}          # tokenize/substitute and tokenize/substitute
$ cat sortdata          # no errors, check results
...                     # list of file names in descending order
```

- ○ 1st tokenize/substitute gives **eval** `′ls′ ′|′ ′sort′ ′-r′ ′>′ ′${out}′`
- ○ 2nd tokenize/substitute gives `′ls | sort -r > sortdata′`, which shell executes

1.8.2 Arithmetic

- Shell variables have type string, which has no arithmetic: `"3" + "17"`.

```
$ i=3           # i has string value “3” not integer 3
```

- Arithmetic is performed by:
  - ○ converting a string to an integer (if possible),
  - ○ performing an integer operation,
  - ○ and converting the integer result back to a string.
- bash performs these steps with shell-command operator $((*expression*)).

```
$ echo $((3 + 4 - 1))
6
$ echo $((3 + ${i} * 2))
9
$ echo $((3 + ${k}))                    # k is unset
bash: 3 + : syntax error: operand expected (error token is " ")
```

- Basic integer operations, +, -, *, /, % (modulus), with usual precedence, and ().
- For shells without arithmetic shell-command (e.g., sh, csh), use system command expr.

```
$ echo `expr 3 + 4 - 1`                 # for sh, csh
6
$ echo `expr 3 + ${i} \* 2`             # escape *
9
$ echo `expr 3 + ${k}`                  # k is unset
expr: non-numeric argument
```

1.8.3 Routine

- A routine is defined as follows:

```
routine_name() {        # number of parameters depends on call
    # commands
}
```

- Invoke like a command.

```
routine_name [ args ... ]
```

- E.g., create a routine to print incorrect usage-message.

```
usage() {
    echo "Usage: ${0} -t -g -e input-file [ output-file ]"
    exit 1              # terminate script with non-zero exit code
}
usage                   # call, no arguments
```

- Special parameter variables to access arguments/result.
  - ○ ${#} number of arguments, not including script name
  - ○ ${0} name of shell script

```
$ echo ${0}             # shell you are using (not csh)
bash
```

  - ○ ${*n*} refers to the arguments by position, i.e., 1st, 2nd, 3rd, ...

- ${*} arguments as a single string, e.g., `"${1} ${2} . . ."`, not including script name
- ${@} arguments as separate strings, e.g., `"${1}" "${2}"` . . ., not including script name
- ${?} exit status of the last routine/command executed; 0 often ⇒ exited normally.
- ${$} process id of executing script.

```
$ cat scriptfile
#!/bin/bash
rtn() {
        echo ${#}           # number of command-line arguments
        echo ${0} ${1} ${2} ${3} ${4} # arguments
        echo ${*}           # arguments as a single string
        echo ${@}           # arguments as separate strings
        echo ${$}           # process id of executing subshell
        return 17           # routine exit status
}
rtn a1 a2 a3 a4 a5      # invoke routine
echo ${?}               # print routine exit status
exit 21                 # script exit status
```

```
$ ./scriptfile              # run script
5                           # number of arguments
scriptfile a1 a2 a3 a4 # script-name / args 1-5
a1 a2 a3 a4 a5              # args 1-5, 1 string
a1 a2 a3 a4 a5              # args 1-5, 5 strings
27028                       # process id of subshell
17                          # routine exit status
$ echo ${?}                 # print script exit status
21
```

- **shift** *[* N *]* : destructively shift parameters to the left N positions, i.e., ${1}=${N+1}, ${2}=${N+2}, etc., and ${#} is reduced by N.
  - If no N, 1 is assumed.
  - ***If* N *is 0 or greater than* ${#}*, there is no shift.***

```
$ cat scriptfile
#!/bin/bash
rtn() {
        echo ${1};  shift 1
        echo ${1};  shift 2
        echo ${1};  shift 3
        echo ${1}
}
rtn 1 2 3 4 5 6 7 8
```

```
$ ./scriptfile
1
2
4
7
```

- Routines/variables must be created before used, are then visible throughout the script, and can be removed.

```
rtn1() {
        var=3                   # new variable
        rtn2                    # call rtn2, see all routines
        unset rtn2              # remove routine!!!
}
rtn2() {
        echo ${var}             # see all variables
        unset var               # remove variable!!!
}
rtn1                            # call
```

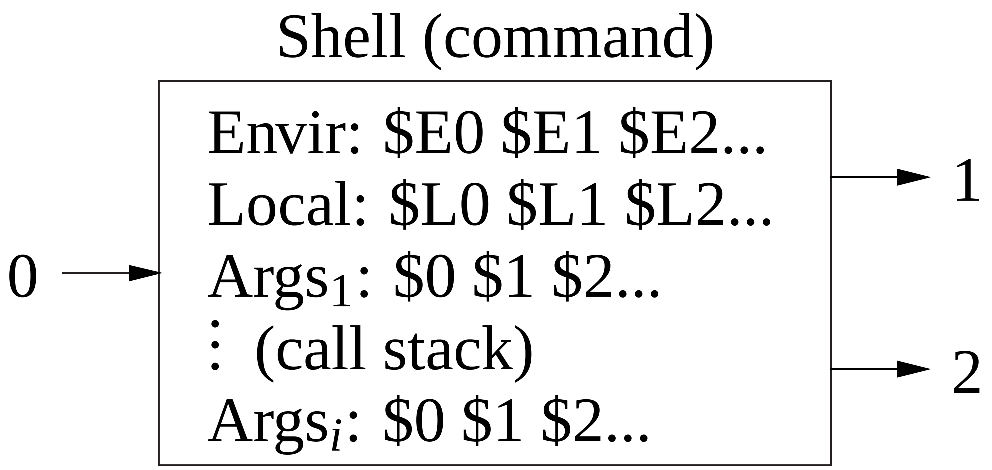
- **source** filename : execute commands from a file in the current shell.
  - For convenience or code sharing, a script may be subdivided into multiple files.
  - E.g., put commonly used routines or set of commands into separate files.
  - No "#!. . . " necessary at top, because not invoked directly like a script.
  - Sourcing a file ***includes*** it into the current shell script and ***evaluates*** the lines.

```
source ./aliases      # include/evaluate aliases into .shellrc file
source ./usage.bash   # include/evaluate usage routine into scriptfile
```

  - Created or modified variables/routines from sourced file immediately affect current shell.

1.8.4 Environment Variables

- Each shell has a list of environment (global) and script (local/parameters) variables.
- Temporally, a shell has a N lists of variables: environment, local, arguments for calls C_{1-i}.



Shell (command)

Envir: \$E0 \$E1 \$E2...
Local: \$L0 \$L1 \$L2...
Args_1: \$0 \$1 \$2...
⋮ (call stack)
Args_i: \$0 \$1 \$2...

0

1

2

- A new variable starts on the local list.

```
$ var=3               # new local variable
```

- A variable is moved to environment list if exported.

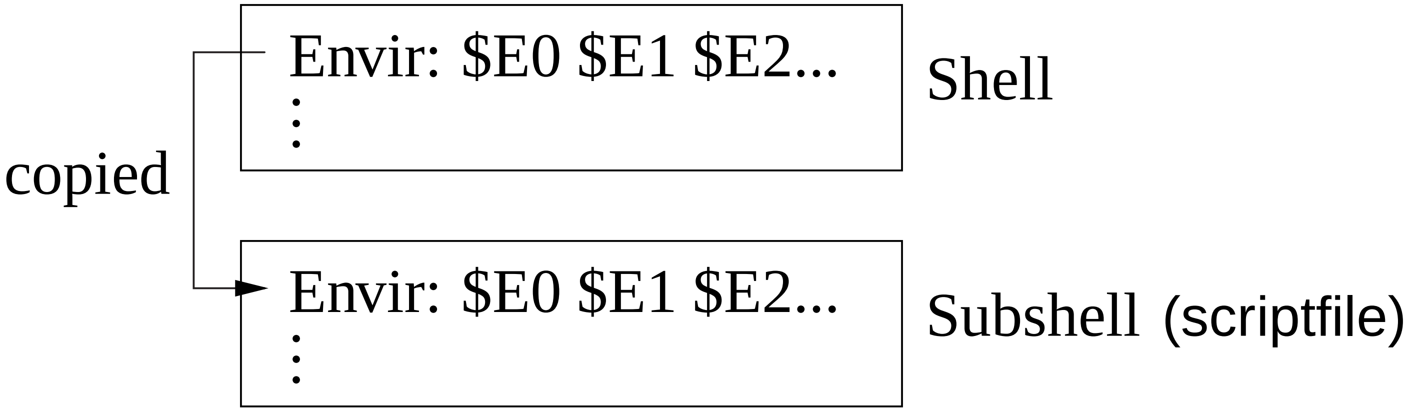
```
$ export var          # move from local to environment list
```

- Login shell starts with a number of useful environment variables, e.g.:

```
$ set                 # print variables (and values) on environment list
HOME=/u/jfdoe         # home directory
HOSTNAME=linux006.student.cs    # host computer
PATH=. . .            # lookup directories for OS commands
SHELL=/bin/bash # login shell
. . .
```

- A script executes in its own subshell with a ***copy*** of calling shell's environment variables (works across different shells).

```
$ ./scriptfile        # execute script in subshell
```



- When a (sub)shell ends, changes to its environment variables do not affect its containing shell (***environment variables only affect subshells***).
- ***Only put a variable in the environment list to make it accessible by subshells.***

1.8.5 Control Structures

- Shell provides control structures for conditional and iterative execution; syntax for bash is presented (csh is different).

1.8.5.1 Test

- **test** ([ ]) command compares strings, integers and queries files.
- **test** expression is constructed using the following:

test	**operation**	**priority**
`! expr`	not	**high**
`\( expr \)`	evaluation order (***must be escaped***)	
`expr1 -a expr2`	logical and (***not short-circuit***)	
`expr1 -o expr2`	logical or (***not short-circuit***)	**low**

- **test** comparison is performed using the following:

test	operation
string1 = string2	equal (***not ==***)
string1 != string2	not equal
integer1 -eq integer2	equal
integer1 -ne integer2	not equal
integer1 -ge integer2	greater or equal
integer1 -gt integer2	greater
integer1 -le integer2	less or equal
integer1 -lt integer2	less
-d file	exists and directory
-e file	exists
-f file	exists and regular file
-r file	exists with read permission
-w file	exists with write permission
-x file	exists with executable or searchable

- Logical operators -a (and) and -o (or) evaluate both operands.
- **test** returns 0 if expression is true and 1 otherwise (counter intuitive).

```
$ test 3 -lt 4                         # integer test
$ echo ${?}                            # true
0
$ test `whoami` = jfdoe                # string test
$ echo ${?}                            # false
1
$ test 2 -lt ${i} -o `whoami` = jfdoe # compound test
$ echo ${?}                            # true
0
$ [ -e q1.cc ]                         # file test, alternate syntax
$ echo ${?}                            # true
0
```

1.8.5.2 Selection

- An **if** statement provides conditional control-flow.

```
if test-command                 if test-command ; then
  then
      commands                        commands
elif test-command                     elif test-command ; then
  then
      commands                        commands
...                             ...
else                            else
      commands                        commands
fi                              fi
```

Semi-colon is necessary to separate test-command from keyword.

- test-command is evaluated; exit status of zero implies true, otherwise false.
- Check for different conditions:

```
if test "`whoami`" = "jfdoe" ; then
    echo "valid userid"
else
    echo "invalid userid"
fi
```

```
if diff file1 file2 > /dev/null ; then  # ignore diff output
    echo "same files"
else
    echo "different files"
fi
if [ -x /usr/bin/cat ] ; then           # alternate syntax for test
    echo "cat command available"
else
    echo "no cat command"
fi
```

- ***Beware unset variables or values with blanks.***

```
if [ ${var} = ′yes′ ] ; then . . .  # var unset => if [ = ′yes′ ]
bash: [: =: unary operator expected
if [ ${var} = ′yes′ ] ; then . . .  # var=“a b c” => if [ a b c = ′yes′ ]
bash: [: too many arguments
if [ "${var}" = ′yes′ ] ; then . . .  # var unset => if [ “” = ′yes′ ]
if [ "${var}" = ′yes′ ] ; then . . .  # var=“a b c” => if [ “a b c” = ′yes′ ]
```

When dereferencing, always quote variables!

- A **case** statement selectively executes one of *N* alternatives based on matching a string expression with a series of patterns (globbing), e.g.:

```
case expression in
     pattern | pattern | . . . )   commands   ;;
     . . .
     * )   commands ;;                  # optional match anything
esac
```

- When a pattern is matched, the commands are executed up to ";;", and control exits the **case** statement.
- If no pattern is matched, the **case** statement does nothing.
- E.g., for simple command with only one of these options:

  -h, --help, -v, -verbose, -f file

  use **case** statement to process single command-line arguments:

```
usage() { . . . }                   # print message and terminate script
case "${1}" in                      # process single command-line argument
  ′-h′ | ′--help′ ) usage ;;
  ′-v′ | ′--verbose′ ) verbose=yes ;;
  ′-f′ | ′--file′ )                 # has additional argument
     shift 1                        # access argument
     file="${1}"
     ;;
  * ) usage ;;                      # default, has to be one argument
esac
if [ ${#} -ne 1 ] ; then usage ; fi # check no other arguments
. . .                               # execute remainder of command
```

1.8.5.3 Looping

- **while** statement executes its commands zero or more times.

```
while test-command          while test-command ; do
  do
     commands                    commands
done                        done
```

- test-command is evaluated; exit status of zero implies true, otherwise false.

- Check for different conditions:

```
# search command-line parameters for “-x”
while [ "${1}" != "-x" ] ; do # string compare
      shift                          # destructive
done
i=1
while [ ${i} -le ${#} ] ; do         # process parameters, non-destructive
      eval arg="\${${i}}"            # 1st step ${1}, 2nd step argument 1
      echo "${arg}"                  # process value
      i=$((${i} + 1))
done
i=1
file=data${i}
while [ -f "${file}" ] ; do          # file regular and exists?
      . . .                          # process file
      i=$((${i} + 1))                # advance to next file
      file=data${i}
done
```

- **for** statement is a specialized **while** statement for iterating with an index over list of strings.

```
for index [ in list ] ; do
      commands
done

for arg in "${@}" ; do           # process parameters, non-destructive
      echo ${arg}
done
```

If no list, iterate over parameters, i.e., ${@}.

- Or over a set of values:

```
for (( init-expr; test-expr; incr-expr )); do   # double parenthesis
      commands
done

for (( i = 1; i <= ${#}; i += 1 )); do
      eval echo "\${${i}}"    # ${1-#}
done
```

- Use directly on command line:

```
$ for file in *.C ; do cp "${file}" "${file}".old ; done
```

- A **while**/**for** loop may contain **continue** and **break** to advance to the next loop iteration or terminate loop.

```
for count in "one" "two" "three & four" ; do
    ...
  if [ "`whoami`" = "jfdoe" ] ; then continue ; fi  # next iteration
    ...
  if [ ${?} -ne 0 ] ; then break ; fi              # exit loop
    ...
done
```

1.9 Cleanup Script

```
#!/bin/bash
#
# List and remove unnecessary files in directories
#
# Usage: cleanup [ [ -r|R ] [ -i|f ] directory-name ]+
#     -r|-R clean specified directory and all subdirectories
#     -i|-f prompt or not prompt for each file removal
# Examples:
#     $ cleanup jfdoe
#     $ cleanup -R .
#     $ cleanup -r dir1 -i dir2 -r -f dir3
# Limitations:
#     * only removes files named: core, a.out, *.o, *.d
#     * does not handle file names with special characters

usage() {                                    # print usage message & terminate
    echo "Usage: ${0} [ [ -r | -R ] [-i | -f] directory-name ]+"
    exit 1
}
defaults() {                                 # defaults for each directory
    prompt="-i"                              # do not prompt for removal
    depth="-maxdepth 1"                      # not recursive
}
```

```
remove() {
    for file in `find "${1}" ${depth} -type f -a \( -name 'core' -o \
        -name 'a.out' -o -name '*.o' -o -name '*.d' \)`
    do
        echo "${file}"                      # print removed file
        rm "${prompt}" "${file}"
    done
}
if [ ${#} -eq 0 ] ; then usage ; fi        # no arguments ?
defaults                                    # set defaults for directory
while [ "${#}" -gt 0 ] ; do                 # process command-line arguments
    case "${1}" in
      "-h" ) usage ;;                       # help ?
      "-r" | "-R" ) depth="" ;;             # recursive ?
      "-i" | "-f") prompt="${1}" ;;         # prompt for deletion ?
      * )                                   # directory name ?
        remove "${1}"                       # remove files in this directory
        defaults                            # set defaults for directory
        ;;
    esac
    shift                                   # remove argument
done
```

1.10 Regress Script

```
#!/bin/bash
#
# Compare output from two programs printing any differences.
#
# Usage: regress program1 ′program1-options′ program2 ′program2-options′ arg
#
# Examples:
#    regress cat ′′ cat ′-n′ regress regress
#    regress regress “cat ′′ cat ′-n′” regress “cat ′′ cat ′-n′” regress regress
#    regress myprog ′-w′ samplesoln ′-w′ 27 100 -2 -100

usage() {
    echo "Usage: ${0} program1 \"program1-options\"" \
            "program2 \"program2-options\" argument-list"
    exit 1
}
```

```
# check command-line arguements
if [ ${#} -lt 5 ] ; then usage ; fi
if [ ! -x "`type -p ${1}`" ] ; then echo "program1 is not executable" ; usa
if [ ! -x "`type -p ${3}`" ] ; then echo "program2 is not executable" ; usa

prog1=${1}                                # copy first 4 parameters
opts1=${2}
prog2=${3}
opts2=${4}
shift 4                                   # remove first 4 parameters

for parm in "${@}" ; do                   # process remaining parameters
    # must use eval to reevaluate parameters
    eval ${prog1} ${opts1} ${parm} > tmp1_${$} 2>&1 # run programs and save
    eval ${prog2} ${opts2} ${parm} > tmp2_${$} 2>&1
    diff tmp1_${$} tmp2_${$}              # compare output from programs
    if [ ${?} -eq 0 ] ; then              # check return code
        echo "identical output"
    fi
    rm tmp1_${$} tmp2_${$}                # clean up temporary files
done
```

2 C++

2.1 First Program

- **Java**

```
import java.lang.*; // implicit
class Hello {
  public static
    void main( String[] args ) {
      System.out.println("Hello!");
      System.exit( 0 );
    }
}
```

- **C/C++**

```
#include <stdio.h>


int main() {
  printf( "Hello!\n" );
  return 0;
}
```

```
#include <iostream> // access to output
using namespace std; // direct naming

int main() { // program starts here
  cout << "Hello!" << endl;
  return 0; // return 0 to shell, optional
}
```

- **#include** <iostream> copies (imports) basic I/O descriptions (no equivalent in Java).
- **using namespace** std allows imported I/O names to be accessed directly (otherwise qualification is necessary).
- **int** main() is the routine where execution starts.
- curly braces, { ... }, denote a block of code, i.e., routine body of main.
- cout << `"Hello!"` << endl prints `"Hello!"` to standard output, called cout (System.out in Java, stdout in C).
- endl starts a newline after `"Hello!"` (println in Java, ′`\n`′ in C).
- Optional **return** 0 returns zero to the shell indicating successful completion of the program; non-zero usually indicates an error.
- **main *magic! If no value is returned, 0 is implicitly returned.***
- Routine exit (Java System.exit) stops a program at any location and returns a code to the shell, e.g., exit(0) (**#include** <cstdlib>).
  - ◦ Literals EXIT_SUCCESS and EXIT_FAILURE indicate successful or unsuccessful termination status, e.g., **return** EXIT_SUCCESS or exit(EXIT_FAILURE).

- Java/C/C++ program must be transformed from human readable form (text) to machine readable form (numbers) for execution by computer, called **compilation**.
- Compilation is performed by a **compiler**; several different compilers exist for C++.
- Compile with g++ command:

```
$ g++ firstprogram.cc      # compile program, generate executable "a.out
$ ./a.out                  # execute program; execution permission
```

C program-files use suffix .c; C++ program-files use suffixes .C / .cpp / **.cc**.

2.2 Program Structure

- A C++ program is composed of comments for people, and statements for both people and the compiler.
- A source file contains a mixture of comments and statements.
- The C/C++ compiler only reads the statements and ignores the comments.

2.2.1 Comment

- Comments document what a program does and how it does it.

- A comment may be placed anywhere a whitespace (space, tab, newline) is allowed.
- There are two kinds of comments in C/C++ (same in Java):

	Java / C / C++
1	*/* . . . */*
2	*// remainder of line*

- First comment begins with the start symbol, /*, and ends with the terminator symbol, */, and hence, can extend over multiple lines.
- **Cannot be nested one within another**:

```
/*  ...  /* ...  */  ...  */
                 ↑    ↑
     end comment      treated as statements
```

- Be extremely careful in using this comment to elide/comment-out code:

```
/* attempt to comment-out a number of statements
while ( ... ) {
    /* ... nested comment causes errors */
    if ( ... ) {
        /* ... nested comment causes errors */
    }
}
*/
```

- Second comment begins with the start symbol, //, and continues to the end of the line, i.e., only one line long.
- Can be nested one within another:

```
// ... // ... nested comment
```

so it can be used to comment-out code:

```
// while ( ... ) {
//     /* ... nested comment does not cause errors */
//     if ( ... ) {
//         // ... nested comment does not cause errors
//     }
// }
```

2.2.2 Statement

- The syntax for a C/C++ statement is a series of tokens separated by whitespace and terminated by a semicolon (except for a block, {}).

2.3 Declaration

- A declaration introduces names or redeclares names from previous declarations.

2.3.1 Identifier

- name used to refer to a variable or type.
- syntax : [_a-zA-Z][_a-zA-Z0-9]* where "*" is wildcard qualifier
- case-sensitive:

```
VeryLongVariableName        Page1      Income_Tax      _75
```

- Some identifiers are reserved (e.g., **if**, **while**), and hence, **keyword**s.

2.3.2 Basic Types

Java	**C / C++**	
boolean	**bool** (C <stdbool.h>)	
char	**char** / **wchar_t**	ASCII / unicode character
byte	**char** / **wchar_t**	integral types
int	**int**	
float	**float**	real-floating types
double	**double**	
		label type, implicit

- C/C++ treat **char** / **wchar_t** as character and integral type.
- Java types **short** and **long** are created using type qualifiers.

2.3.3 Variable Declaration

- Declaration in C/C++ type followed by list of identifiers, except label which has implicit type (same in Java).

Java / C / C++

char a, b, c, d;
int i, j, k;
double x, y, z;
id :

- Declarations may have an initializing assignment (except for fields in **struct**/**class**):

```
int i = 3;              int i = 3, j = 4, k = 5;
int j = 4;
int k = 5;
```

- Value of an **uninitialized variable** is usually undefined.

```
int i;
cout << i << endl;    // i has undefined value
```

  Some C/C++ compilers check for uninitialized variables (use -Wall option).

2.3.4 Type Qualifier

- C/C++ provide two basic integral types **char** and **int**.

- Other integral types are generated using type qualifiers to modify the basic types.
- C/C++ provide size and signed-ness (positive/negative)/(positive only) qualifiers.
- **#include** <climits> specifies names for lower and upper bounds of a type's range of values.

integral types	range (lower/upper bound name)
char (**signed char**)	SCHAR_MIN to SCHAR_MAX, e.g., -128 to 127
unsigned char	0 to UCHAR_MAX, e.g. 0 to 255
short (**signed short int**)	SHRT_MIN to SHRT_MAX, e.g., -32768 to 32767
unsigned short (**unsigned short int**)	0 to USHRT_MAX, e.g., 0 to 65535
int (**signed int**)	INT_MIN to INT_MAX, e.g., -2147483648 to 2147483647
unsigned int	0 to UINT_MAX, e.g., 0 to 4294967295
long (**signed long int**)	(LONG_MIN to LONG_MAX), e.g., -2147483648 to 2147483647
unsigned long (**unsigned long int**)	0 to (ULONG_MAX, e.g. 0 to 4294967295
long long (**signed long long int**)	LLONG_MIN to LLONG_MAX, e.g., -9223372036854775808 to 9223372036854775807
unsigned long long (**unsigned long long int**)	0 to (ULLONG_MAX), e.g., 0 to 18446744073709551615

- **int** range is machine specific: e.g., 2 bytes for 16-bit computer and 4 bytes for 32/64-bit computer.

- **long** range is at least as large as **int**: e.g., 2/4 bytes for 16-bit computer and 4/8 bytes for 32/64-bit computer.
- **#include** <stdint.h> provides ***absolute*** types [u]intN_t for **signed**/**unsigned** N = 8, 16, 32, 64 bits.

integral types	range (lower/upper bound name)
int8_t	INT8_MIN to INT8_MAX, e.g., -128 to 127
uint8_t	0 to UINT8_MAX, e.g., 0 to 255
int16_t	INT16_MIN to INT16_MAX, e.g., -32768 to 32767
uint16_t	0 to UINT16_MAX, e.g., 0 to 65535
int32_t	INT32_MIN to INT32_MAX, e.g., -2147483648 to 2147483647
uint32_t	0 to UINT32_MAX, e.g., 0 to 4294967295
int64_t	INT64_MIN to INT64_MAX, e.g., -9223372036854775808 to 9223372036854775807
uint64_t	0 to UINT64_MAX, e.g., 0 to 18446744073709551615

- C/C++ provide two basic real-floating types **float** and **double**, and one real-floating type generated with type qualifier.
- **#include** <cfloat> specifies names for precision and magnitude of real-floating values.

real-float types	range (precision, magnitude)
float	FLT_DIG precision, FLT_MIN_10_EXP to FLT_MAX_10_EXP, e.g,. 6+ digits over range 10^{-38} to 10^{38}, IEEE (4 bytes)
double	DBL_DIG precision, DBL_MIN_10_EXP to DBL_MAX_10_EXP, e.g., 15+ digits over range 10^{-308} to 10^{308}, IEEE (8 bytes)
long double	LDBL_DIG precision, LDBL_MIN_10_EXP to LDBL_MAX_10_EXP, e.g., 18-33+ digits over range 10^{-4932} to 10^{4932}, IEEE (12-16 bytes)

float : ±1.17549435e-38 to ±3.40282347e+38

double : ±2.2250738585072014e-308 to ±1.7976931348623157e+308

long double : ±3.36210314311209350626e-4932 to ±1.18973149535723176502e+4

2.3.5 Literals

- Variables contain values, and each value has a **constant** (C) or **literal** (C++) meaning.
- E.g., the integral value 3 is constant/literal, i.e., it cannot change, it always means 3.

```
3 = 7;    // disallowed
```

- Every basic type has a set of literals that define its values.

- A variable's value always starts with a literal, and changes via another literal or computation.
- C/C++ and Java share almost all the same literals for the basic types.

type	**literals**
boolean	**false**, **true**
character	'a', '\''
integral	decimal : 123, -456, 123456789 octal, prefix 0 : 0144, -045, 04576132 hexadecimal, prefix 0X / 0x : 0xfe, -0X1f, 0xe89abc3d
real-floating	.1, 1., -1., 0.52, -7.3E3, -6.6e-2, E/e exponent

- Use the right literal for a variable's type:

```
bool b = true;              // not 1
int i = 1;                  // not 1.0
double d = 1.0              // not 1
char c = 'a';               // not 97
```

- Escape sequence provides quoting of special characters in a **char** literal using a , \.

`'\\'`	backslash
`'\''`	single quote
`'\t'`, `'\n'`	(special names) tab, newline, ...
`'\0'`	zero, string termination character
`'\ooo'`	octal value, `ooo` up to 3 octal digits
`'\xhh'`	hexadecimal value, `hh` up to 2 hexadecimal digits for **char**, up to 4 hexadecimal digits for **wchar_t** (not Java)

```
cout << '\\' << endl
     << '\'' << endl
     << '\t' << '\t' << 'x' << '\n' // newline value 10
     << 'y' << '\12'        // octal 10
     << 'z' << '\xa';       // hexadecimal 10
```

```
\
'
                x
y
z
```

- C/C++ provide user named literals (write-once/read-only variables) with type qualifier **const** (Java **final**).

Java	**C/C++**

final char Initial = ′D′;
final short int Size = 3, SupSize;
SupSize = Size + 7;
final double PI = 3.14159;
	const char Initial = ′D′;
	const short int Size = 3, SupSize = Size + 7;
	disallowed
	const double PI = 3.14159;

- C/C++ **const** variable ***must*** be assigned a value at declaration (or by a constructor’s declaration); the value can be the result of an expression.
- A constant variable can (only) appear in contexts where a literal can appear.

```
Size = 7;    // disallowed
```

- Good practise is to name literals so all usages can be changed via the initialization value.
- There are trillions of literals ⇒ cannot all be stored in memory.
- Only literals used in a program occupy storage, some are embedded directly into computer instructions.

2.4 Expression

	Java	**C/C++**	**priority**
unary	., (), [], call	::, ., ->, (), [], call, **dynamic_cast**	**high**
	cast, +, -, !, ~ **new**	cast, +, -, !, ~, &, * **new**, **delete**, **sizeof**	
binary	*, /, %	*, /, %	
	+, -	+, -	
bit shift	<<, >>, >>>	<<, >>	
relational	<, <=, >, >=, instanceof	<, <=, >, >=	
equality	==, !=	==, !=	
bitwise	& and	&	
	^ exclusive-or	^	
	| or	|	
logical	&& short-circuit	&&	
	||	||	
conditional	?:	?:	
assignment	=, +=, -=, *=, /=, %= <<=, >>=, >>>=, &=, ^=, |=	=, +=, -=, *=, /=, %= <<=, >>=, &=, ^=, |=	
comma		,	**low**

- Expression evaluation is like algebra:
  - predefined operations exist and are invoked using name with parenthesized argument(s).

    `abs( -3 );` $|-3|$

    `sqrt( x );` $\sqrt{x}$

    `pow( x, y );` x^y

  - operators are prioritized and performed from high to low.

    ```
    x + y * sqrt( z );     // call, multiple, add
    ```

  - operators with same priority are done left to right

    ```
    x + y - z;             // add, subtract
    3.0 / v * w;           // divide, multiple
    ```

    except for unary, `?`, and assignment operators, which associate right to left.

    ```
    -~x;                   // complement, negate
    *&p;                   // address-of, dereference
    x = y = z;             // z to y to x
    ```

  - parentheses are used to control order of evaluation, i.e., override rules.

```
    x + y * z / w;          // multiple, divide, add
    ((x + y) * (z / w);     // add, divide, multiple
```

- Order of subexpressions and argument evaluation is unspecified (Java left to right).

```
( i + j ) * ( k + j );          // either + done first
( i = j ) + ( j = i );          // either = done first
g( i ) + f( k ) + h( j );       // g, f, or h called in any order
f( p++, p++, p++ );             // arguments evaluated in any order
```

- C++ relational/equality return **false**/**true**; C return 0/1.
- Referencing (address-of), &, and dereference, *, operators do not exist in Java because access to storage is restricted.
- Pseudo-routine **sizeof** returns the number of bytes for a type or variable (not in Java):

```
long int i;
sizeof(long int);               // type, at least 4
sizeof(i);                      // variable, at least 4
```

The **sizeof** a pointer (type or variable) is the size of the pointer on that particular computer and not the size of the type the pointer references.

- Bit-shift operators, << (left), and >> (right) shift bits in integral variables left and right.
  - left shift is multiplying by 2, modulus variable’s size;
  - right shift is dividing by 2 if unsigned or positive (like Java >>>); otherwise undefined.

```
int x, b, c;
x = y = z = 1;
cout << (x << 1) << ′ ′ << (y << 2) << ′ ′ << (z << 3) << endl;
x = y = z = 16;
cout << (x >> 1) << ′ ′ << (y >> 2) << ′ ′ << (z >> 3) << endl;
2 4 8
8 4 2
```

Why are parenthesis necessary?

- Division operator, /, accepts integral and real-float operands, but truncates for integrals.

```
3 / 4               // 0 not 0.75
3.0 / 4.0           // 0.75
```

- Remainder (modulus) operator, %, only accepts integral operands.
  - If either operand is negative, the sign of the remainder is implementation

defined, e.g., -3 % 4, 3 % -4, -3 % -4 can be 3 or -3.

- Assignment is an operator; useful for **cascade assignment** to initialize multiple variables of the same type:

  ```
  a = b = c = 0;          // cascade assignment
  x = y = z + 4;
  ```

  - **Other uses of assignment in an expression are discouraged!**; i.e., assignments only on left side.

- General assignment operators, e.g., lhs += rhs does NOT mean:

  ```
  lhs = lhs + rhs;
  ```

  instead, implicitly rewritten as:

  ```
  temp = &(lhs);  *temp = *temp + rhs;
  ```

  hence, the left-hand side, lhs, is evaluated only once:

  ```
  v[ f(3) ] += 1;                // only calls once
  v[ f(3) ] = v[ f(3) ] + 1;     // calls twice
  ```

- Comma expression allows multiple expressions to be evaluated in a context where only a single expression is allowed.

  ```
  x,  f + g,  sqrt( 3 ) / 2,  m[ i ][ j ]  ← value returned
  ```

  - Expressions evaluated left to right with the value of rightmost expression returned.

- Operators ++ / -- are discouraged because subsumed by general += / -=.

```
i += 1;   versus   i ++
i += 3;   versus   i ++ ++ ++; // disallowed
```

2.4.1 Conversion

- **Conversion** transforms a value from one type to another by changing the value to the new type's representation.
- Conversions can occur implicitly by the compiler or explicitly by the programmer.
- Two kinds of conversions:
  - **widening**/**promotion** conversion, no information is lost:

```
bool → char → short int → long int →        double
true      1           1            1    1.000000000000000
```

    where **false** → 0; **true** → 1

  - **narrowing** conversion, information can be lost:

double	→	**long int**	→	**short int**	→	**char**	→	**bool**
77777.77777777777		77777		12241		209		**true**

where 0 → **false**; non-zero → **true**

- C/C++ have implicit widening and narrowing conversions (Java only implicit widening).
- **Implicit narrowing conversions can cause problems:**

```
int i;    double r;
i = r = 3.5;   // r -> 3.5
r = i = 3.5;   // r -> 3.0 ???
```

- Good practice is to perform narrowing conversions explicitly as documentation using C **cast** operator or C++ **static_cast** operator.

```
int i;    double x = 7.2, y = 3.5;
i = (int) x;                    // explicit narrowing conversion
i = (int) x / (int) y;     // explicit narrowing conversions for integer division
i = static_cast<int>(x / y);   // alternative technique after integer division
```

- C/C++ supports casting among the basic types and user defined types.

2.4.2 Coercion

- **Coercion** ***forces*** a transformation of a value to another type but the result is not meaningful in the new type's representation.
- Some narrowing conversions are considered coercions.
  - E.g., when a value is truncated or converting non-zero to **true**, the result is nonsense in the new type's representation.
- Also, having type **char** represent ASCII characters ***and*** integral (byte) values allows:

  ```
  char ch = 'z' - 'a';        // character arithmetic!
  ```

  which may or may not be reasonable as it might generate an invalid character.
- But the most common coercion is through pointers:

  ```
  int i, *ip = &i;
  double d, *dp = &d;
  dp = (double *)ip;     // dp points at an integer not a double!!!
  ```

  Using the explicit cast, programmer has lied to the compiler about the type of ip.
- ***Good practice is to limit narrowing conversions and NEVER lie about a variable's types.***

2.4.3 Math Operations

- **#include** <cmath> provides overloaded real-float mathematical-routines for types **float**, **double** and **long double**:

operation	routine	operation	routine
$\lvert x\rvert$	abs(x)	$x \bmod y$	fmod(x, y)
$\arccos x$	acos(x)	$\ln x$	log(x)
$\arcsin x$	asin(x)	$\log x$	log10(x)
$\arctan x$	atan(x)	x^y	pow(x, y)
$\lceil x\rceil$	ceil(x)	$\sin x$	sin(x)
$\cos x$	cos(x)	$\sinh x$	sinh(x)
$\cosh x$	cosh(x)	$\sqrt{x}$	sqrt(x)
e^x	exp(x)	$\tan x$	tan(x)
$\lfloor x\rfloor$	floor(x)	$\tanh x$	tanh(x)

and math literals:

M_E	2.7182818284590452354	*// e*
M_LOG2E	1.4426950408889634074	*// log_2 e*
M_LOG10E	0.43429448190325182765	*// log_10 e*
M_LN2	0.69314718055994530942	*// log_e 2*
M_LN10	2.30258509299404568402	*// log_e 10*
M_PI	3.14159265358979323846	*// pi*
M_PI_2	1.57079632679489661923	*// pi/2*
M_PI_4	0.78539816339744830962	*// pi/4*
M_1_PI	0.31830988618379067154	*// 1/pi*
M_2_PI	0.63661977236758134308	*// 2/pi*
M_2_SQRTPI	1.12837916709551257390	*// 2/sqrt(pi)*
M_SQRT2	1.41421356237309504880	*// sqrt(2)*
M_SQRT1_2	0.70710678118654752440	*// 1/sqrt(2)*

- Some systems also provide **long double** math literals.
- pow(x,y) (x^y) is computed using logarithms, $10^{y \log x}$ (versus repeated multiplication), when y is non-integral value $\Rightarrow y \geq 0$

pow(2.0, -3.0); $2^{-3} = \frac{1}{2^3} = \frac{1}{2\times2\times2} = \frac{1}{8} = 0.125$

pow(-2.0, -3.1); $2^{-3.1} = \frac{1}{2^{3.1}} =$ nan (not a number) $\log -3.1$ undefined

- Quadratic roots:

$$r = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
#include <iostream>
#include <cmath>
using namespace std;

int main() {
    double a = 3.5, b = 2.1, c = -1.2;
    double dis = b * b - 4.0 * a * c, dem = 2.0 * a;
    cout << "root1: " << ( -b + sqrt( dis ) ) / dem << endl;
    cout << "root2: " << ( -b - sqrt( dis ) ) / dem << endl;
}
```

- Must explicitly link in the math library:

```
$ g++ roots.cc -lm  # link math library
```

2.5 Control Structures

	Java	**C/C++**
block	{ *intermixed decls/stmts* }	{ *intermixed decls/stmts* }

selection	**if** (*bool-expr1*) *stmt1*
	else if (*bool-expr2*) *stmt2*
	. . .
	else *stmtN*
		if (*bool-expr1*) *stmt1*
		else if (*bool-expr2*) *stmt2*
		. . .
		else *stmtN*

	switch (*integral-expr*) {
	  case c1: *stmts1*; **break**;
	  . . .
	  case cN: *stmtsN*; **break**;
	  default: *stmts0*;
	}
		switch (*integral-expr*) {
		  case c1: *stmts1*; **break**;
		  . . .
		  case cN: *stmtsN*; **break**;
		  default: *stmts0*;
		}

looping	**while** (*bool-expr*) *stmt*	**while** (*bool-expr*) *stmt*
	do *stmt* **while** (*bool-expr*) ;	**do** *stmt* **while** (*bool-expr*) ;
	for (*init-expr*;*bool-expr*;*incr-expr*) *stmt*	**for** (*init-expr*;*bool-expr*;*incr-expr*) *stmt*
transfer	**break** *[label]*	**break**
	continue *[label]*	**continue**
		goto *label*
	return *[expr]*	**return** *[expr]*
	throw *[expr]*	**throw** *[expr]*
label	*label* : *stmt*	*label* : *stmt*

2.5.1 Block

- **Block** is a series of statements bracketed by braces, {. . .}, which can be nested.
- A block forms a complete statement and does not have to be terminated with a semicolon.
- Block serves two purposes: bracket several statements into a single statement and introduce local declarations.
- Good practice is to always use a block versus single statement to allow easy insertion and removal of statements to or from block.

```
if ( x > y )          // no block
    x = 0;            // cannot directly add statements

if ( x > y ) {        // block
    x = 0;            // can directly add/remove statements
}
```

  Does the shell have this problem?

- Declarations may be intermixed among executable statements in a block.
- Variables in blocks are allocated in first-in first-out (FIFO) order from a memory area called the **stack**.

- Localizing declarations in nested blocks helps reduce declaration clutter at the beginning of a block.

```
int i, j, k; // global      int i;
... // use i, j, k          ... // use i
                            {
                                ...
                                int j; // local
                                ... // use i, j
                                {
                                    int k; // local
                                    ... use i, j, k
```

  However, it can also make locating declarations more difficult.

- Variable names can be reused in different blocks, i.e., possibly **overriding** (hiding) prior variables.

```
int i = 1; ...           // first i
{ int k = i, i = 2;...  // second i (override first), both i′s used in block!
    { int i = 3;...      // third i (override second)
```

2.5.2 Selection

- C/C++ selection statements are **if** and **switch** (same as Java).

- An **if** statement selectively executes one of two alternatives based on the result of a comparison, e.g.:

```
if ( x > y ) max = x;
else max = y;
```

- For nested **if** statements, **else** matches with the closest **if**, which results in the **dangling else** problem.
- E.g., reward WIDGET salesperson who sold more than $10,000 worth of WIDGETS and dock pay of those who sold less than $5,000.

Dangling Else	Fix Using Null Else	Fix Using Blocks

if (sales < 10000)
    if (sales < 5000)
        income -= penalty;
else *// incorrect match!!!*
    income += bonus;
	if (sales < 10000)
	    if (sales < 5000)
	        income -= penalty;
	    else ; *// null statement*
	else
	    income += bonus;
		if (sales < 10000) {
		    if (sales < 5000) **{**
		        income -= penalty;
		    } *// block*
		} **else** {
		    income += bonus;
		}

- Unnecessary equality for boolean as value is already **true** or **false**.

```
bool b;
if ( b == true ) . . .  // if ( b )
```

- Common mistake to assign y to x and converts x to **bool** (possible in Java for one type).

```
if ( x = y )...
```

- A **switch** statement selectively executes one of *N* alternatives based on matching an integral value with a series of case clauses, e.g.:

```
switch ( day ) {                     // integral expression
  case MON: case TUE: case WED: case THU:  // case value list
     cout << "PROGRAM" << endl;
     break;                          // exit switch
  case FRI:
     wallet += pay;
     // FALL THROUGH
  case SAT:
     cout << "PARTY" << endl;
     wallet -= party;
     break;                          // exit switch
  case SUN:
     cout << "REST" << endl;
     break;                          // exit switch
  default:
     cerr << "ERROR: bad day" << endl;
     exit( EXIT_FAILURE );           // terminate program
}
```

- Only one label for each **case** clause but a list of **case** clauses is allowed.
- Once a case clause is matched, its statements are executed, and control continues to the ***next*** statement.

- If no case clause is matched and there is a **default** clause, its statements are executed, and control continues to the ***next*** statement.
- Unless there is a **break** statement to prematurely exit the **switch** statement.
- **It is a common error to forget the break in a case clause.**
- Otherwise, the **switch** statement does nothing.

2.5.3 Conditional Expression Evaluation

- **Conditional expression evaluation** performs partial (**short-circuit**) expression evaluation.

&&	only evaluates the right operand if the left operand is true
||	only evaluates the right operand if the left operand is false
?:	only evaluates one of two alternative parts of an expression

- && and || are similar to logical & and | for bitwise (boolean) operands, i.e., both produce a logical conjunctive or disjunctive result.
- However, short-circuit operators evaluate operands lazily until a result is determined, short circuiting the evaluation of other operands.

```
d != 0 && n / d > 5  // may not evaluate right operand, prevents division b
```

false and anything is?

- Hence, short-circuit operators are control structures in the middle of an expression because `e1 && e2` ≠ `&&( e1, e2 )` (unless lazy evaluation).
- Logical & and | evaluate operands eagerly, evaluating both operands.
- Conditional `?:` evaluates one of two expressions, and returns the result of the evaluated expression.
- Acts like an **if** statement in an expression and can eliminate temporary variables.

```
f( ( a < 0 ? -a : a ) + 2 );
```

```
int temp;
if ( a < 0 ) temp = -a;
else temp = a;
f( temp + 2 );
```

2.5.4 Looping

- C/C++ looping statements are **while**, **do** and **for** (same as Java).
- **while** statement executes its statement **zero or more times**.

```
while ( x < 5 ) {
    . . .        // executes 0 or more times
}
```

- **Beware of accidental infinite loops.**

```
x = 0;                                        x = 0;
while (x < 5); // extra semicolon!            while (x < 5) // missing block
    x = x + 1;                                    y = y + x;
                                                  x = x + 1;
```

- **do** statement executes its statement **one or more times**.

```
do {
    . . .        // executes one or more times
} while ( x < 5 );
```

- **for** statement is a specialized **while** statement for iterating with an index.

```
init-expr;
while ( bool-expr ) {          for ( init-expr; bool-expr; incr-expr ) {
    stmts;                         stmts;
    incr-expr;
}                              }
```

- If *init-expr* is a declaration, the scope of its variables is the remainder of the declaration, the other two expressions, and the loop body.

```
for ( int i = 0, j = i; i < j; i += 1 ) { // i and j declared
        // i and j visible
} // i and j deallocated and invisible
```

- Many ways to use the **for** statement to construct iteration:

```
for ( i = 1; i <= 10; i += 1 ) {                    // count up
        // loop 10 times
} // i has value 11 on exit
for ( i = 10; 1 <= i; i -= 1 ) {                    // count down
        // loop 10 times
} // i has value 0 on exit
for ( p = s; p != NULL; p = p->link ) {             // pointer index
        // loop through list structure
} // p has the value NULL on exit
for ( i = 1, p = s; i <= 10 & p != NULL; i += 1, p = p->link ) {  // 2 indice
        // loop until 10th node or end of list encountered
}
```

- Comma expression is used to initialize and increment 2 indices in a context where normally only a single expression is allowed.

- Default **true** value inserted if no conditional is specified in **for** statement.

```
for ( ; ; )          // rewritten as: for ( ; true ; )
```

- **break** statement terminates enclosing loop body.
- **continue** statement advances to the next loop iteration.

2.6 Structured Programming

- **Structured programming** is about managing (restricting) control flow using a fixed set of well-defined control-structures.
- A small set of control structures used with a particular programming style make programs easier to write and understand, as well as maintain.
- Most programmers adopt this approach so there is a universal (common) approach to managing control flow (e.g., like traffic rules).
- Developed during the 1970's to overcome the indiscriminant use of the GOTO statement.
- GOTO leads to convoluted logic in programs (i.e., does NOT support a methodical thought process).

- I.e., arbitrary transfer of control makes programs difficult to understand and maintain.
- Restricted transfer reduces the points where flow of control changes, and therefore, is easy to understand.
- There are 3 levels of structured programming:

classical

- ○ sequence: series of statements
- ○ if-then-else: conditional structure for making decisions
- ○ while: structure for loops with test at top

Can write any program (actually only need **while**s or one **while** and **if**s).

extended

- ○ use the classical control-structures and add:
  - ∗ case/switch: conditional structure for making decisions
  - ∗ for: while with initialization/test/increment
  - ∗ repeat-until/do-while: structure for loops with test at bottom

modified

- ○ use the extended control-structures and add:
  - ∗ one or more exits from arbitrary points in a loop

  - ∗ exits from multiple nested control structures
  - ∗ exits from multiple nested routine calls

2.6.1 Multi-Exit Loop

- A **multi-exit loop** (or mid-test loop) is a loop with one or more exit locations occurring *within* the body of the loop.
- While-loop has 1 exit located at the top:

```
while i < 10 do     loop                      -- infinite loop
                       exit when i >= 10; -- loop exit
    . . .                 . . .         ↑ reverse condition
end while           end loop
```

- Repeat-loop has 1 exit located at the bottom:

```
do                  loop                      -- infinite loop
    . . .                 . . .
                       exit when i >= 10; -- loop exit
while ( i < 10 )    end loop            ↑ reverse condition
```

- Exit should not be restricted to only top and bottom, i.e., can appear in the loop body:

```
loop
      . . .
   exit when i >= 10;
      . . .
end loop
```

- Or allow multiple exit conditions:

```
loop
      . . .
   exit when i >= 10;
      . . .
   exit when j >= 10;
      . . .
end loop
```

- Eliminates priming (duplicated) code necessary with **while**:

```
read( input, d );               loop
while ! eof( input ) do               read( input, d );
      . . .                        exit when eof( input );
      read( input, d );               . . .
end while                       end loop
```

- ***Good practice is to reduce or eliminate duplicate code.*** Why?

- The loop exit is outdented or clearly commented (or both) so it can be found without having to search the entire loop body.
- Same indentation rule as for the **else** of the if-then-else (outdent **else**):

```
if . . . then     if . . . then
    XXX               XXX
    else            else
    XXX               XXX
end if            end if
```

- A multi-exit loop can be written in C/C++ in the following ways:

```
for ( ;; ) {                  while ( true ) {              do {
    . . .                          . . .                        . . .
  if ( i >= 10 ) break;         if ( i >= 10 ) break;         if ( i >= 10 ) break;
    . . .                          . . .                        . . .
  if ( j >= 10 ) break;         if ( j >= 10 ) break;         if ( j >= 10 ) break;
    . . .                          . . .                        . . .
}                             }                             } while( true );
```

- The **for** version is more general as it can be easily modified to have a loop index or a while condition.

```
for ( int i = 0; i < 10; i += 1 ) { // loop index
for ( ; x < y; ) { // while condition
```

- In general, the programming language and your code-entry style should allow insertion of new code without having to change existing code.
- Eliminate **else** on loop exits:

BAD	GOOD

for (;;) {
S1
if (C1) {
S2
} **else** {
break;
}
S3
}
	for (;;) {
	S1
	if (! C1) **break**;
	S2
	S3
	}

for (;;) {
S1
if (C1) {
break;
} **else** {
S2
}
S3
}
	for (;;) {
	S1
	if (C1) **break**;
	S2
	S3
	}

S2 is logically part of loop body not part of an **if**.

- E.g., write linear search such that:

  - ○ no invalid subscript for unsuccessful search
  - ○ index points at the location of the key for successful search.

- Using only control-flow constructs **if** and **while**:

```
int i = -1;  bool found = false;
while ( i < size - 1 & ! found ) { // rewrite: &(i<size-1, !found)
      i += 1;
      found = key == list[i];
}
if ( found ) { . . .        // found
} else { . . .              // not found
}
```

Why must the program be written this way?

- Allow third construct structure: short-circuit operators.

```
for ( i = 0; i < size && key != list[i]; i += 1 ); // using for not while
      // rewrite: if ( i < size ) if ( key != list[i] )
if ( i < size ) { . . .          // found
} else { . . .                   // not found
}
```

- How does && prevent subscript error?

- Short-circuit && does not exist in all programming languages, and requires knowledge of Boolean algebra (false and anything is?).
- Multi-exit loop can be used if no && exits and does not require Boolean algebra.

```
for ( i = 0; ; i += 1 ) { // or  for ( i = 0; i < size; i += 1 )
  if ( i >= size ) break;
  if ( key == list[i] ) break;
}
if ( i < size ) { . . .          // found
} else { . . .                   // not found
}
```

- When loop ends, it is known if the key is found or not found.
- Why is it necessary to re-determine this fact after the loop?
- Can it always be re-determined?
- The extra test after the loop can be eliminated by moving its code into the loop body.

```
for ( i = 0; ; i += 1 ) {
  if ( i >= size ) { . . .      // not found
          break;
     } // exit
  if ( key == list[i] ) { . . .  // found
          break;
     } // exit
} // for
```

- E.g., an element is looked up in a list of items, if it is not in the list, it is added to the end of the list, if it exists in the list its associated list counter is incremented.

```
for ( int i = 0; ; i += 1 ) {
  if ( i >= size ) {
        list[size].count = 1;
        list[size].data = key;
        size += 1;  // needs check for array overflow
        break;
     } // exit
  if ( key == list[i].data ) {
        list[i].count += 1;
        break;
     } // exit
} // for
```

- None of these approaches is best in all cases; select the approach that best fits the problem.

2.6.2 Multi-Level Exit

- **multi-level exit** exits multiple control structures where exit points are *known* at compile time.
- Labelled exit (**break**/**continue**) provides this capability (Java):

```
L1: {
    . . . declarations . . .
    L2: switch ( . . . ) {
        L3: for ( . . . ) {
            . . . break L1; . . . // exit block
            . . . break L2; . . . // exit switch
            . . . break L3; . . . // exit loop
        }
        . . .
    }
    . . .
}
```

- Labelled **break**/**continue** transfer control out of the control structure with the corresponding label, terminating any block that it passes through.
- C/C++ do not have labelled **break**/**continue**; simulate with **goto**.
- **goto** *label* allows arbitrary transfer of control ***within*** a routine from the **goto** to statement marked with label variable.
- **Label variable** is declared by prefixing an identifier with a ":" to a statement.

```
L1: i += 1;                   // associated with expression
L2: if ( . . . ) . . .;       // associated with if statement
L3: ;                         // associated with empty statement
```

- Labels can only be declared in a routine, ***where the label has routine scope***.
  - i.e., label identifier is unique within a routine body ⇒ cannot be overridden in local blocks.

```
int L1;                       // identifier L1
L2: ;                         // identifier L2
{
    double L1;                // can override variable identifier
    double L2;                // cannot override label identifier
}
```

- **goto** transfers control backwards/forwards to labelled statement.

```
L1: ;
. . .
goto L1;                      // transfer backwards, up
goto L2;                      // transfer forward, down
. . .
L2: ;
```

- Why is it good practice to associate a label with an empty statement?

- Transforming labelled **break** to **goto**:

```
{
    ... declarations ...
    switch ( ... ) {
        for ( ... ) {
            ... goto L1; ... // exit block
            ... goto L2; ... // exit switch
            ... goto L3; ... // exit loop
        } L3: ;
        ...
    } L2: ;
    ...
} L1: ;
```

- Why are labels at the end of control structures not as good as at start?
- Multi-level exits are commonly used with nested loops:

```
for ( ;; ) {            // while ( flag1 && . . . )
    for ( ;; ) {            // while ( flag2 && . . . )
        for ( ;; ) {            // while ( flag3 && . . . )
                . . .
  if ( . . . ) goto L1;         // if (. . .) flag1=flag2=flag3=false; else
                . . .
      if ( . . . ) goto L2;         // if (. . .) flag2=flag3=false; else
                . . .
          if ( . . . ) goto L3;         // if (. . .) flag3=false; else
                . . .
        } L3: ;
    } L2: ;
} L1: ;
```

Indentation matches with control-structure terminated.

- Without multi-level exit, multiple “flag variables” are necessary.
  - **flag variable** is used solely to affect control flow, i.e., does not contain data associated with a computation.
- ***Flag variables are the variable equivalent to a goto*** because they can be set/reset/tested at arbitrary locations in a program.
- Multi-level exit allows elimination of all flag variables!

- Simple case (exit 1 level) of multi-level exit is a multi-exit loop.
- Why is it good practice to label all exits?
- **break** and labelled **break** are a **goto** with restrictions:
  - Cannot be used to create a loop (i.e., cause a backward branch); hence, all situations resulting in repeated execution of statements in a program are clearly delineated.
  - Cannot be used to branch ***into*** a control structure.
- The following control-flow pattern appears occasionally:

duplication

```
if ( . . . ) {
    stmts1;
    if ( . . . ) {
        stmts2;
        if ( . . . ) {
            stmts3;
        } else {
            stmts4;
        }
    } else {
        stmts4;
    }
} else {
    stmts4;
}
stmts5;
```

no duplication

```
  if ( . . . ) {
      stmts1;
      if ( . . . ) {
          stmts2;
          if ( . . . ) {
              stmts3;
              goto common:
          }
      }
  }
  stmts4; // only once
common: ;
  stmts5;
```

- If any conditions are false, the same code is executed (e.g., printing an error message), resulting in code duplication.
- Multi-level exit removes all duplication of stmts4.
- **Only use goto to simulate labelled break and continue.**

- **return** statements can simulate multi-exit loop and multi-level exit.
- Multi-level exits appear infrequently, but are extremely concise and execution-time efficient.

2.7 Type Constructor

- A **type constructor** declaration builds a more complex type from the basic types.

constructor	**Java**	**C/C++**
enumeration	**enum** Colour { R, G, B }	**enum** Colour { R, G, B }
pointer		*any-type* *p;
reference	(final) *class-type* r;	*any-type* &r; (C++ only)

array	*any-type* v[ ] = **new** *any-type*[10];
	any-type m[ ][ ] = **new** *any-type*[10][10];
		any-type v[10];
		any-type m[10][10];

structure	**class**	**struct** or **class**

2.7.1 Enumeration

- An **enumeration** is a type defining a set of named literals with only assignment, comparison, and conversion to integer:

```
enum Days {Mon,Tue,Wed,Thu,Fri,Sat,Sun}; // type declaration, implicit numbering
Days day = Sat;                           // variable declaration, initialization
enum {Yes, No} vote = Yes;                // anonymous type and variable declaration
enum Colour {R=0x1, G=0x2, B=0x4} colour; // type/variable declaration, explicit num
colour = B;                               // assignment
```

- Identifiers in an enumeration are called **enumerators**.
- First enumerator is implicitly numbered 0; thereafter, each enumerator is implicitly numbered +1 the previous enumerator.
- Enumerators can be explicitly numbered.

```
enum { A = 3, B, C = A - 5, D = 3, E }; // 3 4 -2 3 4
enum { Red = 'R', Green = 'G', Blue = 'B' }; // 82, 71, 66
```

- Enumeration in C++ denotes a new type; enumeration in C is alias for **int**.

```
day = Sat;          // enumerator must match enumeration
day = 42;           // disallowed C++, allowed C
day = R;            // disallowed C++, allowed C
day = colour;       // disallowed C++, allowed C
```

- Alternative mechanism to create literals is **const** declaration.

```
const short int Mon=0,Tue=1,Wed=2,Thu=3,Fri=4,Sat=5,Sun=6;
short int day = Sat;
days = 42;              // assignment allowed
```

- C/C++ enumerators must be unique in block.

```
enum CarColour { Red, Green, Blue, Black };
enum PhoneColour { Red, Orange, Yellow, Black };
```

Enumerators Red and Black conflict. (Java enumerators are always qualified).

- In C, "**enum**" must also be specified for a declaration:

```
enum Days day = Sat;    // repeat "enum" on variable declaration
```

- Trick to count enumerators (if no explicit numbering):

```
enum Colour { Red, Green, Yellow, Blue, Black, No_Of_Colours };
```

No_Of_Colours is 5, which is the number of enumerators.

- Iterating over enumerators:

```
for ( Colour c = Red; c < No_Of_Colours; c = (Colour)(c + 1) ) {
    cout << c << endl;
}
```

Why is the cast, (Colour), necessary? Is it a conversion or coercion?

2.7.2 Pointer/Reference

- **pointer**/**reference** is a memory address.
- Used to access the value stored in the memory location at the pointer address.
- ***All*** variables have an address in memory, e.g., **int** x = 5, y = 7:

value type		int		int
identifier/value	x	5	y	7
address		100		200

- Two basic addressing operations:

1. **referencing**: obtain address of a variable; unary operator & in C++:

   $100 \leftarrow \&x$
   $200 \leftarrow \&y$

2. **dereferencing**: retrieve value at an address; unary operator $*$ in C++:

   $5 \leftarrow *(100) \leftarrow *(\&x)$
   $7 \leftarrow *(200) \leftarrow *(\&y)$

Note, unary and binary use of operators &/$*$ for reference/dereference and conjunction/multiplication.

- So what does a variable name mean? For x, is it 5 or 100? It depends!
- A variable name is a symbolic name for the pointer to its value, e.g., x means &x, i.e., symbol x is always replaced by pointer value 100.
- What happens in this expression so it can execute?

```
x = x + 1;
```

- First, each variable name is substituted (rewritten) for its pointer value:

$$\begin{array}{lll} (\textbf{\&x}) & \leftarrow (\textbf{\&x}) \ + 1 & \text{where } x \equiv \textbf{\&x} \\ (100) & \leftarrow (100) + 1 & \end{array}$$

Assign into memory location 100 the value 101? Only partially correct!

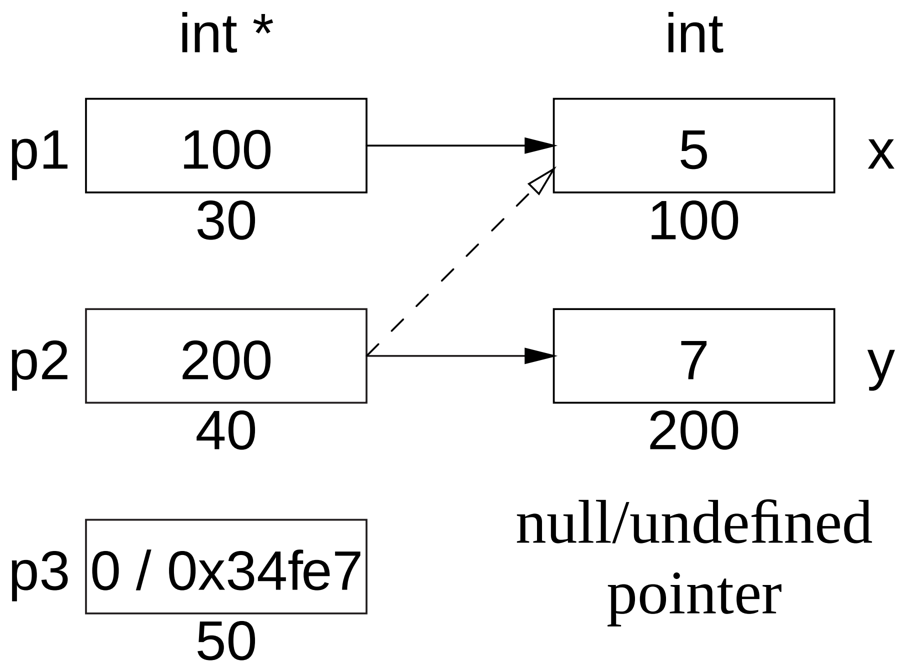
- Second, when a variable name appears on the right-hand side of assignment, it implies the variable's value not its address.

$$\begin{array}{ll} (\textbf{\&x}) & \leftarrow *(\textbf{\&x}) + 1 \\ (100) & \leftarrow *(100) + 1 \\ (100) & \leftarrow 5 + 1 \end{array}$$

Assign into memory location 100 the value 6? Correct!

- Hence, a variable name always means its address, and a variable name is ***also*** implicitly dereferenced on right side of assignment.
- Exception is &x, which just means **&x** not &(**&x**).
- Notice, identifier x (in a particular scope) is a literal (**const**) pointer because it always means the same memory address (e.g., 100).
- Generalize notion of literal variable-name to variable name that can point to more than one memory location (like integer variable versus literal).
- A **pointer variable** is a non-**const** variable that contains different variable addresses ***restricted to a specific type*** in any storage location (i.e., static, stack or heap storage).
  - Java references can only address *object types* on the *heap*.

```
int *p1 = &x, *p2 = &y, *p3 = 0; // or p3 is uninitialized
```



30 ← &p1
40 ← &p2
50 ← &p3
100 ← *&p1
200 ← *&p2
0 ← *&p3
5 ← **&p1
7 ← **&p2
? ← **&p3

- Storage is needed for different address values, so a pointer variable also has an address!
- By convention, no variable is placed at the **null address** (pointer), null in Java, 0 in C/C++.
- Hence, an address value is another variable’s address (**indirection**) or null address or an undefined address when uninitialized.
  - ○ null address often means pointer is unused.
- Multiple pointers may point to the same memory address (p2 = p1, dashed line).

- Dereferencing null/undefined pointer is **undefined** as no variable at address (***but not error***).
- Variable pointed-at is the **target variable** and its value is the **target value**.
  - e.g., x is the target variable of p1 with target value 5.
- Can a pointer variable point to itself?
- Same implicit reference/dereference rules apply for pointer variables.

  ```
  p1 = &x;            // pointer assignment
  (&p1) ← &x          // no rewrite rule for x, why?
  (30)  ← 100
  ```

  Assign to memory location 30 the value 100.

  ```
  p2 = p1;            // pointer assignment
  (&p2) ← *(&p1) // rewrite rules
  (40)   ← *(30)
  (40)   ← 100
  ```

  Assign to memory location 40 the value 100.
- Value assignment requires explicit dereferencing to access values:

```
*p2 = *p1;          // value assignment, y = x
*(&p2) ← *(*(&p1)) // rewrite rules
*(40)  ← *(*(30))
200    ← *(100)
200    ← 5
```

Assign to memory location 200 the value 5.

- Often the target value is used more than the pointer value.

```
*p2 = ((*p1 + *p2) * (*p2 - *p1)) / (*p1 - *p2);
```

Less tedious and error prone to write:

```
p2 = ((p1 + p2) * (p2 - p1)) / (p1 - p2);
```

- C++ reference pointer provides extra implicit dereference to access target value:

```
int &r1 = x, &r2 = y;
r2 = ((r1 + r2) * (r2 - r1)) / (r1 - r2);
```

- **Hence, difference between plain and reference pointer is an extra implicit dereference.**
  - I.e., do you want to write the “*”, or let the compiler write the “*”?

- However, extra implicit dereference generates a problem for pointer assignment.

```
r2 = r1;
*(&r2) ← *(*(&r1)) // value assignment
(&r2)  ← *(&r1)    // not pointer assignment
```

- C++ solves this problem by making reference pointers literals (**const**), like a plain variable.
  - Hence, a reference pointer cannot be assigned after its declaration, so pointer assignment is impossible.
  - As a literal, initialization must occur at declaration, but initializing expression has implicit referencing because address is ***always*** required.

```
int &r1 = &x; // error, unnecessary & before x
```

- Java solves this problem by only using reference pointers, only having pointer assignment, and using a different mechanism for value assignment (clone).
- Is there one more solution?
- Since reference means its target's value, address of a reference means its target's address.

```
int i;
int &r = i;
&r;             *(&r) ⇒ &i not &r
```

- Hence, cannot initialize reference to reference or pointer to reference.

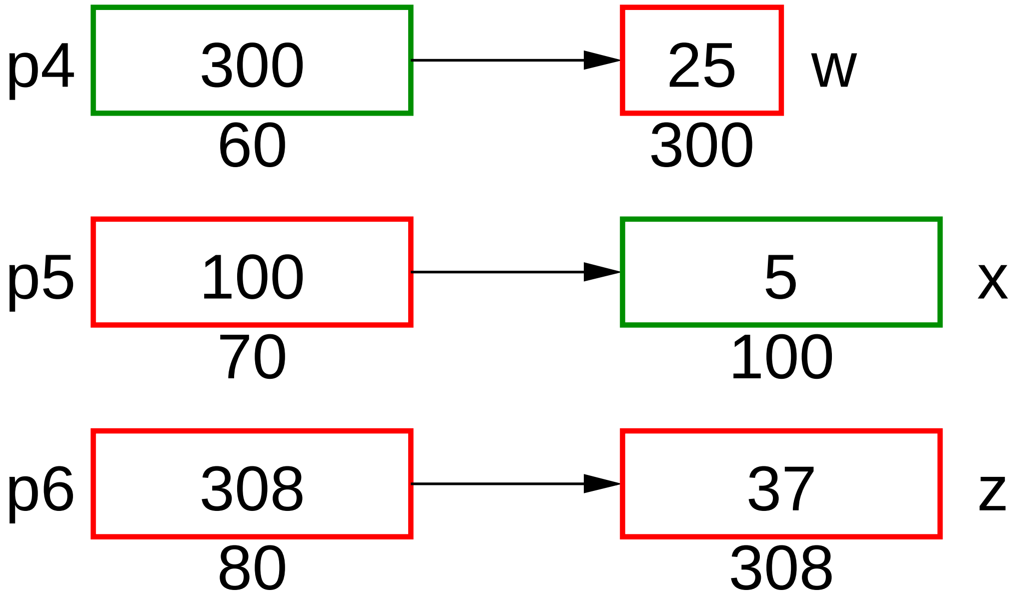
```
int & &rr = r;     // reference to reference, rewritten &r
int &*pr = &r;    // pointer to reference
```

- As well, an array of reference is disallowed (reason unknown).

```
int &ra[3] = { i, i, i };    // array of reference
```

- Type qualifiers can be used to modify pointer types.

```
const short int w = 25;
const short int *p4 = &w;

int * const p5 = &x;
int &p5 = x;

const long int z = 37;
const long int * const p6 = &z;
```



p4 300 60 → 25 300 w

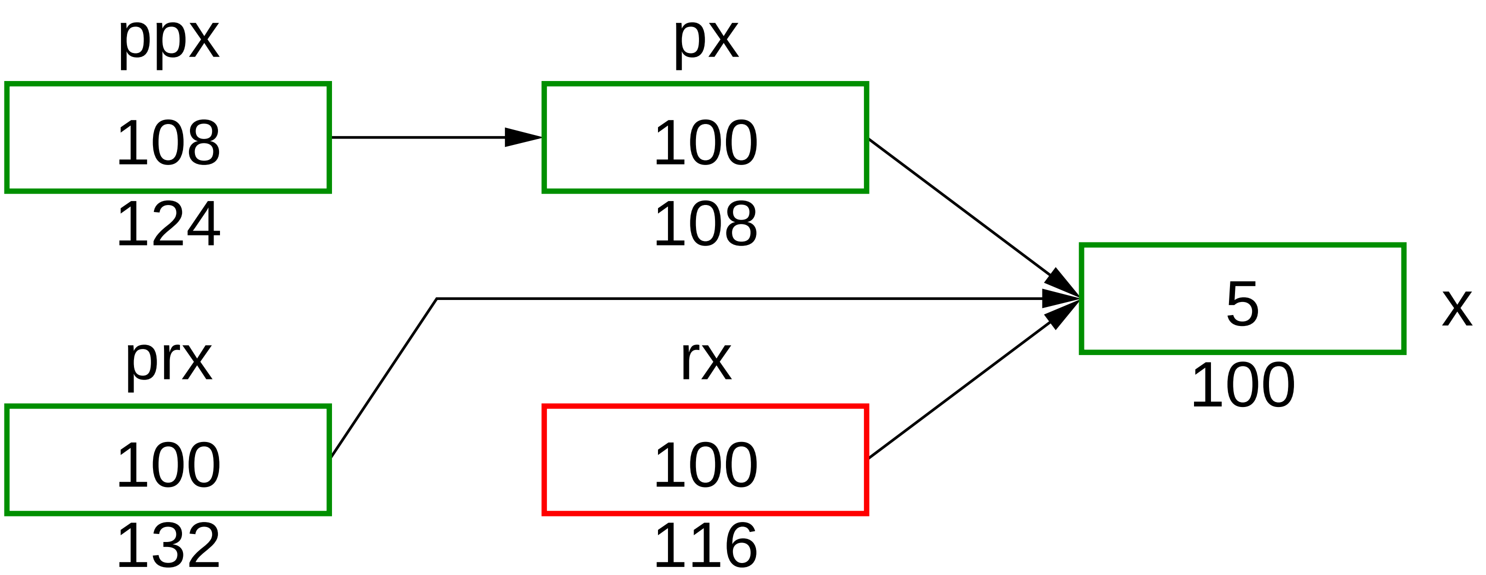
p5 100 70 → 5 100 x

p6 308 80 → 37 308 z

- p4 may point at ***any*** **short int** variable (**const** or non-**const**) and may not change its value.

  Why can p4 point to a non-**const** variable?
- p5 may only point at the **int** variable x and may change the value of x through the pointer.
  - ∗ **const** and & are literal pointers but ∗ **const** has no implicit dereferencing like &.
- p6 may only point at the **long int** variable z and may not change its value.
- Pointer variable has memory address, so it is possible for a pointer to address another pointer or object containing a pointer.

```
int *px = &x, **ppx = &px,
        &rx = x, *prx = &rx;        &prx ← *(&rx)
```



- **Pointer/reference type-constructor is not distributed across the identifier list.**

int * p1, p2;	p1 is a pointer, p2 is an integer	**int** *p1, *p2;
int & rx = i, ry = i;	rx is a reference, ry is an integer	**int** &rx =i, &ry = i;

- C++ idiom for declaring pointers/references is misleading; only works for single versus list of variables.

```
int*    i;              int*    i, k;
double& x = d;          double& x = d, y = d;
```

Gives false impression of distribution across the identifier list.

2.7.3 Aggregates

- Aggregates are a set of homogeneous/heterogeneous values and a mechanism to access the values in the set.

2.7.3.1 Array

- **Array** is a set of **homogeneous values**.

```
int array[10];          // 10 int values
```

- Array type, **int**, is the type of each set value; array **dimension**, 10, is the maximum number of values in the set.
- An array can be structured to have multiple dimensions.

```
int matrix[10][20];     // 10 rows, 20 columns => 200 int values
char cube[5][6][7];     // 5 rows, 6 columns, 7 deep => 210 char values
```

  Common dimension mistake: matrix[10, 20]; means matrix[20] because 10, 20 is a comma expression not a dimension list.

- Number of dimensions is fixed at compile time, but dimension size may be:
  - static (compile time),
  - block dynamic (static in block),

- ○ or dynamic (change at any time).

- C++ only supports a compile-time dimension value; g++ allows a runtime expression.

```
int r, c;
cin >> r >> c;          // input dimensions
int array[r];           // dynamic dimension, g++ only
int matrix[r][c];       // dynamic dimension, g++ only
```

- Array values (elements) are accessed by **subscripts**, “[ ]” (look like dimensions).
- A dimension is subscripted from 0 to dimension-1.

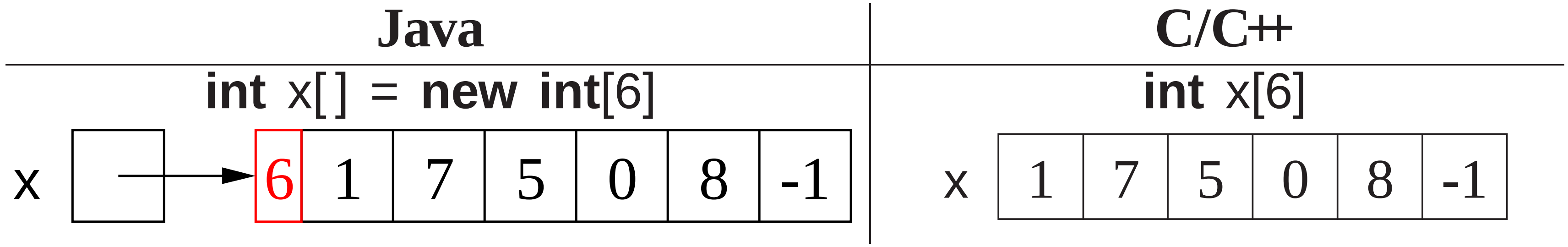
```
array[5] = 3;           // location at column 5
i = matrix[0][2] + 1;  // value at row 0, column 2
c = cube[2][0][3];      // value at row 2, column 0, depth 3
```

Common subscript mistake: `matrix[3, 4]` means `matrix[4]`, 4th row of matrix.

- An array name without a subscript means the first element.

```
array ⇒ array[0]
matrix ⇒ matrix[0][0]
cube ⇒ cube[0][0][0]
```

- C/C++ array is a contiguous set of elements not a reference to the element set as in Java.



Java	**C/C++**
int x[ ] = **new int**[6]	**int** x[6]

- **C/C++ do not store dimension information in the array!**
- Hence, cannot query dimension sizes, ***no subscript checking***, and no array assignment.
- Declaration of a pointer to an array is complex in C/C++ .
- Because no array-size information, the dimension value for an array pointer is unspecified.

```
int i, arr[10];
int *parr = arr;          // think parr[], pointer to array of N ints
```

- However, no dimension information results in the following ambiguity:

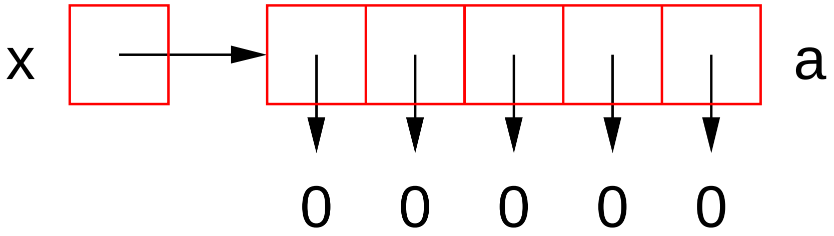
```
int *pvar = &i;           // think pvar[] and i[1]
int *parr = arr;          // think parr[]
```

- ***Variables* pvar *and* parr *have same type but one points at a variable and other an array!***
- Programmer decides if one or many by not using or using subscripting.

```
*pvar                 // one
*parr                 // one, arr[0]
parr[0], parr[3]      // many, many
pvar[3]               // many, but wrong
```

- ASIDE: Practise reading a complex declaration:
  - ○ parenthesize type qualifiers based on priority,
  - ○ read inside parenthesis outwards,
  - ○ start with variable name.
  - ○ end with type name on the left.

```
const long int * const a[5] = {0,0,0,0,0};
const long int * const (&x)[5] = a;
const long int ( * const ( (&x)[5] ) ) = a;
```



x : reference to an array of 5 constant pointers to constant long integers

2.7.3.2 Structure

- **Structure** is a set of **heterogeneous values**, including (nested) structures.

Java	**C/C++**

class Foo {
    int i = 3;
    . . . *// more fields*
}
	struct Foo {
	    int i; *// no initialization*
	    . . . *// more members*
	}; *// semi-colon terminated*

- Components of a structure are called **member**s subdivided into data and routine/function members[1] in C++.
- All members of a structure are accessible (public) by default.
- A structure member cannot be directly initialized (unlike Java) .
- ***A structure is terminated with a semicolon***.
- Structure can be defined and instances declared in a single statement.

```
struct Complex { double re, im; } s; // definition and declaration
```

- In C, "**struct**" must also be specified for a declaration:

```
struct Complex a, b;    // repeat "struct" on variable declaration
```

[1] Java subdivides members into fields (data) and methods (routines).

- Structures with the same type can be assigned but not compared.

```
struct Student {
    struct Name {           // nested structure
        char first[20];     // array
        char last[20];      // array
    } name;
    double age;
    int marks[10];          // array
} s1, s2, *sp1 = &s1;
s1 = s2;                    // allowed
s1 == s2;                   // disallowed, no structure relational operations
```

  Notice, arrays in the structures are copied, but there is no array copy. How?

- Structures ***must*** be compared member by member.
  - comparing bits (e.g., memcmp) fails as alignment padding leaves undefined values between members.
- **Recursive types** (lists, trees) are defined using a self-referential pointer in a structure:

```
struct Node {
        . . .                   // data members
        Node *link;             // pointer to another Node
};
```

- Structure members are accessed by **member selection**, "." and "->".

```
s1.name.first[0] = 'a';      // dot usually with variable
s1.name.last[3] = 'b';
(*sp1).age = 3;
sp1->age = 3;                // -> usually with pointer
(&s1)->marks[5] = 95;
```

- C/C++ are unique for having the priority of selection operator "." incorrectly higher than dereference operator "*".
  - Hence, *p.f executes as *(p.f) instead of (*p).f.
  - -> operator performs a dereference and member selection in the correct order, i.e., p->f is implicitly rewritten as (*p).f.
  - For reference pointers, r.f means (*r).f, so r.f makes more sense than (&r)->f.
- A **bit field** allows direct access to individual bits of memory:

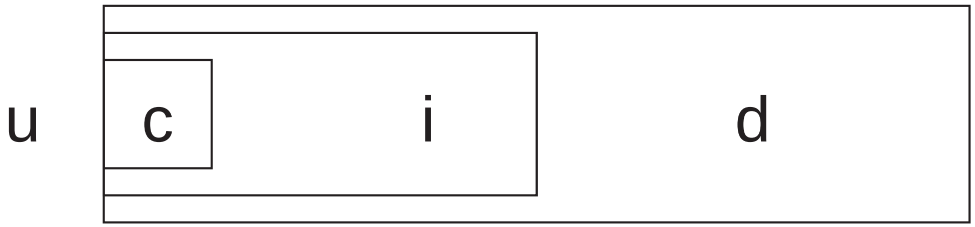
```
struct S {
        int i : 3;      // 3 bits
        int j : 7;      // 7 bits
        int k : 6;      // 6 bits
} s;
s.i = 2;        // 010
s.j = 5;        // 0000101
s.k = 9;        // 001001
```

- A bit field must be an integral type.
- Unfortunately allocation of bit-fields is implementation defined ⇒ not portable (maybe left to right or right to left!).
- Hence, the bit-fields in variable s above must be reversed.
- While it is unfortunate C/C++ bit-fields lack portability, they are the highest-level mechanism to manipulate bit-specific information.

2.7.3.3 Union

- **Union** is a set of **heterogeneous values**, including (nested) structures, **where all members overlay the same storage**.

```
union U {
    char c;
    int i;
    double d;
} u;
```



- Used to access internal representation or save storage by reusing it for different purposes at different times.

```
union U {
    float f;
    struct {
        unsigned int sign : 1;  // may need to be reversed
        unsigned int exp : 8;
        unsigned int frac : 23;
    } s;
    int i;
} u;
u.f = 3.5;          cout << u.f << '\t' << hex << u.i << endl;
u.i = 3;            cout << u.i << '\t' << u.f << endl;
u.f = 3.5e3;        cout << u.s.sign << '\t' << u.s.exp << '\t' << u.s.frac << endl;
u.f = -3.5e-3;      cout << u.s.sign << '\t' << u.s.exp << '\t' << u.s.frac << endl;
```

produces:

```
3.5  40600000
3    4.2039e-45
0    8a   5ac000
1    76   656042
```

- ***Reusing storage is dangerous and can usually be accomplished via other techniques.***

2.7.4 Type Equivalence

- In Java/C/C++, two types are equivalent if they have the same name, called **name equivalence**.

```
struct T1 {                         struct T2 {    // identical structure
    int i, j, k;                        int i, j, k;
    double x, y, z;                     double x, y, z;
}                                   }
T1 t1, t11 = t1;    // allowed, t1, t11 have compatible types
T2 t2 = t1;         // disallowed, t2, t1 have incompatible types
T2 t2 = (T2)t1;     // disallowed, no conversion from type T1 to T2
```

- Types T1 and T2 are **structurally equivalent**, but have different names so they are incompatible, i.e., initialization of variable t2 is disallowed.
- An **alias** is a different name for same type, so alias types are equivalent.

- C/C++ provides **typedef** to create a alias for an existing type:

```
typedef short int shrint1;      // shrint1 => short int
typedef shrint1 shrint2;        // shrint2 => short int
typedef short int shrint3;      // shrint3 => short int
shrint1 s1;      // implicitly rewritten as: short int s1
shrint2 s2;      // implicitly rewritten as: short int s2
shrint3 s3;      // implicitly rewritten as: short int s3
```

- All combinations of assignments are allowed among s1, s2 and s3, because they have the same type name "**short int**".
- Java provides no mechanism to alias types.

2.7.5 Type Nesting

- Type nesting is useful for organizing and controlling visibility for type names:

```
enum Colour { R, G, B, Y, C, M };
struct Foo {
    enum Colour { R, G, B };        // nested type
    struct Bar {                    // nested type
        Colour c[5];                // type defined outside (1 level)
    };
    ::Colour c[5];                  // type defined outside (top level)
    Colour cc;                      // type defined same level
    Bar bars[10];                   // type defined same level
};
Colour c1 = R;                      // type/enum defined same level
Foo::Colour c2 = Foo::R;            // type/enum defined inside
Foo::Bar bar;                       // type defined inside
```

- Variables/types at top nesting-level are accessible with unqualified “::”.
- References to types inside the nested type do not require qualification (like declarations in nested blocks).
- References to types nested inside another type must be qualified with type operator “::”.
- With nested types, Colour (and its enumerators) and Foo in top-level scope; without nested types need:

```
enum Colour { R, G, B, Y, C, M };
enum Colour2 { R2, G2, B2 };      // prevent name clashes
struct Bar {
    Colour2 c[5];
};
struct Foo {
    Colour c[5];
    Colour2 cc;
    Bar bars[10];
};
Colour c1 = R;
Colour2 c2 = R2;
Bar bar;
```

- ***Do not pollute lexical scopes with unnecessary names (name clashes).***

2.7.6 Type-Constructor Literal

enumeration	enumerators
pointer	0 or NULL indicates a null pointer
structure	**struct** { **double** r, i; } c = { 3.0, 2.1 };
array	**int** v[3] = { 1, 2, 3 };

- C/C++ use 0 to initialize pointers (Java null).

- System include-file defines the preprocessor variable NULL as 0.
- Structure and array initialization can occur as part of a declaration.

```
struct { int i; struct { double r, i; } s; } d = { 1, { 3.0, 2.1 } };  // nested structure
int m[2][3] = { {93, 67, 72}, {77, 81, 86} };  // multidimensional array
```

- A nested structure or multidimensional array is created using nested braces.
- Initialization values are placed into a variable starting at beginning of the structure or array.
- Not all the members/elements must be initialized.
  - Uninitialized values are **default initialized**, which means zero-filled for basic types.

    ```
    int b[10];              // uninitialized
    int b[10] = {};         // zero initialized
    ```

- g++ has a cast extension allowing construction of structure and array literals in executable statements not just declarations:

```
void rtn( const int m[2][3] );
struct Complex { double r, i; } c;
rtn( (int [2][3]){ {93, 67, 72}, {77, 81, 86} } );  // g++ only
c = (Complex){ 2.1, 3.4 };  // g++ only
```

- In both cases, a cast indicates the type and structure of the literal.
- String literals can be used as a shorthand array initializer value:

```
char s[6] = "abcde";   rewritten as   char s[6] = { 'a', 'b', 'c', 'd', 'e', '\0' };
```

- It is possible to leave out the first dimension, and its value is inferred from the number of literals in that dimension:

```
    char s[] = "abcde";    // 1st dimension inferred as 6 (Why 6?)
    int  v[] = { 0, 1, 2, 3, 4 } // 1st dimension inferred as 5
    int  m[][3] = { {93, 67, 72}, {77, 81, 86} }; // 1st dimension inferred as 2
```

2.7.7 String

- A **string** is a sequence of characters with specialized operations to manipulate the sequence.
- Strings are provided in C by an array of **char**, string literals, and library facilities.

```
    char s[10];                // string of at most 10 characters
```

- String literal is a double-quoted sequence of characters.

```
"abc"
"a  b  c"
```

- Pointer to a string literal must be **const**.

```
const char *cs = "abc";
```

Why?

- Juxtaposed string literals are concatenated.

```
const char *n1 = "johndoe";
const char *n2 = "john" "doe";  // divide literal for readability
```

- Character escape sequences may appear in string literal.

```
"\\ \" \' \t \n \12 \xa"
```

- Sequence of octal digits is terminated by length (3) or first character not an octal digit; sequence of hex digits is arbitrarily long, but value truncated to fit character type.

```
"\0123\128\xaaa\xaw"
```

How many characters?

- Techniques for preventing escape ambiguity.

  - ○ Octal escape can be written with 3 digits.

    ```
    "\01234"
    ```

  - ○ Octal/hex escape can be written as concatenated strings.

    ```
    "\12" "34" "\xa" "abc" "\x12" "34"
    ```

- Every string literal is implicitly terminated with a character `′\0′`.
  - ○ e.g., string literal `"abc"` is actually 4 characters: `′a′`, `′b′`, `′c′`, and `′\0′`, which occupies 4 bytes of storage.
- Zero value is a **sentinel** used by C-string routines to locate the string end.
- Drawbacks:
  - ○ A string cannot contain a character with the value `′\0′`.
  - ○ To find string length, must linearly search for `′\0′`, which is expensive for long strings.
- Because C-string variable is fixed-sized array, management of variable-sized strings is the programmer's responsibility, requiring complex storage management.
- C++ solves these problems by providing a "string" type using a length member and managing all of the storage for the variable-sized strings.

- Set of powerful operations that perform actions on groups of characters.

Java String	**C char** []	**C++** string
	strcpy, strncpy	=
+, concat	strcat, strncat	+
compareTo	strcmp, strncmp	==, !=, <, <=, >, >=
length	strlen	length
charAt	[]	[]
substring		substr
replace		replace
indexOf, lastIndexOf	strstr	find, rfind
	strcspn	find_first_of, find_last_of
	strspn	find_first_not_of, find_last_not_of
		c_str

- All of the C++ string find members return values of type string::size_type and value string::npos if a search is unsuccessful.

```
string a, b, c;          // declare string variables
cin >> c;                // read white-space delimited sequence of characters
cout << c << endl;       // print string
a = "abc";               // set value, a is “abc”
b = a;                   // copy value, b is “abc”
c = a + b;               // concatenate strings, c is “abcabc”
if ( a == b )            // compare strings, lexigraphical ordering
string::size_type l = c.length(); // string length, l is 6
char ch = c[4];          // subscript, ch is ′b′, zero origin
c[4] = ′x′;              // subscript, c is “abcaxc”, must be character literal
string d = c.substr( 2, 3 ); // extract starting at position 2 (zero origin) for length 3,
c.replace( 2, 1, d);   // replace starting at position 2 for length 1 and insert d, c is “
string::size_type p = c.find( "ax" ); // search for 1st occurrence of string “ax”, p is
p = c.rfind( "ax" );   // search for last occurrence of string “ax”, p is 5
p = c.find_first_of( "aeiou" ); // search for first vowel, p is 0
p = c.find_first_not_of( "aeiou" ); // search for first consonant (not vowel), p is 1
p = c.find_last_of( "aeiou" ); // search for last vowel, p is 5
p = c.find_last_not_of( "aeiou" ); // search for last consonant (not vowel), p is 7
```

- Note different call syntax c.substr(2, 3) versus substr(c, 2, 3).
- Member c_str converts a string to a **char** * pointer (′\0′ terminated).
- Scan string-variable line containing words, and count and print words.

```
unsigned int count = 0;
string line, alpha = "abcdefghijklmnopqrstuvwxyz"
                     "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
. . . // line is initialized with text
line += "\n";                         // add newline as sentinel
for ( ;; ) {                          // scan words off line
    // find position of 1st alphabetic character
    string::size_type posn = line.find_first_of( alpha );
  if ( posn == string::npos ) break;  // any characters left ?
    line = line.substr( posn );       // remove leading whitespace
    // find position of 1st non-alphabetic character
    posn = line.find_first_not_of( alpha );
    // extract word from start of line
    cout << line.substr( 0, posn ) << endl; // print word
    count += 1;                       // count words
    line = line.substr( posn );       // delete word from line
} // for
```

- ***It is seldom necessary to iterate through the characters of a string variable!***
- Contrast C and C++ style strings (note, management of string storage):

```
#include <string>          // C++ string routines
using namespace std;
#include <string.h>        // C string routines

int main() {
    // C++ string
    const string X = "abc", Y = "def", Z = "ghi";
    string S = X + Y + Z;
    // C string
    const char *x = "abc", *y = "def", *z = "ghi";
    char s[strlen(x)+strlen(y)+strlen(z)+1]; // pre-compute worst-case size
    strcpy( s, "" ); // initialize to null string
    strcat( strcat( strcat( s, x ), y ), z );
}
```

Why “+1” for dimension of `s`?

2.8 Modularization

- **Modularization** is the division of a system into interconnecting smaller parts (components), using some systematic basis, and is the foundation of software engineering.
- Medium and large systems must be modularized.

- **Modules** provide a separation of concerns and improve maintainability by enforcing logical boundaries between components.
- These boundaries are provided by **interfaces** defined through various programming-language mechanisms.
- Hence, modularization provides a mechanism to **abstract** data-structures and algorithms through interfaces.
- Modules eliminate duplicated code by **factoring** common code into a single location.
- Essentially any contiguous block of code can be factored into a routine or classand given a name (or vice versa).

2.9 Routine

- Like algebra, arbitrary operations can be define and invoked, e.g., $f(x) = 3x^2 + 2.5x - 17$, where $f(4.5) = 55$.

```
double f( double x ) { return 3.0 * x * x + 2.5 * x - 17.0; }
f( 4.5 );       // returns 55
```

- A **routine** is the simplest module for factoring an abstraction into code.
- Input and output parameters defined a routine’s interface.

C	C++
[**inline** *]* **void** p(OR T f(T1 a *// pass by value*) { *// routine body* *// intermixed decls/stmts* }	*[* **inline** *]* **void** p(OR T f(T1 a, *// pass by value* T2 &b, *// pass by reference* T3 c = 3 *// optional, default value*) { *// routine body* *// intermixed decls/stmts* }

- Routine is either a **procedure** or a **function** based on the return type.
- Procedure does NOT return a value that can be use in an expression, indicated with return type of **void**:

```
void usage() {
    cout << "Usage: " << ... << endl;      // some usage message
    exit( EXIT_FAILURE );      // TERMINATE
}
```

- Procedure can return values via the argument/parameter mechanism.
- Procedure terminates when control runs off the end of its routine body or a **return** statement is executed:

```
void proc() {
        . . . return; . . .
        . . . // run off end => return
}
```

- Function returns a value that can be used in an expression, and hence, ***must*** execute a **return** statement specifying a value:

```
int func() {
        . . . return 3; . . .
        return a + b;
}
```

- A **return** statement can appear anywhere in a routine body, and multiple return statements are possible.
- Routine with no parameters has parameter **void** in C and empty parameter list in C++:

```
. . . rtn( void ) { . . . }        // C: no parameters
. . . rtn() { . . . }              // C++: no parameters
```

  ◦ In C, empty parameters mean no information about the number or types of the parameters is supplied.

- If a routine is qualified with **inline**, the routine is expanded (maybe) at the call site, i.e., unmodularize, to increase speed at the cost of storage (no call).
- Routine cannot be nested in another routine (possible in gcc).
- Java requires all routines to be defined in a **class**.
- Each routine call creates a new block on the stack containing its parameters and local variables, and returning removes the block.
- Variables declared outside of routines are defined in an implicit **static block**.

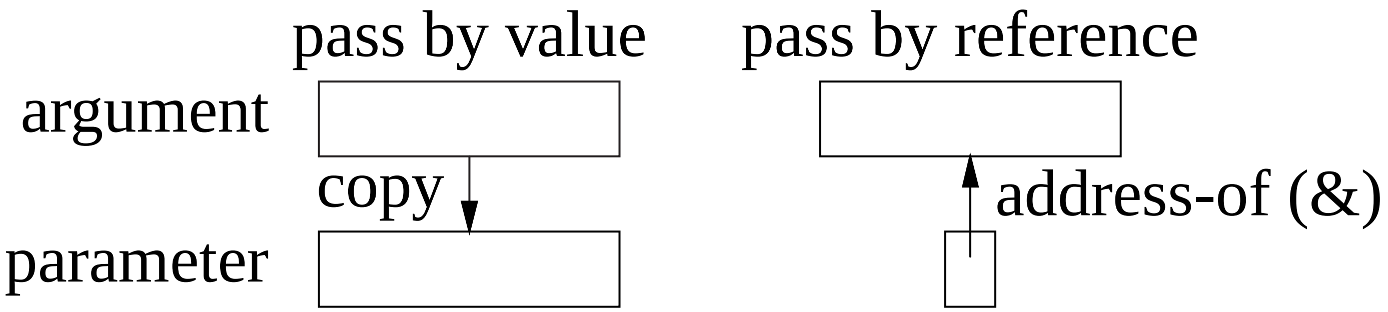
```
int i;                          // static block, global
const double PI = 3.14159;
int rtn( double d )             // static block
{   . . . return 4;                 // create stack block
}                                   // remove stack block
int main()                      // static block
{   int j;                          // create stack block
    {   int k;                      // create stack block
        rtn( 3.0 );
    }                               // remove stack block
}                                   // remove stack block
```

i, PI, rtn, main in static block.

- Static block is a separate memory from the stack and heap and ***is always zero filled***.
- Good practise is to ONLY use static block for literals/variables accessed throughout program.

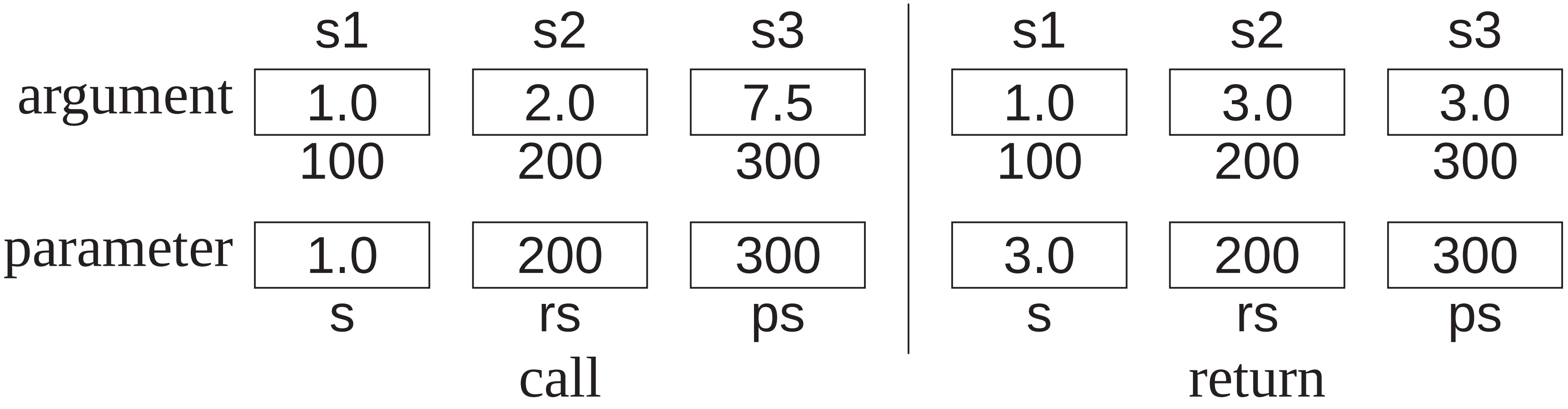
2.9.1 Argument/Parameter Passing

- Modularization without communication is useless; information needs to flow from call to routine and back to call.
- Communication is achieved by passing arguments to parameters and back to arguments or return values.
  - ○ **value parameter**: parameter is initialized by copying argument (input only).
  - ○ **reference parameter**: parameter is a reference to the argument and is initialized to the argument's address (input/output).



- Java/C, parameter passing is by value, i.e., basic types and object references are copied.
- C++, parameter passing is by value or reference depending on the type of the parameter.
- Argument expressions are evaluated ***in any order***.
- For value parameters, each argument-expression result is copied into the corresponding parameter in the routine's block on the stack, ***which may involve an implicit conversion***.
- For reference parameters, each argument-expression result is referenced (address of) and this address is pushed on the stack as the corresponding reference parameter.

```
struct S { double d; };
void r1( S s, S &rs, S * const ps ) {
      s.d = rs.d = ps->d = 3.0;
}
int main() {
     S s1 = {1.0}, s2 = {2.0}, s3 = {7.5};
     r1( s1, s2, &s3 );
     // s1.d = 1.0, s2.d = 3.0, s3.d = 3.0
}
```



	call			return		
	s1	s2	s3	s1	s2	s3
argument	1.0	2.0	7.5	1.0	3.0	3.0
	100	200	300	100	200	300
parameter	1.0	200	300	3.0	200	300
	s	rs	ps	s	rs	ps

- C-style pointer-parameter simulates the reference parameter, but requires & on argument and use of -> with parameter.
- Value passing is most efficient for small values or for large values with high referencing because the values are accessed directly (not through pointer).
- Reference passing is most efficient for large values with low/medium referencing because the values are not duplicated in the routine but accessed via pointers.
- Problem: cannot change a literal or temporary variable via parameter!

```
void r2( int &i, Complex &c, int v[] );
r2( i + j, (Complex){ 1.0, 7.0 }, (int [3]){ 3, 2, 7 } );  // disallowed!
```

- Use type qualifiers to create read-only reference parameters so the corresponding argument is guaranteed not to change:

```
void r2( const int &i, const Complex &c, const int v[] ) {
    i = 3;    // disallowed, read only!
    c.re = 3.0;
    v[0] = 3;
}
r2( i + j, (Complex){ 1.0, 7.0 }, (int [5]){ 3, 2, 7, 9, 0 } );
```

- Provides efficiency of pass by reference for large variables, security of pass by value as argument cannot change, and allows literals and temporary variables as arguments.
- C++ parameter can have a **default value**, which is passed as the argument value if no argument is specified at the call site.

```
void r3( int i, double g, char c = '*', double h = 3.5 ) { . . . }
r3( 1, 2.0, 'b', 9.3 );     // maximum arguments
r3( 1, 2.0, 'b' );          // h defaults to 3.5
r3( 1, 2.0 );               // c defaults to '*', h defaults to 3.5
```

- In a parameter list, once a parameter has a default value, all parameters to the right must have default values.
- In a call, once an argument is omitted for a parameter with a default value, no more arguments can be specified to the right of it.

2.9.2 Array Parameter

- Array copy is unsupported so arrays cannot be passed by value.
- Instead, array argument is a pointer to the array that is copied into the corresponding array parameter (pass by value).
- A formal parameter array declaration can specify the first dimension with a dimension value, [10] (which is ignored), an empty dimension list, [ ], or a pointer, *:

```
double sum( double v[5] );    double sum( double v[] );    double sum( double *v );
double sum( double *m[5] );   double sum( double *m[] );   double sum( double **m );
```

- Good practice uses the middle form as it clearly indicates the variable can be subscripted.
- An actual declaration cannot use [ ]; it must use *:

```
double sum( double v[] ) {   // formal declaration
    double *cv;                  // actual declaration, think cv[]
    cv = v;                      // address assignment
```

- Routine to add up the elements of an arbitrary-sized array or matrix:

```
double sum( int cols, double v[] ) {
    double total = 0.0;
    for ( int c = 0; c < cols; c += 1 )
        total += v[c];
    return total;
}
```

```
double sum( int rows, int cols, double *m[] ) {
    double total = 0.0;
    for ( int r = 0; r < rows; r += 1 )
        for ( int c = 0; c < cols; c += 1 )
            total += m[r][c];
    return total;
}
```

2.10 Input/Output

- Input/Output (I/O) is divided into two kinds:

1. **Formatted I/O** transfers data with implicit conversion of internal values to/from human-readable form.
2. **Unformatted I/O** transfers data without conversion, e.g., internal integer and real-floating values.

2.10.1 Formatted I/O

Java	C	C++

import java.io.*;
import java.util.Scanner;
	#include <stdio.h>	**#include** <iostream>

File, Scanner, PrintStream	FILE	ifstream, ofstream
Scanner in = **new** Scanner(**new** File("f"))	in = fopen("f", "r");	ifstream in("f");
PrintStream out = **new** PrintStream("g")	out = fopen("g", "w")	ofstream out("g")
in.close()	close(in)	scope ends, in.close()
out.close()	close(out)	scope ends, out.close()
nextInt()	fscanf(in, "%d", &i)	in >> T
nextFloat()	fscanf(in, "%f", &f)	
nextByte()	fscanf(in, "%c", &c)	
next()	fscanf(in, "%s", &s)	
hasNext()	feof(in)	in.fail()
hasNextT()	fscanf return value	in.fail()
		in.clear()
skip("regexp")	fscanf(in, "%*[regexp]")	in.ignore(n, c)
out.print(String)	fprintf(out, "%d", i)	out << T
	fprintf(out, "%f", f)	
	fprintf(out, "%c", c)	
	fprintf(out, "%s", s)	

- Both I/O libraries can cascade multiple I/O operations, i.e., input or output multiple values in a single expression.

2.10.1.1 Formats

- Format of input/output values is controlled via **manipulators** defined in **#include** <iomanip>.

oct	integral values in octal
dec	integral values in decimal
hex	integral values in hexadecimal
left / right (default)	values with padding after / before values
boolalpha / noboolalpha (default)	bool values as false/true instead of 0/1
showbase / noshowbase (default)	values with / without prefix 0 for octal & 0x for hex
showpoint / noshowpoint (default)	print decimal point if no fraction
fixed (default) / scientific	float-point values without / with exponent
setprecision(N)	fraction of float-point values in maximum of N columns
setfill(′ch′)	padding character before/after value (default blank)
setw(N)	NEXT VALUE ONLY in minimum of N columns
endl	flush output buffer and start new line (**output only**)
skipws (default) / noskipws	skip whitespace characters (**input only**)

- **Manipulators are not variables for input/output**, but control I/O formatting for all literals/variables after it, continuing to the next I/O expression for a specific stream file.
- **Except manipulator setw, which only applies to the next value in the I/O expression.**
- endl is not the same as ′\n′, as ′\n′ does not flush buffered data.
- During input, skipsw/noskipws toggle between ignoring whitespace between input tokens and reading the whitespace characters (i.e., tokenize versus raw input).

2.10.1.2 Input

- C/C++ formatted input has ***implicit*** character conversion for all basic types and is extensible to user-defined types (Java uses an ***explicit*** Scanner).

Java	**C**	**C++**

import java.io.*;
import java.util.Scanner;
Scanner in =
    new Scanner(**new** File("f"));
PrintStream out =
    new PrintStream("g");
int i, j;
while (in.hasNext()) {
   i = in.nextInt(); j = in.nextInt();
   out.println("i:"+i+" j:"+j);
}
in.close();
out.close();
	#include <stdio.h>
	FILE *in = fopen("f", "r");
	
	FILE *out = fopen("g", "w");
	
	int i, j;
	for (;;) {
	   fscanf(in, "%d%d", &i, &j);
	 if (feof(in)) **break**;
	   fprintf(out,"i:%d j:%d\n",i,j);
	}
	close(in);
	close(out);
		#include <fstream>
		ifstream in("f");
		
		ofstream out("g");
		
		int i, j;
		for (;;) {
		   in >> i >> j;
		 if (in.fail()) **break**;
		   out << "i:" << i
		        <<"j:"<<j<<endl;
		}
		// in/out closed implicitly

- Input values for a stream file are C/C++ literals: 3, 3.5e-1, etc., separated by whitespace.
- Except for characters and character strings, ***which are not in quotes***.
- Type of operand indicates the kind of literal expected in the stream, e.g., an integer operand means an integer literal is expected.
- To read strings containing white spaces use routine getline(stream, string, **char**), which allows different delimiting characters

on input:

```
string s;
getline( cin, s, ′ ′ ); // read characters until ′ ′ => cin >> c
getline( cin, s, ′@′ ); // read characters until ′@′
getline( cin, s, ′\n′ ); // read characters until newline (default)
```

- Input starts reading where the last input left off, and scans lines to obtain necessary number of literals.
- Hence, placement of input values on lines of a file is often arbitrary.
- C/C++ must attempt to read ***before*** end-of-file is set and can be tested.
- **End of file** is the detection of the physical end of a file; **there is no end-of-file character**.
- From a shell, typing <ctrl>-d (C-d), i.e., press <ctrl> and d keys simultaneously, causes the shell to close the current input file marking its physical end.
- In C++, end of file can be explicitly detected in two ways:
  - stream member eof returns **true** if the end of file is reached and **false** otherwise.

  - ○ stream member fail returns **true** for invalid literal OR no literal if end of file is reached, and **false** otherwise.

- Safer to check fail and then check eof.

```
for ( ;; ) {
      cin >> i;
   if ( cin.eof() ) break;           // should use "fail()"
      cout << i << endl;
}
```

- If `"abc"` is entered (invalid integer literal), fail becomes **true** but eof is **false**.
- Generates infinite loop as invalid data is not skipped for subsequent reads.
- Streams also have coercion to **void** *: if fail(), null pointer; otherwise non-null pointer.

```
cout << cin;                 // print fail() status of stream cin
while ( cin >> i ) . . .     // read and check pointer to != 0
```

- When bad data is read, ***stream must be reset and bad data cleared***:

```
#include <iostream>
#include <limits>                                            // numeric_limits
using namespace std;
int main() {
    int n;
    cout << showbase;                                        // prefix hex with 0x
    cin >> hex;                                              // input hex literals
    for ( ;; ) {
        cout << "Enter hexadecimal number: ";
        cin >> n;
        if ( cin.fail() ) {                                  // problem ?
      if ( cin.eof() ) break;                                // eof ?
            cout << "Invalid hexadecimal number" << endl;
            cin.clear();                                     // reset stream failure
            cin.ignore( numeric_limits<int>::max(), '\n' ); // skip until newline
        } else {
            cout << hex << "hex:" << n << dec << " dec:" << n << endl;
        }
    }
    cout << endl;
}
```

- After an unsuccessful read, clear() resets the stream.

- ignore skips *n* characters, e.g., cin.ignore(5) or until a specified character.
- Read in file-names, which may contain spaces, and process each file:

```
#include <fstream>
using namespace std;
int main() {
    ifstream fileNames( "fileNames" );    // requires char * argument
    string fileName;

    for ( ;; ) {                          // process each file
        getline( fileNames, fileName );   // may contain spaces
      if ( fileNames.fail() ) break;      // handle no terminating newlin
        ifstream file( fileName.c_str() ); // access char *
        // read and process file
    }
}
```

- In C, routine feof returns **true** when eof is reached and fscanf returns EOF.
- Parameters in C are always passed by value, so arguments to fscanf must be preceded with & (except arrays) so they can be changed.

2.10.1.3 Output

- Java output style converts values to strings, concatenates strings, and prints final long string:

```
System.out.println( i + " " + j );        // build a string and print it
```

- C/C++ output style has a list of formats and values, and output operation generates strings:

```
cout << i << " " << j << endl;            // print each string as formed
```

- No implicit conversion from the basic types to string in C++ (but one can be constructed).
- **While it is possible to use the Java string-concatenation style in C++, it is incorrect style.**
- Use manipulators to generate specific output formats:

```
#include <iostream>      // cin, cout, cerr
#include <iomanip>       // manipulators
using namespace std;
int i = 7; double r = 2.5; char c = 'z'; const char *s = "abc";
cout << "i:" << setw(2) << i
     << " r:" << fixed << setw(7) << setprecision(2) << r
     << " c:" << c << " s:" << s << endl;

#include <stdio.h>
fprintf( stdout, "i:%2d r:%7.2f c:%c s:%s\n", i, r, c, s );
```

```
i: 7 r:   2.50 c:z s:abc
```

2.10.2 Unformatted I/O

- Expensive to convert from internal (computer) to external (human) forms (bits ⇔ characters).
- When data does not have to be seen by a human, use efficient unformatted I/O so no conversions.
- Uses same mechanisms as formatted I/O to connect variable to file (open/close).

- read and write routines directly transfer bytes from/to a file, where each takes a pointer to the data and number of bytes of data.

```
read( char *data, streamsize num );
write( char *data, streamsize num );
```

- Read/write of types other than characters requires a coercion castor C++ **reinterpret_cast**.

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    ofstream outfile( "myfile" );           // open output file “myfile”
    if ( outfile.fail() ) . . .              // unsuccessful open ?
    double d = 3.0;
    outfile.write( (char *)&d, sizeof( d ) );   // coercion
    outfile.close();                         // close file before attempting read

    ifstream infile( "myfile" );            // open input file “myfile”
    if ( infile.fail() ) . . .               // unsuccessful open ?
    double e;
    infile.read( reinterpret_cast<char *>(&e), sizeof( e ) );   // coercion
    if ( d != e ) . . .                      // problem
    infile.close();
}
```

- Coercion would be unnecessary if buffer type was **void** *.

2.11 Command-line Arguments

- Starting routine main has two overloaded prototypes.

```
int main(); // C: int main( void );
int main( int argc, char *argv[] ); // parameter names may be different
```

- Second form is used to receive command-line arguments from the shell, where the command-line string-tokens are transformed into C/C++ parameters.
- argc is the number of string-tokens on the command line, including the command name.
- ***Java does not include command name, so number of tokens is one less***.
- argv is an array of pointers to C character strings that make up token arguments.

```
% ./a.out -option infile.cc outfile.cc
      0       1        2         3
argc     = 4                        // number of command-line tokens
argv[0]  = ./a.out\0                // not included in Java
argv[1]  = -option\0
argv[2]  = infile.cc\0
argv[3]  = outfile.cc\0
argv[4]  = 0                        // mark end of variable length list
```

- Because shell only has string variables, a shell argument of `"32"` does not mean integer 32, and may have to converted.
- Routine main usually begins by checking `argc` for command-line arguments.

Java	**C/C++**
class Prog {	
public static void main(String[] args) {	**int** main(**int** argc, **char** *argv[]) {
switch (args.length) {	**switch**(argc) {
case 0: . . . *// no args*	**case** 1: . . . *// no args*
break;	**break**;
case 1: . . . args[0] . . . *// 1 arg*	**case** 2: . . . args[1] . . . *// 1 arg*
break;	**break**;
case . . . *// others args*	**case** . . . *// others args*
break;	**break**;
default: . . . *// usage message*	**default**: . . . *// usage message*
System.exit(1);	exit(EXIT_FAILURE);
}	}
. . .	. . .

- Arguments are processed in the range argv[1] through argv[argc - 1] (one greater than Java).
- Process following arguments from shell command line for command:

```
cmd [ size (> 0) [ code (> 0) [ input-file [ output-file ] ] ] ]
```

- Note, dynamic allocation, stringstream (atoi does not indicate errors), and no duplicate code.

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <cstdlib>                               // exit
using namespace std;                             // direct access to std

bool convert( int &val, char *buffer ) {         // convert C string to integer
    std::stringstream ss( buffer );              // connect stream and buffer
    ss >> dec >> val;                            // convert integer from buffer
    return ! ss.fail() &&                        // conversion successful ?
        // characters after conversion all blank ?
        string( buffer ).find_first_not_of( " ", ss.tellg() ) == string::npos;
} // convert

enum { sizeDeflt = 20, codeDeflt = 5 };          // global defaults

void usage( char *argv[] ) {
    cerr << "Usage: " << argv[0] << " [ size (>= 0 : " << sizeDeflt << ") [ cod
     << codeDeflt << ") [ input-file [ output-file ] ] ] ]" << endl;
    exit( EXIT_FAILURE );                        // TERMINATE
} // usage

int main( int argc, char *argv[] ) {
    int size = sizeDeflt, code = codeDeflt;      // default value
    istream *infile = &cin;                      // default value
    ostream *outfile = &cout;                       // default value
    switch ( argc ) {
      case 5:
         outfile = new ofstream( argv[4] );
         if ( outfile->fail() ) usage( argv ); // open failed ?
         // FALL THROUGH
      case 4:
         infile = new ifstream( argv[3] );
         if ( infile->fail() ) usage( argv ); // open failed ?
         // FALL THROUGH
      case 3:
         if ( ! convert( code, argv[2] ) || code < 0 ) usage( argv ) ; // invalid integer
         // FALL THROUGH
      case 2:
         if ( ! convert( size, argv[1] ) || size < 0 ) usage( argv ); // invalid integer ?
         // FALL THROUGH
      case 1:                                       // all defaults
         break;
      default:                                      // wrong number of options
         usage( argv );
    }
    // program body
    if ( infile != &cin ) delete infile;            // close file, do not delete cin!
    if ( outfile != &cout ) delete outfile;         // close file, do not delete cout!
} // main
```

2.12 Preprocessor

- Preprocessor manipulates the text of the program ***before*** compilation.
- **Program you see is not what the compiler sees!**
- A preprocessor statement starts with a # character, followed by a series of tokens separated by whitespace, which is usually a single line and not terminated by punctuation.
- The three most commonly used preprocessor facilities are substitution, file inclusion, and conditional inclusion.

2.12.1 Variables/Substitution

- **#define** statement declares a preprocessor string variable, and its value is all the text after the name up to the end of line.

```
#define Integer int
#define begin {
#define end }
#define gets =
#define set
#define with =
Integer main() begin            // same as: int main() {
    Integer x gets 3, y;        // same as: int x = 3, y;
    x gets 5;                   // same as: x = 5;
    set y with x;               // same as: y = x;
end                             // same as: }
```

- Preprocessor can transform the syntax of C/C++ program (**discouraged**).
- Preprocessor variables can be defined and initialized on the compilation command with option -D.

  ```
  % g++ -DDEBUG="2" -DASSN . . . source-files
  ```

  Initialization value is text after =.
- Same as putting the following **#define**s in a program without changing the program:

  ```
  #define DEBUG 2
  #define ASSN 1
  ```

- **Cannot have both -D and #define for the same variable.**
- Predefined preprocessor-variables exist identifying hardware and software environment, e.g., mcpu is kind of CPU.
- Replace **#define** with **enum** for integral types; otherwise use **const** declarations (Java **final**).

```
enum { arraySize = 100 };            #define arraySize 100
enum { PageSize = 4 * 1024 };        #define PageSize (4 * 1024)
const double PI = 3.14159;           #define PI 3.14159
int array[arraySize], pageSize = PageSize;
double x = PI;
```

  enum uses no storage while **const** declarations might.

- **#define** can declare macros with parameters, which expand during compilation, textually substituting arguments for parameters, e.g.:

```
#define MAX( a, b ) ((a > b) ? a : b)
z = MAX( x, y );        // implicitly rewritten as: z = ((x > y) ? x : y)
```

- Use **inline** routines in C/C++ rather that **#define** macros.

```
inline int MAX( int a, int b ) { return a > b ? a : b; }
```

2.12.2 File Inclusion

- File inclusion copies text from a file into a C/C++ program.
- An included file may contain anything.
- An include file normally imports preprocessor and C/C++ templates/declarations for use in a program.
- All included text goes through every compilation step, i.e., preprocessor, compiler, etc.
- Java implicitly includes by matching class names with file names in CLASSPATH directories, then extracting and including declarations.
- The **#include** statement specifies the file to be included.
- C convention uses suffix “.h” for include files containing C declarations.
- C++ convention drops suffix “.h” for its standard libraries and has special file names for equivalent C files, e.g., cstdio versus stdio.h.

```
#include <stdio.h>          // C style
#include <cstdio>           // C++ style
#include "user.h"
```

- A file name can be enclosed in <> or "".

- <> means preprocessor only looks in the system include directories.
- "" means preprocessor starts looking for the file in the same directory as the file being compiled, then in the system include directories.
- System files limits.h (climit) and stddef.h (cstddef) contain many useful **#define**s.
  - e.g., null pointer literal NULL and min/max values for types (e.g., see /usr/include/limits.h).

2.12.3 Conditional Inclusion

- Preprocessor has an **if** statement, which may be nested, to conditionally add/remove code from a program.
- Conditional **if** uses the same relational and logical operators as C/C++, but operands can only be integer or character values.

```
#define DEBUG 0          // declare and initialize preprocessor variable
...
#if DEBUG == 1           // level 1 debugging
#   include "debug1.h"
...
#elif DEBUG == 2         // level 2 debugging
#   include "debug2.h"
...
#else                    // non-debugging code
...
#endif
```

- By changing value of preprocessor variable DEBUG, different parts of the program are included for compilation.
- To exclude code (comment-out), use 0 conditional as 0 implies false.

```
#if 0
...                                     // code commented out
#endif
```

- It is also possible to check if a preprocessor variable is defined or not defined by using **#ifdef** or **#ifndef**:

```
#ifndef __MYDEFS_H__      // if not defined
#define __MYDEFS_H__ 1    //    make it so
. . .
#endif
```

- Used in an **#include** file to ensure its contents are only expanded once.
- Note difference between checking if a preprocessor variable is defined and checking the value of the variable.
- The former capability does not exist in most programming languages, i.e., checking if a variable is declared before trying to use it.

2.13 Assertions

- **Assertion**s document program assumptions:
  - pre-conditions – things true before a computation (e.g., all values are positive),
  - invariants – things true across the computation (e.g., all values during the computation are positive, because only $+, *, /$ operations),
  - post-conditions – things true after the computation (e.g., all results are positive).

- Assumptions cannot reflect external usage, where there is no control.
  - E.g., at interface points, a routine call can be made with incorrect values.
  - Checking interface parameters is not an assumption about program behaviour, rather about user behaviour.
- Assertions occur ***after*** usage checks when a program has control over its computation.
  - E.g., after checking a “car” is passed to a routine to calculate braking distance, an assumption of correct behaviour is a positive braking distance.
  - Therefore, routine can assert post-condition “braking distance is greater than zero” before returning.
- Macro `assert` tests a boolean expression representing a logical assumption:

```
#include <cassert>
unsigned int stopping_distance( Car car ) {
    if ( car != ... ) exit( EXIT_FAILURE );  // check parameter

    brakes = ... ;
    assert( brakes > 0 );        // pre-condition

    temp = brakes ... ;
    assert( temp > 0 );          // invariant
    temp = ... ;
    assert( temp > 0 );          // invariant

    distance = ... ;
    assert( distance > 0) ); // post-condition
    return distance;
}
```

- If assert fails (false result), it aborts program and prints expression:

```
a.out: test.cc:19: unsigned int stopping_distance(Car):
    Assertion ′distance > 0′ failed.
```

- Use comma expression to add documentation to assertion message.

```
assert( ("Internal error, please report", distance > 0) );
a.out: test.cc:19: unsigned int stopping_distance(Car):
        Assertion (´"Internal error, please report", distance
```

- Assertions in **hot spot**, i.e., point of high execution, can significantly increase program cost.
- Compiling a program with preprocessor variable NDEBUG defined removes all asserts.

```
% g++ -DNDEBUG ... # all asserts removed
```

- Therefore, never put computations needed by a program into an assertion.

```
assert( needed_computation(...) > 0 );   // may not be executed
```

2.14 Debugging

- **Debugging** is the process of determining why a program does not have an intended behaviour.
- Often debugging is associated with fixing a program after a failure.
- However, debugging can be applied to fixing other kinds of problems, like poor performance.

- Before using debugger tools it is important to understand what you are looking for and if you need them.

2.14.1 Debug Print Statements

- An excellent way to debug a program is to *start* by inserting debug print statements (i.e., as the program is written).
- It takes more time, but the alternative is wasting hours trying to figure out what the program is doing.
- The two aspects of a program that you need to know are: where the program is executing and what values it is calculating.
- Debug print statements show the flow of control through a program and print out intermediate values.
- E.g., every routine should have a debug print statement at the beginning and end, as in:

```
int p( ... ) {
    // declarations
    cerr << "Enter p " << parameter variables << endl;
    ...
    cerr << "Exit p " << return value(s) << endl;
    return r;
}
```

- Result is a high-level audit trail of where the program is executing and what values are being passed around.
- Finer resolution requires more debug print statements in important control structures:

```
if ( a > b ) {
    cerr << "a > b" << endl ;                                // debug print
    for ( ... ) {
        cerr << "x=" << x << ", y=" << y << endl;       // debug print
        ...
    }
} else {
    cerr << "a <= b" << endl;                                // debug print
    ...
}
```

- By examining the control paths taken and intermediate values generated, it is possible to determine if the program is executing correctly.
- Unfortunately, debug print statements can generate enormous amounts of output.

  > *It is of the highest importance in the art of detection to be able to recognize out of a number of facts which are incidental and which vital. (Sherlock Holmes, The Reigate Squires)*

- Gradually comment out debug statements as parts of the program begin to work to remove clutter from the output, but do not delete them until the program works.
- When you go for help, your program should contain debug print-statements to indicate some attempted at understanding the problem.
- Use a preprocessor macro to simplify debug prints:

```
#define DPRT( title, expr ) \
  { std::cerr << #title "\t\"" << __PRETTY_FUNCTION__ << "\" " << \
     expr << " in " << __FILE__ << " at line " << __LINE__ << std::e
```

  for printing entry, intermediate, and exit locations and data:

```
#include <iostream>
#include "DPRT.h"
int test( int a, int b ) {
    DPRT( ENTER, "a:" << a << " b:" << b );
    if ( a < b ) {
        DPRT( a < b, "a:" << a << " b:" << b );
    }
    DPRT( , a + b );      // empty title
    DPRT( HERE, "" );   // empty expression
    DPRT( EXIT, a );
    return a;
}
```

which generates debug output:

```
ENTER "int test(int, int)" a:3 b:4 in test.cc at line 14
a < b  "int test(int, int)" a:3 b:4 in test.cc at line 16
    "int test(int, int)" 7 in test.cc at line 18
HERE  "int test(int, int)"  in test.cc at line 19
EXIT   "int test(int, int)" 3 in test.cc at line 20
```

2.14.2 Errors

- Debug print statements do not prevent errors, they simply aid in finding errors.
- What you do about an error depends on the kind of error.
- Errors fall into two basic categories: syntax and semantic.
- **Syntax error** is in the arrangement of the tokens in the programming language.
- These errors correspond to spelling or punctuation errors when writing in a human language.
- Fixing syntax errors is usually straight forward especially if the compiler generates a meaningful error message.
- Always ***read*** the error message carefully and ***check*** the statement in error.

  You see (Watson), but do not observe. (Sherlock Holmes, A Scandal in Bohemia)

- Difficult syntax errors are:
  - missing closing " or */, as the remainder of the program is *swallowed* as part of the character string or comment.

  - missing { or }, especially if the program is properly indented (editors can help here)
  - missing semi-colon at end of structure

- **Semantic error** is incorrect behaviour or logic in the program.
- These errors correspond to incorrect meaning when writing in a human language.
- Semantic errors are harder to find and fix than syntax errors.
- A semantic or execution error message only tells why the program stopped not what caused the error.
- In general, when a program stops with a semantic error, the statement in error is often not the one that must be fixed.
- Must work backwards from the error to determine the cause of the problem.

  > *In solving a problem of this sort, the grand thing is to be able to reason backwards. That is a very useful accomplishment, and a very easy one, but people do not practise it much. In the everyday affairs of life it is more useful to reason forward, and so the other comes to be neglected. (Sherlock Holmes, A Study in Scarlet)*

- Reason from the particular (error symptoms) to the general (error cause).

- ○ locate pertinent data : categorize as correct or incorrect
- ○ look for contradictions
- ○ list possible causes
- ○ devise a hypothesis for the cause of the problem
- ○ use data to find contradictions to eliminate hypotheses
- ○ refine any remaining hypotheses
- ○ prove hypothesis is consistent with both correct and incorrect results, and accounts for all errors

- E.g., an infinite loop with nothing wrong with the loop.

```
i = 10;
while ( i != 5 ) {
    . . .
    i += 2;
}
```

The initialization is wrong.

- Difficult semantic errors are:
  - ○ uninitialized variable
  - ○ invalid subscript or pointer value
  - ○ off-by-one error

- Finally, if a statement appears not to be working properly, but looks correct, check the syntax.

  ```
  if ( a = b ) {
      cerr << "a == b" << endl;
  }
  ```

  When you have eliminated the impossible whatever remains, however improbable must be the truth. (Sherlock Holmes, Sign of Four)

2.15 Dynamic Storage Management

- Java/Scheme are **managed language**s because the language controls all memory management, e.g., **garbage collection** to free dynamically allocated storage.
- C/C++ are **unmanaged language**s because the programmer is involved in memory management, e.g., no garbage collection so dynamic storage must be explicitly freed.
- C++ provides dynamic storage-management operations **new**/**delete** and C provides malloc/free.
- ***Do not mix the two forms in a C++ program.***

Java	C	C++

class Foo { **char** c1, c2; }
Foo r = **new** Foo();
r.c1 = ′X′;
// r garbage collected
	struct Foo { **char** c1, c2; };
	struct Foo *p =
	(**struct** Foo *) _// coerce_
	malloc(_// allocate_
	sizeof(**struct** Foo) _// size_
	);
	p->c1 = ′X′;
	free(p); _// explicit free_
		struct Foo { **char** c1, c2; };
		Foo *p = **new** Foo();
		p->c1 = ′X′;
		delete p; _// explicit free_
		Foo &r = ***new** Foo();
		r.c1 = ′X′;
		delete &r; _// explicit free_

- Allocation has 3 steps:
  1. determine size of allocation,
  2. allocate heap storage of correct size/alignment,
  3. coerce undefined storage to correct type.
- C++ operator **new** performs all 3 steps implicitly; each step is explicit in C.
- Coercion cast is required in C++ for malloc but optional in C.
  - C has implicit cast from **void** * (pointer to anything) to specific pointer (***dangerous!***).
  - Good practise in C is to use a cast so compiler can verify type usage after allocation.

- Parenthesis after the type name in the **new** operation are optional.
- For reference r, why is there a “∗” before **new** and an “&” in the **delete**?
- Storage for dynamic allocation comes from a memory area called the **heap**.
- If heap is full (i.e., no more storage available), malloc returns 0, and **new** generates an error.
- Before storage can be used, it ***must*** be allocated.

```
Foo *p;           // forget to initialize pointer with “new”
p->c1 = ′R′;      // places ′R′ at some random location in memory
```

  Called an uninitialized variable.

- After storage is no longer needed it ***must*** be explicitly deleted.

```
Foo *p = new Foo;
p = new Foo;      // forgot to free previous storage
```

  Called a **memory leak**.

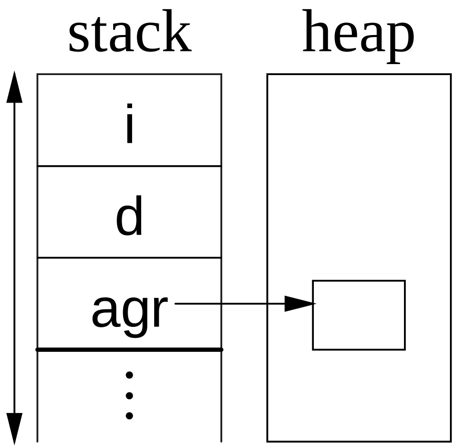
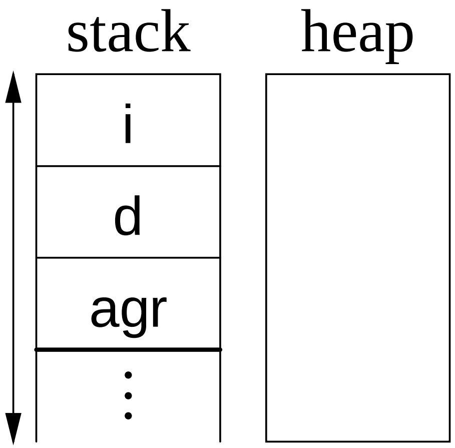
- After storage is deleted, it ***must*** not be used:

```
delete p;
p->c1 = ′R′;          // result of dereference is undefined
```

  Called a **dangling pointer**.

- Unlike Java, C/C++ allow ***all*** types to be dynamically allocated not just object types, e.g., **new int**.
- As well, C/C++ allow ***all*** types to be allocated on the stack, i.e., local variables of a block:

Java	**C++**

{ *// basic & reference*
  int i;
  double d;
  AggrType agr =
    new AggrType();
  . . .
} *// garbage collected*

 	
	{ *// all types*
	  int i;
	  double d;
	  AggrType agr;
	  . . .
	} *// implicit delete*
	

- Stack allocation eliminates explicit storage-management (simpler) and is more efficient than heap allocation — **use it whenever possible.**

```
{ // good, use stack      { // bad, unnecessary dynamic allocation
      int i;                    int *ip = new int;
      . . . // use i            . . . // use *ip
                                delete ip;
}                         }
```

- **Dynamic allocation in C++ should be used only when a variable's storage must outlive the block in which it is allocated**.

```
Type *rtn(...) {
        Type *tp = new Type;      // MUST USE HEAP
        ...                       // initialize/compute using tp
        return tp;                // storage outlives block
}                                 // tp deleted later
```

- Declaration of a pointer to an array is complex in C/C++ .
- Because no array-size information, no dimension for an array pointer.

```
int *parr = new int[10];          // think parr[ ], pointer to array of 10 ints
```

- No dimension information results in the following ambiguity:

```
int *pvar = new int;              // basic "new"
int *parr = new int[10];          // parr[ ], array "new"
```

- Variables pvar and parr have the same type but one is allocated with the basic **new** and the other with the array **new**.
- Special syntax ***must*** be used to call the corresponding deletion operation for a variable or an array (any dimensions):

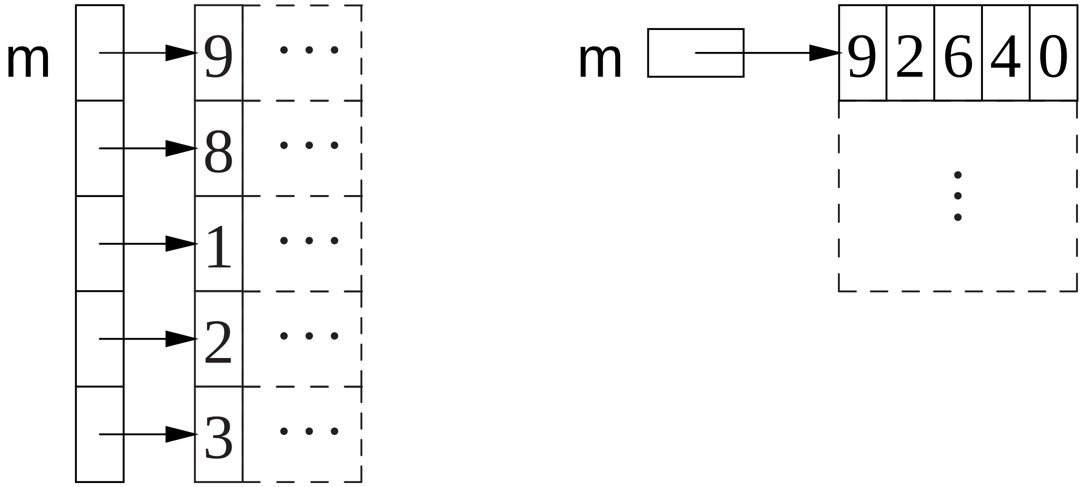
```
delete pvar;        // basic delete : single element
delete [] parr;     // array delete : multiple elements (any dimension)
```

- If basic **delete** is used on an array, only the first element is freed (memory leak).
- If array **delete** is used on a variable, storage after the variable is also freed (often failure).
- **Never do this:**

```
delete [] parr, pvar; // => (delete [] parr), pvar;
```

which is an incorrect use of a comma expression; pvar is not deleted.

- Declaration of a pointer to a matrix is complex in C/C++, e.g., **int** *m[5] could mean:



- Left: array of 5 pointers to an array of unknown number of integers.
- Right: pointer to matrix of unknown number of rows with 5 columns of integers.
- Dimension is higher priority so declaration is interpreted as **int** (*(m[5])) (left).
- Right example cannot be generalized to a dynamically-sized matrix.

```
int R = 5, C = 4;                    // 5 rows, 4 columns
int (*m)[R] = new int[R][C];         // C must be literal, e.g, 4
```

  Compiler must know the stride (number of columns) to compute row.

- Left example can be generalized to a dynamically-sized matrix.

```
int main() {
    int R = 5, C = 4;                // cin >> R >> C;
    int *m[R];                       // R rows
    for ( int r = 0; r < R; r += 1 ) {
        m[r] = new int[C];           // C columns per row
        for ( int c = 0; c < C; c += 1 ) {
            m[r][c] = r + c;         // initialize matrix
        }
    }
    for ( int r = 0; r < R; r += 1 ) { // print matrix
        for ( int c = 0; c < C; c += 1 ) {
            cout << m[r][c] << ", ";
        }
        cout << endl;
    }
    for ( int r = 0; r < R; r += 1 ) {
        delete [] m[r];                 // delete each row
    }
}                                       // implicitly delete array “m”
```

2.16 Overloading

- **Overloading** occurs when a name has multiple meanings in the same context.
- Most languages have overloading, e.g., most built-in operators are overloaded on both integral and real-floating operands, i.e., + operator is different for `1 + 2` than for `1.0 + 2.0`.
- Overloading requires disambiguating among identical names based on some criteria.
- Normal criterion is type information.

- In general, overloading is done on operations not variables:

```
int i;          // disallowed : variable overloading
double i;
void r( int ) { . . . }  // allowed : routine overloading
void r( double ) { . . . }
```

- ***Power of overloading occurs when a variable's type is changed: operations on the variable are implicitly reselected for the variable's new type.***
- E.g., after changing a variable's type from **int** to **double**, all operations implicitly change from integral to real-floating.
- Number and ***unique*** parameter types ***but not the return type*** are used to select among a name's different meanings:

```
int r( int i, int j ) { . . . }  // overload name r three different ways
int r( double x, double y ) { . . . }
int r( int k ) { . . . }
r( 1, 2 );          // invoke 1st r based on integer arguments
r( 1.0, 2.0 );      // invoke 2nd r based on double arguments
r( 3 );             // invoke 3rd r based on number of arguments
```

Subtle cases:

```
int i;  unsigned int ui;  long int li;
void r( int i ) { . . . }  // overload name r three different ways
void r( unsigned int i ) { . . . }
void r( long int i ) { . . . }
r( i );                    // int
r( ui );                   // unsigned int
r( li );                   // long int
```

- Parameter types with qualifiers other than **short**/**long**/**signed**/**unsigned** or reference with same base type are not unique:

```
int r( int i ) {. . .}                   // rewritten: int r( signed int )
int r( signed int i ) {. . .}            // disallowed : redefinition
int r( const int i ) {. . .}             // disallowed : redefinition
int r( int &i ) {. . .}                  // disallowed : ambiguous
int r( const int &i ) {. . .}            // disallowed : ambiguous
r( i );       // all routines look the same
```

- Implicit conversions between arguments and parameters can cause ambiguities:

```
r( 1, 2.0 ); // ambiguous, convert either argument to integer or double
```

- Use explicit cast to disambiguate:

```
r( 1, (int)2.0 )          // 1st r
r( (double)1, 2.0 )       // 2nd r
```

- Overload/conversion confusion: I/O operator << is overloaded with **char** * to print a C string and **void** * to print pointers.

```
char c;  int i;
cout << &c << " " << &i << endl;  // print address of variables
```

type of &c is char *, so printed as C string, which is undefined; type of &i is **int** *, which is converted to **void** *, so printed as an address.

- Fix using coercion.

```
cout << (void *)&c << " " << &i << endl;  // print address of variables
```

- Overlap between overloading and default arguments for parameters with same type:

Overloading	Default Argument

int r(**int** i, **int** j) { . . . }
int r(**int** i) { **int** j = 2; . . . }
r(3); *// 2nd r*
	int r(**int** i, **int** j = 2) { . . . }
	
	r(3); *// default argument of 2*

If the overloaded routine bodies are essentially the same, use a default argument, otherwise use overloaded routines.

2.17 Routine Pointer

- The flexibility and expressiveness of a routine comes from the argument/parameter mechanism, which generalizes a routine across any argument variables of matching type.
- However, the code within the routine is the same for all data in these variables.
- To generalize a routine further, code can be passed as an argument, which is executed within the routine body.
- Most programming languages allow a routine pointer for further generalization and reuse. (Java does not as its routines only appear in a class.)
- As for data parameters, routine pointers are specified with a type (return type, and number and types of parameters), and any routine matching this type can be passed as an argument, e.g.:

```
int f( int v, int (*p)( int ) ) { return p( v * 2 ) + 2; }
int g( int i ) { return i - 1; }
int h( int i ) { return i / 2; }
cout << f( 4, g ) << endl;      // pass routines g and h as arguments
cout << f( 4, h ) << endl;
```

- Routine f is generalized to accept any routine argument of the form: returns an **int** and takes an **int** parameter.
- Within the body of f, the parameter p is called with an appropriate **int** argument, and the result of calling p is further modified before it is returned.
- A routine pointer is passed as a constant reference in virtually all programming languages; in general, it makes no sense to change or copy routine code, like copying a data value.
- C/C++ require the programmer to explicitly specify the reference via a pointer, while other languages implicitly create a reference.
- Two common uses of routine parameters are fix-up and call-back routines.
- A **fix-up routine** is passed to another routine and called if an unusual situation is encountered during a computation.
- E.g., a matrix is not invertible if its determinant is 0 (singular).
- Rather than halt the program for a singular matrix, invert routine calls a user supplied fix-up routine to possibly recover and continue with a correction (e.g., modify the matrix):

```
int singularDefault( int matrix[ ][10], int rows, int cols ) { return 0; }
int invert( int matrix[ ][10], int rows, int cols,
            int (*singular)( int matrix[ ][10], int rows, int cols ) = singularDefault ) {
        . . .
        if ( determinant( matrix, rows, cols ) == 0 ) {
            correction = singular( matrix, rows, cols ); // compute correction
        }
        . . .
}
```

- A fix-up parameter generalizes a routine as the corrective action is specified for each call, and the action can be tailored to a particular usage.
- Giving the fix-up parameter a default value eliminates having to provide a fix-up argument.
- A **call-back routine** is used in event programming.
- When an event occurs, one or more call-back routines are called (triggered) and each one performs an action specific for that event.
- E.g., a graphical user interface has an assortment of interactive “widgets”, such as buttons, sliders and scrollbars.
- When a user manipulates the widget, events are generated representing the new state of the widget, e.g., button down or up.

- A program registers interest in transitions for different widgets by creating and registering a call-back routine.

```
int closedown( /* info about event */ ) {
        // close down because close button press
        // return status of callback action
}
// inform when close button pressed for "widget"
registerCB( widget, closeButton, closedown );
```

- widget maintains list of registered callbacks.
- A widget calls specific call-back routine(s) when the widget changes state, passing new state of the widget to each call-back routine.

2.18 Object

- ***Object***-oriented programming was developed in the mid-1960s by Dahl and Nygaard and first implemented in SIMULA67.
- Object programming is based on structures, used for organizing logically related data:

unorganized	organized

int people_age[30];
bool people_sex[30];
char people_name[30][50];
	struct Person {
	    int age;
	    bool sex;
	    char name[50];
	} people[30];

- Both approaches create an identical amount of information.
- Difference is solely in the information organization (and memory layout).
- Computer does not care as the information and its manipulation is largely the same.
- Structuring is an administrative tool for programmer understanding and convenience.
- Objects extend organizational capabilities of a structure by allowing routine members.
- C++ does not subscribe to the Java notion that everything is either a basic type or an object, i.e., routines can exist without being embedded in a **struct**/**class**.

structure form	object form

struct Complex {
    double re, im;
};
double abs(**const** Complex &This) {
    return sqrt(This.re * This.re +
                  This.im * This.im);
}
Complex x;   *// structure*
abs(x);   *// call abs*
	struct Complex {
	    double re, im;
	    double abs() **const** {
	        return sqrt(re * re +
	                        im * im);
	    }
	};
	Complex x; *// object*
	x.abs(); *// call abs*

- ***An object provides both data and the operations necessary to manipulate that data in one self-contained package.***
- Both approaches use routines as an abstraction mechanism to create an interface to the information in the structure.
- Interface separates usage from implementation at the interface boundary, allowing an object's implementation to change without affecting usage.
- E.g., if programmers do not access Complex's implementation, it can change from Cartesian to polar coordinates and maintain same interface.
- ***Developing good interfaces for objects is important.***

- ○ e.g., mathematical types (like complex) should use value semantics (functional style) versus reference to prevent changing temporary values.

2.18.1 Object Member

- A routine member in a class is constant, and cannot be assigned (e.g., **const** member).
- What is the scope of a routine member?
- Structure creates a scope, and therefore, a routine member can access the structure members, e.g., abs member can refer to members re and im.
- Structure scope is implemented via a T * **const** this parameter, implicitly passed to each routine member (like left example).

```
double abs() const {
    return sqrt( this->re * this->re + this->im * this->im );
}
```

Since implicit parameter "this" is a const pointer, it should be a reference.

- Except for the syntactic differences, the two forms are identical.
- *The use of implicit parameter this, e.g., this->f, is seldom necessary.*

- Member routine declared **const** is read-only, i.e., cannot change member variables.
- Member routines are accessed like other members, using member selection, x.abs, and called with the same form, x.abs().
- No parameter needed because of implicit structure scoping via **this** parameter.
- ***Nesting of object types only allows static not dynamic scoping*** (Java allows dynamic scoping).

```
struct Foo {
    int g;
    int r() { . . . }
    struct Bar {                    // nested object type
        int s() { g = 3; r(); }   // disallowed, dynamic reference
    };                              //    to specific object
} x, y, z;
```

References in s to members g and r in Foo disallowed because must know the **this** for specific Foo object, i.e., which x, y or z.

- Extend type Complex by inserting an arithmetic addition operation:

```
struct Complex {
    . . .
    Complex add( Complex c ) {
        return (Complex){ re + c.re, im + c.im };
    }
};
```

- To sum x and y, write x.add(y), which looks different from normal addition, x + y.
- Because addition is a binary operation, add needs a parameter as well as the implicit context in which it executes.
- Like outside a type, C++ allows overloading members in a type.

2.18.2 Operator Member

- It is possible to use operator symbols for routine names:

```
struct Complex {
    . . .
    Complex operator+( Complex c ) { // rename add member
        return (Complex){ re + c.re, im + c.im };
    }
};
```

- Addition routine is called +, and x and y can be added by x.**operator**+(y) or y.**operator**+(x), which looks slightly better.
- Fortunately, C++ implicitly rewrites x + y as x.**operator**+(y).

```
Complex x = { 3.0, 5.2 }, y = { -9.1, 7.4 };
cout << "x:" << x.re << "+" << x.im << "i" << endl;
cout << "y:" << y.re << "+" << y.im << "i" << endl;
Complex sum = x + y;     // rewritten as  x.operator+( y )
cout << "sum:" << sum.re << "+" << sum.im << "i" << endl;
```

2.18.3 Constructor

- A **constructor** is a special member used to ***implicitly*** perform initialization after object allocation to ensure the object is valid before use.

```
struct Complex {
    double re, im;
    Complex() { re = 0.; im = 0.; } // default constructor
    . . . // other members
};
```

- Constructor name is overloaded with the type name of the structure (normally disallowed).
- Constructor without parameters is the **default constructor**, for initializing a new object to a default value.

```
                                                Complex x;  x.Complex();
Complex x;                    implicitly        Complex *y = new Complex;
Complex *y = new Complex;     rewritten as                   y->Complex();
```

- Unlike Java, C++ does not initialize all object members to default values.
- Constructor is responsible for initializing members ***not initialized via other constructors***, i.e., some members are objects with their own constructors.
- Because a constructor is a routine, arbitrary execution can be performed (e.g., loops, routine calls, etc.) to perform initialization.
- A constructor may have parameters but no return type (not even **void**).
- ***Never put parentheses to invoke default constructor for local declarations.***

```
    Complex x(); // routine prototype, no parameters returning a complex
```

- ***Once a constructor is specified, structure initialization is disallowed:***

```
Complex x = { 3.2 };    // disallowed
Complex y = { 3.2, 4.5 };    // disallowed
```

- Replace using constructor(s) with parameters:

```
struct Complex {
    double re, im;
    Complex( double r = 0.0, double i = 0.0 ) { re = r; im = i; }
    . . .
};
```

Note, use of default values for parameters.

- Unlike Java, constructor argument(s) can be specified *after* a variable for local declarations:

```
Complex x, y(1.0), z(6.1, 7.2);
```

implicitly rewritten as

```
Complex x; x.Complex(0.0, 0.0);
Complex y; y.Complex(1.0, 0.0);
Complex z; z.Complex(6.1, 7.2);
```

- Dynamic allocation is same as Java:

```
Complex *x = new Complex(); // parentheses optional
Complex *y = new Complex(1.0);
Complex *z = new Complex(6.1, 7.2);
```

- Constructor may force dynamic allocation when initializating an array of objects.

```
Complex ac[10];               // complex array initialized to 0.0
for ( int i = 0; i < 10; i += 1 ) {
    ac[i] = (Complex){ i, 2.0 }   // disallowed
}
// MUST USE DYNAMIC ALLOCATION
Complex *ap[10];              // array of complex pointers
for ( int i = 0; i < 10; i += 1 ) {
    ap[i] = new Complex( i, 2.0 );   // allowed
}
```

- ***If only non-default constructors are specified, i.e., ones with parameters, an object cannot be declared without an initialization value:***

```
struct Foo {
    // no default constructor
    Foo( int i ) { . . . }
};
Foo x;   // disallowed!!!
Foo x( 1 );   // allowed
```

- Unlike Java, constructor cannot be called explicitly in another constructor, so constructor reuse is done through a separate member:

Java	**C++**

class Foo {
    int i, j;
 	
    Foo() { **this**(2); } *// explicit call*
    Foo(**int** p) { i = p; j = 1; }
}
	struct Foo {
	    int i, j;
	    void common(**int** p) { i = p; j = 1; }
	    Foo() { common(2); }
	    Foo(**int** p) { common(p); }
	};

2.18.3.1 Literal

- Constructors can be used to create object literals (like g++ type-constructor literals):

```
Complex x, y, z;
x = Complex( 3.2 );               // complex literal value 3.2+0.0i
y = x + Complex(1.3, 7.2);        // complex literal 1.3+7.2i
z = Complex( 2 ); // 2 widened to 2.0, complex literal value 2.0+0.0i
```

- Previous operator + for Complex is changed because type-constructor literals are disallowed for a type with constructors:

```
Complex operator+( Complex c ) {
    return Complex( re + c.re, im + c.im ); // create new complex value
}
```

2.18.3.2 Conversion

- Constructors are implicitly used for conversions:

```
int i;
double d;
Complex x, y;

x = 3.2;                         x = Complex( 3.2 );
y = x + 1.3;  implicitly         y = x.operator+( Complex(1.3) );
y = x + i;    rewritten as       y = x.operator+( Complex( (double)i );
y = x + d;                       y = x.operator+( Complex( d ) );
```

- Allows built-in literals and types to interact with user-defined types.
- Note, two implicit conversions are performed on variable i in x + i: **int** to **double** and then **double** to Complex.
- Can require only explicit conversions with qualifier **explicit** on constructor:

```
struct Complex {
    // turn off implicit conversion
    explicit Complex( double r = 0.0, double i = 0.0 ) { re = r; im = i; }
    . . .
};
```

- Problem: implicit conversion disallowed for commutative binary operators.
- `1.3 + x`, disallowed because it is rewritten as `(1.3).`**operator**`+(x)`, but member **double operator**+(Complex) does not exist in built-in type **double**.
- Solution, move operator + out of the object type and made into a routine, which can also be called in infix form:

```
struct Complex { . . . }; // same as before, except operator + removed
Complex operator+( Complex a, Complex b ) { // 2 parameters
    return Complex( a.re + b.re, a.im + b.im );
}
```

`x + y;`	implicitly rewritten as	**operator**+(x, y)
`1.3 + x;`		**operator**+(Complex(1.3), x)
`x + 1.3;`		**operator**+(x, Complex(1.3))

- Compiler first checks for an appropriate operator in object type, and if found, applies conversions only on the second operand.

- If no appropriate operator in object type, the compiler checks for an appropriate routine (it is ambiguous to have both), and if found, applies applicable conversions to *both* operands.
- In general, commutative binary operators should be written as routines to allow implicit conversion on both operands.
- I/O operators << and >> often overloaded for user types:

```
ostream &operator<<( ostream &os, Complex c ) {
    return os << c.re << "+" << c.im << "i";
}
cout << "x:" << x;   // rewritten as: <<( cout.operator<<("x:"), x )
```

- Standard C++ convention for I/O operators to take and return a stream reference to allow cascading stream operations.
- << operator in object cout is used to first print string value, then overloaded routine << to print the complex variable x.
- Why write as a routine versus a member?

2.18.4 Destructor

- A **destructor** (finalize in Java) is a special member used to perform uninitialization at object deallocation:

Java	**C++**

class Foo {
. . .
finalize() { . . . }
}
	struct Foo {
	. . .
	~Foo() { . . . } *// destructor*
	};

- An object type has one destructor; its name is the character "~" followed by the type name (like a constructor).
- A destructor has no parameters nor return type (not even **void**):
- ***A destructor is only necessary if an object is non-contiguous, i.e., composed of multiple pieces within its environment***, e.g., files, dynamically allocated storage, etc.
- A **contiguous object**, like a Complex object, requires no destructor as it is self-contained.
- A destructor is invoked *before* an object is deallocated, either implicitly at the end of a block or explicitly by a **delete**:

```
{
        Foo x, y( x );
        Foo *z = new Foo;
        ...
        delete z;
        ...

}
```

implicitly
rewritten as

```
{ // allocate local storage
        Foo x, y;  x.Foo(); y.Foo( x );
        Foo *z = new Foo; z->Foo();
        ...
        z->~Foo();  delete z;
        ...
        y.~Foo();  x.~Foo();
} // deallocate local storage
```

- For local variables in a block, destructors ***must be*** called in reverse order to constructors because of dependencies, e.g., y depends on x.
- A destructor is more common in C++ than a finalize in Java due to the lack of garbage collection in C++.
- ***If an object type performs dynamic storage allocation, it is non-contiguous and needs a destructor to free the storage:***

```
struct Foo {
        int *i;     // think int i[ ]
        Foo( int size ) { i = new int[size]; } // dynamic allocation
        ~Foo() { delete [ ] i; }        // must deallocate storage
        ...
};
```

Exception is when the dynamic object is transfered to another object for deallocation.

- C++ destructor is invoked at a deterministic time (block termination or **delete**), ensuring prompt cleanup of the execution environment.
- Java finalize is invoked at a non-deterministic time during garbage collection or *not at all*, so cleanup of the execution environment is unknown.

2.18.5 Copy Constructor / Assignment

- There are multiple contexts where an object is copied.

  1. declaration initialization (ObjType obj2 = obj1)
  2. pass by value (argument to parameter)
  3. return by value (routine to temporary at call site)
  4. assignment (obj2 = obj1)

- Cases 1 to 3 involve a newly allocated object with undefined values.
- Case 4 involves an existing object that may contain previously computed values.
- C++ differentiates between these situations: initialization and assignment.

- Constructor with a **const** reference parameter of class type is used for initialization (declarations/parameters/return), called the **copy constructor**:

```
Complex( const Complex &c ) { . . . }
```

- Declaration initialization:

```
Complex y = x;   implicitly rewritten as   Complex y; y.Complex( x );
```

  - ○ "=" is misleading as copy constructor is called not assignment operator.
  - ○ value on the right-hand side of "=" is argument to copy constructor.

- Parameter/return initialization:

```
Complex rtn( Complex a, Complex b ) { . . . return a; }
Complex x, y;
x = rtn( x, y );          // creates temporary before assignment
```

  - ○ call results in the following implicit action in rtn:

```
Complex rtn( Complex a, Complex b ) {
        a.Complex( x ); b.Complex( y ); // initialize parameters with argumer
        . . .
```

  - ○ return results in a temporary created at the call site to hold the result:

```
                                            Complex temp;
x = rtn(...);    implicitly rewritten as    temp.Complex( rtn(...) );
                                            x = temp;
```

- Assignment routine is used for assignment:

  ```
  Complex &operator=( const Complex &rhs ) { ... }
  ```

  ○ value on the right-hand side of "=" is argument to assignment operator.

  ```
  x = y;   implicitly rewritten as   x.operator=( y );
  ```

  ○ usually most efficient to use reference for parameter and return type.

- If a copy constructor or assignment operator is not defined, an implicit one is generated that does a **memberwise copy** of each subobject.

  ○ basic type, **bitwise copy**
  ○ class type, use class's copy constructor
  ○ array, each element is copied appropriate to the element type

```
struct B {
    B() {}
    B( const B &c ) { cout << "B(&) "; }
    B &operator=( const B &rhs ) { cout << "B= "; }
};
struct D {          // implicit copy and assignment
    int i;          // basic type, bitwise
    B b;            // object type, memberwise
    B ab[5];        // array, element/memberwise
};
int main() {
    D i;            // B′s default constructor
    D d = i;        // D′s default copy-constructor
    d = i;          // D′s default assignment
}
```

outputs the following:

```
B(&) B(&) B(&) B(&) B(&) B(&) B= B= B= B= B= B=
b    ab                       b   ab
```

- Often only a bitwise copy as subobjects have no copy constructor or assignment operator.

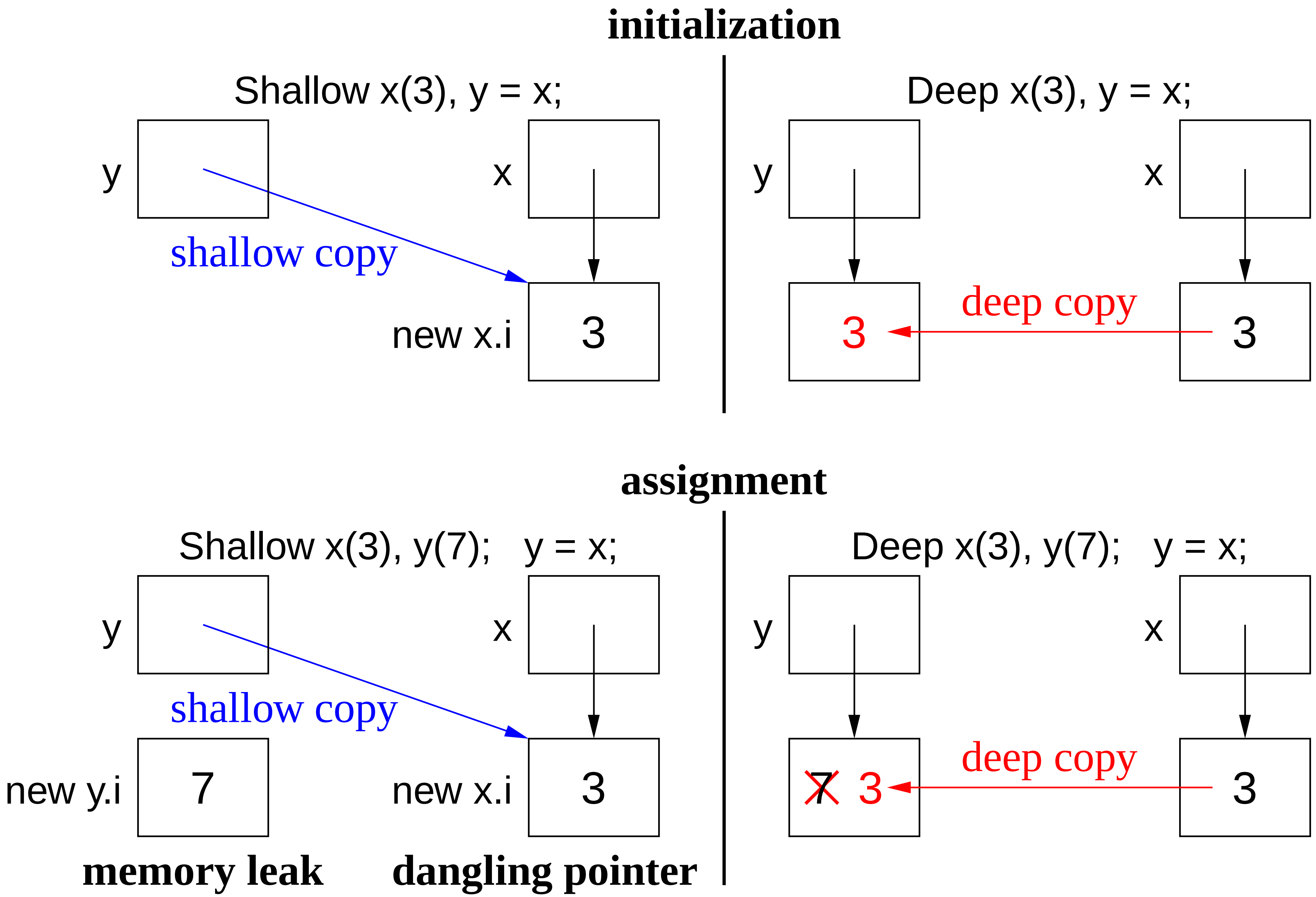
- If D defines a copy-constructor/assignment, it is used rather than that in any subobject.

```
struct D {
    int i;    B b;    B ab[5];
    D( const D &c ) : i( c.i ), b( c.b ), ab( c.ab ) {}
    D &operator=( const D &rhs ) {
        i = rhs.i;  b = rhs.b;
        for ( int i = 0; i < 5; i += 1 ) ab[i] = rhs.ab[i];
        return *this;
    }
};
```

  Must manually copy each subobject (same output as before). **Note array copy!**

- When an object type has pointers, it is often necessary to do a deep copy, i.e, copy the contents of the pointed-to storage rather than the pointers.

```
struct Shallow {
    int *i;
    Shallow( int v ) { i = new int; *i = v; }
    ~Shallow() { delete i; }
};
struct Deep {
    int *i;
    Deep( int v ) { i = new int;  *i = v; }
    ~Deep() { delete i; }
    Deep( Deep &d ) { i = new int; *i = *d.i; }      // copy value
    Deep &operator=( const Deep &rhs ) {
        *i = *rhs.i;  return *this;                  // copy value
    }
};
```



initialization

Shallow x(3), y = x;

y

x

shallow copy

new x.i 3

Deep x(3), y = x;

y

x

3

deep copy

3

assignment

Shallow x(3), y(7); y = x;

y

x

shallow copy

new y.i 7

new x.i 3

memory leak

dangling pointer

Deep x(3), y(7); y = x;

y

x

~~7~~ 3

deep copy

3

- For shallow copy:

  - memory leak occurs on the assignment
  - dangling pointer occurs after `x` or `y` is deallocated; when the other object is deallocated, it reuses this pointer to delete the same storage.

- Deep copy does not change the pointers only the values associated within the pointers.
- Beware **self-assignment** for variable-sized types:

```
struct Varray {                          // variable-sized array
    unsigned int size;
    int *a;
    Varray( unsigned int s ) { size = s; a = new int[size]; }
    . . .  // other members
    Varray &operator=( const Varray &rhs ) {  // deep copy
        delete [] a;                     // delete old storage
        size = rhs.size;                 // set new size
        a = new int[size];               // create storage for new array
        for ( unsigned int i = 0; i < size; i += 1 ) a[i] = rhs.a[i]; // copy v
        return *this;
    }
};
Varray x( 5 ), y( 10 );
x = y;    // works
y = y;   // fails
```

- How can this problem be fixed?
- Which pointer problem is this, and why can it go undetected?
- For deep copy, it is often necessary to define a equality operator (**operator==**) performing a deep compare, i.e., compare values not pointers.

2.18.6 Initialize const / Object Member

- C/C++ **const** members and local objects of a structure must be initialized at declaration:

Ideal (Java-like)	Structure
struct Bar {	**struct** Bar {
Bar(**int** i) {. . .}	Bar(**int** i) {. . .}
// no default constructor	*// no default constructor*
} bar(3);	} bar(3);
struct Foo {	**struct** Foo {
const int i **= 3**;	**const int** i;
Bar * **const** p **= &bar**;	Bar * **const** p;
Bar &rp **= bar**;	Bar &rp;
Bar b**(7)**;	Bar b;
} x;	} x = { 3, &bar, bar**, 7** };

- Left: disallowed because fields cannot be directly initialized.
- Right: disallowed because Bar has a constructor so b must use constructor syntax.
- Try using a constructor:

Constructor/assignment

```
struct Foo {
    const int i;
    Bar * const p;
    Bar &rp;
    Bar b;
    Foo() {
        i = 3;  // after declaration
        p = &bar;
        rp = bar;
        b( 7 ); // not a statement
    }
};
```

Constructor/initialize

```
struct Foo {
    const int i;
    Bar * const p;
    Bar &rp;
    Bar b;
    Foo() : // declaration order
        i( 3 ),
        p( &bar ),
        rp( bar ),
        b( 7 )  {
    }
};
```

- Left: disallowed because **const** has to be initialized at point of declaration.
- Right: special syntax to indicate initialized at point of declaration.
- Ensures **const**/object members are initialized before used in constructor body.
- ***Must be initialized in declaration order to prevent use before initialization.***
- Syntax may also be used to initialize any local members:

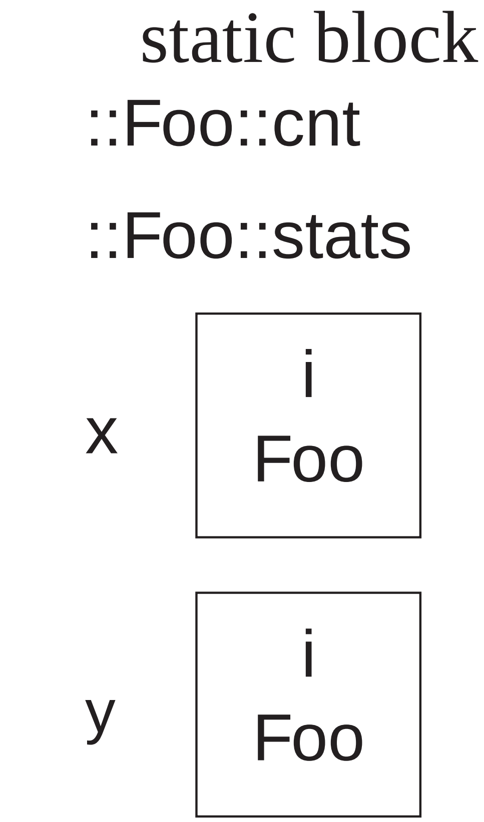
```
struct Foo {
    Complex c;
    int k;
    Foo() : c( 1, 2 ), k( 14 ) {        // initialize c, k
        c = Complex( 1, 2 );            // or assign c, k
        k = 14;
    }
};
```

Initialization may be more efficient versus default constructor and assignment.

2.18.7 Static Member

- Static members create a single instance for object type versus for object instances, e.g., maintain statistics across all objects.
- Members qualified with **static** are declared in the static block not within an object.

```
struct Foo {
    int i;
    static int cnt;
    Foo() {
        cnt += 1;      // allowed
        stats();       // allowed
    }
    static void stats() {
        cout << cnt;   // allowed
        i = 3;      // disallowed
        mem();   // disallowed
    }
} x, y;
int Foo::cnt; // declaration (optional initialization)
```



static block

::Foo::cnt

::Foo::stats

x

i
Foo

y

i
Foo

- Object members mem can reference j and rtn in static block.
- Static member rtn not logically nested in type foo, so it cannot reference members i and mem.
- ***Static class-variables must be declared once (versus defined in the type) in a .cc file.***

2.19 Random Numbers

- **Random numbers** are values generated independently, i.e., new values do not depend on previous values (independent trials).
- E.g., lottery numbers, suit/value of shuffled cards, value of rolled dice, coin flipping.
- While programmers spend most of their time ensuring computed values are not random, random values are useful:
  - gambling, simulation, cryptography, games, etc.
- A **random-number generator** is an algorithm that computes independent values.
- If the algorithm uses deterministic computation, it generates **pseudo random-numbers** versus "true" random numbers, as sequence is predictable.
- All **pseudo random-number generator**s (PRNG) involve some technique that scrambles the bits of a value, e.g., multiplicative recurrence:

```
seed_ = 36969 * (seed_ & 65535) + (seed_ >> 16); // scramble bits
```

- Multiplication of large values adds new least-significant bits and drops most-significant bits.

bits 63-32	bits 31-0
`0`	`3e8e36`
`5f`	`718c25e1`
`ad3e`	`7b5f1dbe`
`bc3b`	`ac69ff19`
`1070f`	`2d258dc6`

- By dropping bits 63-32, bits 31-0 become scrambled after each multiply.
- E.g., class PRNG generates a ***fixed*** sequence of LARGE random values that repeats after 2^{32} values (but might repeat earlier):

```
class PRNG {
        uint32_t  seed_;      // same results on 32/64-bit architectures
   public:
        PRNG( uint32_t  s = 362436069 ) {
                seed_ = s;                                  // set seed
        }
        void seed( uint32_t  s ) {                          // reset seed
                seed_ = s;                                  // set seed
        }
        uint32_t operator()() {                             // [0,UINT_MAX]
                seed_ = 36969 * (seed_ & 65535) + (seed_ >> 16); // scramble
                return seed_;
        }
        uint32_t operator()( uint32_t u ) {                 // [0,u]
                return operator()() % (u + 1);              // call operator()()
        }
        uint32_t operator()( uint32_t l, uint32_t u ) {  // [l,u]
                return operator()( u - l ) + l;             // call operator()( uint32_t )
        }
};
```

- Creating a member with the function-call operator name, (), (**functor**) allows these objects to behave like a routine.

```
PRNG prng;        // often create single generator
prng();           // [0,UINT_MAX]
prng( 5 );        // [0,5]
prng( 5, 10 );    // [5,10]
```

- Large values are scaled using modulus; e.g., generate 10 random number between 5-21:

```
PRNG prng;
for ( int i = 0; i < 10; i += 1 ) {
    cout << prng() % 17 + 5 << endl; // values 0-16 + 5 = 5-21
    cout << prng( 16 ) + 5 << endl;
    cout << prng( 5, 21 ) << endl;
}
```

- By initializing PRNG with a different “seed” each time the program is run, the generated sequence is different:

```
PRNG prng( getpid() );     // process id of program
prng.seed( time() );       // current time
```

- **#include** <cstdlib> provides C random routines srand and rand to set a seed and generate random values, respectively.

```
srand( getpid() );          // seed random genrator
r = rand();                 // obtain next random value
```

2.20 Declaration Before Use

- C/C++ have **Declaration Before Use** (DBU), e.g., a variable declaration must appear before its usage in a block:
- In theory, a compiler could handle some DBU situations:

  ```
  {
      cout << i << endl;      // prints 4 ?
      int i = 4;              // declaration after usage
  }
  ```

  but ambiguous cases make this impractical:

  ```
  int i = 3;
  {
      cout << i << endl;      // which i?
      int i = 4;
      cout << i << endl;
  }
  ```

- C always requires DBU.

- C++ requires DBU in a block and among types but not within a type.
- Java only requires DBU in a block, but not for declarations in or among classes.
- DBU has a fundamental problem specifying **mutually recursive** references:

```
void f() {      // f calls g
        g();    // g is not defined and being used
}
void g() {      // g calls f
        f();    // f is defined and can be used
}
```

Caution: these calls cause infinite recursion as there is no base case.

- Cannot type-check the call to g in f to ensure matching number and type of arguments and the return value is used correctly.
- Interchanging the two routines does not solve the problem.
- A **forward declaration** introduces a routine's type (called a **prototype**/**signature**) before its actual declaration:

```
int f( int i, double ); // routine prototype: parameter names optional
. . .                   // and no routine body
int f( int i, double d ) { // type repeated and checked with prototype
    . . .
}
```

- Prototype parameter names are optional (good documentation).
- Actual routine declaration repeats routine type, which must match prototype.
- Routine prototypes also useful for organizing routines in a source file.

```
int main();             // forward declarations, any order
void g( int i );
void f( int i );
int main() {            // actual declarations, any order
    f( 5 );
    g( 4 );
}
void g( int i ) { . . . }
void f( int i ) { . . . }
```

- E.g., allowing main routine to appear first, and for separate compilation.
- Like Java, C++ does not always require DBU within a type:

Java	**C++**

class T {
    void f() { c = Colour.R; g(); }
    void g() { c = Colour.G; f(); }
    Colour c;
    enum Colour { R, G, B };
};
	void g() {} *// not selected by call in T::f*
	struct T {
	    void f() { c = R; g(); } *// c, R, g not DBU*
	    void g() { c = G; f(); } *// c, G not DBU*
	    enum Colour { R, G, B }; *// type must be DBU*
	    Colour c;
	};

- Unlike Java, C++ requires a forward declaration for mutually-recursive declarations ***among*** types:

Java	**C++**

class T1 {
    T2 t2;
    T1() { t2 = **new** T2(); }
};
class T2 {
    T1 t1;
    T2() { t1 = **new** T1(); }
};
T1 t1 = **new** T1();
	struct T1 {
	    T2 t2; *// DBU failure, T2 size?*
	
	};
	struct T2 {
	    T1 t1;
	
	};
	T1 t1;

Caution: these types cause infinite expansion as there is no base case.

- Java version compiles because t1/t2 are references not objects, and Java can look ahead at T2; C++ version disallowed because DBU on T2 means it does not know the size of T2.
- An object declaration and usage requires the object's size and members so storage can be allocated, initialized, and usages type-checked.
- Solve using Java approach: break definition cycle using a forward declaration and pointer.

Java	**C++**

class T1 {
    T2 t2;
    T1() { t2 = **new** T2(); }
};
class T2 {
    T1 t1;
    T2() { t1 = **new** T1(); }
};
	struct T2; *// forward*
	struct T1 {
	    T2 &t2; *// pointer, break cycle*
	    T1() : t2(*__new__ T2) {} *// DBU failure, size?*
	};
	struct T2 {
	    T1 t1;
	};

- Forward declaration of T2 allows the declaration of variable T1::t2.

- Note, a forward declaration only introduces the name of a type.
- Given just a type name, only pointer/reference declarations to the type are possible, which allocate storage for an address versus an object.
- C++'s solution still does not work as the constructor cannot use type T2.
- Use forward declaration and syntactic trick to move member definition ***after both types are defined***:

```
struct T2;      // forward
struct T1 {
        T2 &t2; // pointer, break cycle
        T1();   // forward declaration
};
struct T2 {
        T1 t1;
};
T1::T1() : t2( *new T2 ) {} // can now see type T2
```

- Use of qualified name T1::T1 allows a member to be logically declared in T1 but physically located later.

2.21 Encapsulation

- **Encapsulation** hides implementation to force abstraction (**access control**).
- Access control applies to types NOT objects, i.e., all objects of the same type have identical levels of encapsulation.
- ***Abstraction and encapsulation are neither essential nor required to develop software.***
- E.g., programmers could follow a convention of not directly accessing the implementation.
- However, relying on programmers to follow conventions is dangerous.
- **Abstract data-type** (ADT) is a user-defined type that practices abstraction and encapsulation.
- Encapsulation is provided by a combination of C and C++ features.
- C features work largely among source files, and are indirectly tied into separate compilation.
- C++ features work both within and among source files.
- C++ provides 3 levels of access control for object types:

Java	**C++**

class Foo {
private ...
...
protected ...
...
public ...
...
};
	struct Foo {
	private: *// within and friends*
	// private members
	protected: *// within, friends, inherited*
	// protected members
	public: *// within, friends, inherited, users*
	// public members
	};

- Java requires encapsulation specification for each member.
- C++ groups members with the same encapsulation, i.e., all members after a label, **private**, **protected** or **public**, have that visibility.
- Visibility labels can occur in any order and multiple times in an object type.
- To enforce abstraction, all implementation members are private, and all interface members are public.
- ***Nevertheless, private and protected members are still visible but cannot be accessed.***

```
struct Complex {
  private:
    double re, im;  // cannot access but still visible
  public:
    // interface routines
};
```

- **struct** has an implicit **public** inserted at beginning, i.e., by default all members are public.
- **class** has an implicit **private** inserted at beginning, i.e., by default all members are private.

```
struct S {        class C {
  // public:        // private:
    int z;            int x;
  private:          protected:
    int x;            int y;
  protected:        public:
    int y;            int z;
};                };
```

- Use encapsulation to preclude object copying by hiding copy constructor and assignment operator:

```
class Foo {
        Foo( const Foo & );          // definitions not required
        Foo &operator=( Foo & );
   public:
        Foo() {...}
        ...
};
void rtn( Foo f ) {...}
Foo x, y;
rtn( x );   // disallowed, no copy constructor for pass by value
x = y;    // disallowed, no assignment operator for assignment
```

- Prevent object forgery (lock, boarding-pass, receipt) or copying that does not make sense (file, database).
- Encapsulation introduces problems when factoring for modularization, e.g., previously accessible data becomes inaccessible.

```
class Complex {
    double re, im;
  public:
    Complex operator+(Complex c);
    . . .
};
ostream &operator<<(ostream &os,
                    Complex c);
```

```
class Cartesian { // implementation type
    double re, im;
};
class Complex {
    Cartesian impl;
  public:
    . . .
};
Complex operator+(Complex a, Complex b);
ostream &operator<<(ostream &os,
                    Complex c);
```

- Implementation is factored into a new type Cartesian, “+” operator is factored into a routine outside and output “<<” operator must be outside.
- Both Complex and “+” operator need to access Cartesian implementation, i.e., re and im.
- Creating get and set interface members for Cartesian provides no advantage over full access.
- C++ provides a mechanism to state that an outside type/routine is allowed access to its implementation, called **friendship** (similar to package visibility in Java).

```
class Complex;   // forward
class Cartesian { // implementation type
    friend Complex operator+( Complex a, Complex b );
    friend ostream &operator<<( ostream &os, Complex c );
    friend class Complex;
    double re, im;
};
class Complex {
    friend Complex operator+( Complex a, Complex b );
    friend ostream &operator<<( ostream &os, Complex c );
    Cartesian impl;
  public:
    . . .
};
Complex operator+( Complex a, Complex b ) {
    return Complex( a.impl.re + b.impl.re, a.impl.im + b.impl.im );
}
ostream &operator<<( ostream &os, Complex c ) {
    return os << c.impl.re << "+" << c.impl.im << "i";
}
```

- Cartesian makes re/im accessible to friends, and Complex makes impl accessible to friends.

- Alternative design is to nest the implementation type in Complex and remove encapsulation (use **struct**).

```
class Complex {
        friend Complex operator+( Complex a, Complex b );
        friend ostream &operator<<( ostream &os, Complex c );
        struct Cartesian { // implementation type
                double re, im;
        } impl;
    public:
        Complex( double r = 0.0, double i = 0.0 ) {
                impl.re = r; impl.im = i;
        }
};
. . .
```

Complex makes Cartesian, re, im and impl accessible to friends.

2.22 System Modelling

- **System modelling** involves describing a complex system in an abstract way to help understand, design and construct the system.
- Modelling is useful at various stages:

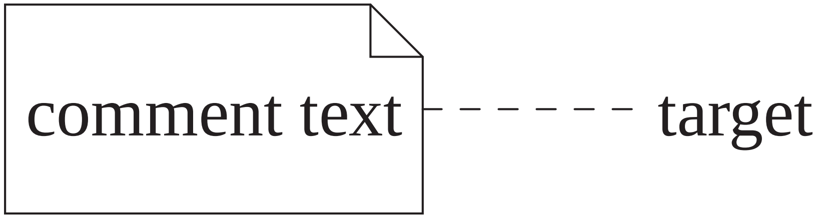
  - analysis : system function, services, requirements (outline for design)
  - design : system parts/structure, interactions, behaviour (outline for programming)
  - programming : converting model into implementation
- Model grows from nothing to sufficient detail to be transformed into a functioning system.
- Model provides high-level documentation of the system for understanding (education) and for making changes in a systematic manner.
- Top-down successive refinement is a foundational mechanism used in system design.
- Multiple design tools (past and present) for supporting system design, most are graphical and all are programming-language independent:
  - flowcharts (1920-1970)
  - pseudo-code
  - Warnier-Orr Diagrams
  - Hierarchy Input Process Output (HIPO)
  - UML
- Design tools can be used in various ways:

  - **sketch** out high-level design or complex parts of a system,
  - **blueprint** the system abstractly with high accuracy,
  - **generate** interfaces/code directly.
- Key advantage is design tool provides a generic, abstract model of a system, which is transformable into different formats.
- Key disadvantage is design tool seldom linked to implementation mechanism so two often differ. **(CODE = TRUTH)**
- Currently, UML is the most popular design tool.

2.22.1 UML

- **Unified Modelling Language** (UML) is a graphical notation for describing and designing software systems, with emphasis on the object-oriented style.
- UML modelling has multiple viewpoints:
  - **class model** : describes static structure of the system for creating objects
  - **object model** : describes dynamic (temporal) structure of system objects
  - **interaction model** : describes the kinds of interactions among objects

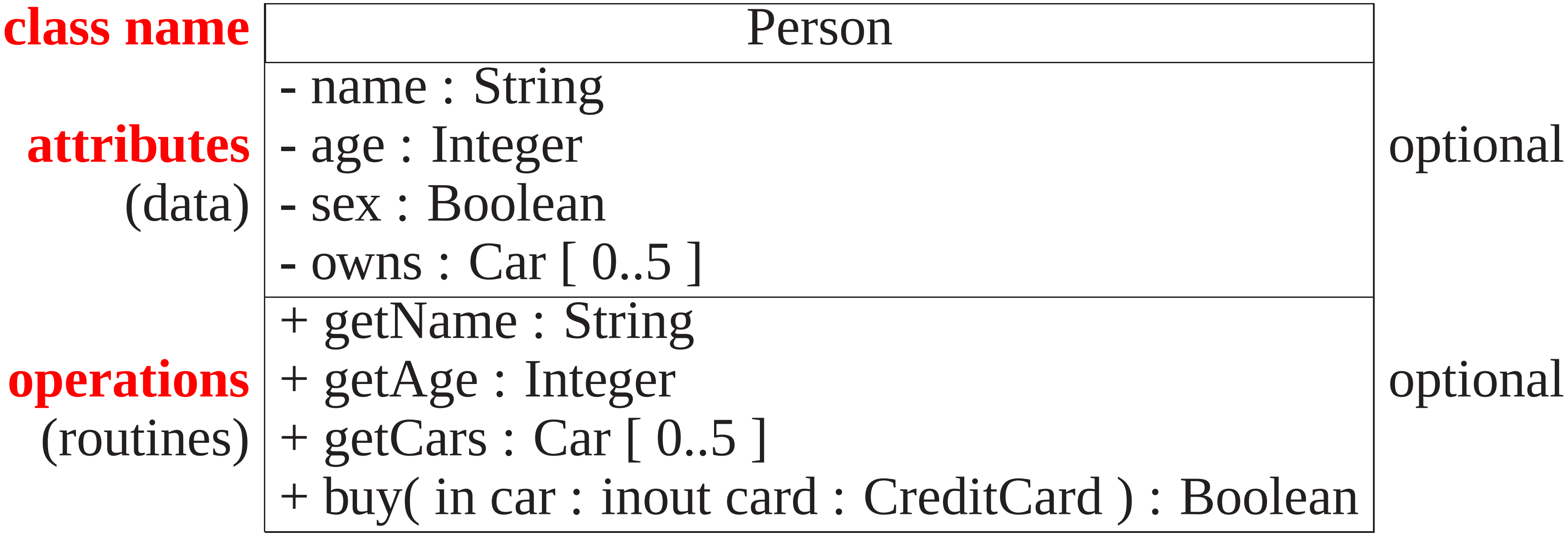
  Focus on class and object modelling.
- Note / comment



comment text

target

- **Classes diagram** defines class-based modelling, where a class is a type for instantiating objects.
- Class has a name, attributes and operations, and may participate in inheritance hierarchies.



class name

Person

attributes (data)

- name : String
- age : Integer
- sex : Boolean
- owns : Car [0..5]

optional

operations (routines)

+ getName : String
+ getAge : Integer
+ getCars : Car [0..5]
+ buy(in car : inout card : CreditCard) : Boolean

optional

- **Attribute** describes a property in a class.

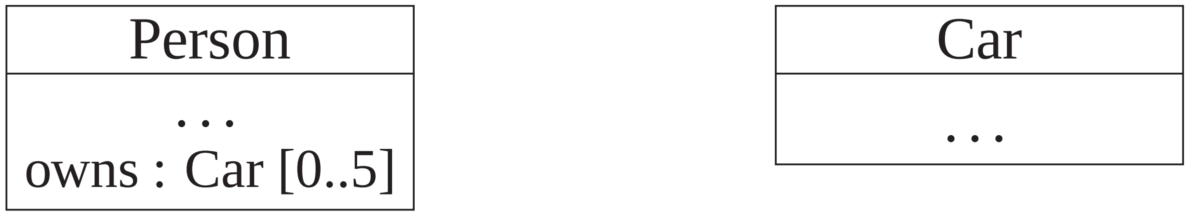
  [visibility] name [“:” [type] [“[” multiplicity “]”] [“=” default]]

- ◦ visibility : access to property
  $+ \Rightarrow$ public, $- \Rightarrow$ private, # $\Rightarrow$ protected, $\sim \Rightarrow$ package
- ◦ name : identifier for property (like field name in structure)
- ◦ type : kind of property
  Boolean, Integer, Float, String, class-name
- ◦ multiplicity : cardinality for instantiation of property
  $0..(N|*)$, from 0 to N or unlimited, N short for $N..N$, $*$ short for $0..*$
  Defaults to 1
- ◦ default : expression that evaluates to default value (or values) for property

- **operation** : action invoked in context of object from the class
  [visibility] name ["(" [parameter-list] ")"] [":" return-type] ["[" multiplicity "]"]
  - ◦ visibility : access to operation
    $+ \Rightarrow$ public, $- \Rightarrow$ private, # $\Rightarrow$ protected, $\sim \Rightarrow$ package
  - ◦ name : identifier for operation (like method name in structure)
  - ◦ parameter-list : input/output types for operation
    [direction] parameter-name ":" type ["[" multiplicity "]"]
    ["=" default] ["{" modifier-list "}"]]

  - ○ direction : direction of parameter data flow
    “in” (default) | “out” | “inout”
  - ○ return-type : output type from operation

- Only specify attributes/operations useful in modelling: no flags, counters, temporaries, constructors, helper routines, etc.
- Attribute with type other than basic type has an **association**.



Person

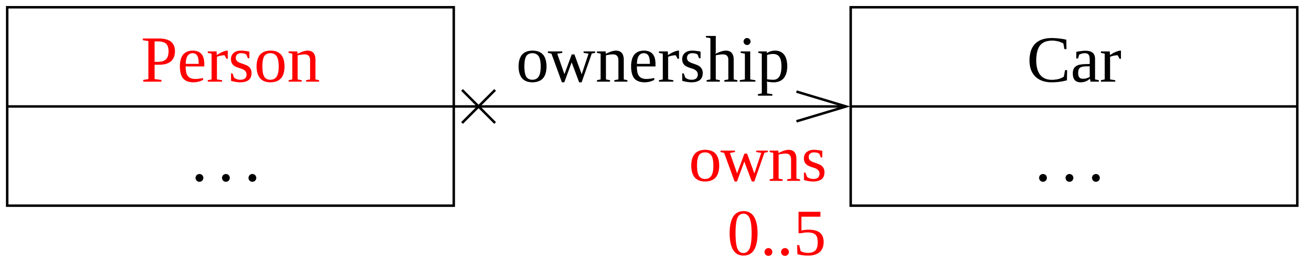
. . .
owns : Car [0..5]

Car

. . .

  - ○ Class Person has attribute owns with multiplicity constraint 0..5 forming unidirectional association with class Car, i.e., person owns (has) 0 to 5 cars.

- Alternatively, association can be represented via a line (possibly named):



Person

. . .

ownership

owns

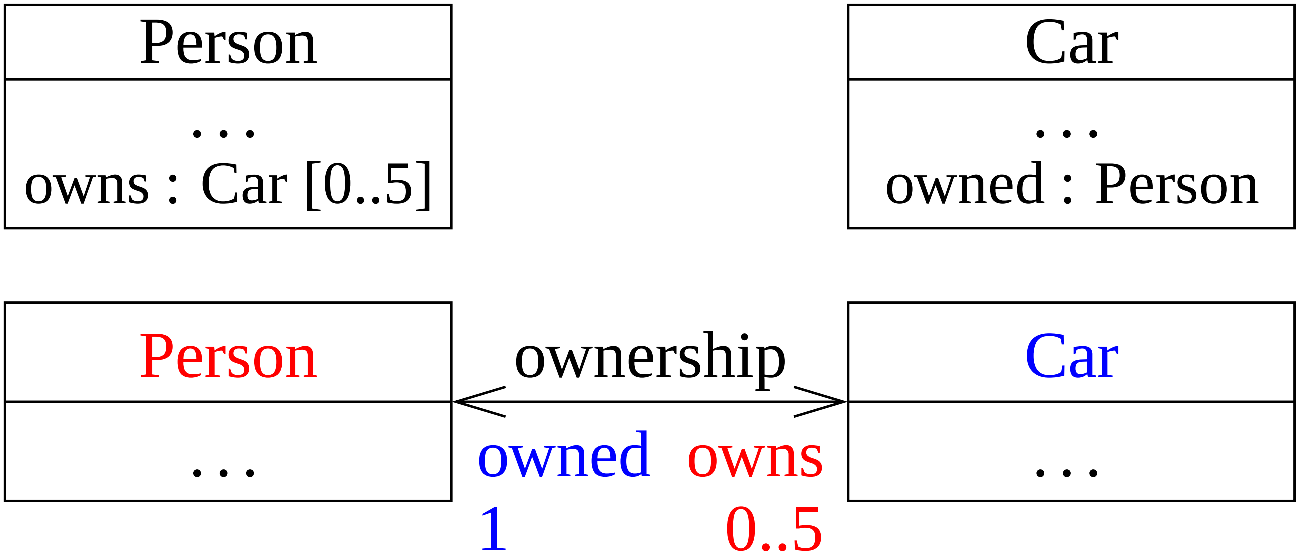
0..5

Car

. . .

- ○ Class Person has attribute owns with multiplicity constraint 0..5 (at target end) forming a unidirectional association with class Car and association is named “ownership”.

- Association can also be bidirectional.



Person

. . .

owns : Car [0..5]

Car

. . .

owned : Person

Person

. . .

ownership

Car

. . .

owned owns

1 0..5

- ○ Association “ownership” also has class Car having attribute owned with multiplicity constraint 1 person, i.e., a car can only be owned by 1 person.

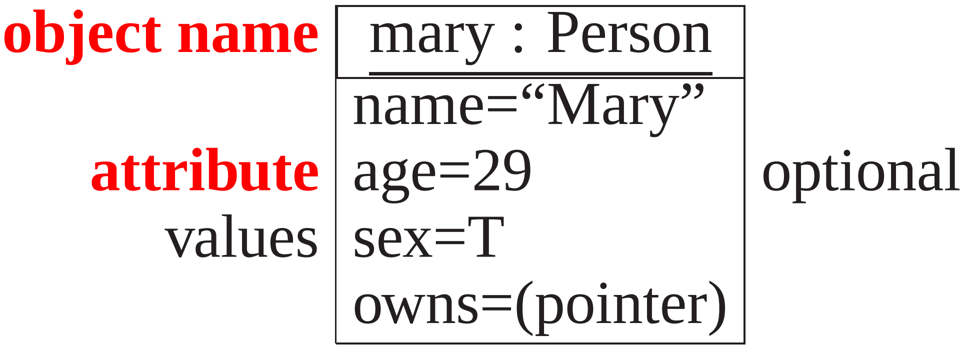
- If UML graph is cluttered with lines, create association in class rather than using a line.
  - ○ E.g., if 20 classes associated with Car, replace 20 lines with attributes in each class.

- Alternatively, multiple lines to same aggregate may be merged into a single segment.
  - Any adornments on that segment apply to all of the aggregation ends.
- $<$ (arrowhead) ⇒ **navigable**:
  - instances of association can be accessed efficiently at the association end (arrowhead) (car is accessible from person)
  - opposite association end "owns" the association's implementation (person has a car)
- X ⇒ not navigable.
- Adornments options:
  - show all arrows and Xs (completely explicit)
  - suppress all arrows and Xs ⇒ no inference about navigation

    often convenient to suppress some of the arrows/Xs and only show special cases
  - show only unidirectional association arrows, and suppress bidirectional associations

    ⇒ two-way navigability cannot be distinguished from no navigation at all, but latter case occurs rarely in practice.

- Navigability may be implemented in a number of ways:
  - ○ pointer/reference from one object to another
  - ○ elements in arrays
- **Object diagram** : is a snaphot of class instances at one moment during execution.
- Object can specify values of class : “name : class-type” (underlined), attribute values.



object name

mary : Person

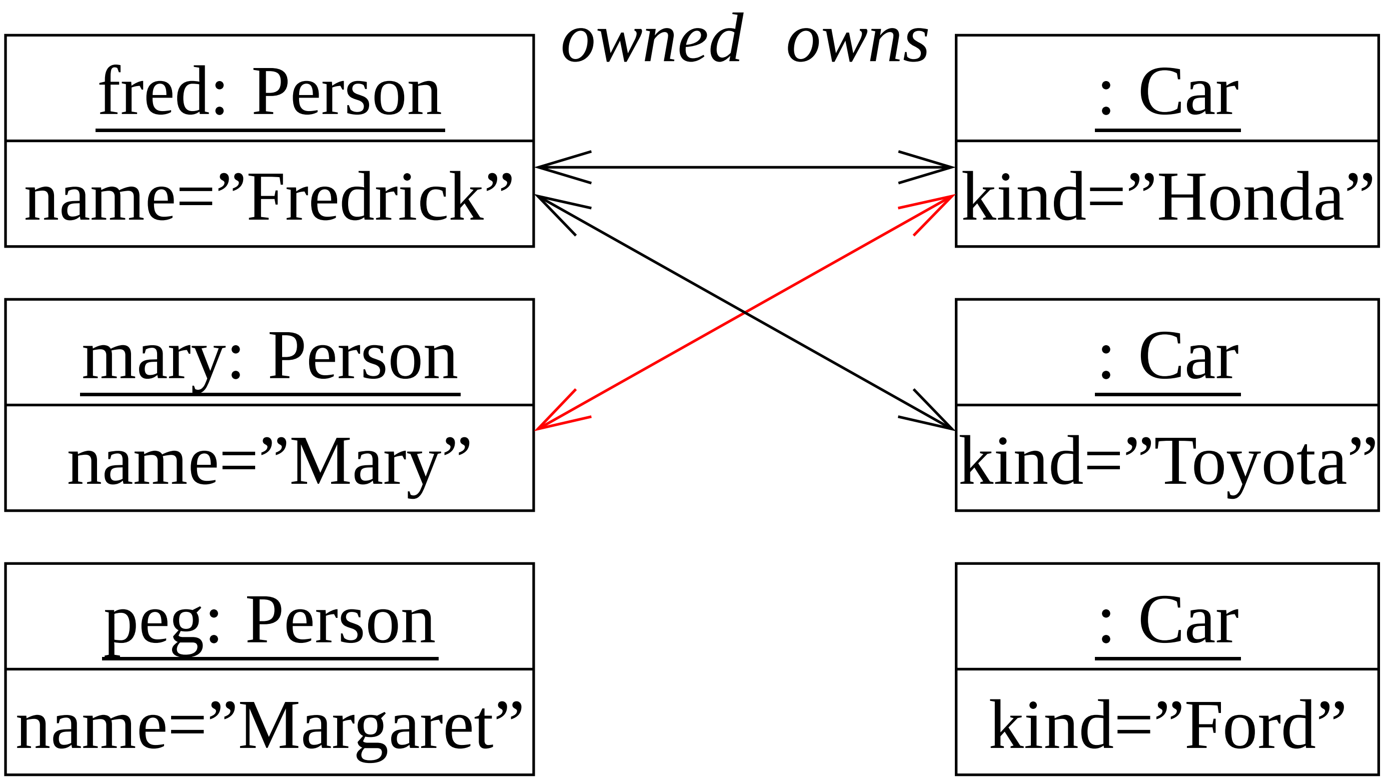
name=“Mary”
age=29
sex=T
owns=(pointer)

attribute values

optional

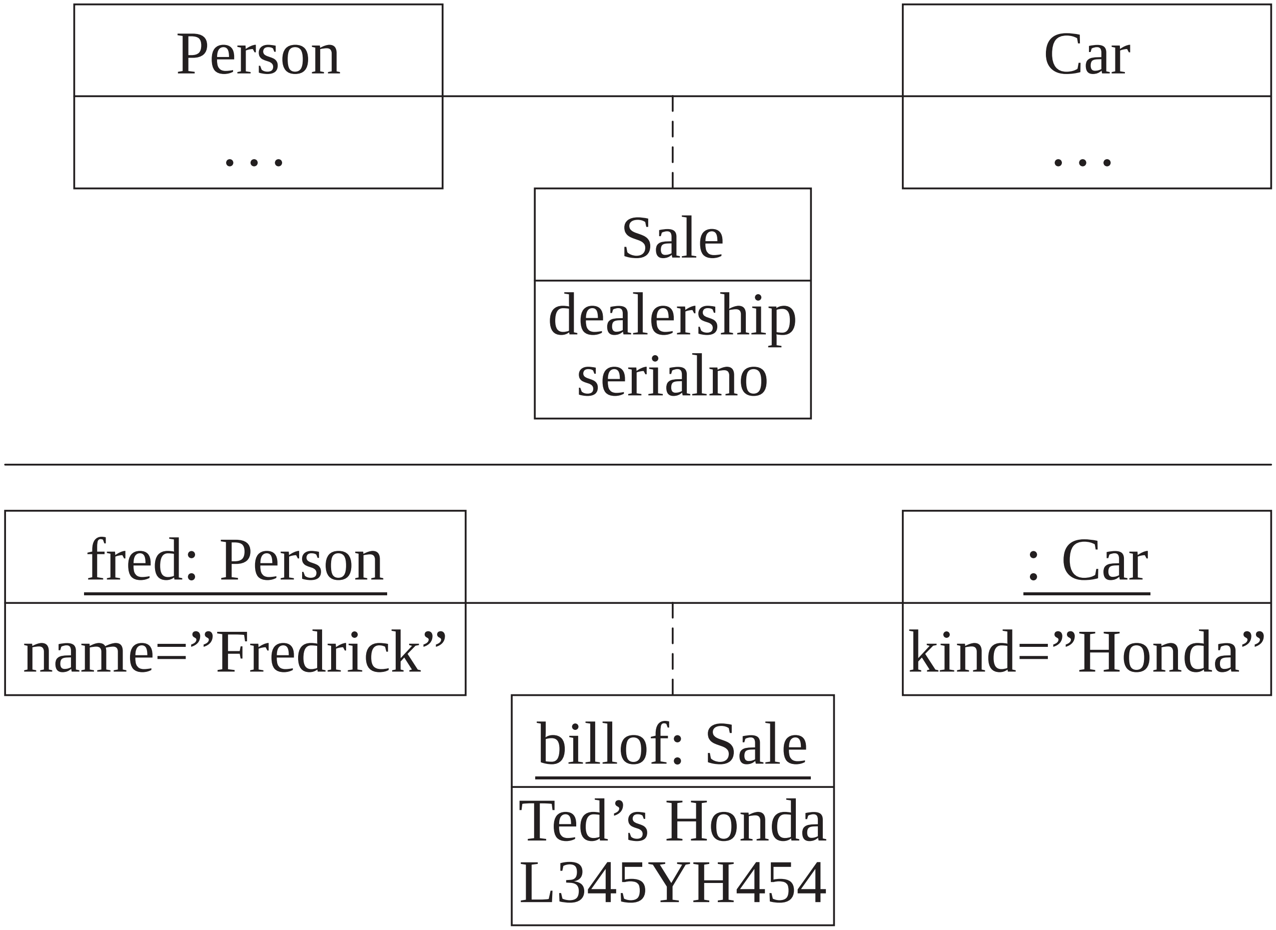
Object may not have a name (dynamically allocated).

- Objects associated with “ownership” are linked.



Which associations are valid/invalid/missing?

- **Association Class** : optional aspects of association (dashed line).



Person

. . .

Car

. . .

Sale

dealership
serialno

fred: Person

name=”Fredrick”

: Car

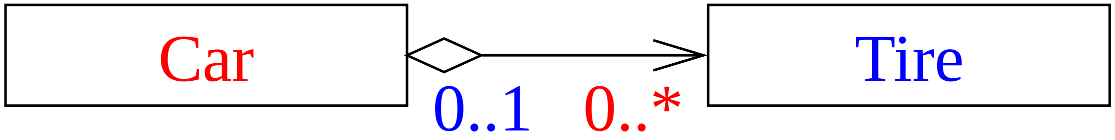
kind=”Honda”

billof: Sale

Ted’s Honda
L345YH454

- ○ cars sold through dealership (versus gift) need bill of sale
- ○ association class cannot exist without association (no owner)

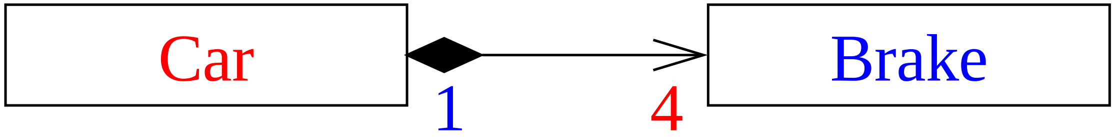
- **Aggregation** (◊) is an association between an aggregate attribute and its parts.



- ○ car can have 0 or more tires and a tire can only be on 0 or 1 car
- ○ aggregate may not create/destroy its parts, e.g., many different tires during car's lifetime and tires may exist after car's lifetime (snow tires).

```
class Car {
        Tires *tires[4];  // array of pointers to tires
```

- **Composition** (♦) is a stronger aggregation where a part is included in at most one composite at a time.

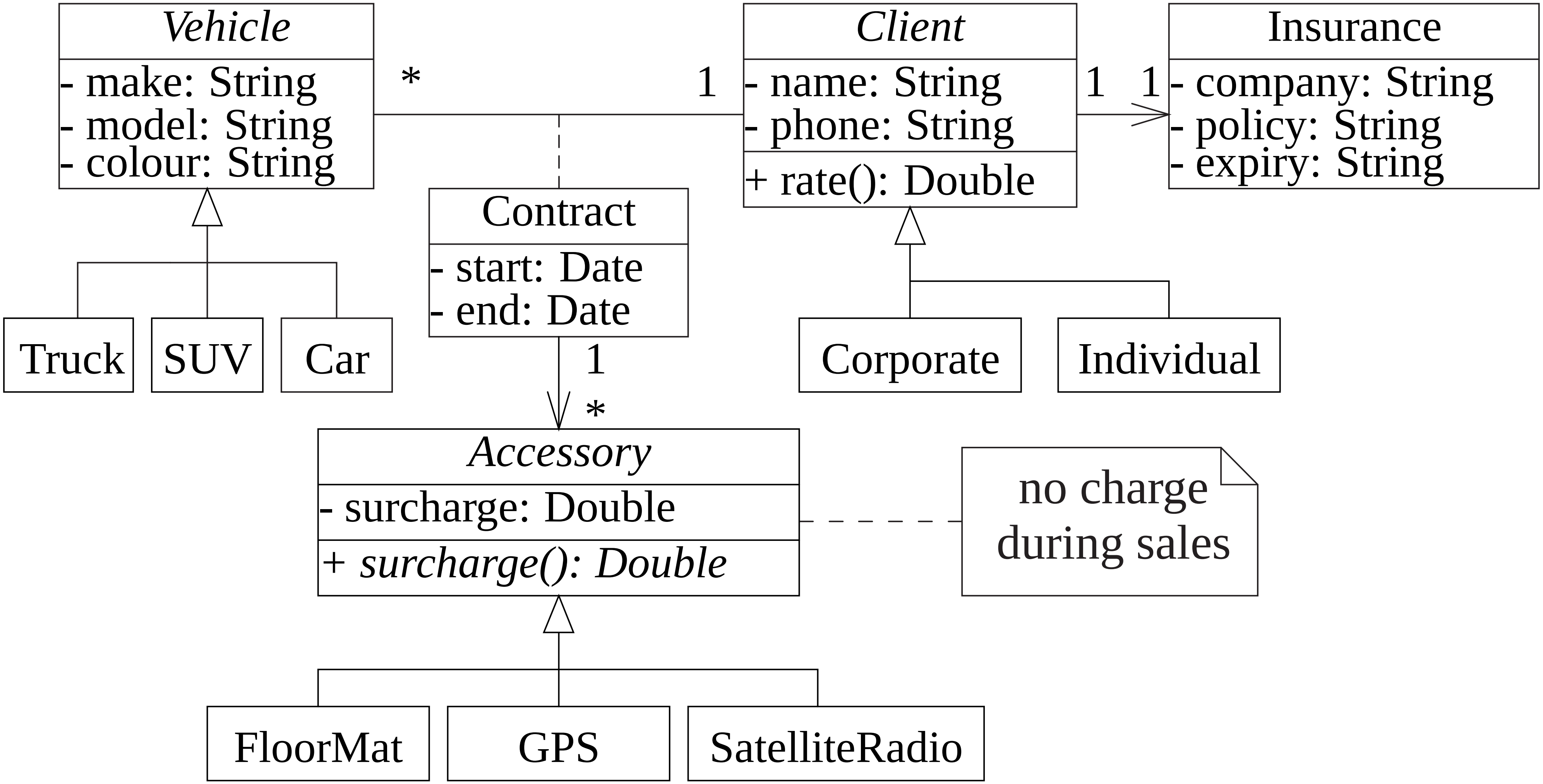


- ○ car has 4 brakes and each brake is on 1 car
- ○ composite aggregate often does create/destroy its parts, i.e., same brakes for lifetime of car and brakes deleted when car deleted (unless brakes removed at junkyard)

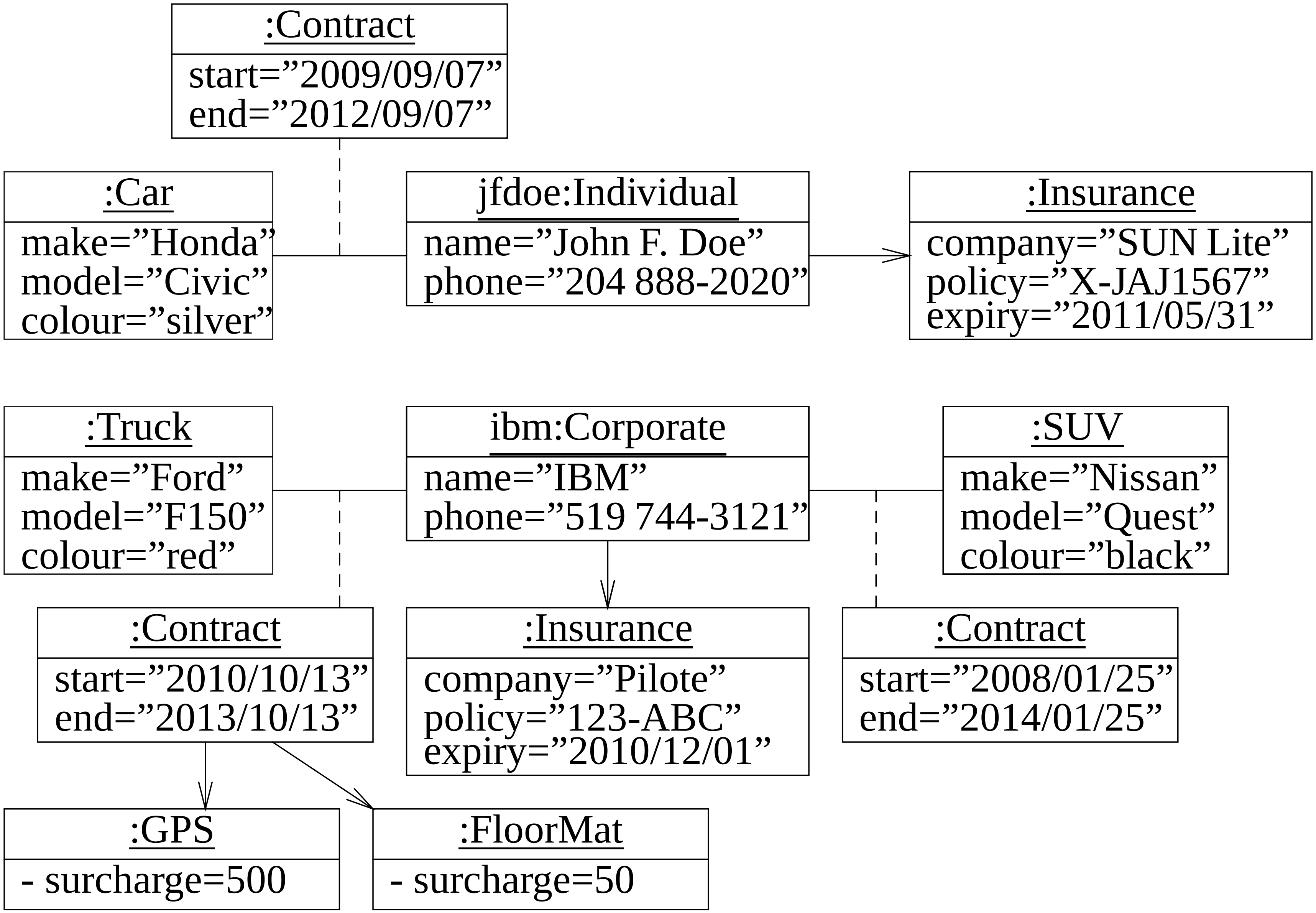
```
class Car {
        DiscBrake brakes[4];  // array of brakes
```

- UML has many more facilities, supporting very complex descriptions of relationships among entities.
  - VERY large visual mechanisms, with several confusing graphical representations.
- **UML diagram is too complex if it contains more than about 25 boxes.**

Classes Diagram



Object Diagram



:Contract
start=”2009/09/07”
end=”2012/09/07”

:Car
make=”Honda”
model=”Civic”
colour=”silver”

jfdoe:Individual
name=”John F. Doe”
phone=”204 888-2020”

:Insurance
company=”SUN Lite”
policy=”X-JAJ1567”
expiry=”2011/05/31”

:Truck
make=”Ford”
model=”F150”
colour=”red”

ibm:Corporate
name=”IBM”
phone=”519 744-3121”

:SUV
make=”Nissan”
model=”Quest”
colour=”black”

:Contract
start=”2010/10/13”
end=”2013/10/13”

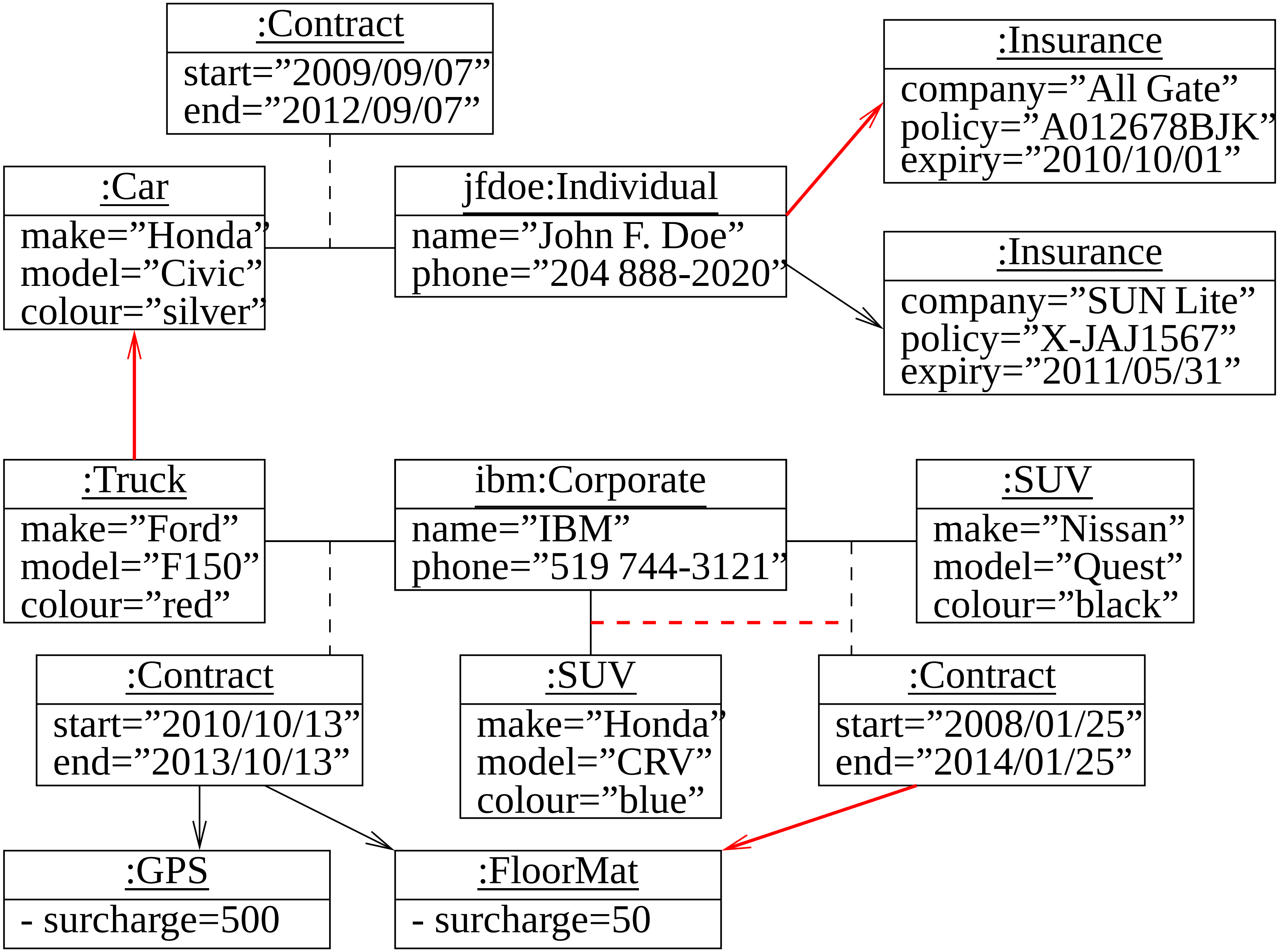
:Insurance
company=”Pilote”
policy=”123-ABC”
expiry=”2010/12/01”

:Contract
start=”2008/01/25”
end=”2014/01/25”

:GPS
- surcharge=500

:FloorMat
- surcharge=50

Invalid Object Diagram



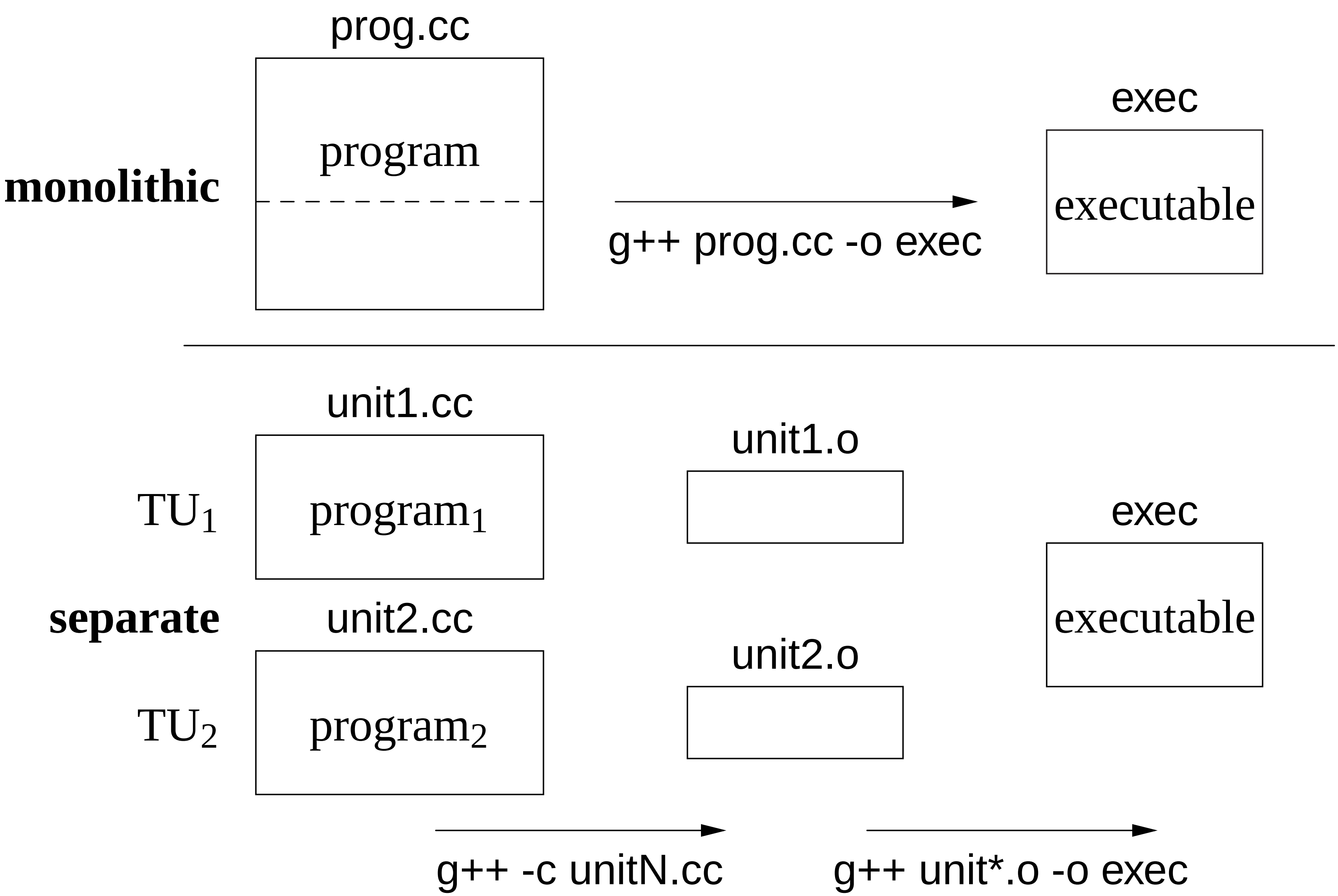
2.23 Separate Compilation

- As program size increases, so does cost of compilation.
- **Separate compilation** divides a program into units, where each unit can be independently compiled.
- Advantage: saves time by recompiling only program unit(s) that change.
  - In theory, if an expression is changed, only that expression needs to be recompiled.
  - In practice, compilation unit is coarser: **translation unit** (TU), which is a file in C/C++.
  - In theory, each line of code (expression) could be put in a separate file, but impractical.
  - ***So a TU should not be too big and not be too small.***
- Disadvantage: TUs depend on each other because a program shares many forms of information, especially types (done automatically in Java).
  - Hence, need mechanism to **import** information from referenced TUs and **export** information needed to referencing TUs.
- For example, simple program in file prog.cc using complex numbers:

prog.cc

```
#include <iostream>              // import
#include <cmath>
using namespace std;
class Complex {
    friend Complex operator+( Complex a, Complex b );
    friend ostream &operator<<( ostream &os, Complex c );
    static int objects;          // shared counter
    double re, im;
  public:
    Complex( double r = 0.0, double i = 0.0 ) { objects += 1; . . .}
    double abs() const { return sqrt( re * re + im * im ); };
    static void stats() { cout << objects << endl; }
};
int Complex::objects;            // declare
Complex operator+( Complex a, Complex b ) {. . .}
. . . // other arithmetic and logical operators
ostream &operator<<( ostream &os, Complex c ) {. . .}
const Complex C_1( 1.0, 0.0 );
int main() {
    Complex a( 1.3 ), b( 2., 4.5 ), c( -3, -4 );
    cout << a + b + c + C_1 << c.abs() << endl;
    Complex::stats();
}
```

- TU prog.cc has referenes to items in iostream and cmath.
- As well, there are many references within the TU, e.g., main references Complex.
- Subdividing program into TUs in C/C++ is complicated because of import/export mechanism.



prog.cc

program

monolithic

exec

executable

g++ prog.cc -o exec

unit1.cc

unit1.o

TU_1

program_1

exec

separate

executable

unit2.cc

unit2.o

TU_2

program_2

g++ -c unitN.cc

g++ unit*.o -o exec

- TU_i is NOT a program; program formed by combining TUs.
- Compile each TU_i with -c compiler flag to generate executable code in .o

file (Java has .class file).

```
$ g++ -c unit1.cc ...   // compile only modified TUs
```

generates files unit1.o containing a compiled version of source code.

- Combine TU$_i$ with -o compiler flag to generate executable program.

```
$ g++ unit*.o -o exec    // create new excutable program “exec”
```

- Separate original program into two TUs in files complex.cc and prog.cc:

complex.cc

```
#include <iostream>                 // import
#include <cmath>
using namespace std;
class Complex {
        friend Complex operator+( Complex a, Complex b );
        friend ostream &operator<<( ostream &os, Complex c );
        static int objects;         // shared counter
        double re, im;              // implementation
    public:
        Complex( double r = 0.0, double i = 0.0 ) { objects += 1; ...}
        double abs() const { return sqrt( re * re + im * im ); }
        static void stats() { cout << objects << endl; }
};
int Complex::objects;               // declare
Complex operator+( Complex a, Complex b ) {...}
... // other arithmetic and logical operators
ostream &operator<<( ostream &os, Complex c ) {...}
const Complex C_1( 1.0, 0.0 );
```

TU complex.cc has referenes to items in iostream and cmath.

prog.cc

```
int main() {
    Complex a( 1.3 ), b( 2., 4.5 ), c( -3, -4 );
    cout << a + b + c + C_1 << c.abs() << endl;
    Complex::stats();
}
```

TU prog.cc has referenes to items in iostream and complex.cc.

- How can TU prog.cc access Complex? By importing description of Complex.
- How are descriptions imported?

  TU imports information using preprocessor **#include**.
- Why not include complex.cc into prog.cc?

  Because all of complex.cc is compiled each time prog.cc is compiled so there is no advantage to the separation (program is still monolithic).
- Hence, must separate complex.cc into interface for import and implementation for code.
- Complex interface placed into file complex.h, for inclusion (import) into TUs.

complex.h

```
#ifndef __COMPLEX_H__
#define __COMPLEX_H__    // protect against multiple inclusion
#include <iostream>          // import
// NO “using namespace std”, use qualification to prevent polluting sco
class Complex {
    friend Complex operator+( Complex a, Complex b );
    friend std::ostream &operator<<( std::ostream &os, Complex c
    static int objects;          // shared counter
    double re, im;               // implementation
  public:
    Complex( double r = 0.0, double i = 0.0 );
    double abs() const;
    static void stats();
};
extern Complex operator+( Complex a, Complex b );
. . . // other arithmetic and logical operator descriptions
extern std::ostream &operator<<( std::ostream &os, Complex c );
extern const Complex C_1;
#endif // __COMPLEX_H__
```

- (Usually) no code, just descriptions : preprecessor variables, C/C++ types and forward declarations.

- **extern** qualifier means variable or routine definition is located elsewhere (not for types).
- Complex implementation placed in file complex.cc.

```
complex.cc
     #include "complex.h"         // do not copy interface
     #include <cmath>             // import
     using namespace std;         // ok to pollute implementation scope
     int Complex::objects;        // defaults to 0
     void Complex::stats() { cout << Complex::objects << endl; }
     Complex::Complex( double r, double i ) { objects += 1; . . .}
     double Complex::abs() const { return sqrt( re * re + im * im ); }
     Complex operator+( Complex a, Complex b ) {
          return Complex( a.re + b.re, a.im + b.im );
     }
     ostream &operator<<( ostream &os, Complex c ) {
          return os << c.re << "+" << c.im << "i";
     }
     const Complex C_1( 1.0, 0.0 );
```

- Implementation is composed of actual declarations and code.
- ***.cc file includes the .h file so that there is only one copy of the constants, declarations, and prototype information.***

- Why is **#include** <cmath> in complex.cc instead of complex.h?
- Compile TU complex.cc to generate complex.o.

```
$ g++ -c complex.cc
```

- What variables/routines are exported from complex.o?

```
$ nm -C complex.o | egrep ′ T | B ′
C_1
Complex::stats()
Complex::objects
Complex::Complex(double, double)
Complex::Complex(double, double)
Complex::abs() const
operator<<(std::ostream&, Complex)
operator+(Complex, Complex)
```

- In general, type names are not in the .o file?
- To compile prog.cc, it must import complex.h

prog.cc

```
#include "complex.h"
#include <iostream>            // included twice!
using namespace std;

int main() {
    Complex a( 1.3 ), b( 2., 4.5 ), c( -3, -4 );
    cout << a + b + c + C_1 << c.abs() << endl;
    Complex::stats();
}
```

- Why is **#include** <iostream> in prog.cc when it is already imported by complex.h?
- Compile TU prog.cc to generate prog.o.

```
$ g++ -c prog.cc
```

- Link together TUs complex.o and prog.o to generate exec.

```
$ g++ prog.o complex.o -o exec
```

- ***All .o files MUST be compiled for the same hardware architecture, e.g., all x86.***

- To hide global variables/routines (but NOT class members) in TU, qualify with **static**.

```
complex.cc
    . . .
    static Complex operator+( Complex a, Complex b ) {. . .}
    static ostream &operator<<( ostream &os, Complex c ) {. . .}
    static Complex C_1( 1.0, 0.0 );
```

  - here **static** means linkage NOT allocation.
- Encapsulation is provided by giving a user access to the include file(s) (.h) and the compiled source file(s) (.o), but not the implementation in the source file(s) (.cc).
- Note, while the .h file encapsulates the implementation, the implementation is still visible.
- To completely hide the implementation requires a (more expensive) reference:

complex.h

```
#ifndef __COMPLEX_H__
#define __COMPLEX_H__       // protect against multiple inclusion
#include <iostream>         // import
// NO “using namespace std”, use qualification to prevent polluting sco
class Complex {
        friend Complex operator+( Complex a, Complex b );
        friend std::ostream &operator<<( std::ostream &os, Complex c );
        static int objects;         // shared counter
        struct ComplexImpl;         // hidden implementation, nested class
        ComplexImpl &impl;          // indirection to implementation
    public:
        Complex( double r = 0.0, double i = 0.0 );
        Complex( const Complex &c );   // copy constructor
        ~Complex();                 // destructor
        Complex &operator=( const Complex &c ); // assignment opera
        double abs() const;
        static void stats();
};
extern Complex operator+( Complex a, Complex b );
extern std::ostream &operator<<( std::ostream &os, Complex c );
extern const Complex C_1;
#endif // __COMPLEX_H__
```

complex.cc

```
    #include "complex.h"              // do not copy interface
    #include <cmath>                  // import
    using namespace std;              // ok to pollute implementation scope
    int Complex::objects;             // defaults to 0
    struct Complex::ComplexImpl { double re, im; }; // implementation
    Complex::Complex( double r, double i ) : impl(*new ComplexImpl) {
        objects += 1; impl.re = r; impl.im = i;
    }
    Complex::Complex( const Complex &c ) : impl(*new ComplexImpl
        objects += 1; impl.re = c.impl.re; impl.im = c.impl.im;
    }
    Complex::~Complex() { delete &impl; }
    Complex &Complex::operator=( const Complex &c ) {
        impl.re = c.impl.re; impl.im = c.impl.im; return *this;
    }
    double Complex::abs() { return sqrt( impl.re * impl.re + impl.im * impl
    void Complex::stats() { cout << Complex::objects << endl; }
    Complex operator+( Complex a, Complex b ) {
        return Complex( a.impl.re + b.impl.re, a.impl.im + b.impl.im );
    }
    ostream &operator<<( ostream &os, Complex c ) {
        return os << c.impl.re << "+" << c.impl.im << "i";
    }
```

- A copy constructor and assignment operator are used because complex objects now contain a reference pointer to the implementation.

2.24 Inheritance

- Object-***oriented*** languages provide **inheritance** for writing reusable program-components.

Java	**C++**

class Base { . . . }
class Derived **extends** Base { . . . }
	struct Base { . . . }
	struct Derived : **public** Base { . . . };

- Inheritance has two orthogonal sharing concepts: implementation and type.
- Implementation inheritance provides reuse of code ***inside*** an object type; type inheritance provides reuse ***outside*** the object type by allowing existing code to access the base type.

2.24.1 Implementation Inheritance

- Implementation inheritance reuses program components by composing a new object's implementation from an existing object, taking advantage of previously written and tested code.

- Substantially reduces the time to generate and debug a new object type.
- One way to understand implementation inheritance is to model it via composition:

Composition	Inheritance

struct Base {
    int i;
    int r(...) { ... }
    Base() { ... }
};
struct Derived {
    Base b; *// explicit composition*
    int s(...) { b.i = 3; b.r(...); ... }
    Derived() { ... }
} d;
d.b.i = 3; *// composition reference*
d.b.r(...); *// composition reference*
d.s(...); *// direct reference*
	struct Base {
	    int i;
	    int r(...) { ... }
	    Base() { ... }
	};
	struct Derived : **public** Base { *// implicit*
	    // composition
	    int s(...) { i = 3; r(...); ... }
	    Derived() { ... }
	} d;
	d.i = 3; *// direct reference*
	d.r(...); *// direct reference*
	d.s(...); *// direct reference*

- Composition implies explicitly create an object member, b, to aid in the implementation, i.e., Derived has-a Base.

- Inheritance, "**public** Base" clause, implies implicitly:
  - ○ create an anonymous base-class object-member,
  - ○ ***open*** the scope of anonymous member so its members are accessible without qualification, both inside and outside the inheriting object type.
- Constructors and destructors must be invoked for all implicitly declared objects in the inheritance hierarchy as done for an explicit member in the composition.

```
                                  Base b;   b.Base(); // implicit, hidden declaration
Derived d;     implicitly         Derived d;   d.Derived();
. . .          rewritten as       . . .
                                  d.~Derived();   b.~Base(); // reverse order of constru
```

- If base type has members with the same name as derived type, it works like nested blocks: inner-scope name overrides outer-scope name.
- Still possible to access outer-scope names using "::" qualification to specify the particular nesting level.

Java	**C++**
class Base1 {	**struct** Base1 {
int i;	**int** i;
}	};
class Base2 **extends** Base1 {	**struct** Base2 : **public** Base1 {
int i;	**int** i; *// overrides Base1::i*
}	};
class Derived **extends** Base2 {	**struct** Derived : **public** Base2 {
int i;	**int** i; *// overrides Base2::i*
void s() {	**void** r() {
int i = 3;	**int** i = 3; *// overrides Derived::i*
this.i = 3;	Derived::i = 3; *// this.i*
((Base2)**this**).i = 3; *// super.i*	Base2::i = 3;
((Base1)**this**).i = 3;	Base2::Base1::i = 3; *// or Base1::i*
}	}
}	};

- E.g., Derived declaration first creates an invisible Base object in the Derived object, like composition, for the implicit references to Base::i and Base::r in Derived::s.
- ***Friendship is not inherited.***

```
class C {
        friend class Base;
        . . .
};
class Base {
        // access C′s private members
        . . .
};
class Derived : public Base {
        // not friend of C
};
```

- Unfortunately, having to inherit all of the members is not always desirable; some members may be inappropriate for the new type (e.g, large array).
- As a result, both the inherited and inheriting object must be very similar to have so much common code.

2.24.2 Type Inheritance

- Type inheritance extends name equivalence to allow routines to handle multiple types, called **polymorphism**, e.g.:

```
struct Foo {                    struct Bar {
    int i;                          int i;
    double d;                       double d;
                                    . . .
} f;                            } b;
void r( Foo f ) { . . . }
r( f );    // allowed
r( b );    // disallowed, name equivalence
```

- Since types Foo and Bar are structurally equivalent, instances of either type should work as arguments to routine r.
- Even if type Bar has more members at the end, routine r only accesses the common ones at the beginning as its parameter is type Foo.
- However, name equivalence precludes the call r(b).
- ***Type inheritance relaxes name equivalence by aliasing the derived name with its base-type names.***

```
struct Foo {                    struct Bar : public Foo { // inheritance
    int i;                          // remove Foo members
    double d;
                                    . . .
} f;                            } b;
void r( Foo f ) { . . . }
r( f );    // valid call, derived name matches
r( b );    // valid call because of inheritance, base name matches
```

- E.g., create a new type Mycomplex that counts the number of times abs is called for each Mycomplex object.
- Use both implementation and type inheritance to simplify building type Mycomplex:

```
struct Mycomplex : public Complex {
    int cntCalls;                               // add
    Mycomplex() : cntCalls(0) {}                // add
    double abs() {    // override, reuse complex′s abs routine
        cntCalls += 1;
        return Complex::abs();
    }
    int calls() { return cntCalls; }            // add
};
```

- Derived type Mycomplex uses the implementation of the base type Complex, adds new members, and overrides abs to count each call.
- Why is the qualification Complex:: necessary in Mycomplex::abs?
- Allows reuse of Complex's addition and output operation for Mycomplex values, because of the relaxed name equivalence provided by type inheritance between argument and parameter.
- Redeclare Complex variables to Mycomplex to get new abs, and member calls returns the current number of calls to abs for any Mycomplex object.
- Two significant problems with type inheritance.

1. ○ Complex routine **operator+** is used to add the Mycomplex values because of the relaxed name equivalence provided by type inheritance:

```
int main() {
    Mycomplex x;
    x = x + x;
}
```

   ○ However, result type from **operator+** is Complex, not Mycomplex.

   ○ Assignment of a complex (base type) to Mycomplex (derived type) disallowed because the Complex value is missing the cntCalls member!

- ○ Hence, a Mycomplex can mimic a Complex but not vice versa.
- ○ This fundamental problem of type inheritance is called **contra-variance**.
- ○ C++ provides various solutions, all of which have problems and are beyond this course.

2.

```
void r( Complex &c ) {
        c.abs();
}
int main() {
        Mycomplex x;
        x.abs();        // direct call of abs
        r( x );         // indirect call of abs
        cout << "x:" << x.calls() << endl;
}
```

- ○ While there are two calls to abs on object x, only one is counted!

- **public** inheritance means both implementation and type inheritance.
- **private** inheritance means only implementation inheritance.

```
class bus : private car { . . .
```

Use implementation from car, but bus is not a car.

- No direct mechanism in C++ for type inheritance without implementation inheritance.

2.24.3 Constructor/Destructor

- Constructors are *implicitly* executed top-down, from base to most derived type.
- Mandated by scope rules, which allow a derived-type constructor to use a base type's variables so the base type must be initialized first.
- Destructors are *implicitly* executed bottom-up, from most derived to base type.
- Mandated by the scope rules, which allow a derived-type destructor to use a base type's variables so the base type must be uninitialized last.
- Java finalize must be *explicitly* called from derived to base type.
- Unlike Java, C++ disallows calls to other constructors at the start of a constructor.
- To pass arguments to other constructors, use same syntax as for initializing **const** members.

Java	**C++**

class Base {
    Base(**int** i) { . . . }
};
class Derived **extends** Base {
    Derived() { **super**(3); . . . }
    Derived(**int** i) { **super**(i); . . . }
};
	struct Base {
	    Base(**int** i) { . . . }
	};
	struct Derived : **public** Base {
	    Derived() : Base(3) { . . . }
	    Derived(**int** i) : Base(i) {. . .}
	};

2.24.4 Copy Constructor / Assignment

- If a copy constructor or assignment operator is not defined in the derived class, it inherits from the base class.

```
struct B {
    B() {}
    B( const B &c ) { cout << "B(&) "; }
    B &operator=( const B &rhs ) { cout << "B= "; }
};
struct D : public B {       // inherit copy and assignment
    int i;          // basic type, bitwise
};
int main() {
    D d = d;        // bitwise/memberwise copy
    d = d;          // bitwise/memberwise assignment
}
```

outputs the following:

```
B(&) B=
```

- If D defines a copy-constructor/assignment, it is used rather than that in any base class.

```
struct D : public B {
    int i;          // basic type, bitwise
    D( const D &c ) : B( c ), i( c.i ) {}
    D &operator=( const D &rhs ) {
        i = rhs.i;  (B &)*this = rhs;  return *this;
    }
};
```

Must manually copy each subobject (same output as before). **Note coercion!**

2.24.5 Overloading

- Overloading a member routine in a derived class overrides all overloaded routines in the base class with the same name.

```
class Base {
  public:
     void mem( int i ) {}
     void mem( char c ) {}
};
class Derived : public Base {
  public:
     void mem() {}    // overrides both versions of mem in base class
};
```

- Hidden base-class members can still be accessed:
  - Provide explicit wrapper members for each hidden one.

```
class Derived : public Base {
  public:
     void mem() {}
     void mem( int i ) { Base::mem( i ); }
     void mem( char c ) { Base::mem( c ); }
};
```

  - Collectively provide implicit members for all of them.

```
class Derived : public Base {
  public:
     void mem() {}
     using Base::mem; // all base mem routines visible
};
```

- Use explicit qualification to call members (violates abstraction).

```
Derived d;
d.Base::mem( 3 );
d.Base::mem( 'a' );
d.mem();
```

2.24.6 Virtual Routine

- When a member is called, it is usually obvious which one is invoked even with overriding:

```
struct Base {
    void r() { . . . }
};
struct Derived : public Base {
    void r() { . . . }        // override Base::r
};
Base b;
b.r();     // call Base::r
Derived d;
d.r();     // call Derived::r
```

- However, it is not obvious for arguments/parameters and pointers/references:

```
void s( Base &b ) { b.r(); }
s( d );            // inheritance allows call: Base::r or Derived::r ?
Base &bp = d;      // assignment allowed because of inheritance
bp.r();            // Base::r or Derived::r ?
```

- Inheritance masks the actual object type, but both calls should invoke Derived::r because argument b and reference bp point at an object of type Derived.
- If variable d is replaced with b, the calls should invoke Base::r.

- To invoke routine defined in referenced object, qualify member routine with **virtual**.
- To invoke routine defined by type of pointer/reference, do not qualify member routine with **virtual**.
- C++ uses non-virtual as the default because it is more efficient.
- Java ***always*** uses virtual for all calls to objects.
- Once a base type qualifies a member as virtual, ***it is virtual in all derived types regardless of the derived type's qualification for that member***.
- Programmer may want to access members in Base even if the actual object is of type Derived, which is possible because Derived *contains* a Base.
- C++ provides mechanism to override the default at the call site.

Java	C++
class Base {	struct Base {
public void f() {} // virtual	void f() {} // non-virtual
public void g() {} // virtual	void g() {} // non-virtual
public void h() {} // virtual	virtual void h() {} // virtual
}	};
class Derived extends Base {	struct Derived : public Base {
public void g() {} // virtual	void g() {}; // non-virtual
public void h() {} // virtual	void h() {}; // virtual
}	};
final Base bp = new Derived();	Base &bp = *new Derived(); // polymorphic assig
bp.f(); // Base.f	bp.f(); // Base::f, pointer type
((Base)bp).g(); // Derived.g	bp.g(); // Base::g, pointer type
bp.g(); // Derived.g	((Derived &)bp).g(); // Derived::g, pointer type
((Base)bp).h(); // Derived.h	bp.Base::h(); // Base::h, explicit selection
bp.h(); // Derived.h	bp.h(); // Derived::h, object type

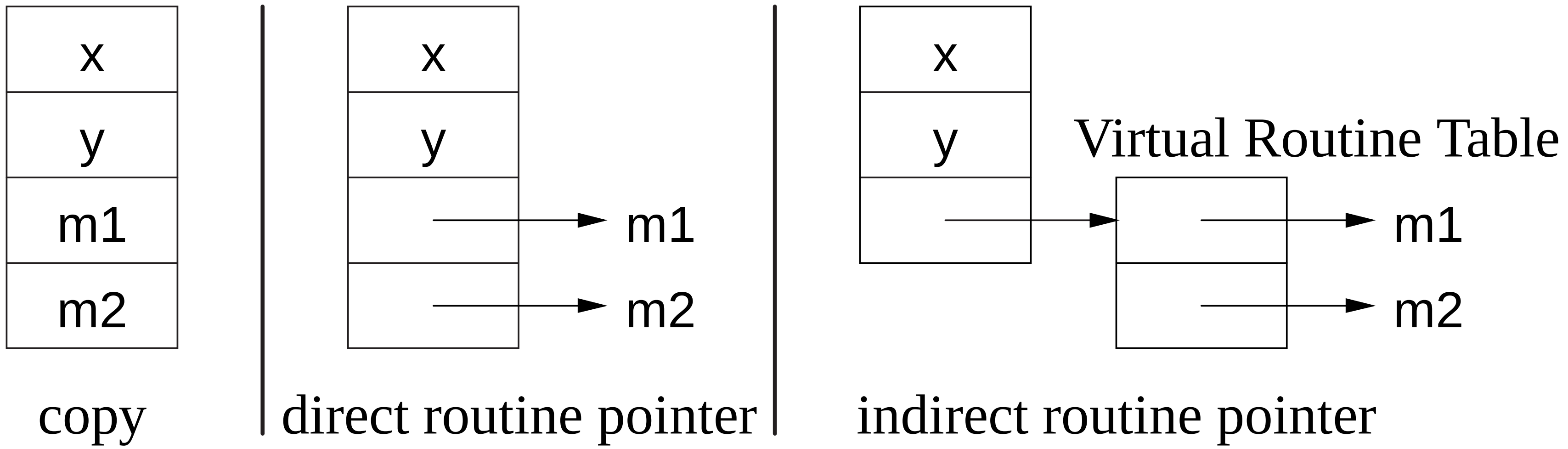
- Java casting does not provide access to base-type's member routines.
- ***Virtual members are only necessary to access derived members through a base-type reference or pointer.***
- If a type is not involved in inheritance (final class in Java), virtual members

are unnecessary so use more efficient call to its members.

- C++ virtual members are qualified in the base type as opposed to the derived type.
- Hence, C++ requires the base-type definer to presuppose how derived definers might want the call default to work.
- ***Good practice for inheritable object types is to make all routine members virtual.***
- Any type with virtual members and a destructor needs to make the destructor virtual so the most derived destructor is called through a base-type pointer/reference.
- Virtual routines are normally implemented by routine pointers.

```
class Base {
    int x, y;                   // data members
    virtual void m1(. . .);     // routine members
    virtual void m2(. . .);
};
```

- May be implemented in a number of ways:



2.24.7 Downcast

- Type inheritance can mask the actual type of an object through a pointer/reference.
- A **downcast** dynamically determines the actual type of an object pointed to by a polymorphic pointer/reference.
- The Java operator instanceof and the C++ **dynamic_cast** operator perform a dynamic check of the object addressed by a pointer/reference (not coercion):

Java	C++

Base bp = **new** Derived();
if (**bp instanceof Derived**)
    ((Derived)bp).rtn();
	Base *bp = **new** Derived;
	Derived *dp;
	dp = **dynamic_cast<Derived *>(bp)**;
	if (dp != 0) { *// 0 => not Derived*
	    dp->rtn(); *// only in Derived*

- ***To use* dynamic_cast *on a type, the type must have at least one virtual member.***

2.24.8 Slicing

- Polymorphic copy or assignment can result in object truncation, called **slicing**.

```
struct B {
        int i;
};
struct D : public B {
        int j;
};
void f( B b ) {. . .}
int main() {
        B b;
        D d;
        f( d );            // truncate D to B
        b = d;             // truncate D to B
}
```

- ***Avoid polymorphic value copy/assignment; use polymorphic pointers.***

2.24.9 Protected Members

- Inherited object types can access and modify public and protected members allowing access to some of an object's implementation.

```
class Base {
  private:
     int x;
  protected:
     int y;
  public:
     int z;
};
class Derived : public Base {
  public:
     Derived() { x; y; z; };   // x disallowed; y, z allowed
};
int main() {
     Derived d;
     d.x; d.y; d.z;           // x, y disallowed; z allowed
}
```

2.24.10 Abstract Class

- **Abstract class** combines type and implementation inheritance for structuring new types.
- Contains at least one pure virtual member that ***must*** be implemented by

derived class.

```
class Shape {
        int colour;
   public:
        virtual void move( int x, int y ) = 0; // pure virtual member
};
```

- Strange initialization to 0 means pure virtual member.
- Define type hierarchy (taxonomy) of abstract classes moving common data and operations are high as possible in the hierarchy.

Java	**C++**

abstract class Shape {
    protected int colour = White;
    public
 	
    abstract void move(int x, int y);
}
abstract class Polygon **extends** Shape {
    protected int edges;
    public **abstract int sides();**
}
class Rectangle **extends** Polygon {
    protected int x1, y1, x2, y2;
 	
    public Rectangle(. . .) {. . .}
    public void move(**int** x, **int** y) {. . .}
    public int sides() { **return** 4; }
}
class Square **extends** Rectangle {
    // check square
    Square(. . .) { **super**(. . .); . . .}
}
	class Shape {
	  protected: **int** colour;
	  public:
	    Shape() { colour = White; }
	    virtual void move(int x, int y) = 0;
	};
	class Polygon : **public** Shape {
	  protected: **int** edges;
	  public: **virtual int sides() = 0;**
	};
	class Rectangle : **public** Polygon {
	  protected: **int** x1, y1, x2, y2;
	  public:
	    Rectangle(. . .) {. . .} *// init corners*
	    void move(**int** x, **int** y) {. . .}
	    int sides() { **return** 4; }
	};
	struct Square : **public** Rectangle {
	    // check square
	    Square(. . .) : Rectangle(. . .) {. . .}
	};

- Use **public**/**protected** to define interface and implementation access for derived classes.

- Provide (pure) virtual member to allow overriding and force implementation by derived class.
- Provide default variable initialization and implementation for **virtual** routine (non-abstract) to simplify derived class.
- Provide non-virtual routine to ***force*** specific implementation; ***derived class should not override these routines***.
- **Concrete class** inherits from one or more abstract classes defining all pure virtual members, i.e., can be instantiated.
- ***Cannot instantiate an abstract class, but can declare pointer/reference to it.***
- Pointer/reference used to write polymorphic data structures and routines:

```
void move3D( Shape &s ) { . . . s.move(. . .); . . . }
Polygon *polys[10] = { new Rectangle(), new Square(), . . . };
for ( unsigned int i = 0; i < 10; i += 1 ) {
    cout << polys[i]->sides() << endl;  // polymorphism
    move3D( *polys[i] );  // polymorphism
}
```

- To maximize polymorphism, ***write code to the highest level of abstraction***, i.e. use Shape over Polygon, use Polygon over Rectangle, etc.

2.24.11 Multiple Inheritance

- **Multiple inheritance** allows a new type to apply type and implementation inheritance multiple times.

```
class X : public Y, public Z, private P, private Q { . . . }
```

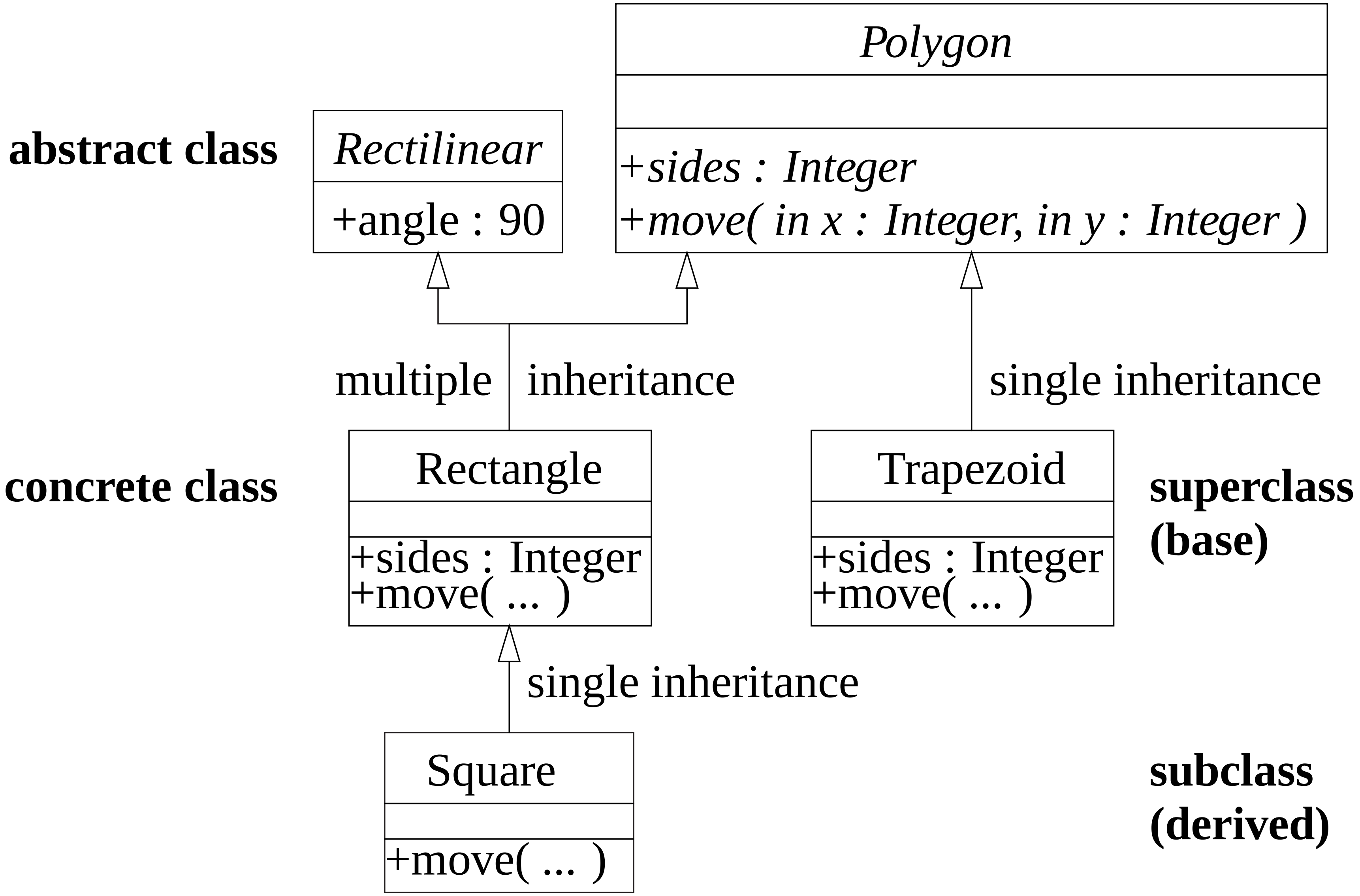
- X type is aliased to types Y and Z with implementation, and also uses implementation from P and Q.
- **Interface class** (**pure abstract-class**) provides only types and constants, providing type inheritance.
- Java only allows multiple inheritance for interface class.

Java	C++
interface Polygon {	**struct** Polygon {
int sides();	**virtual** **int** sides() = 0;
void move(**int** x, **int** y);	**virtual** **void** move(**int** x, **int** y) = 0;
}	};
interface Rectilinear {	**struct** Rectilinear {
final **int** angle = 90;	**enum** { angle = 90 };
}	};
class Rectangle **implements** Rectilinear,	**class** Rectangle : **public** Polygon,
Polygon {	**public** Rectilinear {
private **int** x1, y1, x2, y2;	**int** x1, y1, x2, y2;
	public:
public **void** move(**int** x, **int** y) {}	**void** move(**int** x, **int** y) {}
public **int** sides() { **return** 4; }	**int** sides() { **return** 4; }
}	};
class Square **extends** Rectangle {	**struct** Square : **public** Rectangle {
public **void** move(**int** x, **int** y) {}	**void** move(**int** x, **int** y) {}
}	};

- Multiple inheritance has ***many*** problems (beyond this course).
- ***Safe if restrict multiple inheritance to one*** **public** ***type and one or two*** **private** ***types.***

2.24.12 UML

- **Generalization** : reuse through forms of inheritance.



  - Inheritance establishes “**is-a**” relationship on type, and reuse of attributes and operations.
  - Association class can be implemented with forms of multiple inheritance (mixin).

- For abstract class, the class name and abstract operations are *italicized*.
- For concrete class, abstract operations that are implemented appear in the class diagram.

2.25 Inheritance / Composition Design

- Duality between “has-a” (composition) and “is-a” (inheritance) relationship.
- Types created from multiple composite classes; types created from multiple superclasses.

Composition	Inheritance
class A {. . .};	**class** A {. . .};
class B { A a; . . .};	**class** B : A {. . .};
class C {. . .};	**class** C {. . .};
class D { B b; C c; . . .};	**class** D : B, C {. . .};

- Both approaches:
  - ○ remove duplicated code (variable/code sharing)
  - ○ have separation of concern into components/superclasses.
- Choose inheritance when evolving hierarchical types (taxonomy) needing polymorphism.

```
Vehicle
  Construction
    Heavy Machinery
      Crane, Grader, Back-hoe
    Haulage
      Semi-trailer, Flatbed
  Passenger
    Commercial
      Bus, Fire-truck, Limousine, Police-motorcycle
    Personal
      Car, SUV, Motorcycle
```

- For maximum reuse and to eliminate duplicate code, place variables/operations as high in the hierarchy as possible.
- ***Polymorphism requires derived class maintain base class's interface (substitutability).***

  - ○ derived class should also have **behavioural** compatibility with base class.

- However, all taxonomies are an organizational compromise: when is a car a limousine and vice versa.
- Not all objects fit into taxonomy: flying-car, boat-car.
- Inheritance is rigid hierarchy.
- Choose composition when implementation can be **delegated**.

```
class Car {
        SteeringWheel s;         // fixed
        Donut spare;
        Wheel *wheels[4];        // dynamic
        Engine *eng;
        Transmission *trany;
    public:
        Car( Engine *e = fourcyl, Transmission *t = manual ) :
                eng( e ), trany( t ) { wheels[i] = . . .}
        rotate() {. . .}         // rotate tires
        wheels( Wheels *w[4] ) {. . .} // change wheels
        engine( Engine *e ) {. . .}  // change engine
};
```

- Composition may be fixed or dynamic (pointer/reference).

- Composition still uses hierarchical types to generalize components.
    - Engine is abstract class that is specialized to different kinds of engines, e.g., 3,4,6,8 cylinder, gas/diesel/hybrid, etc.

2.26 Template

- Inheritance provides reuse for types organized into a hierarchy that extends name equivalence.
- **Template** provides alternate kind of reuse with no type hierarchy and types are not equivalent.
- E.g., overloading, where there is identical code but different types:

```
int max( int a, int b ) { return a > b ? a : b; }
double max( double a, double b ) { return a > b ? a : b; }
```

- **Template routine** eliminates duplicate code by using types as compile-time parameters:

```
template<typename T>  T max( T a, T b ) { return a > b ? a : b }
```

- **template** introduces type parameter T used to declare return and parameter types.

- At a call, compiler infers type T from argument(s), and constructs a specialized routine with inferred type(s):

```
cout << max( 1, 3 ) << " " << max( -1, -4 ) << endl; // T -> int
cout << max( 1.1, 3.5 ) << " " << max( -1.1, -4.5 ) << endl; // T -> doubl
```

- Inferred type must supply all operations used within the template routine.
  - e.g., types used with template routine max must supply **operator**>.
- **Template type** prevents duplicating code that manipulates different types.
- E.g., collection data-structures (e.g., stack), have common code to manipulate data structure, but type stored in collection varies:

```
template<typename T=int, unsigned int N=10> // default type/value
  struct Stack {                        // NO ERROR CHECKING
    T elems[N];                         // maximum N elements
    unsigned int size;                  // position of free element after top
    Stack() { size = 0; }
    T top() { return elems[size - 1]; }
    void push( T e ) { elems[size] = e; size += 1; }
    T pop() { size -= 1; return elems[size]; }
};
template<typename T, unsigned int N>  // print stack
  ostream &operator<<( ostream &os, const Stack<T, N> &stk ) {
    for ( int i = 0; i < stk.size; i += 1 ) os << stk.elems[i] << " ";
    return os;
}
```

- Type parameter, T, specifies the element type of array elems, and return and parameter types of the member routines.
- Integer parameter, N, denotes the maximum stack size.
- Unlike template routines, the compiler cannot infer the type parameter for template types, so it must be explicitly specified:

```
Stack<> si;                     // stack of int, 10
Stack<double> sd;               // stack of double, 10
Stack<Stack<int>,20> ssi;       // stack of (stack of int, 10), 20
si.push( 3 );                   // si : 3
si.push( 4 );                   // si : 3 4
sd.push( 5.1 );                 // sd : 5.1
sd.push( 6.2 );                 // sd : 5.1 6.2
ssi.push( si );                 // ssi : (3 4)
ssi.push( si );                 // ssi : (3 4) (3 4)
ssi.push( si );                 // ssi : (3 4) (3 4) (3 4)
cout << si.top() << endl;       // 4
cout << sd << endl;             // 5.1 6.2
cout << ssi << endl;            // 3 4  3 4  3 4
int i = si.pop();               // i : 4, si : 3
double d = sd.pop();            // d : 6.2, sd : 5.1
si = ssi.pop();                 // si : 3 4, ssi : (3 4) (3 4)
```

Why does cout << ssi << endl have 2 spaces between the stacks?

- Specified type must supply all operations used within the template type.
- ***There must be a space between the two ending chevrons or >> is parsed as operator>>.***

```
template<typename T> struct Foo { . . . };
Foo<Stack<int>> foo;   // syntax error
Foo<Stack<int> > foo; // space between chevrons
```

- ***Compiler requires a template definition for each usage so both the interface and implementation of a template must be in a .h file, precluding some forms of encapsulation.***

2.26.1 Standard Library

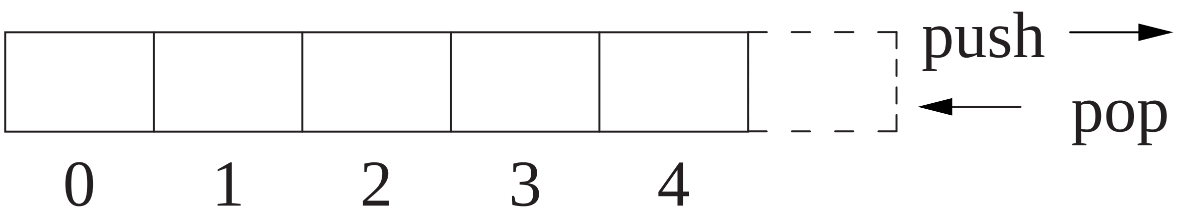
- C++ Standard Library is a collection of (template) classes and routines providing: I/O, strings, data structures, and algorithms (sorting/searching).
- Data structures are called **container**s: vector, map, list (stack, queue, deque).
- In general, nodes of a data structure are either in a container or pointed-to from the container.
- To copy a node requires its type have a default and/or copy constructor so instances can be created without constructor arguments.
- ***Standard library containers use copying ⇒ node type must have default constructor.***

- All containers are dynamic sized so nodes are allocated in the heap.
- To provide encapsulation, containers use a nested **iterator** typeto traverse nodes.
  - Knowledge about container implementation is completely hidden.
- Iterator capabilities often depend on kind of container:
  - singly-linked list has unidirectional traversal
  - doubly-linked list has bidirectional traversal
  - hashing list has random traversal
- Iterator operator "++" moves forward to the next node, until ***past*** the end of the container.
- For bidirectional iterators, operator "--" moves in the reverse direction to "++".

2.26.1.1 Vector

- vector has random access, length, subscript checking (at), and assignment (like Java array).

std::vector<T>	
vector()	create empty vector
vector(**int** N)	create vector with N empty elements
int size()	vector size
bool empty()	size() == 0
T &**operator**[ ](**int** i)	access ith element, NO subscript checking
T &at(**int** i)	access ith element, subscript checking
vector &**operator**=(**const** vector &)	vector assignment
void push_back(**const** T &x)	add x after last element
void pop_back()	remove last element
void resize(**int** n)	add or erase elements at end so size() == n
void clear()	erase all elements



- vector is alternative to C/C++ arrays.

```
#include <vector>
int i, elem;
vector<int> v;                       // think: int v[0]
for ( ;; ) {                         // create/assign vector
     cin >> elem;
  if ( cin.fail() ) break;
     v.push_back( elem );            // add elem to vector
}
vector<int> c;                       // think: int c[0]
c = v;                               // array assignment
for ( i = c.size() - 1; 0 <= i; i -= 1 ) {
     cout << c.at(i) << " ";         // subscript checking
}
cout << endl;
v.clear();                           // remove ALL elements
```

- Vector declaration ***may*** specify an initial size, e.g., vector<**int**> v(size), like a dimension.
- To reduce dynamic allocation, it is more efficient to dimension, when the size is known.

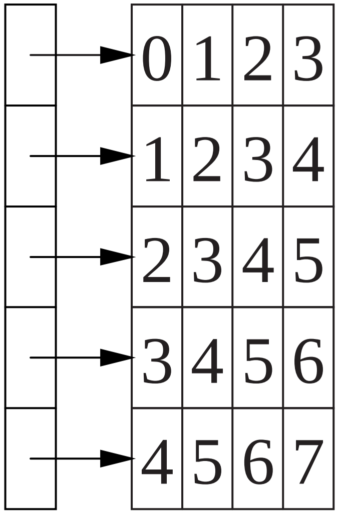
```
int size;
cin >> size;                    // read dimension
vector<int> v(size);            // think int v[size]
```

- Matrix declaration is a vector of vectors:

```
vector< vector<int> > m;
```

- Again, it is more efficient to dimension, when size is known.

```
#include <vector>
vector< vector<int> > m( 5, vector<int>(4) );
for ( int r = 0; r < m.size(); r += 1 ) {
    for ( int c = 0; c < m[r].size(); c += 1 ) {
        m[r][c] = r+c;        // or m.at(r).at(c)
    }
}
for ( int r = 0; r < m.size(); r += 1 ) {
    for ( int c = 0; c < m[r].size(); c += 1 ) {
        cout << m[r][c] << ", ";
    }
    cout << endl;
}
```

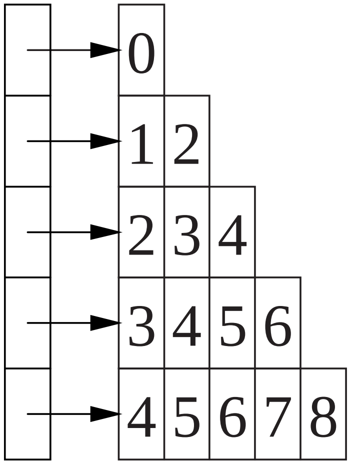


```
0 1 2 3
1 2 3 4
2 3 4 5
3 4 5 6
4 5 6 7
```

- Optional second argument is initialization value for each element, i.e., 5 rows of vectors each initialized to a vector of 4 integers initialized to zero.

- All loop bounds use dynamic size of row or column (columns may not be same length).
- Alternatively, each row is dynamically dimensioned to a specific size, e.g., triangular matrix.

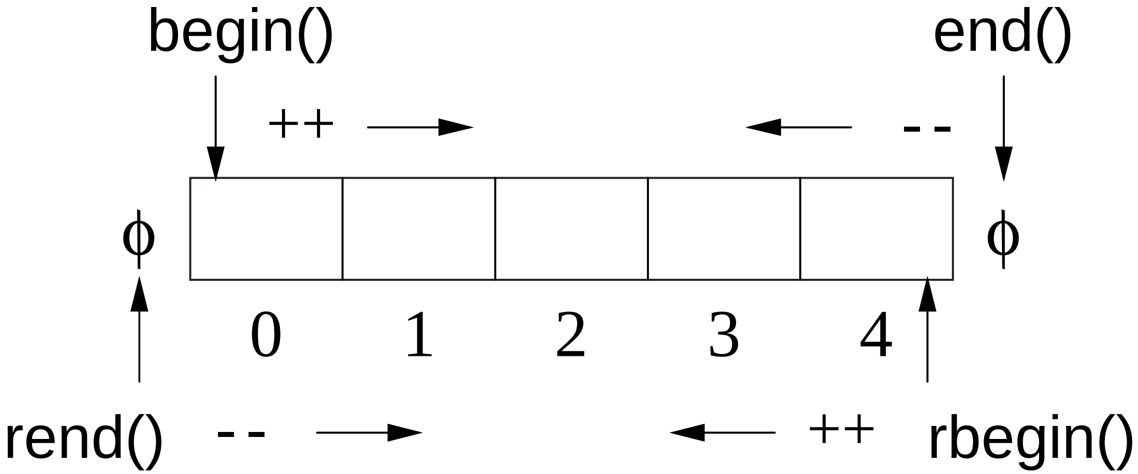
```
vector< vector<int> > m( 5 ); // 5 rows
for ( int r = 0; r < m.size(); r += 1 ) {
    m[r].resize( r + 1 ); // different length
    for ( int c = 0; c < m[r].size(); c += 1 ) {
        m[r][c] = r+c;      // or m.at(r).at(c)
    }
}
```



0				
1	2			
2	3	4		
3	4	5	6	
4	5	6	7	8

- Iterator allows traversal in insertion order or random order.

std::vector<T>::iterator	
iterator begin()	iterator pointing to first element
iterator end()	iterator pointing **AFTER** last element
iterator rbegin()	iterator pointing to last element
iterator rend()	iterator pointing **BEFORE** first elemen
iterator insert(iterator posn, **const** T &x)	insert x before posn
iterator erase(iterator posn)	erase element at posn
++, --, +, +=, -, -= (insertion / random order)	forward/backward operations



- Iterator's value is a pointer to its current vector element ⇒ dereference to access element.

```
vector<int> v(3);
vector<int>::iterator it;
v[0] = 2;                  // initialize first element
it = v.begin();            // intialize iterator to first element
cout << v[0] << " " << *v.begin() << " " << *it << endl;
```

- ***If*** **erase** ***and*** **insert** ***took subscript argument, no iterator necessary!***
- Use iterator like subscript by adding/subtracting from begin/end.

```
v.erase( v.begin() );      // erase v[0], first
v.erase( v.end() - 1 );    // erase v[N - 1], last (why “- 1”?)
v.erase( v.begin + 3 );    // erase v[3]
```

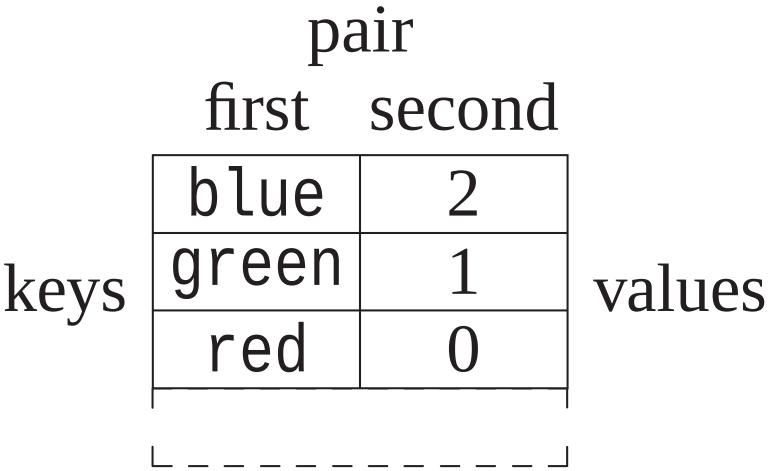
- ***Insert or erase during iteration using an iterator causes failure.***

```
vector<int> v;
for ( int i = 0 ; i < 5; i += 1 ) // create
    v.push_back( 2 * i );          // values: 0, 2, 4, 6, 8

v.erase( v.begin() + 3 );          // remove v[3] : 6

int i;    // find position of value 4 using subscript
for ( i = 0; i < 5 && v[i] != 4; i += 1 );
v.insert( v.begin() + i, 33 );     // insert 33 before value 4

// print reverse order using iterator (versus subscript)
vector<int>::reverse_iterator r;
for ( r = v.rbegin(); r != v.rend(); r ++ ) // ++ move towards rend
    cout << *r << endl;           // values: 8, 4, 33, 2, 0
```

2.26.1.2 Map

- map (dictionary) has random access, sorted, unique-key container of pairs (Key, Val).

std::map<Key,Val> / std::pair<Key,Val>	
map()	create empty map
int size()	map size
bool empty()	size() == 0
Val &**operator**[ ](**const** Key &k)	access pair with Key k
int count(Key key)	0 ⇒ no key, 1 ⇒ key (unique keys)
map &**operator**=(**const** map &)	map assignment
insert(pair<Key,Val>(k, v))	insert pair
erase(Key k)	erase key k
void clear()	erase all pairs



first	second
blue	2
green	1
red	0

```
#include <map>
map<string, int> m, c;          // Key => string, Val => int
m["green"] = 1;                 // create, set to 1
m["blue"] = 2;                  // create, set to 2
m["red"];                       // create, set to 0 for int
m["green"] = 5;                 // overwrite 1 with 5
cout << m["green"] << endl;     // print 5
c = m;                          // map assignment
m.insert( pair<string,int>( "yellow", 3 ) ); // m[“yellow”] = 3
if ( m.count( "black" ) )       // check for key “black”
m.erase( "blue" );              // erase pair( “blue”, 2 )
```

- First subscript for key creates an entry and initializes it to default or specified value.
- Iterator can search and return values in key order.

std::map<T>::iterator / std::map<T>::reverse_iterator	
iterator begin()	iterator pointing to first pair
iterator end()	iterator pointing **AFTER** last pair
iterator rbegin()	iterator pointing to last pair
iterator rend()	iterator pointing **BEFORE** first pair
iterator find(Key &k)	find position of key k
iterator insert(iterator posn, **const** T &x)	insert x before posn
iterator erase(iterator posn)	erase pair at posn
++, -- (sorted order)	forward/backward operations

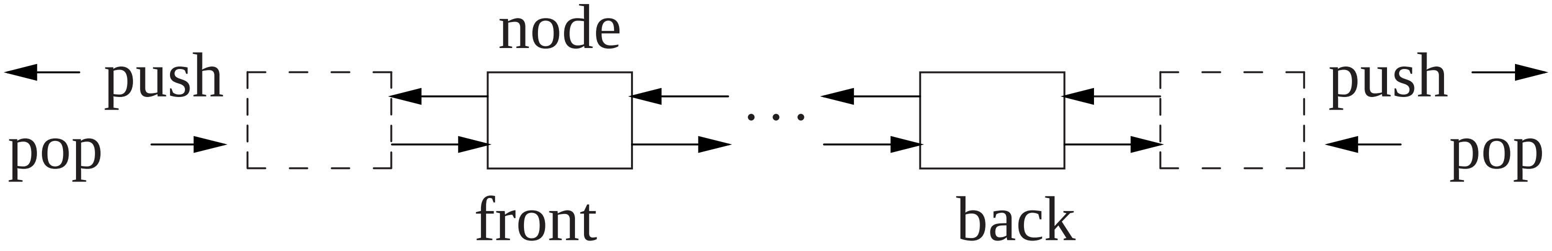
- Iterator returns a pointer to a pair, with fields first (key) and second (value).

```
#include <map>
map<string,int>::iterator f = m.find( "green" ); // find key position
if ( f != m.end() )                       // found ?
    cout << "found " << f->first << ′ ′ << f->second << endl;

for ( f = m.begin(); f != m.end(); f ++ )           // increasing order
    cout << f->first << ′ ′ << f->second << endl;

map<string,int>::reverse_iterator r;
for ( r = m.rbegin(); r != m.rend(); r ++ )        // decreasing order
    cout << r->first << ′ ′ << r->second << endl;
m.clear();                                // remove ALL pairs
```

2.26.1.3 List

- If random access is not required, use more efficient single (stack/queue/deque) or double (list) linked-list container.
- Examine list (arbitrary removal); stack, queue, deque are similar (restricted insertion/removal).

std::list<T>	
list()	create empty list
list(**int** n)	create list with n default nodes
int size()	list size
bool empty()	size() == 0
list &**operator**=(**const** list &)	list assignment
T front()	first node
T back()	last node
void push_front(**const** T &x)	add x before first node
void push_back(**const** T &x)	add x after last node
void pop_front()	remove first node
void pop_back()	remove last node
void clear()	erase all nodes



- Iterator returns a pointer to a node.

std::list<T>::iterator / std::list<T>::reverse_iterator	
iterator begin()	iterator pointing to first node
iterator end()	iterator pointing **AFTER** last node
iterator rbegin()	iterator pointing to last node
iterator rend()	iterator pointing **BEFORE** first node
iterator insert(iterator posn, **const** T &x)	insert x before posn
iterator erase(iterator posn)	erase node at posn
++, -- (insertion order)	forward/backward operations

```
#include <list>
struct Node {
    char c;  int i;  double d;
    Node( char c, int i, double d ) : c(c), i(i), d(d) {}
};
list<Node> dl;                                    // doubly linked list
for ( int i = 0; i < 10; i += 1 ) {               // create list nodes
    dl.push_back( Node( 'a'+i, i, i+0.5 ) ); // push node on end of list
}
list<Node>::iterator f;
for ( f = dl.begin(); f != dl.end(); f ++ ) { // forward order
    cout << "c:" << (*f).c << " i:" << f->i << " d:" << f->d << endl;
}
while ( 0 < dl.size() ) {                         // destroy list nodes
    dl.erase( dl.begin() );                       // remove first node
} // same as dl.clear()
```

2.26.1.4 for_each

- Template routine for_each provides an alternate mechanism to iterate through a container.
- An action routine is called for each node in the container passing the node

to the routine for processing (Lisp apply).

```
#include <iostream>
#include <list>
#include <vector>
#include <algorithm>                                      // for_each
using namespace std;
void print( int i ) { cout << i << " "; }    // print node
int main() {
    list< int > int_list;
    vector< int > int_vec;
    for ( int i = 0; i < 10; i += 1 ) {          // create lists
        int_list.push_back( i );
        int_vec.push_back( i );
    }
    for_each( int_list.begin(), int_list.end(), print ); // print each node
    for_each( int_vec.begin(), int_vec.end(), print );
}
```

- Type of the action routine is **void** rtn(T), where T is the type of the container node.
- E.g., print has an **int** parameter matching the container node-type.
- More complex actions are possible using a functor.

- E.g., an action to print on a specified stream must store the stream and have an **operator**() allowing the object to behave like a function:

```
struct Print {
        ostream &stream;                        // stream used for output
        Print( ostream &stream ) : stream( stream ) {}
        void operator()( int i ) { stream << i << " "; }
};
int main() {
        list< int > int_list;
        vector< int > int_vec;
        . . .
        for_each( int_list.begin(), int_list.end(), Print(cout) );
        for_each( int_vec.begin(), int_vec.end(), Print(cerr) );
}
```

- Expression Print(cout) creates a constant Print object, and for_each calls **operator**()(Node) in the object.

2.27 Namespace

- C++ **namespace** is used to organize programs and libraries composed of multiple types and declarations ***to deal with naming conflicts***.

- E.g., namespace std contains all the I/O declarations and container types.
- Names in a namespace form a declaration region, like the scope of block.
- Analogy in Java is a package, but **namespace** does NOT provide abstraction/encapsulation (use .h/.cc files).
- C++ allows multiple namespaces to be defined in a file, as well as among files (unlike Java packages).
- Types and declarations do not have to be added consecutively.

Java source files	**C++ source file**

package Foo; *// file*
public class X . . . *// export one type*
// local types / declarations
	namespace Foo {
	    // types / declarations
	}
	namespace Foo {
	    // more types / declarations
	}
	namespace Bar {
	    // types / declarations
	}

package Foo; *// file* **public** enum Y . . . *// export one type* *// local types / declarations*	
package Bar; *// file* **public class** Z . . . *// export one type* *// local types / declarations*	

- Contents of a namespace are accessed using full-qualified names:

Java	**C++**
Foo.T t = **new** Foo.T();	Foo::T *t = **new** Foo::T();

- Or by importing individual items or importing all of the namespace content.

Java	**C++**

import Foo.T;
import Foo.*;
	using Foo::T; *// declaration*
	using namespace Foo; *// directive*

- **using** declaration ***unconditionally*** introduces an alias (like **typedef**) into the current scope for specified entity in namespace.
  - If name already exists in current scope, **using** fails.

    ```
    namespace Foo { int i = 0; }
    int i = 1;
    using Foo::i; // i exists in scope, conflict failure
    ```

  - May appear in any scope.
- **using** directive ***conditionally*** introduces aliases to current scope for all entities in namespace.
  - If name already exists in current scope, alias is ignored; if name already exists from **using** directive in current scope, **using** fails.

```
namespace Foo { int i = 0; }
namespace Bar { int i = 1; }
{
    int i = 2;
    using namespace Foo; // i exists in scope, alias ignored
}
{
    using namespace Foo;
    using namespace Bar; // i exists from using directive
    i = 0;    // conflict failure, ambiguous reference to ′i′
}
```

- ○ May appear in namespace and block scope, but not class scope.

```
namespace Foo {                 // start namespace
    enum Colour { R, G, B };
    int i = 3;
}
namespace Foo {                 // add more
    class C { int i; };
    int j = 4;
    namespace Bar {             // start nested namespace
        typedef short int shrint;
        char j = ′a′;
        int C();
    }
}
int j = 0;                      // external
int main() {
    int j = 3;                  // local
    using namespace Foo;        // conditional import: Colour, i, C, Bar (not j)
    Colour c;                   // Foo::Colour
    cout << i << endl;          // Foo::i
    C x;                        // Foo::C
    cout << ::j << endl;        // external
    cout << j << endl;          // local
    cout << Foo::j << " " << Bar::j << endl;  // qualification
    using namespace Bar;        // conditional import: shrint, C() (not j)
    shrint s = 4;               // Bar::shrint
    using Foo::j;              // disallowed : unconditional import
    C();                        // disallowed : ambiguous “class C” or “int C()”
}
```

- Never put a **namespace** in a header file (.h) (pollute local namespace) or before **#include** (can affect names in header file).

3 Tools

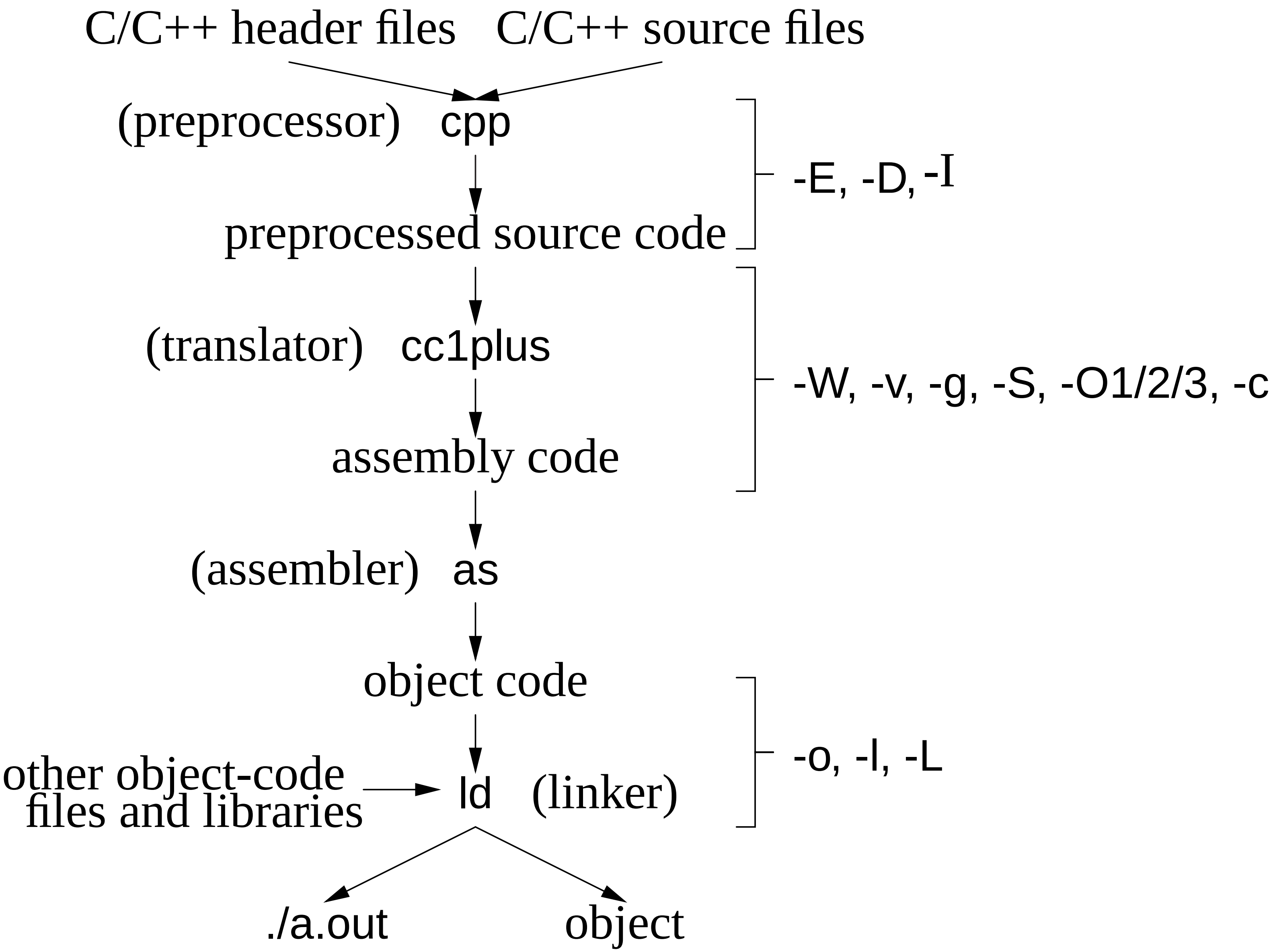
3.1 C/C++ Composition

- C++ is composed of 4 languages:
  1. The preprocessor language (cpp) modifies (text-edits) the program ***before*** compilation .
  2. The template (generic) language adds new types and routines ***during*** compilation .
  3. The C programming language specifying basic declarations and control flow to be executed ***after*** compilation.
  4. The C++ programming language specifying advanced declarations and control flow to be executed ***after*** compilation.
- A programmer uses the four programming languages as follows:

  user edits → **preprocessor edits** → **templates expand** → **compilation**
  (→ linking/loading → execution)

- C is composed of languages 1 & 3.
- The compiler interface controls all of these steps.

3.2 Compilation



- **Compilation** is the process of translating a program from human to machine readable form.
- The translation is performed by a tool called a **compiler**.
- Compilation is subdivided into multiple steps, using a number of tools.
- Often a number of options to control the behaviour of each step.
- Option are presented for g++, but other compilers have similar options.
- General format:

```
g++ option-list *.cc *.o ...
```

3.2.1 Preprocessor

- Preprocessor (cpp) takes a C++ source file, removes comments, and expands **#include**, **#define**, and **#if** directives.
- Options:
  - -E run only the preprocessor step and writes the preprocessor output to standard out.

```
$ g++ -E *.cc ...
... much output from the preprocessor
```

  - ◦ -D define and optionally initialize preprocessor variables from the compilation command:

    ```
    $ g++ -DDEBUG=2 -DASSN ... *.cc *.o ...
    ```

    same as putting the following **#define**s in a program without changing the program:

    ```
    #define DEBUG 2
    #define ASSN
    ```

- -I*directory* search directory for include files;
  - ◦ files within the directory can now be referenced by relative name using **#include** <file-name>.

3.2.2 Translator

- Translator takes a preprocessed file and converts the C++ language into assembly language for the target machine.
- Options:
  - ◦ -W*kind* generate warning message for this “*kind*” of situation.
    - ∗ -Wall print ALL warning messages.
    - ∗ -Werror make warnings into errors so program does not compile.

- -v show each compilation step and its details:

  ```
  $ g++ -v *.cc *.o ...
  ... much output from each compilation step
  ```

  E.g., system include-directories where cpp looks for system includes.

  ```
  #include <...> search starts here:
  /usr/include/c++/3.3
  /usr/include/c++/3.3/i486-linux
  /usr/include/c++/3.3/backward
  /usr/local/include
  /usr/lib/gcc-lib/i486-linux/3.3.5/include
  /usr/include
  ```

- -g add symbol-table information to object file for debugger
- -S compile source file, writing assemble code to file *source-file*.s
- -O1/2/3 optimize translation to different levels, where each level takes more compilation time and possibly more space in executable
- -c compile/assemble source file but do not link, writing object code to file *source-file*.o

3.2.3 Assembler

- Assembler (as) takes an assembly language file and converts it to object code (machine language).

3.2.4 Linker

- Linker (ld) takes the implicit .o file from translated source and explicit .o files from the command line, and combines them into a new object or executable file.
- Linking options:
  - -L*directory* is a directory containing library files of precompiled code.
  - -l*library* search in library directories for given *library*.
  - -o gives the file name where the combined object/ executable is placed.
    - If no name is specified, default name a.out is used.
- Look in library directory "/lib" for math library "m" containing precompiled "sin" routine used in "myprog.cc" naming executable program "calc".

```
$ gcc myprog.cc -L/lib -lm -o calc
```

3.3 Compiling Complex Programs

- As number of TUs grow, so do the references to type/variables (dependencies) among TUs.
- When one TU is changed, other TUs that depend on it must change and be recompiled.
- ***For a large numbers of TUs, the dependencies turn into a nightmare with respect to recompilation.***

3.3.1 Dependencies

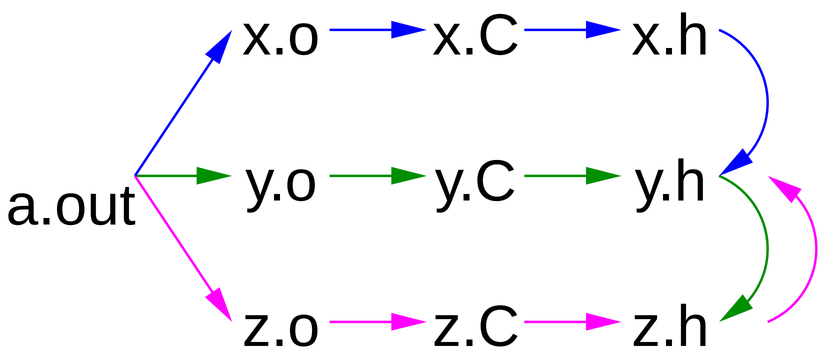
- A **dependence** occurs when a change in one location (entity) requires a change in another.
- Dependencies can be:
  - loosely coupled, e.g., changing source code may require a corresponding change in user documentation, or
  - tightly coupled, changing source code may require recompiling of some or all of the components that compose a program.
- Dependencies in C/C++ occur as follows:
  - executable depends on .o files (linking)

- ○ .o files depend on .C files (compiling)
- ○ .C files depend on .h files (including)

source code

x.h	**#include** "y.h"
x.C	**#include** "x.h"
y.h	**#include** "z.h"
y.C	**#include** "y.h"
z.h	**#include** "y.h"
z.C	**#include** "z.h"

dependence graph

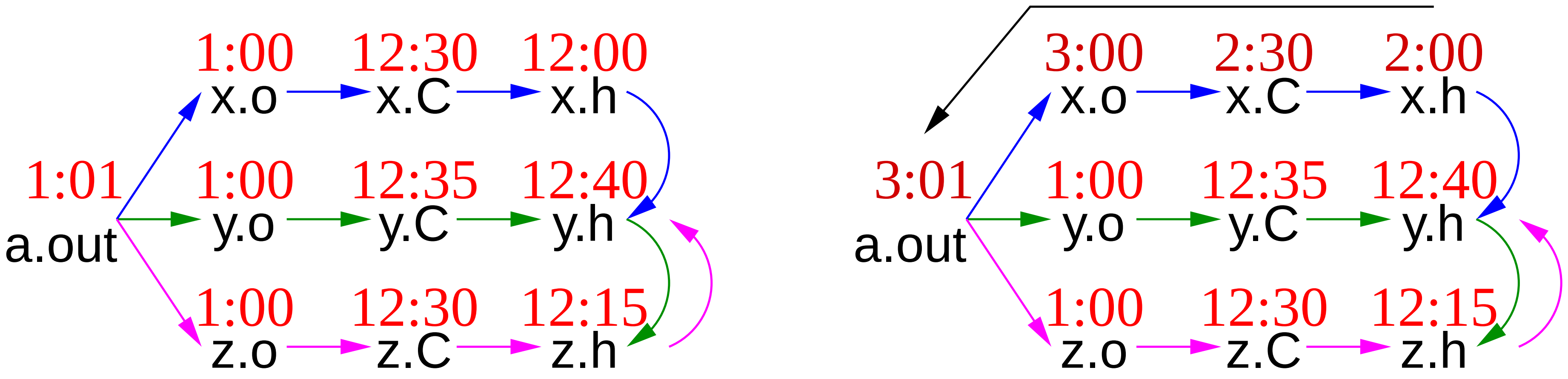


- Cycles in **#include** dependences are broken by **#ifndef** checks (see page 184).
- The executable (a.out) is generated by compilation commands:

```
$ g++ -c z.C          # generates z.o
$ g++ -c y.C          # generates y.o
$ g++ -c x.C          # generates x.o
$ g++ x.o y.o z.o     # generates a.out
```

- However, it is inefficient and defeats the point of separate compilation to recompile all program components after a change.

- If a change is made to y.h, what is the minimum recompilation necessary? (all!)
- Does ***any*** change to y.h require these recompilations?
- Often no mechanism to know the kind of change made within a file, e.g., changing a comment, type, variable.
- Hence, "change" may be coarse grain, i.e., based on ***any*** change to a file.
- One way to denote file change is with **time stamp**s.
- UNIX stores in the directory the time a file is last changed, with second precision.
- Using time to denote change means the dependency graph is a temporal ordering where the root has the newest (or equal) time and the leafs the oldest (or equal) time.

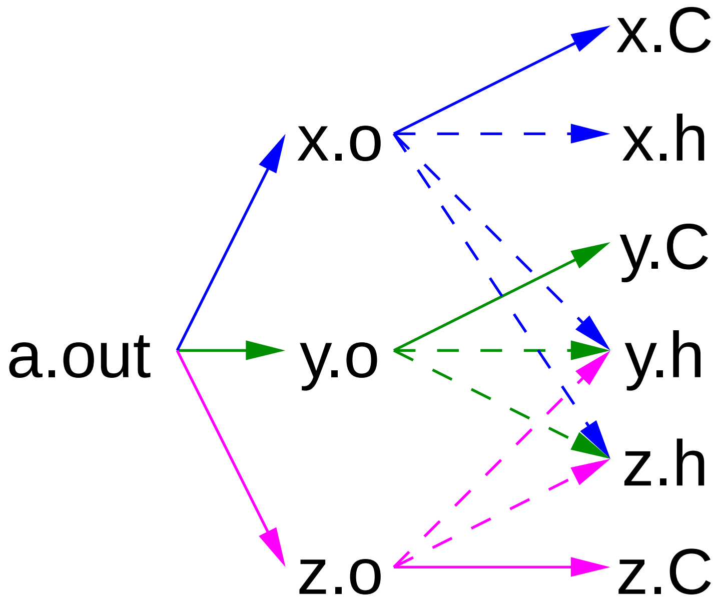


- ◦ Files x.o, y.o and z.o created at 1:00 from compilation of files created before 1:00.
- ◦ File a.out created at 1:01 from link of x.o, y.o and z.o.
- ◦ Changes are subsequently made to x.h and x.C at 2:00 and 2:30.
- ◦ Only files x.o and a.out need to be recreated at 3:00 and 3:01. (Why?)

3.3.2 Make

- **make** is a system command that takes a dependence graph and uses file change-times to trigger rules that bring the dependence graph up to date.
- A make dependence-graph expresses a relationship between a product and a set of sources.

- **make does not understand relationships among sources, one that exists at the source-code level and is crucial.**
- Hence, make dependence-graph loses some of the relationships (dashed lines):



- E.g., source x.C depends on source x.h but x.C is not a product of x.h like x.o is a product of x.C and x.h.
- Two most common UNIX makes are: make and gmake (on Linux, make is gmake).
- Like shells, there is minimal syntax and semantics for make, which is mostly portable across systems.

- Most common non-portable features are specifying dependencies and implicit rules.
- A basic makefile consists of string variables with initialization, and a list of targets and rules.
- This file can have any name, but make implicitly looks for a file called makefile or Makefile if no file name is specified.
- Each target has a list of dependencies, and possibly a set of commands specifying how to re-establish the target.

```
variable = value                              # variable
target : dependency1 dependency2 . . .        # target / dependencies
        command1                              # rules
        command2
        . . .
```

- ***Commands must be indented by one tab character.***
- make is invoked with a target, which is the root or subnode of a dependence graph.
- make builds the dependency graph and decorates the edges with time stamps for the specified files.

- If any of the dependency files (leafs) is newer than the target file, or if the target file does not exist, the commands are executed by the shell to update the target (generating a new product).
- Makefile for previous dependencies:

```
a.out : x.o y.o z.o
	g++ x.o y.o z.o -o a.out
x.o : x.C x.h y.h z.h
	g++ -g -Wall -c x.C
y.o : y.C y.h z.h
	g++ -g -Wall -c y.C
z.o : z.C z.h y.h
	g++ -g -Wall -c z.C
```

- Check dependency relationship (assume source files just created):

```
$ make -n -f Makefile a.out
g++ -g -Wall -c x.C
g++ -g -Wall -c y.C
g++ -g -Wall -c z.C
g++ x.o y.o z.o -o a.out
```

  All necessary commands are triggered to bring target a.out up to date.

  ○ -n builds and checks the dependencies, showing rules to be triggered

(leave off to execute rules)

- ○ -f Makefile is the dependency file (leave off if named [Mm]akefile)
- ○ a.out target name to be updated (leave off if first target)

- Generalize and eliminate duplication using variables:

```
CXX = g++                        # compiler
CXXFLAGS = -g -Wall -c           # compiler flags
OBJECTS = x.o y.o z.o            # object files forming executable
EXEC = a.out                     # executable name

${EXEC} : ${OBJECTS}             # link step
    ${CXX} ${OBJECTS} -o ${EXEC}
x.o : x.C x.h y.h z.h            # targets / dependencies / commands
    ${CXX} ${CXXFLAGS} x.C
y.o : y.C y.h z.h
    ${CXX} ${CXXFLAGS} y.C
z.o : z.C z.h y.h
    ${CXX} ${CXXFLAGS} z.C
```

- Eliminate common rules:
  - ○ make can deduce simple rules when dependency files have specific suffixes.

- E.g., given target with dependencies:

  ```
  x.o : x.C x.h y.h z.h
  ```

  `make` deduces the following rule:

  ```
      ${CXX} ${CXXFLAGS} -c -o x.o     # special variable names
  ```

  where `-o x.o` is redundant as it is implied by `-c`.
- This rule use variables `${CXX}` and `${CXXFLAGS}` for generalization.
- Therefore, all rules for `x.o`, `y.o` and `z.o` can be removed.

  ```
  CXX = g++                           # compiler
  CXXFLAGS = -g -Wall                 # compiler flags, remove -c
  OBJECTS = x.o y.o z.o               # object files forming executable
  EXEC = a.out                        # executable name

  ${EXEC} : ${OBJECTS}                # link step
      ${CXX} ${OBJECTS} -o ${EXEC}
  x.o : x.C x.h y.h z.h               # targets / dependencies
  y.o : y.C y.h z.h
  z.o : z.C z.h y.h
  ```

- Because dependencies are extremely complex in large programs, programmers seldom construct them correctly or maintain them.

- **Without complete and update dependencies, make is useless.**
- Automate targets and dependencies:

```
CXX = g++                               # compiler
CXXFLAGS = -g -Wall -MMD               # compiler flags
OBJECTS = x.o y.o z.o                   # object files forming executable
DEPENDS = ${OBJECTS:.o=.d}              # substitute ".o" with ".d"
EXEC = a.out                            # executable name

${EXEC} : ${OBJECTS}                    # link step
     ${CXX} ${OBJECTS} -o ${EXEC}

-include ${DEPENDS}                     # copies files x.d, y.d, z.d (if exists)

.PHONY : clean                          # not a file name
clean :                                 # remove files that can be regenerated
     rm -rf ${DEPENDS} ${OBJECTS} ${EXEC}   # alternative *.d *.o
```

  - ○ Preprocessor traverses all include files, so it knows all source-file dependencies.
  - ○ g++ flag -MMD writes out a dependency graph for user source-files to file *source-file*.d

file	contents
x.d	x.o: x.C x.h y.h z.h
y.d	y.o: y.C y.h z.h
z.d	z.o: z.C z.h y.h

- g++ flag -MD generates a dependency graph for user/system source-files.
- **-include** reads the .d files containing dependencies.
- .PHONY indicates a target that is not a file name and never created; it is a recipe to be executed every time the target is specified.
  - A phony target avoids a conflict with a file of the same name.
- Phony target clean removes product files that can be rebuilt (save space).

```
$ make clean              # remove all products (don′t create “clean”)
```

- Hence, it is possible to have a universal Makefile for a single or multiple programs.

3.4 Source-Code Management

- As a program develops/matures, it changes in many ways.
  - UNIX files do not support the temporal development of a program (**version control**), i.e., history of program over time.

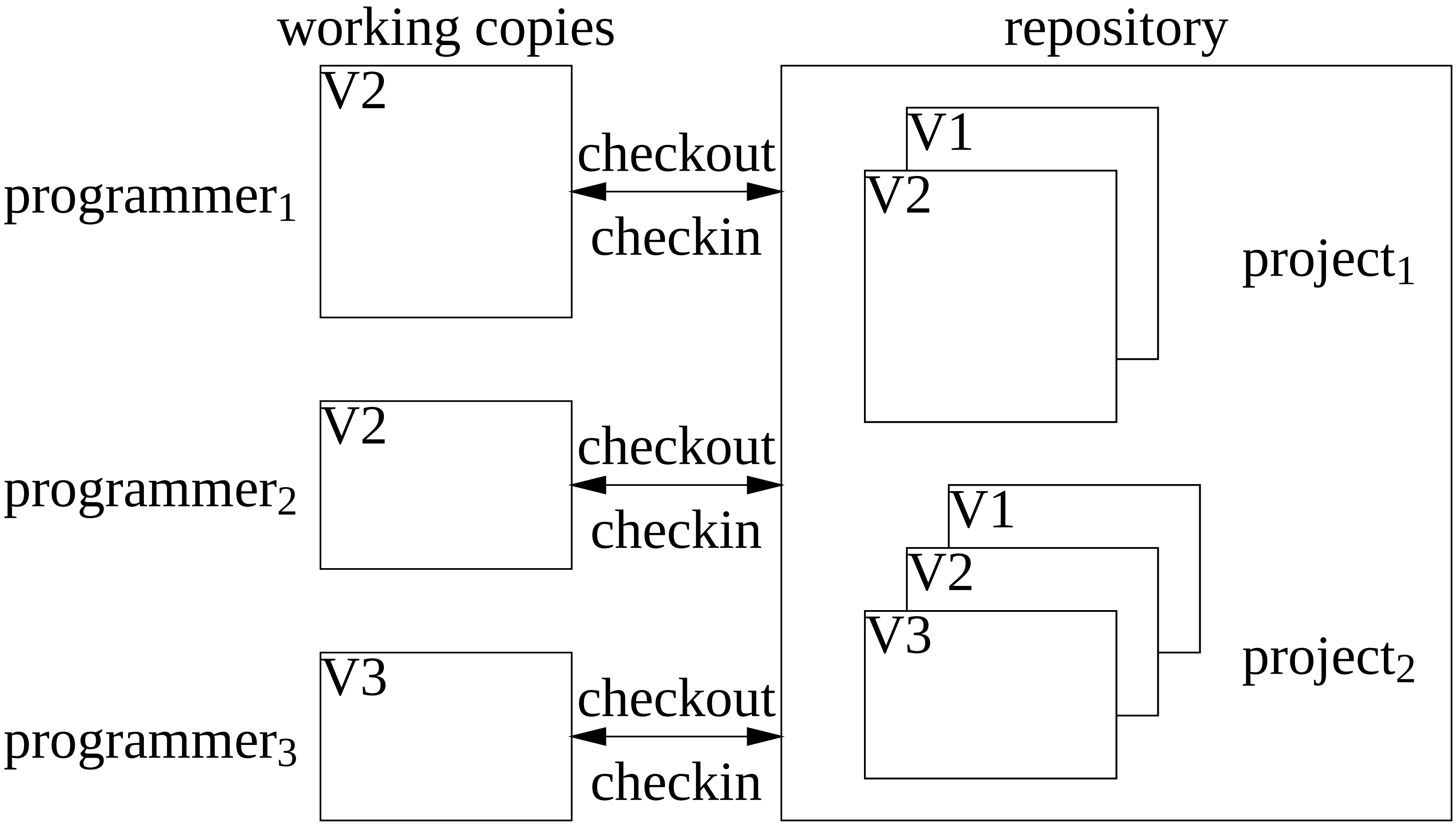
  - ○ Access to older versions of a program is useful, e.g., backing out of changes because of design problems.

- Program development is often performed by multiple developers each making independent changes.
  - ○ Sharing using files can damage file content for simultaneous writes.
  - ○ Merging changes from different developers is tricky and time consuming.
- To solve these problems, a **source-code management-system** is used to provide versioning and control cooperative work.

3.4.1 SVN

- **Subversion** (SVN 1.6) is a source-code management-system using the **copy-modify-merge model**.
  - ○ master copy of all **project** files kept in a **repository**,
  - ○ multiple versions of the project files managed in the repository,
  - ○ developers **checkout** a **working copy** of the project for modification,
  - ○ developers **checkin** changes from working copy with helpful integration using **text merging**.

  SVN works on file content not file time-stamps.



3.4.2 Repository

- The repository is a directory containing multiple projects.

```
courses                         repository
    cs246                       meta-project
        assn1                   project
            x.h, x.C, . . .     project files
        assn2                   project
            . . .               project files
    more meta-projects / projects
```

- svnadmin create command creates and initializes a repository.

  $ svnadmin create courses

- svn mkdir command creates subdirectories for meta-projects and projects.

```
$ svn mkdir file:///u/jfdoe/courses/cs246 -m "create directory cs246"
Committed revision 1.
$ svn mkdir file:///u/jfdoe/courses/cs246/assn1 -m "create subdirectory
Committed revision 2.
```

  - ○ files in repository are designated using URL, so must use absolute pathname
  - ○ -m (message) flag documents repository change.
  - ○ if no -m (message) flag specified, prompts for documentation (using an editor if shell environment variable EDITOR set).

- svn ls command lists directories.

```
$ svn ls file:///u/jfdoe/courses/cs246
assn1/
$ svn ls file:///u/jfdoe/courses/cs246/assn1
```

- If project directory assn1 already exists, it can be added directly to the repository.
- svn import command copies an unversioned directory of files into a repository.

```
$ svn import assn1 file:///u/jfdoe/courses/cs246/assn1
Adding          assn1/z.h
Adding          assn1/x.C
Adding          assn1/y.C
Adding          assn1/z.C
Adding          assn1/Makefile
Adding          assn1/x.h
Adding          assn1/y.h
Committed revision 2.
```

```
$ svn ls file:///u/jfdoe/courses/cs246/assn1
Makefile
x.C
x.h
y.C
y.h
z.C
z.h
```

- For students working together, the shared repository must be made accessible in the file system.

  ```
  $ chgrp -R cs246_75 courses  # set group on directory and subfiles
  $ chmod -R g+rwx courses     # allow group members access to ALL file
  ```

  and for the path to the repository.

- Group name cs246_75 is acquired on a per course basis for each team of students.

3.4.3 Checking Out

- svn checkout command extracts a working copy of a project from the repository.

```
$ svn checkout file:///u/jfdoe/courses/cs246/assn1
Checked out revision 2.
$ ls -AF assn1
.svn/
```

- For first checkout, directory assn1 is created in the current directory (unless it already exists).
- Subdirectory .svn contains administrative information for SVN and ***must not be modified***.
- Working copy is then modified before being merged back into the repository.
- Other developers do not see each others working copy, and will only see modifications when committed.
- To create a working-copy off-campus, use ssh URL:

  ```
  $ svn checkout svn+ssh://jfdoe@student.cs.uwaterloo.ca/u/jfdoe/cours
  ```

  (Replace file URL in subsequent commands with ssh URL.)

3.4.4 Adding

- Introduce files into project directory assn1.

```
$ cd assn1
$ . . .   # create files: Makefile   x.C   x.h   y.C   y.h   z.h z.C
$ ls -AF
.svn/   Makefile   x.C   x.h   y.C   y.h   z.C   z.h
```

- svn add command ***schedules*** addition of files (in current directory) into the repository.

```
$ svn add Makefile x.C x.h y.C y.h z.h z.C
A          Makefile
A          x.C
A          x.h
A          y.C
A          y.h
A          z.h
A          z.C
```

Addition only occurs on next commit.

- Forgetting svn add is a common mistake.
- ***Put only project source-files into repository.***
- Product files, e.g., *.o, *.d, a.out, do not need to be versioned.

3.4.5 Checking In

- svn commit command updates the repository with the changes in working copy.

```
$ svn commit -m "initial project files"
Adding         Makefile
Adding         x.C
Adding         x.h
Adding         y.C
Adding         y.h
Adding         z.C
Adding         z.h
Transmitting file data .......
Committed revision 3.
```

- if no -m (message) flag specified, prompts for commit documentation.

```
$ svn ls file:///u/jfdoe/courses/cs246/assn1
Makefile
x.C
x.h
y.C
y.h
z.C
z.h
```

- ***Always make sure your code compiles and runs before committing;*** it is unfair to pollute a project with bugs.

3.4.6 Modifying

- Editted files in working copy are implicitly ***scheduled*** for update on next commit.

```
$ vi y.h y.C
```

- svn rm command removes files from working copy and ***schedules*** removal of files from the repository.

```
$ ls -AF
.svn/  Makefile  x.C  x.h  y.C  y.h  z.C  z.h
$ svn rm z.h z.C
D         z.h
D         z.C
$ ls -AF
.svn/  Makefile  x.C  x.h  y.C  y.h
```

- svn status command displays changes between working copy and repository.

```
$ svn status
D       z.h
M       y.C
D       z.C
M       y.h
```

Files y.h / y.C have local modifications “M”, and z.h / z.C are deleted “D”.

- Possible to undo scheduled changes by reverting to files from repository.
- svn revert command copies unchanged files from repository to working copy.

```
$ svn revert y.C z.h
Reverted ′y.C′
Reverted ′z.h′
$ ls -AF
.svn/  Makefile  x.C  x.h  y.C  y.h  z.h
```

- Commit edits and removals.

```
$ svn commit -m "changes to y.h and remove z.C"
Sending        y.h
Deleting       z.C
Transmitting file data .
Committed revision 4.
$ svn ls file:///u/jfdoe/courses/cs246/assn1
Makefile
x.C
x.h
y.C
y.h
z.h
```

- Files in the repository can be renamed and copied.
- svn mv command renames file in working copy and ***schedules*** renaming in

the repository.

```
$ svn mv x.h w.h
A          w.h
D          x.h
$ ls -AF
.svn/  Makefile  w.h  x.C  y.C  y.h
```

- svn cp command copies file in working copy and ***schedules*** copying in the repository:

```
$ svn cp w.h k.h
A          k.h
$ ls -AF
.svn/  Makefile  k.h  w.h  x.C  y.C  y.h
```

- Commit renaming and copying.

```
$ svn commit -m "renaming and copying"
Adding         k.h
Adding         w.h
Deleting       x.h
Committed revision 5.
$ svn ls file:///u/jfdoe/courses/cs246/assn1
Makefile
k.h
w.h
x.C
y.C
y.h
```

3.4.7 Revision Number

- Each commit receives a revision number (currently 5).
- Information in older versions is accessible using suffix @N on URL.
- E.g., print file z.C, which last existed in revision 3.
- svn cat command prints specified file from the repository.

```
$ svn cat file:///u/jfdoe/courses/cs246/assn1/z.C@3
#include "z.h"
```

- Copy deleted file z.C from repository into working copy and modify.

```
$ svn copy file:///u/jfdoe/courses/cs246/assn1/z.C@3 z.C
A          z.C
$ ls -AF
.svn/  Makefile  k.h  w.h  x.C  y.C  y.h  z.C  z.h
$ . . . # change z.C
$ svn commit -m "bring back z.C and modify"
Adding          z.C
Transmitting file data .
Committed revision 6.
$ svn cat file:///u/jfdoe/courses/cs246/assn1/z.C@6
#include "z.h"
new text
```

3.4.8 Updating

- Synchronize working copy with commits in the repository from other developers.

jfdoe	kdsmith
modify x.C	modify x.C & y.C
	remove k.h
	add t.C

- Assume kdsmith has committed their changes.
- jfdoe attempts to committed their changes.

```
$ svn commit -m "modify x.C"
Sending        x.C
svn: Commit failed (details follow):
svn: File ′/cs246/assn1/x.C′ is out of date
```

- jfdoe must resolve differences between their working copy and the current revision in the repository.
- svn update command attempts to update working copy from most recent revision.

```
$ svn update
D    k.h            file k.h deleted
U    y.C            file y.C updated without conflicts
A    t.C            file t.C added
Conflict discovered in ′x.C′.
Select: (p) postpone, (df) diff-full, (e) edit,
        (mc) mine-conflict, (tc) theirs-conflict,
        (mf) mine-full, (tf) theirs-full,
        (s) show all options: df
--- .svn/text-base/x.C.svn-base     Sun May  2 09:54:08 2010
+++ .svn/tmp/x.C.tmp      Sun May  2 11:28:42 2010
@@ -1 +1,6 @@
 #include "x.h"
+<<<<<<< .mine
+jfdoe new text
+=======
+kdsmith new text
+>>>>>>> .r7
Select: (p) postpone, (df) diff-full, (e) edit, (r) resolved,
        (mc) mine-conflict, (tc) theirs-conflict,
        (mf) mine-full, (tf) theirs-full,
        (s) show all options: tc
G    x.C            file x.C merGed with kdsmith version
Updated to revision 7.
```

  - ◦ (p) postpone : mark conflict to be resolved later
  - ◦ (df) diff-full : show changes to merge file
  - ◦ (e) edit : change merged file in an editor
  - ◦ (r) resolved : after editing version
  - ◦ (mc) mine-conflict : accept my version for conflicts
  - ◦ (tc) theirs-conflict : accept their version for conflicts
  - ◦ (mf) mine-full : accept my file (no conflicts resolved)
  - ◦ (tf) theirs-full : accept their file (no conflicts resolved)
- Merge algorithm is generally very good if changes do not overlap.
- Overlapping changes result in a conflict, which must be resolved.
- If unsure about how to deal with a conflict, it can be postponed for each file.

```
$ svn update
D    k.h              file k.h deleted
U    y.C              file y.C updated without conflicts
A    t.C              file t.C added
Conflict discovered in ′x.C′.
Select: (p) postpone, (df) diff-full, (e) edit,
        (mc) mine-conflict, (tc) theirs-conflict,
        (mf) mine-full, (tf) theirs-full,
        (s) show all options: p
C    x.C              file x.C conflict
Updated to revision 7.
Summary of conflicts:
  Text conflicts: 1
```

- Working copy now contains the following files:

x.C	x.C.mine
#include "x.h" <<<<<<< .mine jfdoe new text ======= kdsmith new text >>>>>>> .r7	**#include** "x.h" jfdoe new text
x.C.r3	x.C.r7
#include "x.h"	**#include** "x.h" kdsmith new text

- ○ x.C : with conflicts
- ○ x.C.mine : jfdoe version of x.C
- ○ x.C.r3 : previous jfdoe version of x.C
- ○ x.C.r7 : kdsmith version of x.C in repository

- No further commits allowed until conflict is resolved.
- svn resolve --accept ARG command resolves conflict with version specified by ARG, for ARG options:
  - ○ base : x.C.r3 previous version in repository
  - ○ working : x.C current version in my working copy (***needs modification***)

  - mine-conflict : x.C.mine accept my version for conflicts
  - theirs-conflict : x.C.r7 accept their version for conflicts
  - mine-full : x.C.mine accept my file (no conflicts resolved)
  - theirs-full : x.C.r7 accept their file (no conflicts resolved)

```
$ svn resolve --accept theirs-conflict x.C
Resolved conflicted state of ′x.C′
```

- Removes 3 conflict files, x.C.mine, x.C.r3, x.C.r7, and sets x.C to the ARG version.

```
$ svn commit -m "modified x.C"
Sending        x.C
Transmitting file data .
Committed revision 8.
```

3.5 Debugger

- An interactive, symbolic **debugger** effectively allows debug print statements to be added and removed to/from a program dynamically.
- You should not rely solely on a debugger to debug a program.

- You may work on a system without a debugger or the debugger may not work for certain kinds of problems.
- A good programmer uses a combination of debug print statements and a debugger when debugging a complex program.
- A debugger does not debug your program for you, it merely helps in the debugging process.
- Therefore, you must have some idea about what is wrong with a program before starting to look or you will simply waste your time.

3.5.1 GDB

- The two most common UNIX debuggers are: dbx and gdb.
- File test.cc contains:

```
int r( int a[] ) {
    int i = 100000000;
    a[i] += 1;    // really bad subscript error
    return a[i];
}
int main() {
    int a[10] = { 0, 1 };
    r( a );
}
```

- Compile program using the -g flag to include names of variables and routines for symbolic debugging:

```
$ g++ -g test.cc
```

- Start gdb:

```
$ gdb ./a.out
. . .  gdb disclaimer
(gdb) ← gdb prompt
```

- Like a shell, gdb uses a command line to accept debugging commands.
- <Enter> without a command repeats the last command.
- **r**un command begins execution of the program:

```
(gdb) run
Starting program: /u/userid/cs246/a.out
Program received signal SIGSEGV, Segmentation fault.
0x000106f8 in r (a=0xffbefa20) at test.cc:3
3           a[i] += 1;      // really bad subscript error
```

- ○ If there are no errors in a program, running in GDB is the same as running in a shell.
- ○ If there is an error, control returns to gdb to allow examination.
- ○ If program is not compiled with -g flag, only routine names given.

- **ba**cktrace command prints a stack trace of called routines.

```
(gdb) backtrace
#0  0x000106f8 in r (a=0xffbefa08) at test.cc:3
#1  0x00010764 in main () at test.cc:8
```

- ○ stack has 2 frames main (#1) and r (#0) because error occurred in call to r.

- **p**rint command prints variables accessible in the current routine, object, or external area.

```
(gdb) print i
$1 = 100000000
```

- Can print any C++ expression:

```
(gdb) print a
$2 = (int *) 0xffbefa20
(gdb) p *a
$3 = 0
(gdb) p a[1]
$4 = 1
(gdb) p a[1]+1
$5 = 2
```

- **set var**iable command changes the value of a variable in the current routine, object or external area.

```
(gdb) set variable i = 7
(gdb) p i
$6 = 7
(gdb) set var a[0] = 3
(gdb) p a[0]
$7 = 3
```

Change the values of variables while debugging to:

  ◦ investigate how the program behaves with new values without recompile and restarting the program,

  - to make local corrections and then continue execution.

- **f**rame *[n]* command moves the **current stack frame** to the nth routine call on the stack.

```
(gdb) f 0
#0  0x000106f8 in r (a=0xffbefa08) at test.cc:3
3           a[i] += 1;      // really bad subscript error
(gdb) f 1
#1  0x00010764 in main () at test.cc:8
8           r( a );
```

  - If n is not present, prints the current frame
  - Once moved to a new frame, it becomes the current frame.
  - All subsequent commands apply to the current frame.

- To trace program execution, **breakpoint**s are used.

- **b**reak command establishes a point in the program where execution suspends and control returns to the debugger.

```
(gdb) break main
Breakpoint 1 at 0x10710: file test.cc, line 7.
(gdb) break test.cc:3
Breakpoint 2 at 0x106d8: file test.cc, line 3.
```

  - Set breakpoint using routine name or source-file:line-number.
  - info breakpoints command prints all breakpoints currently set.

```
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x00010710 in main at test.cc:7
2   breakpoint     keep y   0x000106d8 in r(int*) at test.cc:3
```

- Run program again to get to the breakpoint:

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /u/userid/cs246/a.out
Breakpoint 1, main () at test.cc:7
7           int a[10] = { 0, 1 };
(gdb) p a[7]
$8 = 0
```

- Once a breakpoint is reached, execution of the program can be continued in several ways.
- step *[n]* command executes the next n lines of the program and stops, so control enters routine calls.

```
(gdb) step
8          r( a );
(gdb) s
r (a=0xffbefa20) at test.cc:2
2          int i = 100000000;
(gdb) s
Breakpoint 2, r (a=0xffbefa20) at test.cc:3
3          a[i] += 1;     // really bad subscript error
(gdb) <Enter>
Program received signal SIGSEGV, Segmentation fault.
0x000106f8 in r (a=0xffbefa20) at test.cc:3
3          a[i] += 1;     // really bad subscript error
(gdb) s
Program terminated with signal SIGSEGV, Segmentation fault.
The program no longer exists.
```

  - ○ If n is not present, 1 is assumed.
  - ○ If the next line is a routine call, control enters the routine and stops at the first line.

- **n**ext *[n]* command executes the next n lines of the current routine and stops, so routine calls are not entered (treated as a single statement).

```
(gdb) run
...
Breakpoint 1, main () at test.cc:7
7           int a[10] = { 0, 1 };
(gdb) next
8           r( a );
(gdb) n
Breakpoint 2, r (a=0xffbefa20) at test.cc:3
3           a[i] += 1;     // really bad subscript error
(gdb) n
Program received signal SIGSEGV, Segmentation fault.
0x000106f8 in r (a=0xffbefa20) at test.cc:3
3           a[i] += 1;     // really bad subscript error
```

- continue *[n]* command continues execution until the next breakpoint is reached.

```
(gdb) run
...
Breakpoint 1, main () at test.cc:7
7         int a[10] = { 0, 1 };
(gdb) c
Breakpoint 2, r (a=0x7fffffffe7d0) at test.cc:3
3         a[i] += 1;    // really bad subscript error
(gdb) p i
$9 = 100000000
(gdb) set var i = 3
(gdb) c
Continuing.
Program exited normally.
```

- **l**ist command lists source code.

```
(gdb) list
1       int r( int a[] ) {
2               int i = 100000000;
3               a[i] += 1;      // really bad subscript error
4               return a[i];
5       }
6       int main() {
7               int a[10] = { 0, 1 };
8               r( a );
9       }
```

- ○ with no argument, list code around current execution location
- ○ with argument line number, list code around line number

- **q**uit command terminate gdb.

```
(gdb) run
. . .
Breakpoint 1, main () at test.cc:7
7               int a[10] = { 0, 1 };
1: a[0] = 67568
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

4 Software Engineering

- **Software Engineering** (SE) is the social process of designing, writing, and maintaining computer programs.
- SE attempts to find good ways to help people understand and develop software.
- However, what is good for people is not necessarily good for the computer.
- Many SE approaches are counter productive in the development of high-performance software.

  1. The computer does not execute the documentation!
     - Documentation is unnecessary to the computer, and significant amounts of time are spent building it so it can be ignored (program comments).
     - Remember, the ***truth*** is always in the code.
     - However, without documentation, developers have difficulty designing and understanding software.
  2. Designing by anthropomorphizing the computer is seldom a good approach (desktops/graphical interfaces).
  3. Compiler spends significant amounts of time ***undoing*** SE design and coding approaches to generate efficient programs.

- It is important to know these differences to achieve a balance between programs that are good for people and good for the computer.

4.1 Software Crisis

- Large software systems ($>$ 100,000 lines of code) require many people and months to develop.
- These projects too often emerge late, over budget, and do not work well.
- Today, hardware costs are low, and people costs are high.
- While commodity software is available, someone still has to write it.
- Since people produce software $\Rightarrow$ software cost is great.
- Coupled with a shortage of software personnel $\Rightarrow$ problems.
- Unfortunately, software is complex and precise, which requires time and patience.
- Errors occur and cost money if not lives, e.g., Ariane 5, Therac–25, Intel Pentium division error, Mars Climate Orbiter, UK Child Support Agency, etc.

4.2 Software Development

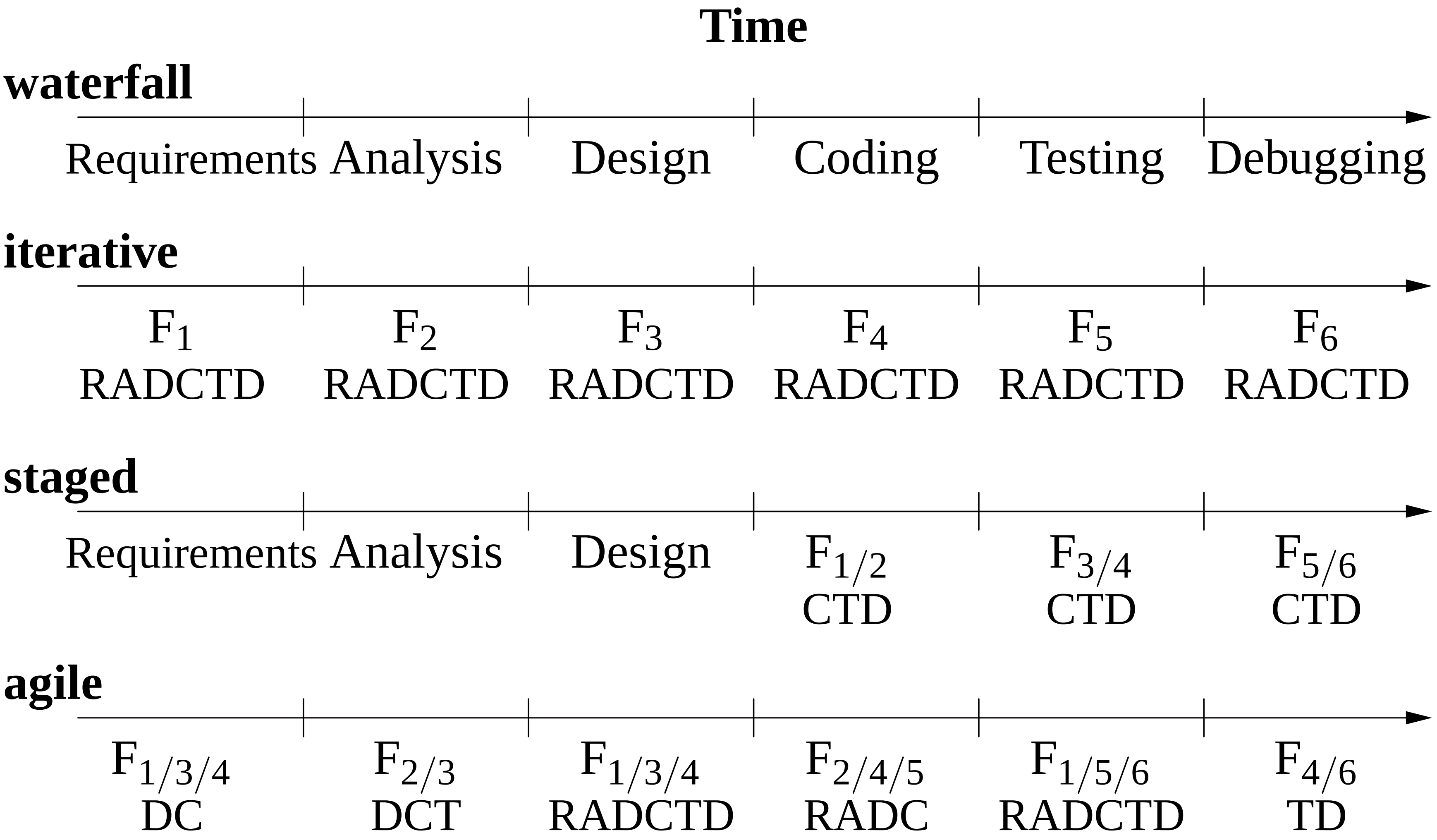
- Techniques for program development for small, medium, and large systems.
- Objectives:
  - plan and schedule project (requirements documents, UML, time-lines)
  - produce reliable, flexible, efficient programs
  - produce programs that are easily maintained
  - reduce the cost of software
  - reduce program failure
- E.g., a typical software project:
  - estimate 12 months of work
  - hire 3 people for 4 months
  - make up milestones for the end of each month
- However, first milestone is reached after 2 months instead of 1.
- To finish on time, hire 2 more people, but:
  - new people require training
  - work must be redivided

  This takes at least 1 month.

- Now 2 months behind with 9 months of work to be done in 1 month by 5 people.
- To get the project done:
  - must reschedule
  - trim project goals
- Often, adding manpower to a late software project makes it later.
- Illustrates the need for a methodology to aid in the development of software projects.

4.3 Development Processes

- There are different conceptual approaches for developing software:



Time

waterfall

Requirements Analysis | Design | Coding | Testing | Debugging

iterative

F_1 RADCTD | F_2 RADCTD | F_3 RADCTD | F_4 RADCTD | F_5 RADCTD | F_6 RADCTD

staged

Requirements Analysis | Design | $F_{1/2}$ CTD | $F_{3/4}$ CTD | $F_{5/6}$ CTD

agile

$F_{1/3/4}$ DC | $F_{2/3}$ DCT | $F_{1/3/4}$ RADCTD | $F_{2/4/5}$ RADC | $F_{1/5/6}$ RADCTD | $F_{4/6}$ TD

waterfall : break down project based on activities that flow (down stream) across a timeline.

- activities : (cycle of) requirements, analysis, design, coding, testing, debugging (RADCTD).
- timeline : assign time to accomplish each activity up to project

completion time

iterative/spiral : break down project based on functionality and divide functions across a timeline

- ○ functions : (cycle of) acquire/verify data, process data, generate data reports
- ○ timeline : assign time to perform software cycle on each function up to project completion time

staged delivery : combination of waterfall and iterative

- ○ start with waterfall for analysis/design, and finish with iterative for coding/testing

agile/extreme : short, intense iterations focused largely on code (versus documentation)

- ○ often analysis and design are done iteratively
- ○ often coding/testing done in pairs

- Pure waterfall is problematic because all coding/testing comes at end $\Rightarrow$ major problems can appear near project deadline.
- Pure agile can leave a project with "just" working code, and little or no testing / documentation.
- Selecting a process depends on:

  - kind/size of system
  - quality of system (mission critical?)
  - hardware/software technology used
  - kind/size of programming team
  - working style of teams
  - nature of completion risk
  - consequences of failure
  - culture of company

- Meta-processes specifying the effectiveness of processes:
  - Capability Maturity Model Integration (CMMI)
  - International Organization for Standardization (ISO) 9000
- Meta-requirements
  - procedures cover key aspects of processes
  - monitoring mechanisms
  - adequate records
  - checking for defects, with appropriate and corrective action
  - regularly reviewing processes and its quality
  - facilitating continual improvement

4.4 Software Methodology

- System Analysis (next year)
  - Study the problem, the existing systems, the requirements, the feasibility.
  - Analysis is a set of requirements describing the system inputs, outputs, processing, and constraints.
- System Design
  - Breakdown of requirements into modules, with their relationships and data flows.
  - Results in a description of the various modules required, and the data interrelating these.
- Implementation
  - writing the program
- Testing & Debugging
  - get it working
- Operation & Review
  - was it what the customer wanted and worth the effort?
- Feedback

  - If possible, go back to the above steps and augment the project as needed.

4.4.1 System Design

- Two basic strategies exist to systematically modularize a system:
  - top-down or functional decomposition
  - bottom-up
- Both techniques have much in common and so examine only one.

4.4.2 Top-Down

- Start at highest level of abstraction and break down problem into cohesive units, i.e., divide & conquer.
- Then refine each unit further generating more detail at each division.
- Each subunit is divided until a level is reached where the parts are comprehensible, and can be coded directly.
- This recursive process is called **successive refinement** or **factoring**.
- Unit are independent of a programming language, but ultimately must be mapped into constructs like:
  - generics (templates)

  - modules
  - classes
  - routines

- Details look at data and control flow within and among units.
- Implementation programming language is often chosen only after the system design.
- Factoring goals:
  - reduce module size : $\approx$ 30-60 lines of code, i.e., 1-2 screens with documentation
  - make system easier to understand
  - eliminate duplicate code
  - localize modifications
- Stop factoring when:
  - cannot find a well defined function to factor out
  - interface becomes too complex
- Avoid having the same function performed in more than one module (create useful general purpose modules)

- Separate work from management:
  - Higher-level modules only make decisions (management) and call other routines to do the work.
  - Lower-level modules become increasingly detailed and specific, performing finer grain operations.
- In general:
  - do not worry about little inefficiencies unless the code is executed a LARGE number of times
  - put thought into readability of program

4.5 Design Quality

- System design is a general plan for attacking a problem, but leads to multiple solutions.
- Need the ability to compare designs.
- 2 measures: coupling and cohesion
- Low (loose) coupling is a sign of good structured and design; high cohesion supports readability and maintainability.

4.5.1 Coupling

- **Coupling** measures the degree of interdependence among programming “modules”.
- Aim is to achieve lowest coupling or highest independence (i.e., each module can stand alone or close to it).
- A module can be read and understood as a unit, so that changes have minimal effect on other modules and possible to isolate it for testing purposes (like stereo components).
- 5 types of coupling in order of loose to tight (low to high):

  1. **Data** : modules communicate using arguments/parameters containing minimal data.
     - E.g., `sin( x )`, `avg( marks )`
  2. **Stamp** : modules communicate using only arguments/parameters containing extra data.
     - E.g., pass aggregate data (array/structure) with some elements/fields unused
     - problem: accidentally change other data
     - modules may be less general (e.g., average routine passed an array of records)

- ◦ stamp coupling is common because data grouping is more important than coupling

3. **Control** : pass data using arguments/parameters to effect control flow.
   - ◦ E.g., module calculate 2 different things depending on a flag
   - ◦ bad when flag is passed down, worse when flag is passed up
4. **Common** : modules share global data.
   - ◦ cannot control access since scope rule allows many modules to access the global variables
   - ◦ difficult to find all references reading/writing global variables
5. **Content** : modules share information about type, size and structure of data, or methods of calculation
   - ◦ changes effect many different modules (good/bad)
   - ◦ avoid **friend** routine/class unless friend module is logically nested but extracted for technical reasons.

4.5.2 Cohesion

- **Cohesion** measures degree of association among elements within a module (how focused).
- Elements can be a statement, group of statements, or calls to other modules.

- Alternate names for cohesion: binding, functionality, modular strength.
- Highly cohesive module has strongly and genuinely related elements.
- If modules have low cohesion (module elements are related) ⇒ tight coupling.
- If modules have high cohesion (module elements are NOT related) ⇒ loose coupling.
- 7 types of cohesion (high to low):

1. **Functional** : modules elements all contribute to computation of one and only one problem related task (Single Responsibility Principle).
   - ○ E.g., sin(x), avg(marks), Car {. . .}, Driver {. . .}
   - ○ coupling is excellent
2. **Sequential** : module elements interact as producer/consumer, i.e., output data from one activity is input data to next.

   ```
   print( process( getword( word ) ) ); // read -> process -> print (shell pipe
   ```

   - ○ similar to functional, except possibly mandates sequences of use
   - ○ coupling is good
3. **Communicational** : module elements contribute to activities that use the same data.

```
find( book, title );
find( book, price );
find( book, ISBN );
find( book, author );
```

- ○ all have same input data
- ○ like sequential but order is not important
- ○ coupling is acceptable
- ○ usually improve maintainability by splitting common module into separate, functional ones

4. **Procedural** : module elements involved in different and possibly unrelated activities, but which flow from one activity to the next.

```
file = open( filename );          // open connection to file name
read( file );                     // read file contents
close( file );                    // close connection to file name
```

- ○ related by order of execution rather than by any single problem-related function
- ○ typically data sent to procedure modules is unrelated to data sent back
- ○ procedural modules pass around partial results

5. **Temporal** : module elements involved in activities related in time.

```
initialization
   - turn things on
   - turn things off
   - set things to 0
   - set things to 1
   - set things to ′ ′
```

- unrelated except carried out at particular time
- each initialization is more closely related to the modules that make use of it ⇒ tight coupling
- want to re-initialize only some of the entities in initialization routine
- like procedural, except order of execution is more important in procedural

6. **Logical** : module elements contribute to same general category, where activity is selected from outside the module.

```
#include <algorithms>
find ...
swap ...
search ...
sort ...
inner_product ...
```

- modules contain number of activities of some general kind

  - ○ to use, pick out just one of the pieces needed
  - ○ interface weak, and contains code sharing common lines of code and/or data areas

7. **Coincidental** : module elements grouped arbitrarily.
   - ○ activities are related neither by flow of data nor control
   - ○ like logical, internal activity must be externally selected, but worse since categories in the module are very weakly related

4.6 Design Principles

- low coupling, high cohesion (logical modularization)
- good interfaces (abstraction and encapsulation)
- type reuse (type inheritance)
- code reuse (implementation inheritance, physical modularization)
- indirection (data/routine pointers) to generalize objects

4.7 Design Patterns

- **Design patterns** have existed since people/trades developed formal approaches.

- E.g., chef's cooking meals, musician's writing/playing music, mason's building pyramid/cathedral.
- **Pattern** is a common/repeated issue; it can be a problem or a solution.
- Name and codify common patterns for educational and communication purposes.
- Software pattern are solutions to problems:
  - name : descriptive name
  - problem : kind of issues pattern can solve
  - solution : general elements composing the design, with relationships, responsibilities, and collaborations
  - consequences : results/trade-offs of pattern (alternative/implementation issues)
- Patterns help:
  - extend developers' vocabulary

    > **Squadron Leader** : Top hole. Bally Jerry pranged his kite right in the how's your father. Hairy blighter, dicky-birdied, feathered back on his Sammy, took a waspy, flipped over on his Betty Harper's and caught his can in the Bertie.
    > – RAF Banter, Monty Python

- ○ offer higher-level abstractions than routines or classes

4.7.1 Pattern Catalog

	creational	structural	behavioural
class	**factory method**	**adapter**	interpreter **template**
object	**abstract factory** builder prototype **singleton**	**adapter** bridge **composite** **decorator** facade flyweight **proxy**	responsibility chain command **iterator** mediator memento **observer** state strategy **visitor**

- Scope : applies to classes or objects
  - ○ **class pattern** – relationships among classes and subclasses (static inheritance)
  - ○ **object pattern** – relationships among objects (dynamic creation and association)

- Purpose : what a pattern does
  - creational : classes defer construction through inhertiance / objects defer creation to other objects
  - structural : composition via inherited classes or assembled objects
  - behavioural : classes describes algorithm or control-flow / objects cooperate to perform task

4.7.1.1 Class Patterns

factory method : generalize creation of product with multiple variants

```
struct Pizza {...};                           // product
struct Pizzeria {                             // creator
    enum Kind { It, Mg, Ch, Dd };             // styles
    virtual Pizza *order( Kind p ) = 0;
};
struct Italian : public Pizzeria {            // concrete creator (factory)
    Pizza *order( Kind p );           // create italian/margarita style
};
struct Chicago : public Pizzeria {            // concrete creator
    Pizza *order( Kind p );           // create chicago/deep-dish style
};
```

```
Italian italian;  Chicago chicago;          // factories
enum Kind { It, Mg, Ch, Dd };
Pizza *dispatch( Kind pizza ) {             // parameterized creator
    switch ( pizza ) {
      case It: case Mg: return italian.order( Pizzeria::Mg );
      case Ch: case Dd: return chicago.order( Pizzeria::Dd );
      default: ; // error
    }
}
Pizza *p = dispatch( It );
p = dispatch( Ch );
```

- product (Pizza) objects are consistent across all factories (could be subclassed)
- clients get a concrete product (Pizza) from the creator (directly or indirectly), but product type is unknown
- client interacts with product object through its abstract interface (Pizza)

adapter/**wrapper** : convert interface into another

```
struct Stack {                          struct Vector {
    virtual void push(...);                 virtual push_back(...);
    virtual void pop(...);                  virtual pop_back(...);
};                                      };
struct VStack : public Stack, private Vector {  // adapter/wrapper
    void push(...) { ... push_back(...); ... }
    void pop(...) { pop_back(...); }
};
void p( Stack &s ) { ... }
VStack vs;   // use VStack code with Stack routine
p( vs );
```

- VStack is polymorphic with Stack but implements push/pop with Vector::push_back/ Vector::pop_back.

template method : provide algorithm but defer some details to subclass

```
class PriceTag {                          // template method
    virtual string label() = 0;           // details for subclass
    virtual string price() = 0;
    virtual string currency() = 0;
  public:
    string tag() { return label() + price() + currency(); }
};
class FurnitureTag : public PriceTag {  // actual method
    string label() { return "furniture "; }
    string price() { return "$1000 "; }
    string currency() { return "Cdn"; }
};
FurnitureTag ft;
cout << ft.tag() << endl;
```

- template-method routines are non-virtual, i.e., not overridden

4.7.1.2 Object Patterns

abstract factory : generalize creation of family of products with multiple variants

```
struct Food {...};                        // abstract product
struct Pizza : public Food {...};         // concrete product
struct Burger : public Food {...};        // concrete product
struct Restaurant {                       // abstract factory product
    enum Kind { Pizza, Burger };
    virtual Food *order( Kind f ) = 0;
    virtual int staff() = 0;
};
struct Pizzeria : public Restaurant {    // concrete factory product
    Food *order( Kind f ) {}
    int staff() {...}
};
struct Burgers : public Restaurant {     // concrete factory product
    Food *order( Kind f ) {}
    int staff() {...}
};
```

```
enum Type { PizzaHut, BugerKing };
struct RestaurantFactory {                // abstract factory
    Restaurant *create( Type t ) {}
};
struct PizzeriaFactory : RestaurantFactory { // concrete factory
    Restaurant *create( Type t ) {}
};
struct BurgerFactory : RestaurantFactory { // concrete factory
    Restaurant *create( Type t ) {}
};
PizzeriaFactory pizzeriaFactory;
BurgerFactory burgerFactory;
Restaurant *pizzaHut = pizzeriaFactory.create( PizzaHut );
Restaurant *burgerKing = burgerFactory.create( BugerKing );
Food *dispatch( Restaurant::Kind food ) { // parameterized creator
    switch ( food ) {
      case Restaurant::Pizza: return pizzaHut->order( Restaurant::Pizza );
      case Restaurant::Burger: return burgerKing->order( Restaurant::Burg
      default: ; // error
    }
}
```

- use factory-method pattern to construct generated product (Food)

- use factory-method pattern to construct generated factory (Restaurant)
- clients obtains a concrete product (Pizza, Burger) from a concrete factory (PizzaHut, BugerKing), but product type is unknown
- client interacts with product object through its abstract interface (Food)

singleton : single instance of class

.h file

```
class Singleton {
      struct Impl {
            int x, y;
            Impl( int x, int y );
      };
      static Impl impl;
   public:
      void m();
};
```

.cc file

```
#include "Singleton.h"
Singleton::Impl Singleton::impl( 3, 4 );
Singleton::Impl::Impl( int x, int y )
      : x(x), y(y) {}
void Singleton::m() { . . . }
```

```
Singleton x, y, z;       // all access same value
```

- Allow different users to have they own declaration but still access same value.

```
Database database;   // user 1
Database db;         // user 2
Database info;       // user 3
```

- Alternative is global variable, which forces name and may violate abstraction.

composite : interface for complex composite object

```
struct Assembly {            // composite type
    string partNo();
    string name();
    double price();
    void insert( Assembly assm );
    void remove( string partNo );
    struct Iterator {...};
};
class Engine : public Assembly {...};
class Transmission : public Assembly{...};
class Wheel : public Assembly {...};
class Car : public Assembly {...};
class Stove : public Assembly {...};
// create parts for car
Car c;                       // composite object
c.insert( engine );
c.insert( transmission );
c.insert( wheel );
c.insert( wheel );
```

- recursive assembly type creates arbitrary complex assembly object.

- vertices are subassemblies; leafs are parts
- since composite type defines both vertices and leaf, all members may not apply to both

iterator : abstract mechanism to traverse composite object

```
double price = 0.0;
Assembly::Iterator c( car );
for ( part = c.begin( engine ); part != c.end(); ++part ) { // engine cost
    price += part->price();
}
```

- iteration control: multiple starting/ending locations; depth-first/breath-first, forward/backward, etc.; level of traversal
- iterator may exist independently of a composite design-pattern

adapter : convert interface into another

```
struct Stack {                          struct Vector {
    virtual void push(...);                 virtual push_back(...);
    virtual void pop(...);                  virtual pop_back(...);
};                                      };
struct VecToStack : public Stack {  // adapter/wrapper
    Vector &vec;
    VectortoStack( Vector &vec ) : vec( vec ) {}
    void push(...) { ... vec.push_back(...); ... }
    void pop(...) { vec.pop_back(...); }
};
void p( Stack &s ) { ... }
Vector vec;
VecToStack vtos( vec );            // any Vector
p( vtos );
```

- specific conversion from Vector to Stack

proxy : frontend for another object to control access

```
struct DVD {
    void play(...);
    void pause(...);
};
struct SPVR : public DVD {              // static
    void play(...) { ... DVD::play(...); ... }
    void pause(...) { ... DVD::pause(...); ... }
};
struct DPVR : public DVD {              // dynamic
    DVD *dvd;
    DPVR() { dvd = NULL; }
    ~DPVR() { if ( dvd != NULL ) delete dvd; }
    void play(...) { if ( dvd == NULL ) dvd = new T; dvd->play(...); ... }
    void pause(...) { ... don't need dvd, no pause ... }
};
```

- proxy extends object's type
- reverse structure of template method
- dynamic approach lazily creates control object

decorator : attach additional responsibilities to an object dynamically

```
struct Window {
    virtual void move(. . .) {. . .}
    virtual void lower(. . .) {. . .}
    . . .
};                                   };
struct Scrollbar : public Window {       // specialize
    enum Kind { Hor, Ver };
    Window &window;
    Scrollbar( Window &window, Kind k ) : window( &window ), . . . {}
    void scroll( int amt ) {. . .}
};
struct Title : public Window {           // specialize
    . . .
    Title( Window &window, . . . ) : window( window ), . . . {}
    setTitle( string t ) {. . .}
};
Window w;
Title( Scrollbar( Scrollbar( w, Ver ), Hor ), "title" ) decorate;
```

- decorator only mimics object’s type through base class
- allows decorator to be dynamically associated with different object’s, or

same object to be associated with multiple decorators

observer : 1 to many dependency ⇒ change updates dependencies

```
struct Fan {                                // abstract
    Band &band;
    Fan( Band &band ) : band( band ) {}
    virtual void update( CD cd ) = 0;
};
struct Band {
    list<Fan *> fans;                       // list of fans
    static void perform( Fan *fan ) { fan->update(); }
    void attach( Fan &fan ) { fans.push_back( &fan ); }
    void deattach( Fan &fan ) { fans.remove( &fan ); }
    void notify() { for_each( fans.begin(), fans.end(), perform ); }
};
struct Groupie : public Fan {               // specialize
    Groupie( Band &band ) : Fan( band ) { band.attach( *this ); }
    ~Groupie() { band.deattach( *this ); }
    void update( CD cd ) { buy/listen new cd }
};
Band dust;
Groupie g1( dust ), g2( dust );             // register
dust.notify();                              // inform fans about new CD
```

- manage list of interested objects, and push new events to each

- alternative design has interested objects pull the events from the observer
    - ⇒ observer must store events until requested

visitor : perform operation on elements of heterogeneous container

```
struct PrintVisitor {
    void visit( Wheel &w ) { print wheel }
    void visit( Engine &e ) { print engine }
    void visit( Transmission &t ) { print transmission }
    . . .
};
struct Part {
    virtual void action( Visitor &v ) = 0;
};
struct Wheel : public Part {
    void action( Visitor &v ) { v.visit( *this ); } // overload
};
struct Engine : public Part {
    void action( Visitor &v ) { v.visit( *this ); } // overload
};
. . .
```

```
PrintVisitor pv;
list<Part *> ps;
for ( int i = 0; i < 10; i += 1 ) {
    ps.push_back( add different car parts );
}
for ( list<Part *>::iterator pi = ps.begin(); pi != ps.end(); ++pi ) {
    (*pi)->action( pv );
}
```

- each part has a general action that is specialized by visitor
- different visitors perform different actions or dynamically vary the action
- compiler statically selects appropriate overloaded version of visit in action

4.8 Testing

- A major phase in program development is testing ($>$ 50%).
- This phase often requires more time and effort than design and coding phases combined.
- Testing is not debugging.
- **Testing** is the process of "executing" a program with the intent of determining differences between the specification and actual results.

  - Good test is one with a high probability of finding a difference.
  - Successful test is one that finds a difference.
- Debugging is the process of determining why a program does not have an intended testing behaviour and correcting it.

4.8.1 Human Testing

- **Human Testing** : systematic examination of program to discover problems.
- Studies show 30–70% of logic design and coding errors can be detected in this manner.
- **Code inspection** team of 3-6 people led by moderator (team leader) looking for problems, often "grilling" the developer(s):
  - data errors: wrong types, mixed mode, overflow, zero divide, bad subscript, initialization problems, poor data-structure
  - logic errors: comparison problems (== / !=, < / <=), loop initialization / termination, off-by-one errors, boundary values, incorrect formula, end of file, incorrect output
  - interface errors: missing members or member parameters, encapsulation / abstraction issues

- **Walkthrough** : less formal examination of program, possibly only 2-3 developers.
- **Desk checking** : single person “plays computer”, executing program by hand.

4.8.2 Machine Testing

- **Machine Testing** : systematic running of program using test data designed to discover problems.
  - speed up testing, occur more frequently, improve testing coverage, greater consistency and reliability, use less people-time testing
- Commercial products are available.
- Should be done after human testing.
- Exhaustive testing is usually impractical (too many cases).
- **Test-case design** involves determining subset of all possible test cases with the highest probability of detecting the greatest number of errors.
- Two major approaches:
  - **Black-Box Testing** : program’s design / implementation is unknown when test cases are drawn up.

  - **White-Box Testing** : program's design / implementation is used to develop the test cases.
  - **Gray-Box Testing** : only partial knowledge of program's design / implementation know when test cases are drawn up.
- Start with the black-box approach and supplement with white-box tests.
- Black-Box Testing
  - **equivalence partitioning** : completeness without redundancy
    - partition all possible input cases into equivalence classes
    - select only one representative from each class for testing
    - E.g., payroll program with input HOURS

```
           HOURS <= 40
      40 < HOURS <= 45   (time and a half)
      45 < HOURS         (double time)
```

    - 3 equivalence classes, plus invalid hours
    - Since there are many types of invalid data, invalid hours can also be partitioned into equivalence classes
  - **boundary value testing**
    - test cases which are below, on, and above boundary cases

39, 40, 41	(hours)	valid cases
44, 45, 46	”	
0, 1, 2	”	
-2, -1, 0	”	invalid cases
59, 60, 61	”	

- ○ **error guessing**
  - ∗ surmise, through intuition and experience, what the likely errors are and then test for them

- White-Box (logic coverage) Testing
  - ○ develop test cases to cover (exercise) important logic paths through program
  - ○ try to test every decision alternative at least once
  - ○ test all combinations of decisions (often impossible due to size)
  - ○ test every routine and member for each type
  - ○ cannot test all permutations and combinations of execution
- **Test Harness** : a collection of software and test data configured to run a program (unit) under varying conditions and monitor its outputs.

4.8.3 Testing Strategies

- **Unit Testing** : test each routine/class/module separately before integrated into, and tested with, entire program.
  - requires construction of drivers to call the unit and pass it test values
  - requires construction of stub units to simulate the units called during testing
  - allows a greater number of tests to be carried out in parallel
- **Integration Testing** : test if units work together as intended.
  - after each unit is tested, integrate it with tested system.
  - done top-down or bottom-up : higher-level code is drivers, lower-level code is stubs
  - In practice, a combination of top-down and bottom-up testing is usually used.
  - detects interfacing problems earlier
- Once system is integrated:
  - **Functional Testing** : test if performs function correctly.
  - **Regression Testing** : test if new changes produce different effects from previous version of the system (diff results of old / new versions).
  - **System Testing** : test if program complies with its specifications.

  - ○ **Performance Testing** : test if program achieves speed and throughput requirements.
  - ○ **Volume Testing** : test if program handles difference volumes of test data (small ⇔ large), possibly over long period of time.
  - ○ **Stress Testing** : test if program handles extreme volumes of data over a short period of time with fixed resources, e.g., can air-traffic control-system handle 250 planes at same time?
  - ○ **Usability Testing** : test whether users have the skill necessary to operate the system.
  - ○ **Security Testing** : test whether programs and data are secure, i.e., can unauthorized people gain access to programs, files, etc.
  - ○ **Acceptance Testing** : checking if the system satisfies what the client ordered.
- If a problem is discovered, make up additional test cases to zero in on the issue and ultimately add these tests to the test suite for regression testing.

4.8.4 Tester

- A program should not be tested by its writer, but in practice this often occurs.

- Remember, the tester only tests what *they* think it should do.
- Any misunderstandings the writer had while coding the program are carried over into testing.
- Ultimately, any system must be tested by the client to determine if it is acceptable.
- Points to the need for a written specification to protect both the client and developer.