School of Computer Science

Course Notes

CS 246

Object-Oriented Software Development

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

Fall 2009

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),
  - clicking on an icon launches (starts) a program,
  - program may pop up a dialog box for arguments to specify 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 specify its execution.
- Graphical interface is convenient, but seldom is 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** and **csh**, each with slightly different syntax and semantics.
- sh variants: ksh, bash
- csh variants: tcsh
- 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
cs246
$ echo Hi There!Enter        # print any string
Hi There!
```

- Comment begins with a hash (#) and continues to the end of the 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
cs246
Hi There!
```

- Commands can be editted on the command line:
  - position with ◁ and ▷ arrow keys,
  - remove characters with backspace/delete key,
  - add new characters,
  - pressing Enter at any point along the command line.
- Most commands have options, specified with a minus followed by one or more characters, which specify how the command operates.

```
$ uname -p # processor type
sparc
$ uname -s # operating system
SunOS
$ uname -a # all system information
SunOS services16.student.cs 5.8 Generic_117350-56 sun4u sparc SUNW
```

- Options are normally processed left to right; one option may cancel another.
- ***No standardization for option syntax and names.***
- Shells can be nested within each other (**subshell**).

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

  - when the login shell of terminal/xterm terminates, the terminal/xterm terminates.
  - when the login terminal/xterm terminates, you sign off the computer (logout).
- Use command chsh to set which shell you want to use (bash, tcsh, etc.).

1.1 File System

- Shell commands interact extensively with the file system.
- Files are containers for data stored on secondary 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 UNIX commands
    lib         system libraries
    usr
        bin         more UNIX commands
        lib         more system libraries
        include     system include files, .h files
    tmp         system temporary files
    u1          user files
    u2          user files
        . . .
        jfdoe       home directory
            .cshrc, .emacs, .login, . . .       hidden files
            cs246                               course files
                a1                              assignment 1 files
                    q1x.C, q2y.h, q2y.cc
        . . .
    u9          user files
    u or home       magic directory combining what is under u1-u9
```

- Directory named "/" is the root of the file system.
- bin, lib, usr, include : UNIX commands, system library and include files.

- tmp : location of temporary files created by commands.
- u1, . . . , u9 : user files are distributed across these directories.
- u or home : *magic* directory combining all users from user directories.
- 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 directories from the root to the file separated by the character "/".

```
/u2/jfdoe/cs246/a1/q1x.C # => file q1x.C
/u/jfdoe/cs246/a1/q1x.C  # => file q1x.C
```

- A **relative pathname** is a short name for a file provided by the shell 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.
- E.g., if 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/stdio.h       # => /usr/include/stdio.h
```

1.2 Pattern Matching

- Shells provide pattern matching of file names (**globbing**) 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.

- Patterns are composable: multiple wildcards joined into complex pattern.
- E.g., if the current directory is /u/jfdoe/cs246/a1 with leaf files q1x.C, q2y.h, q2y.cc
  - * matches 0 or more characters

    ```
    q*                  # => q1x.C, q2y.h, q2y.cc
    ```

  - ? matches 1 character

    ```
    q*.??               # => q1y.cc
    ```

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

    ```
    *.{cc,cpp,C}        # => q1x.C, q2y.cc
    ```

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

    ```
    q[12]*              # => q1x.C, q2y.h, q2y.cc
    ```

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

    ```
    q[!1]*              # => q2y.h, q2y.cc
    ```

  - 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 any file names starting with -, ?, or *
```

- **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, e.g., .cshrc, .login, etc., ***and*** ".", ". ."
- Pattern .[!.]* does not match "." and ". ." directories.
- On the command line, pressing the tab key after typing several characters of a file name requests the shell to automatically complete the file name.

```
$ echo cotab   # cause completion of file name to counter.cc
```

- If the completion is ambiguity, the shell "beeps", and you must type more characters to uniquely identifier the file name.

1.3 Quoting

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

```
$ echo \w \q \* \? \[ \] \$ \\ \ \ \ \ X
w q * ? [ ] $ \     X
```

Normally multiple spaces are compressed.

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

```
$ echo `whoami`
cs246
```

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

```
$ echo ′\w \q \* \? \[ \] \$ \\ \ \ \ \ X′
\w \q \* \? \[ \] \$ \\ \ \ \ \ X
```

A single quote cannot appear inside single quotes.

- A file name containing special characters is enclosed in single quotes.

```
$ rm ′Book Report 2.txt′   # file name with spaces
```

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

```
$ echo "    * ? [ ] \\  \"`whoami`\""
    * ? [ ] \  "cs246"
```

- Put newline into string for multi-line text.

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

1.4 Shell Commands

- Some commands are executed directly by the shell rather than the OS because they read/write the shell's state.
- **cd** : change the current directory.

  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** ~/bin : move to the bin directory contained in the 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/cs246/teaching/notes
```

- **time** : execute a command and print a time summary.
  - Prints **user time** (program CPU), **system time** (OS CPU), **real time** (wall clock)
  - Different shells print these values differently:

```
$ time a.out          | % time a.out
real        1.2       | 0.94u 0.22s 0:01.16
user        0.9       |
sys         0.2       |
```

  - user + system ≈ real-time (uniprocessor, no OS delay)

- **history** and "!" : print a numbered history of most recent commands entered and access them.

```
$ history          | $ !2
    1  date        | whoami
    2  whoami      | cs246
    3  cd ..       | $ !!
    4  ls xxx      | whoami
    5  cat xxx     | cs246
    6  history     | $ !ls
                   | ls xxx
                   | xxx
```

- !N rerun command N
- !! rerun last command
- !xyz rerun last command starting with the string “xyz”
- Use arrow keys △ / ▽ to move forwards / backwards through history commands.

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

  alias *[command-name [=] string]*

  - sh requires the “=” and does not allow spaces before/after it.
  - string is substituted for command **command**-name.
  - without arguments, print all currently defined alias names and strings.

- provide nickname for frequently used or variations of a command:

```
$ alias d="date"
$ d
Mon Oct 27 12:56:36 EDT 2008
$ alias off="clear; logout"
$ off         # clear screen before logging off
```

  Why are quotes necessary for alias off?

- Good style to always use quotes to prevent problems.
- aliases are composable:

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

- useful 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 the command name:

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

which does not add the -i option.

- alias entered on a command line is only in effect for a shell session.
- two options for making aliases persist across sessions:
  1. insert the **alias** commands in your .*shell*rc file,
  2. place a list of **alias** commands in a file called .aliases in your home directory and execute that file from your .*shell*rc file.

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

```
$ echo  I  like  ice  cream
I like ice cream
$ echo " I  like  ice  cream "
 I  like  ice  cream
```

- **eval** : process each argument and execute.

```
$ echo `date`  `whoami`
`date` `whoami`
$ eval echo `date`  `whoami`
Sat Dec 19 09:12:20 EST 2009 cs246
```

- removes quotes, expands variables, etc., then executes command

- **exit** : terminates shell, with optional integer exit status (return code) N.

  exit *[N]*

  - exit status defaults to zero if unspecified.

1.5 System Commands

- Commands executed by UNIX.
- man : print information about command.

```
$ man bash      # print information about "bash" command
$ man man       # print information about "man" command
```

- which : print pathname of a command.

```
$ which make
/usr/ccs/bin/make
$ which gmake
/software/.admin/bins/bin/gmake
```

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

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

- -a lists *all* files, including those that begin with a dot
- -l generates a *long* listing (details) for each file
- no file/directory name implies current directory

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

  mkdir *directory-name-list*

- cp : copies files, and with the -r option, copies 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 the contents of a source directory to the target directory.

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

  mv *[* -i *] source-file target-file*
  mv *[* -i *] source-file/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.

- rm : removes (deletes) files, and with the -r option, removes directories.

  rm *[-if] file-list*
  rm *[-ifr] file/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 * *(check* .snapshot*).***

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

- lpr/lpq/lprm : add, query and remove files from the printer queues.

  lpr *[-P printer-name] file-list*
  lpq *[-P printer-name]*
  lprm *[-P printer-name] job-number*

- if no printer is specified, the queue is a default printer.
- each job on a printer's queue has a unique number.
- use this number to remove a job from a print queue.

```
$ lpr -P ljp_3016 uml.ps        # print file to printer ljp_3016
$ lpq                           # check status, default printer ljp_3016
Spool queue: lp (ljp_3016)
Rank    Owner       Job  Files                  Total Size
1st     rggowner    308  tt22                   10999276 bytes
2nd     cs246       403  uml.ps                 41262 bytes
$ lprm 403                      # cancel printing
services203.math: cfA403services16.student.cs dequeued
$ lpq                           # 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 minimal differences.

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

- cmp generates the first difference between the files.

file x	file y
a	a
b	b
c	c
d	e
g	h
h	i
	g

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

- diff generates output describing the changes need to change the first file into the second file (used by patch).

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

- grep : (global regular expression print) search and print lines matching

pattern in files (google).

```
grep -i -r "pattern-string" file-list
```

- -i ignore case in both pattern and input files
- -r recursively examine files in directories.
- grep pattern is different from globbing pattern (see man grep).

```
$ grep -i fred names.txt  # list all lines containing fred in any case
$ grep '^\\(begin\|end\){.*}' *.tex
```

^ match start of line, match "\", match "begin" or "end", match "{", match 0 or more characters (notice "."), match "}".

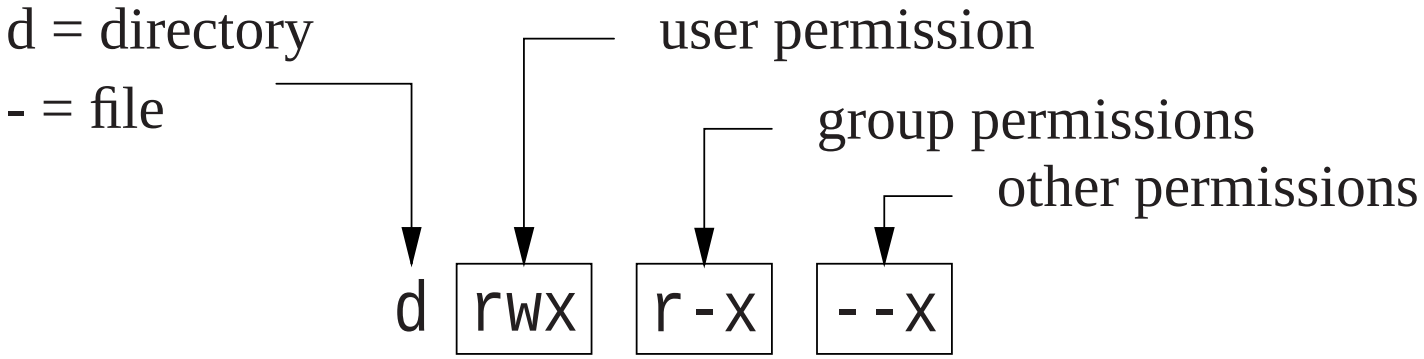
1.6 File Permission

- UNIX file structure supports 3 levels of security on each file or directory:
  - user : owner of the file,
  - group : arbitrary name associated with a number of userids,
  - other : any other user.
- A file or directory can have the following permissions: read, write, and execute/search.

- Readable and writable allow any of the specified users to read or write/change a file/directory.
- Executable for files means the file can be executed as a command, e.g., file contains a program or shell script.
- Executable for directories means the directory can be searched by certain system operations but not read in general.
- `ls -l` prints file-permission information for the current directory:

```
drwx------ 7 cs246 cs246  4096 Oct 20 13:07 ./
drwxr-x--- 5 cs246 cs246  4096 Oct 15 08:07 ../
drwx------ 2 cs246 cs246  4096 Oct 19 18:19 C++/
drwx------ 2 cs246 cs246  4096 Oct 21 08:51 Tools/
-rw------- 1 cs246 cs246 22714 Oct 21 08:50 notes.aux
-rw------- 1 cs246 cs246 63332 Oct 21 08:50 notes.dvi
```

- Columns are permissions, #-files-in-directory, owner, group, file size, change date, file name.
- Permission information is complex:



- 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 on a file are rw-r----- (usually), which means owner has read/write permission, and group has only read permission.
- Default permissions on a directory are rwx------, which means owner has read/write/execute.
- chgrp : change group-name associated with file:

  chgrp *[* -R *]* *group-name* *file-list*

  - -R recursively modify the group of a directory.

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

  chmod *[-R] mode-list file-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.
- The elements of the *mode-list* are separated by commas.
- E.g., to remove read and write permissions from security levels group and other for file xyz.

```
chmod g-r,o-r,g-w,o-w xyz     # long form
chmod go-rw xyz               # short form
chmod -R a+r assn2            # make directory and its subfiles
                              #    readable to everyone
```

1.7 Input/Output Redirection

- Every command has three special files: standard input (0), standard output (1) and standard error (2).
- By default, these are connected to the keyboard (input) and screen (output).
- Shell provides operators < for redirecting input and > for redirecting output to/from other sources.

```
$ ls -l > xxx          # output to file xxx
$ more < xxx           # input from file xxx; output to standard output
$ more < xxx > yyy     # input from file xxx; output to file yyy
```

- Command is (usually) unaware of redirection.
- Normally, standard error (e.g., error messages) is not redirected because of its importance.
- To selectively redirect output:

```
$ a.out  > xxx                 # redirect standard output
$ a.out 1> xxx                 # redirect standard output
$ a.out 2> errors              # redirect standard error
$ a.out 1> data 2> errors      # redirect standard output/error different files
$ a.out > xxx 2>&1             # redirect standard output/error same file
```

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

```
$ a.out 2> /dev/null              # ignore error messages
```

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

```
$ cat xxx | nl          # print xxx with line numbers
$ man ls | more         # paginate manual information for ls
```

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

```
$ a.out 2>&1 | nl       # both standard output and error go through pipe
```

- A pipeline can be arbitrarily long.

1.8 Programming

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

```
#!/bin/tcsh  [ -x ]
...          # shell and OS commands
```

- First line should begin with magic comment: "*#!*" with shell pathname for executing script.
- This line forces a specific shell to be used rather than the invoking shell.
- If the "*#!*" line is missing, the script is run using the invoking shell.
- 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:

```
$ sh scriptfile          # direct invocation
$ chmod u+x scriptfile   # make script file executable
$ ./scriptfile           # command execution, shell specified in script
```

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

1.8.1 Variables

- syntax : (letter | ´_´) (letter | ´_´ | digit)*
- **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 "=":

```
path=/u/cs246/          # declare and assign
```

No spaces before or after "=".

- A variable's value is returned using operator "$".

```
$ echo $path ${path}
/u/cs246/ /u/cs246/
```

braces, "{. . .}", allow unambiguous specification of name.

- Referencing an undefined variables returns the empty string.

```
$ echo $pathA1
```
blank line

- Always use braces to allow concatenation with other text:

```
$ echo $pathA1 ${path}A1      # $pathA1 undefined
/u/cs246/A1
```

- Each shell has a list of local and environment (global) variables.
- New variables are added to the local list.

- Local variables are only visible within a shell's execution context.
- Shell begins by copying containing shell's environment variables (works across different shells).
- Login shell starts with a number of useful environment variables, e.g.:

```
DISPLAY=services16.student.cs:10.0
EDITOR=emacsclient
HOST=services16.student.cs
PATH=. . .
```

- Local variable can be moved to shell's environment list.

  export path

- A variable can be removed from the local/environment list.

  unset path

- When a shell ends, changes to its environment variables do not affect its containing shell (***environment variables only affect subshells***).
- ***Beware commands composed in variables.***

```
$ cmd=´ls | more´  # command as value
$ ${cmd}           # execute command
ls: cannot access |: No such file or directory
ls: cannot access more: No such file or directory
$ eval ${cmd}      # evaluate and execute command
```

- "${cmd}" evaluates as: ´ls´ ´|´ ´more´, so | and more are file names.
- "**eval** ${cmd}" evaluates as: ls | more, so | is pipe and more is a command.

1.8.2 Routine

- A routine is defined as follows:

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

- Routines may be defined in any order.
- 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
}
```

- Invoked like command.

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

- All variables are global to every routine in a script.

```
rtn1() {
        var=3           # create local var
        rtn2            # call rtn2
}
rtn2() {
        echo ${var}     # use local var
        unset var       # destroy local var
}
```

- Special shell variables to access arguments/result:
  - ${#} number of command arguments, not including command name.
  - ${0} refers to script's name.

```
$ echo ${0}         # which shell are you using (except csh)
bash
```

  - ${*n*} refers to the command argument by position, i.e., 1st, 2nd, 3rd, ...
  - ${*} command arguments as a single string, e.g., "${1} ${2} . . .", not including command name

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

• Routine may return an integer exit status, which is examined using ${?}.

```
$ 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 shell
    return 17          # exit status
}
rtn a1 a2 a3 a4 a5  # invoke routine
echo ${?}           # print return value
```

```
$ ./scriptfile
5
scriptfile a1 a2 a3 a4
a1 a2 a3 a4 a5        # 1 string
a1 a2 a3 a4 a5        # 5 strings
27028
17
```

- **shift** *[* N *]* : destructively shift parameters to the left N positions, i.e., ${1}=${2}, ${2}=${3}, etc., and ${#} is reduced by N.
  - If no N, 1 is assumed.

1.8.3 Arithmetic

- Shell variables have type string, which has no arithmetic: `"1" + "17"`.

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

- To perform arithmetic a string is converted to an integer (if possible), an integer operation performed, and the integer result converted back to a string.
- UNIX command expr performs these steps.

- Basic integer operations, +, -, *, /, % (modulus), with usual precedence.

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

- bash supports arithmetic as a shell command:

```
$ echo $((3 + 4 - 1))
7
$ echo $((3 + ${i} * 2))             # no escape
9
$ echo $((3 + ${k}))
bash: 3 + : syntax error: operand expected (error token is " ")
```

1.8.4 Control Structures

- Shell supports several control constructs; syntax for sh/bash is presented (csh is different).

1.8.4.1 Test

- Strings, integers and files can be tested to affect control flow.
- expn is **test** expression, not arithmetic expression.

test	operation
\(expn \)	evaluation order (***must be escaped***)
! expn	not
expn1 -a expn2	logical and (***not short-circuit***)
expn1 -o expn2	logical or (***not short-circuit***)
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

1.8.4.2 Selection

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

```
if [ test ] ; then
    commands
elif [ test ] ; then
    commands
...
else
    commands
fi
```

- E.g.:

```
if [ "`whoami`" = "cs246" ] ; then          # string compare
    echo "valid userid"
else
    echo "invalid userid"
fi
```

```
grep "${user}" /etc/passwd > /dev/null    # ignore output
# check exit status
if [ ${?} -eq 0 ] ; then                  # integer compare
    echo "${user} has an account"
else
    echo "${user} does not have an account"
fi
if [ -x /usr/bin/cat ] ; then             # file check
    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′ ];
```

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, its commands are executed up to ;;, and control exits the **case** statement.
- If no pattern is matched, the **case** statement does nothing.
- E.g.

```
usage() {
    echo "Usage: ${0} -h -v -f input-file"
    exit 1        # terminate script with non-zero exit code
}
case "${1}" in            # process command-line arguments
  '-h' | '--help' ) usage ;;
  '-v' | '--verbose' ) verbose=yes ;;
  '-f' | '--file' )
     shift 1              # access argument
     file="${1}"
     ;;
  * ) usage ;;            # default
esac
```

1.8.4.3 Looping

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

```
while [ test ] ; do
    commands
done
```

- E.g.:

```
# print command-line arguments
while [ "${1}" != "" ] ; do          # string compare
      echo ${1}
      shift                          # destructive
done
i=1
while [ ${i} -lt 5 ] ; do            # integer compare
      echo ${i}
      i=`expr ${i} + 1`
done
while [ -f "${file}" ] ; do          # file check
      ...                            # update file variable
done
```

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

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

- ***Cannot have integer index.***
- If no list, iterate over arguments.

- E.g.:

```
for args in ${@} ; do          # process arguments, non-destructive
    echo ${args}
done
```

```
$ for count in "one" "two" "three & four" ; do echo ${count} ; done
one
two
three & four
```

```
$ 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`" = "cs246" ] ; then continue ; fi   # next iteration
    ...
  if [ ${?} -ne 0 ] ; then break ; fi          # exit loop
    ...
done
```

```
#!/bin/bash
#
# List and remove unnecessary files in directories
#
# Usage: cleanup [ [ -r | R ] [-i] directory-name ]+
#
# Examples:
#    cleanup -R .
#    cleanup -r xxx -i yyy -r -i zzz
#
# 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] directory-name ]+"
    exit 1
}
defaults() {                                # defaults for each directory
    prompt="-i"                             # prompt for removal
    depth="-maxdepth 1"                     # not recursive
}
```

```
remove() {
    for file in `find "${1}" ${depth} -type f \( -name ′core′ -o \
        -name ′a.out′ -o -name ′*.o′ -o -name ′*.d′ \)`
    do
        echo "${file}"                  # print removed file
        rm "${prompt}" "${file}"
    done
}
if [ ${#} -eq 0 ] ; then usage ; fi     # no arguments ?
defaults                                # set defaults
while [ "${1}" != "" ] ; 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                        # reset defaults
        ;;
    esac
    shift                               # remove argument
done
```

2 C++

2.1 Program Structure

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

2.1.1 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 as 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
//      }
// }
```

2.1.2 Statement

- C++ is actually 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.
- A preprocessor statement is a # character, followed by a series of tokens separated by whitespace, which is usually a single line and not terminated by punctuation.
- The syntax for a C/C++ statement (both template and regular) is a series of tokens separated by whitespace and terminated by a semicolon. ({} is an exception)

2.2 First Program

- **Java**

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

- **C++**

```
#include <iostream>       // insert contents of file iostream
using namespace std;      // direct naming of I/O facilities

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

- **#include** <iostream> copies basic I/O descriptions (no equivalent in Java).
- **using namespace** std allows imported I/O names to be accessed directly, i.e., ***without*** qualification.

- **int** main() is the routine where execution starts.
- curly braces, `{ ... }`, denote a block of code, i.e., routine body of main.
- cout `<< "Hello World!" <<` endl prints `"Hello World!"` to standard output, called cout (System.out in Java).
- endl start newline after `"Hello World!"` (println in Java).
- 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`).
- Compile with g++ command:

```
% g++ firstprogram.cc      # compile program
% a.out                    # execute program; execution permission
```

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

2.3 Declaration

- A declaration introduces names or redeclares names from previous declarations in a program.

2.3.1 Identifier

- name used to refer to a variable or type.
- syntax : (letter | ′_′) (letter | ′_′ | digit)$*$
- **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**	
byte	**char** / **wchar_t**	integral types
int	**int**	
float	**float**	real-floating types
double	**double**	
		label type, implicit

- C/C++ treat **char** and **wchar_t** (unicode characters) as an integral type.
- Java types **short** and **long** are created using type qualifiers.

2.3.3 Variable Declaration

- Declaration in C/C++ same as Java: type followed by list of identifiers.

Java / C / C++
char a, b, c, d; **int** i, j, k; **double** x, y, z; *id* :

- Declarations may be intermixed among executable statements in a block.
- Declarations may have an initializing assignment (except for fields in **struct**/**class**):

```
int i = 3;
```

- C/C++ do not check for uninitialized variables. (maybe)

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

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

```
int i; ...                  // first i
{ int k = i, i; ...         // second i (override first), both i's used in bloc
    { int i = i; ...        // third i (override second)
```

- Labels can only be declared in a routine and cannot be overridden; i.e., each label is unique within a routine body.

2.3.4 Type Qualifier

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

- Other integral types are generated using type qualifiers.
- C/C++ provide signed (positive/negative) and unsigned (positive only) integral types.

integral types	range
signed char / **char**	at least -127 to 127 (SCHAR_MIN / SCHAR_MAX)
unsigned char	at least 0 to 255 (UCHAR_MAX)
signed short int / **short**	at least -32767 to 32767 (SHRT_MIN / SHRT_MAX)
unsigned short int / **unsigned short**	at least 0 to 65535 (USHRT_MAX)
signed int / **int**	at least -32767 to 32767 (INT_MIN / INT_MAX)
unsigned int	at least 0 to 65535 (UINT_MAX)
signed long int / **long**	at least -2147483647 to 2147483647 (LONG_MIN / LONG_MAX)
unsigned long int / **unsigned long**	at least 0 to 4294967295 (ULONG_MAX)
signed long long int / **long long**	at least -9223372036854775807 to 9223372036854775807 (LLONG_MIN / LLONG_M
unsigned long long int / **unsigned long long**	at least 0 to 18446744073709551615 (ULLONG_MAX

- Range of values for **int** is machine specific: 2 bytes for 16-bit computers and 4 bytes for 32/64-bit computers.
- **long** is 4 bytes for 16-bit computers and 8 bytes for 32/64-bit computers.
- **#include** <limits.h> provides sizes for integer types (e.g., INT_MAX, etc.).
- **#include** <stdint.h> provides types [u]intN_t for **signed**/**unsigned** N = 8,

16, 32, 64 bits.

integral types	range
int8_t	-127 to 127 (INT8_MIN / INT8_MAX)
uint8_t	0 to 255 (UINT8_MAX)
int16_t	-32767 to 32767 (INT16_MIN / INT16_MAX)
uint16_t	0 to 65535 (UINT16_MAX)
int32_t	-2147483647 to 2147483647 (INT32_MIN / INT32_MAX)
uint32_t	0 to 4294967295 (UINT32_MAX)
int64_t	-92233720368547758O7 to 9223372036854775807 (INT64_MIN / INT64_MAX)
uint64_t	0 to 18446744073709551615 (UINT64_MAX)

- C/C++ provide two basic real-floating types **float** and **double**.
- One additional real-floating type is generated using a type qualifier.

real-float types	range, precision, architecture
float	$\approx 10^{-38}$ to 10^{38}, ≈ 7 digits, IEEE
double	$\approx 10^{-308}$ to 10^{308}, ≈ 16 digits, IEEE
long double	$\approx 10^{-4932}$ to 10^{4932}, ≈ 34 digits, IEEE

- C/C++ support write-once/read-only constant variables with type qualifier

const (Java **final**), in any variable declaration context.

Java	C/C++
`final short x = 3, y;` `y = x + 7;` `final char c = 'x';`	`const short int x = 3, y = x + 7;` **disallowed** `const char c = 'x';`

- C/C++ **const** identifier ***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 appear in read-only contexts after it is initialized.

2.3.5 String

- Strings are supported in C by language and library facilities.
- Language facility ensures string constant is terminated with a character `'\0'`.
- E.g., string constant `"abc"` is actually an array of the 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:

  1. A string cannot contain a character with the value `'\0'`.

2. String operations needing the length of a string must linearly search for ′\0′, which is expensive for long strings.
3. Management of variable-sized strings is the programmer's responsibility, with complex storage management problems.

- C++ solves these drawbacks by providing a string type using a length member and managing all of the storage for the variable-sized strings.
- Unlike Java, instances of the C++ string type are not constant.
- Values can change so a companion type like StringBuffer in Java is unnecessary.
- ***It is seldom necessary to iterate through the characters of a string variable!***

Java String methods	**C char** [] routines	**C++** string members
	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 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 constant
string d = c.substr( 2, 3 ); // extract starting at position 2 (zero origin) for length
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 i
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
```

- Member c_str returns a pointer to **char** * value in a string (´\0´ delimited).
- Routine getline(stream, string, **char**) allows different delimiting characters on input:

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

2.3.6 Constants

- Java and C/C++ share almost all the same constants for the basic types (except for unsigned).
- A **designated constant** indicates its type with suffixes: L/l for long, LL/ll for long long, U/u for unsigned, and F/f for float.
- Unlike Java, there is no D/d suffix for **double** constants.
- The type of an integral **undesignated constant** (octal/decimal/hexadecimal) is the smallest **int** type that holds the value, and the type of an undesignated real-floating constant is **double**.

boolean	**false**, **true**
decimal	123, -456L, 789u, 21UL
octal, prefix 0	0144, -045l, 0223U, 067ULL
hexadecimal, prefix 0X or 0x	0xfe, -0X1fL, 0x11eU, 0xffUL
real-floating	.1, 1., -1., -7.3E3, -6.6e-2F, E/e exponent
character, single character	′a′, ′\′′
string, multi-character	"abc", "\"\""

- Use the right constant with types character or string:

```
char ch = "a";        // use ′a′
char *str = ′a′;      // use “a”
string str = ′a′;     // use “a”
```

- An escape sequence allows special characters to appear in a character or string constant and starts with a backslash, \.

```
"\\ \" \t \n \012 \xf3"
```

- The most common escape sequences are (see a C++ textbook for others):

'\\'	backslash
'\'', "\""	single and double quote
'\t', '\n'	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 (not in Java)

- Sequence of octal/hex digits is terminated by first character not an octal/hex digit.

2.3.7 Type Constructor

- A **type constructor** is a declaration that 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	*class-type* r;	*any-type* &r; (C++ only)
structure	**class**	**struct** or **class**
array	**int** v[] = **new** **int**[10]; **int** m[][] = **new** **int**[10][10];	**int** v[10]; **int** m[10][10];

2.3.7.1 Enumeration

- An **enumeration** is a type defining a set of named constants with only assignment, comparison and implicit cast to integer operations:

```
enum Day {Mon,Tue,Wed,Thu,Fri,Sat,Sun}; // type declaration, implicit numbering
Day 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 nu
colour = B;                             // assignment
day = colour;                           // fails C++, works C
```

- Names in an enumeration are called **enumerator**s.
- Enumerators can be numbered explicitly.
- Enumeration in C++ denotes a new type; enumeration in C is alias for **int**.
- C/C++ enumeration only has underlying type **int**; Java enumeration can give names (and operations) to any value.
- Java enumerator names must always be qualified.
- C/C++ enumerator names are unqualified $\Rightarrow$ unique in a lexical scope.
- Trick to count enums:

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

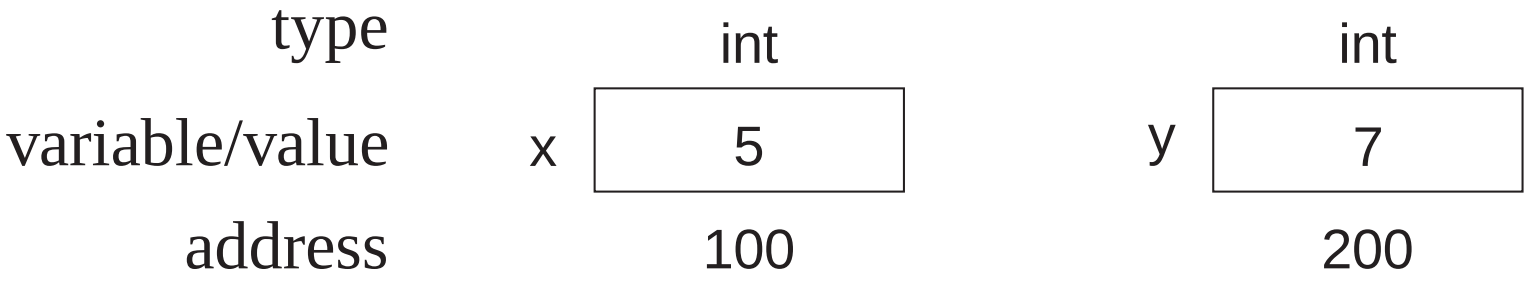
No_Of_Colours is 5, which is the number of enumerator colours.

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

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

2.3.7.2 Pointer/Reference

- **pointer**/**reference** is an indirect mechanism to access a type instance.
- ***All*** variables have an address in memory, e.g., **int** x = 5, y = 7:



- Value of a pointer/reference is the address of a variable.
- Accessing this address is different for a pointer or reference.
- Two basic pointer/reference operations:

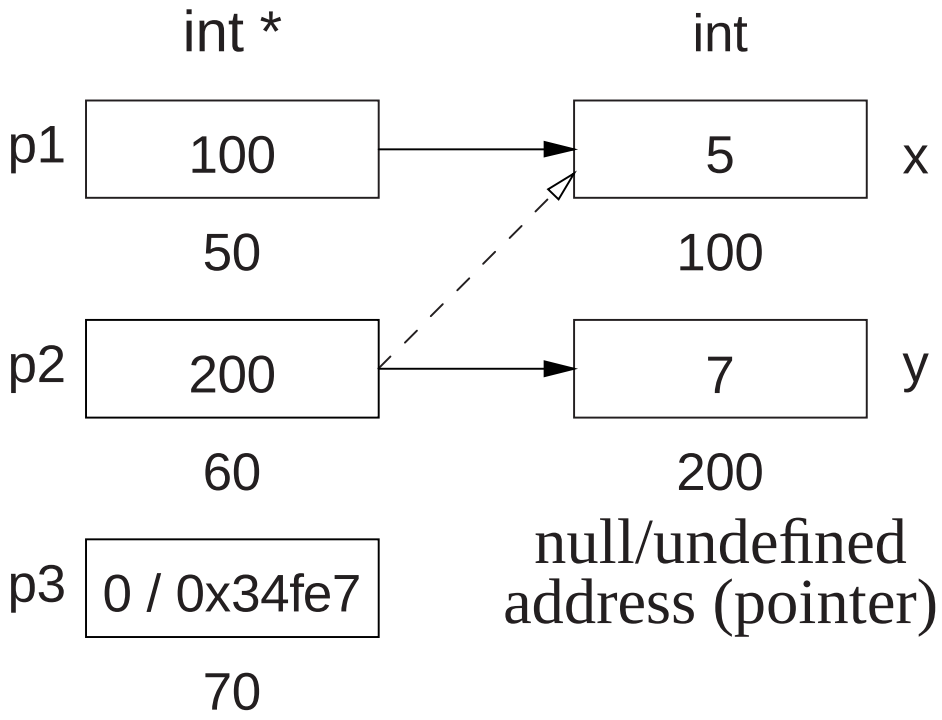
  1. **referencing**: obtain address of a variable; unary operator & in C++:

```
&x → 100
&y → 200
```

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

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

- Compiler automatically does first dereference, so x is really $*(\&x)$.
- Note, unary and binary use of operators &/$*$ for reference/deference and conjunction/multiplication.
- By convention, no variable is placed at the **null address** (pointer), null in Java, 0 in C/C++.
- Pointer/reference variable contains the memory address of another variable (**indirection**) or null pointer (or an undefined address if uninitialized).




```
&p1 → 50
&p2 → 60
&p3 → 70
*(&p1) → 100   p1 = &x;
*(&p2) → 200   p2 = &y; p2 = p1;
*(&p3) → 0      p3 = 0;
*(*(&p1)) → 5  *p1
*(*(&p2)) → 7  *p2
*(*(&p3)) → ?  *p3
```

- Because of implicit 1st dereference, p1 is 100 and *p1 is 5.
- Multiple pointers/references may point to the same memory address (dashed line).
- Dereferencing null/undefined pointer is undefined as no variable at the address (***but not necessarily an error***).
- Explicit dereference is an operation usually associated with a pointer:

```
*p2 = *p1;        ≡   y = x;    // value assignment
*p1 = *p2 * 3;    ≡   x = y * 3;
```

- Address assignment does not require dereferencing:

```
p2 = p1;          // address assignment
```

- p2 is assigned the same memory address as p1, i.e., p2 points at x; values of x and y do not change.
- Having to perform explicit dereferencing can be tedious and error prone.

```
p1 = p2 * 3;      // implicit deference
```

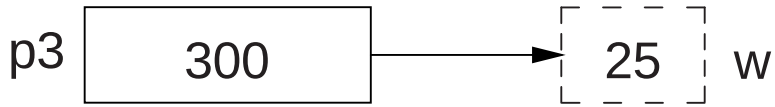
  unreasonable as p1 is assigned address in p2 times 3.
- Reasonable if value pointed to by p1 is assigned value pointed to by p2 times 3.
- A pointer that provides implicit dereferencing is a **reference**.
- However, implicit dereferencing generates an ambiguous situation for:

```
p2 = p1;
```

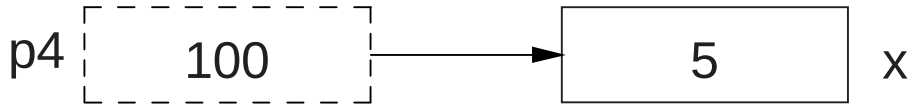
- Should this expression perform address or value assignment, and how are both cases specified?
- C provides only a pointer; C++ provides a pointer and a restricted reference; Java provides only a general reference.

- C/C++ pointer:
  1. created using the * type-constructor,
  2. may point to any type (i.e., basic or object type) in any storage location (i.e., global, stack or heap storage),
  3. and no implicit referencing or dereferencing.
     - Type qualifiers can be used to modify pointer types:

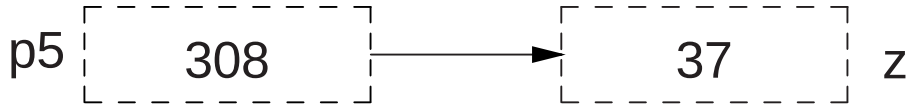
```
const short int w = 25;
const short int *p3 = &w;
```




```
int * const p4 = &x;
( int &p4 = x; )
```




```
const long int z = 37;
const long int * const p5 = &z;
```



     - p3 may point at any **const short int** variable.
     - Pointer can change to point at different variables, but the value of the variables cannot be changed through the pointer.
     - p4 may only point at variable x.

- Pointer cannot change to point at a different variable, but the value of the variable can be changed through the pointer.
- `p5` may only point at variable `z`.
- Pointer cannot change to point at a different variable, and the value of the variable `z` cannot be changed through the pointer.

- C++ reference

1. created using the & type-constructor,
2. may point to any type (i.e., basic or object type) in any storage location (i.e., global, stack or heap storage),
3. restricted to a constant pointer to user created (non-temporary/non-constant) storage,
4. and always has implicit dereferencing.

- Constant-pointer restriction of a C++ reference is equivalent to a Java **final** reference or `*` **const** pointer with implicit dereferencing.
- Java reference can vary what it points to, but it can only point to objects in heap storage.
- C++ constant-pointer restriction has two implications:

1. A C++ reference must be initialized at the point of declaration.

- initializing expression has implicit referencing because an address is ***always*** required;

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

2. No need for address assignment after a C++ reference declaration because the address cannot change.
   - Java interprets reference assignment `r2 = r1` as address assignment and has no mechanism to perform value assignment between reference types.

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

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

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

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

Gives false impression of distribution across the identifier list.

2.3.7.3 Aggregation (Structure/Array)

- Like Java, C++ has "objects", but it does not subscribe to the notion that everything is either a basic type or an object.
- Instead, aggregation is performed by structures and arrays, and computation is performed by routines.
- An object type is the composition of a structure and routines.
- In C++, a routine can exist without being embedded in a **struct**/**class**.

Structure is a mechanism to group together heterogeneous values, including (nested) structures:

Java	C/C++
class Foo { **public 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++.

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

- All members of a structure are accessible (public) by default (excluding Java package visibility).
- A structure member cannot be directly initialized (unlike Java) , and a structure is terminated with a semicolon.
- As for enumerations, a structure can be defined and instances declared in a single statement.

```
struct S { int i; } s; // definition and declaration
```

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

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

- **Recursive type**s (lists, trees) are defined using a pointer in a structure:

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

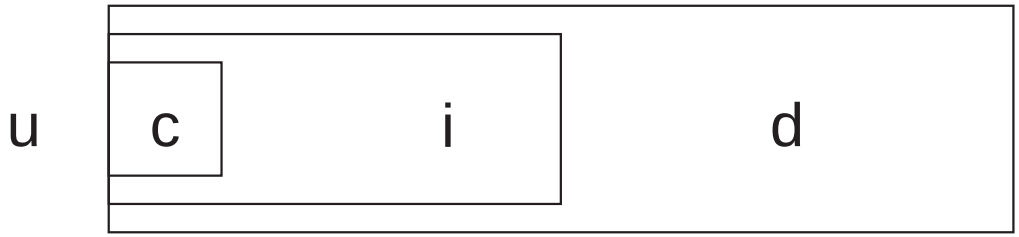
- A **bit field** allows direct access to individual bits of memory:

```
struct S {
        int i : 3;      // 3 bits
        int j : 7;      // 7 bits
        int k : 6;      // 6 bits
};
i = 2;    // 10
j = 5;    // 101
k = 9;    // 1001
```

- A bit field must be an integral type.
- Unfortunately, bit-fields are not portable.
- On little-endian architectures (e.g., like Intel/AMD x86), the compiler reverses the bit order.
- However, the compiler does not implicitly reverse the bit order.
- Hence, the bit-fields in variable s above must be reversed for little-endian architectures.
- While it is unfortunate C/C++ bit-fields lack portability, they are the highest-level mechanism to manipulate bit-specific information.

Union is a heterogeneous aggregation mechanism, 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;
        unsigned int exp : 8;
        unsigned int val : 23;
    } s;
    int i;
} u;
u.f = 3.5;          cout << hex << u.f << "\t" << 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.val << er
u.f = -3.5e-3;      cout << u.s.sign << "\t" << u.s.exp << "\t" << u.s.val << er
```

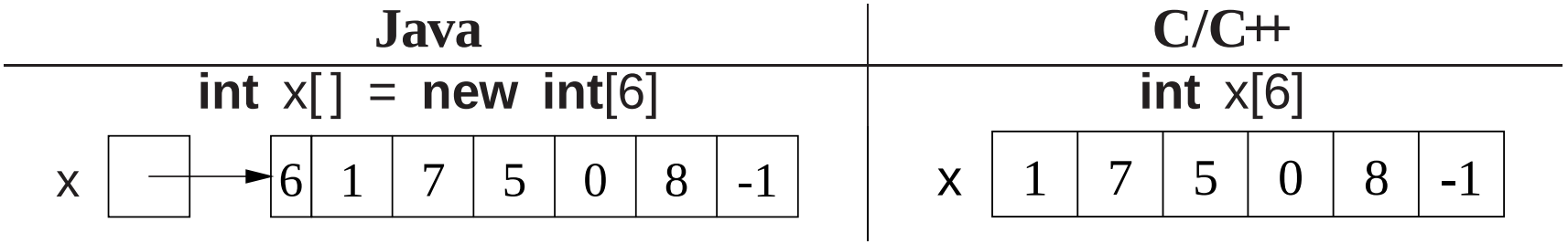
produces:

```
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.***

Array is a mechanism to group together homogeneous values.

- Unlike Java, a C/C++ array is a contiguous sequence of objects not a reference to the object sequence.



- Hence, array variables can have dimensions specified on a declaration and all the array elements are implicitly allocated.
- Be careful not to write:

```
int b[10, 20];          // not int b[10][20]
```

- 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
```

- **Subscripting**, [ ], selects an array element, and can be used on the left and right of assignment.

```
x[3];           // 3rd element
x[i];           // ith element
x[i + 1] = x[ t / 3 ] - y;      // left/right of assignment
```

- An array name without a subscript means &x, i.e., the starting address of the first element.
- Like Java, an array is subscripted from at 0 to dimension - 1.
- However, a C/C++ array is simple because dimension information is not stored with an array object.
- Hence, no equivalent to Java's length member for arrays, ***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:

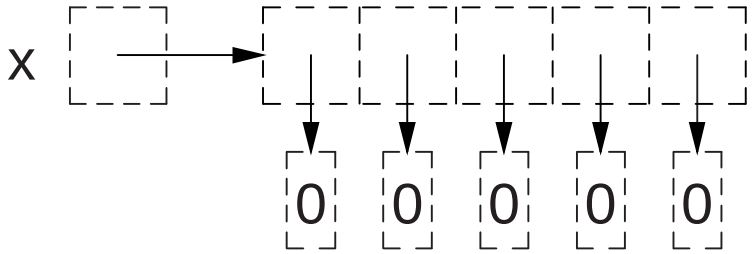
```
int arr[10];
int *parr = arr;        // think parr[], pointer to array of N ints
```

- 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 the same type but one is pointing at a variable and the other an array!***
- To read a complex declaration, parenthesize type qualifiers based on priority, read inside parenthesis outwards, start with variable name and 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

2.3.8 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;             // fails, t2, t1 have incompatible types
```

- Types T1 and T2 are **structurally equivalence**, but have different names so they are incompatible, i.e., initialization of variable t2 fails.
- An **alias** is a different name for same type, so alias types are equivalent.
- C/C++ provides **typedef** to create a synonym 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.

2.3.9 Type-Constructor Constant

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 versus null in Java.
- System include-files define the preprocessor variable NULL as 0.
- Structure and array initialization can only 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
```

- Values in initialization list are placed into a variable starting at the beginning of the structure or array.
- Not all the members/elements must be initialized.
- A nested structure or multidimensional array is created using braces.

- String constants 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 constants 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
```

2.4 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

- Like algebra, operators are prioritize and performed from high to low.
- Operators with same priority are done left to right, except for unary, ?, and assignment operators, which associate right to left.

```
int **a, **b, c, d, *w[10];
**a = **b > c ? ( *a = *b, d - 1 ) : (*w)[3] * 7 + 3;
(*(*a)) = (((*(*b)) > c) ? ( (((*a) = (*b)), (d - 1)) ) : ((((*w)[3]) * 7) + 3));
```

- Order of evaluation 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
```

- Referencing (address-of), &, and dereference, *, operators do not exist in Java because access to storage is restricted.
- Find address of any variable in any storage context, e.g., &x, &s.d, &v[5].
- Arrow operator, ->, is unique to C/C++ and is an anomaly among programming languages.
- Exists because the priority of selection operator “.” is incorrectly higher

than dereference operator "*", so *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.
- Unlike Java, the C/C++ remainder operator, %, only accepts integral operands.
- 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., assignment only on left side.
- C/C++ allows any expression to appear as a statement:

```
3;      j + i;      ( i + j ) * ( k + j );      sin(x);
```

- Complex assignment operators, e.g., lhs += rhs, are implicitly rewritten:

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

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

```
v[ rand() % 5 ] += 1;       // only calls random once
v[ rand() % 5 ] = v[ rand() % 5 ] + 1;   // calls random twice
```

- Comma expression is a series of expressions separated by commas:

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

- Expressions evaluated left to right with the value of rightmost expression returned as result.
- Comma expression allows multiple expressions to be evaluated in a context where only a single expression is allowed.
- Dimension problem m[10, 20] actually means m[20] because 10, 20 is a comma expression not a dimension list.
- Subscripting problem m[3, 4] means m[4], 4th row of matrix.
- Operators ++ / -- are discouraged because subsumed by general += / -=.

2.4.1 Conversion

- Conversion implicitly/explicitly transforms a value from one type to another.
- Two kinds of conversions:

- **widening/promotion** conversion, no information is lost:

char	→	**short int**	→	**long int**	→	**double**
'\x7'		7		7		7.000000000000000

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

double	→	**long int**	→	**short int**	→	**char**
77777.77777777777		77777		12241		'\xd1'

- C/C++ support both 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 ???
```

- Better to perform narrowing conversions explicitly using **cast** operator.

```
int i;    double x, y;
i = (int) x;          // explicit narrowing conversion
i = (int) x / (int) y; // explicit narrowing conversions for integer division
i = (int)(x / y);     // alternative technique
```

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

- g++ has a cast extension allowing construction of structure and array constants 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 meaning and structure of the constant.

2.4.2 Math Operations

- **#include** <cmath> provides real-float mathematical routines.
- All arguments and the return value are type **double**.

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

- Standard math constants are also available.

```
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       2.3025850929940456840   // 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)
```

- These constants are inadequate for computation using **long double**.
- Some systems provide **long double** versions, e.g., M_PIl.

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** (*cond-expr1*) *stmt1* **else if** (*cond-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** (*cond-expr*) *stmt*
	do *stmt* **while** (*bool-expr*) ;	**do** *stmt* **while** (*cond-expr*) ;
	for (*init-expr*;*bool-expr*;*incr-expr*) *stmt*	**for** (*init-expr*;*cond-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.
- Block serves two purposes: bracket several statements into a single statement and introduce local declarations.
- **When a statement is required, good practice is to always use a block to allow easy insertion and removal of statements to or from block.**
- Putting local declarations precisely where they are needed can help reduce declaration clutter at the beginning of an outer block.
- However, it can also make locating them more difficult.

2.5.2 Conditional

- C/C++ uses a **conditional expression** in control structures to cause conditional transfer (Java uses a boolean expression).
- A conditional expression is evaluated and implicitly tested for not equal to zero, i.e., *cond-expr* $\equiv$ *expr* `!= 0`.
- Boolean expressions are converted to 0 for **false** and 1 for **true** before comparison to zero, e.g.:

```
if ( x > y )...     implicitly rewritten as     if ( (x > y) != 0 )...
```

- Hence, other expressions are allowed in a conditional (C/C++ idiom):

```
if ( x ) ...        implicitly rewritten as     if ( (x) != 0 )...
while ( x )...                                  while ( (x) != 0 )...
```

- Watch for the common mistake in a conditional:

```
if ( x = y )...     implicitly rewritten as     if ( (x = y) != 0 )...
```

which assigns y to x and tests x != 0 (possible in Java for one type).

2.5.3 Selection

- C/C++ selection statements are **if** and **switch** (same as Java, except for boolean versus conditional expression).
- 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;
```

- Java/C/C++ have the **dangling else** problem of associating an **else** clause with its matching **if** in nested **if** statements.

- 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 -= pena } } **else** { income += bonus; }

- 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" << endl;
    exit( -1 );                 // terminate program
}
```

- Once a case clause is matched, its statements are executed, and control continues to the ***next*** statement.
- **break** statement is used at end of a case clause to exit **switch** statement.

- **It is a common error to forget the break.**
- If no case clause is matched and there is a **default** clause, its statements are executed, and control continues to the ***next*** statement.
- Otherwise, the **switch** statement does nothing.
- Only one label for each **case** clause but a list of **case** clauses is allowed.

2.5.4 Conditional Expression Evaluation

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

&&	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.

```
i < size && key != array[i]      // may only evaluate left operand of &&
```

- Hence, short-circuit operators are control structures in the middle of an expression because `e1 && e2` $\not\equiv$ `&&( 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:

```
abs2 = ( a < 0 ? -a : a ) + 2
```

```
if ( a < 0 ) {
    abs2 = -a;
} else {
    abs2 = a;
}
abs2 += 2;
```

2.5.5 Looping

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

- **Beware of accidental infinite loops.**

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

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

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

```
do {
    . . .  // executed at least once
} while ( x < 5 );
```

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

```
init-expr;
while ( cond-expr ) {
    stmt;
    incr-expr;
}
```

```
for ( init-expr; cond-expr; incr-expr ) {
    stmt;
}
```

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

```
for ( i = 1; i <= 10; i += 1 ) {          // count up
    // loop 10 times
} // i has the value 11 on exit
```

```
for ( i = 10; 1 <= i; i -= 1 ) {                // count down
    // loop 10 times
} // i has the value 0 on exit
for ( p = l; p != NULL; p = p->link ) {         // pointer index
    // loop through list structure
} // p has the value NULL on exit
for ( i = 1, p = l; i <= 10 & p != NULL; i += 1, p = p->link ) {  // 2 indi
    // 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 ; )
```

- **continue**/**break** statements available in all iteration constructs to advance to the next loop iteration or terminate loop.

```
for ( i = 0; ; i += 1 ) {          // infinite loop, conditional is "true"
    ...
  if ( x > y ) break;              // exit loop
    ...
  if ( x == y ) continue;          // start next iteration
    ...
}
```

- C/C++ **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 and ":" to a statement, where the label has routine scope.

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

- Transfer control backwards/forwards with respect to code in routine body.

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

- Can transfer into and out of control structures.

```
goto L1;                    // highly discouraged
...
for ( i = -5; i < 0; i += 1 ) {
    ...
    L1: ;                   // loop index uninitialized
    ...
    goto L2;
    ...
}
...
L2: ;
```

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 results in programs that are difficult to understand and maintain.
- Restricted transfer reduces the points where flow of control changes, and therefore, is easy to understand.

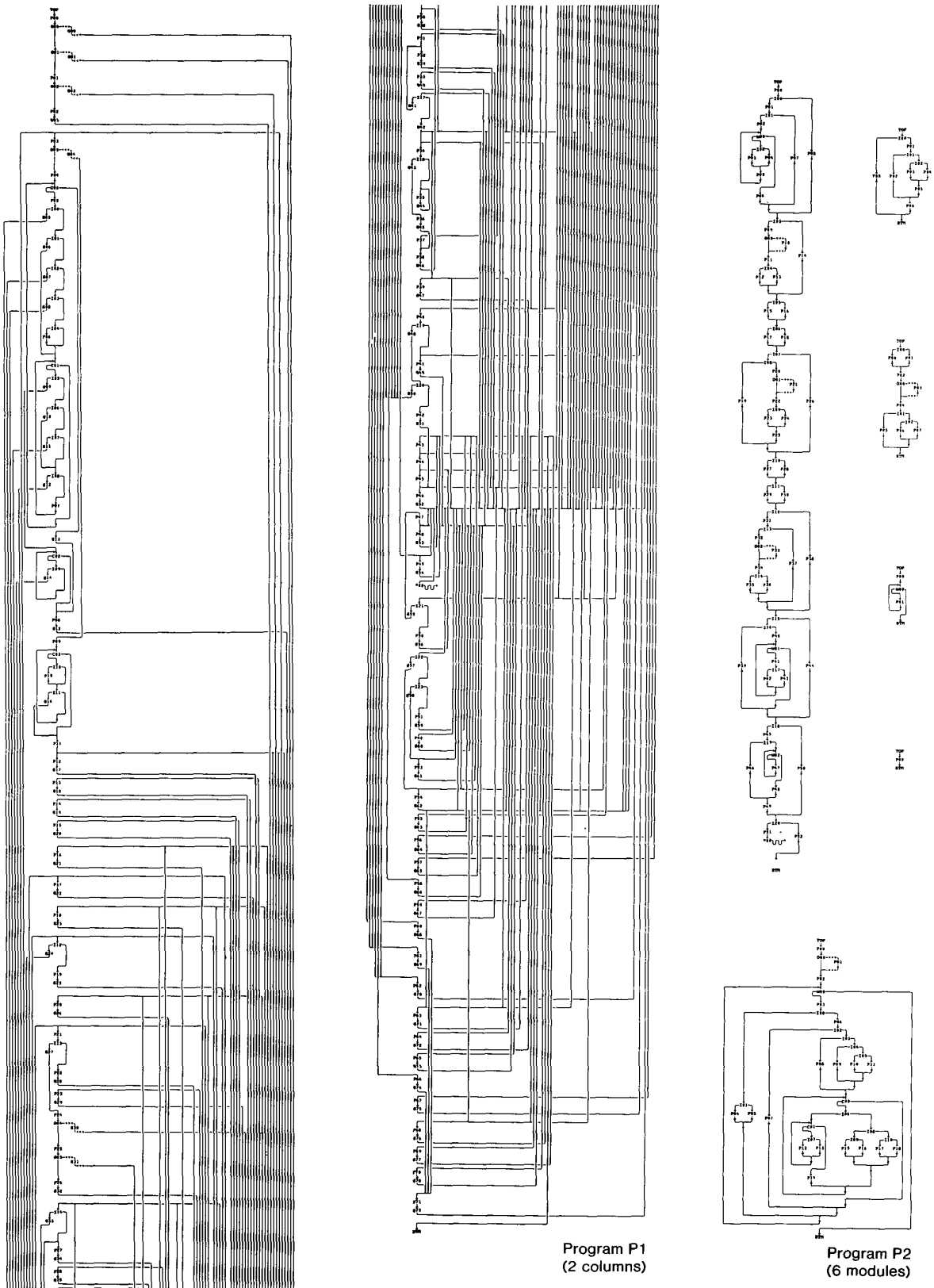


Fig. 1. Logical Control Flows of Two Versions of a Program.

- 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** or one **while** and **if**s).

  extended
  - classical control structures
  - case/switch: conditional structure for making decisions
  - repeat-until/do-while: structure for loops with test at bottom

  modified
  - extended control structures
  - one or more exits from arbitrary points in a loop
  - exits from multiple nested control structures
  - exits from multiple nested routine calls

  Eliminates the need for flag variables.

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 condition can appear in other locations 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 (copied) code necessary with **while**:

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

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

- C/C++ idioms for this situation are:

C	C++
`while ( (d = getc( stdin ) ) != EOF )`	`while ( cin >> d )`

- Results in expression side-effects and precludes analysis of d without code duplication.

- E.g., print the status of stream cin after every read for debugging:

```
while ( cin >> d ) {
    cout << cin.good() << endl;
    ...
}
cout << cin.good() << endl;
```

```
loop
    cin >> d;
    cout << cin.good() << endl;
  exit when cin.fail();
    ...
end loop
```

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

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

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

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

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

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

```
do {
    . . .
  if ( i >= 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 code-typing style should allow insertion of new code without having to change existing code.
- E.g., write linear search such that:
  - no invalid subscript for unsuccessful search
  - index points at the location of the key for successful search.
- Use only **if** and **while**:

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

- Allow short-circuit operators.

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

- Logical & is incorrect because it evaluates both operands.
- Alternatively, use multi-exit loop.

```
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
}
```

- The extra test after the loop can be eliminated by introducing it 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 ( i = 0; ; i += 1 ) {
   if ( i >= size ) {
            list[size].count = 1;
            list[size].data = key;
            size += 1;
      break;
      } // exit
   if ( key == list[i].data ) {
            list[i].count += 1;
      break;
      } // exit
} // for
```

2.6.2 Static Multi-Level Exit

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

Java

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

C / C++

```
{
      ... declarations ...
      switch ( ... ) {
            for ( ... ) {
                  ... goto L1; ...
                  ... goto L2; ...
                  ... goto L3; ...
            } L1: ;
            ...
      } L2: ;
      ...
} L3: ;
```

- Labelled **break**/**continue** transfer control out of the control structure with the corresponding label, terminating any block that it passes through.
- Commonly used with nested loops:

Java

```
L1: for ( ;; ) {          // while ( flag1 && ... )
      L2: for ( ;; ) {          // while ( flag2 && ... )
            L3: for ( ;; ) {          // while ( flag3 && ... )
                  ...
                if ( ... ) break L1; // exit 3 levels
                  ...
                if ( ... ) break L2; // exit 2 levels
                  ...
                if ( ... ) break L3; // or break, exit 1 level
                  ...
            }
      }
}
```

C / C++

```
for ( ;; ) {
      for ( ;; ) {
            for ( ;; ) {
                  ...
    if ( ... ) goto L1;
                  ...
          if ( ... ) goto L2;
                  ...
                if ( ... ) goto L3;
                  ...
            } L3: ;
      } L2: ;
} L1: ;
```

- Eliminates flag variables, which are the variable equivalent to a goto.
- Normal and labelled **break** are a **goto** with restrictions:
  - Cannot be used to create a loop (i.e., cause a backward branch in the program); hence, all situations that result in repeated execution of statements in a program are clearly delineated.
  - Cannot be used to branch *into* a control structure.
- The simple case (exit 1 level) of multi-level exit is a multi-exit loop.

- Why is it good practice to label all exits?
- **Only use goto to simulate labelled break and continue.**
- **return** statements can generate multi-exit loop and multi-level exit.
- Static multi-level exits appear infrequently, but are extremely concise and execution-time efficient.

2.7 Preprocessor

- Preprocessor manipulates the text of the program ***before*** compilation.
- **Program you see is not what the compiler sees!**
- The three most commonly used preprocessor facilities are substitution, file inclusion, and conditional inclusion.

2.7.1 Substitution

- **#define** statement declares a preprocessor 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 PI 3.14159
#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 PI;                // same as: x = 3.14159;
    set y with x;             // same as: y = x;
end                           // same as: }
```

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

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

Same as putting the following **#define**s in a program without changing the program:

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

- Predefined preprocessor-variables exist identifying hardware and software environment, e.g., mcpu is kind of CPU.
- Replace **#define** with **enum** (see Section 2.3.7.1, p. 65) for integral types; otherwise use **const** declarations (see Section 2.3.4, p. 55) (**final** in Java).

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

- **enum** uses no storage while **const** declarations do.
- **#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.

2.7.2 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 inclusions 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 and unistd.h contains many useful **#define**s, like the null pointer constant NULL (e.g., see /usr/include/limits.h).

2.7.3 Conditional Inclusion

- Preprocessor has an **if** statement, which may be nested, to conditionally add/remove code from a program.
- Conditional of **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
```

  Independent of language structure, can overlap definitions and routines.

- 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.

2.8 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.
     - Conversion is based on the type of variables and format codes.
  2. **Unformatted I/O** transfers data without conversion, e.g., internal integer and real-floating values.

2.8.1 Formatted I/O

Java	C	C++
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)	

- Formatted I/O occurs to/from a **stream file**.
- C++ has three implicit stream files: cin, cout and cerr, which are automatically declared and opened (Java has in, out and err).
- C has stdin, stdout and stderr, which are automatically declared and opened.
- Include iostream has all necessary declarations for cin, cout and cerr.
- cin reads input from the keyboard (unless redirected by shell).
- cout writes to the terminal screen (unless redirected by shell).
- cerr writes to the terminal screen even when cout output is redirected.
- ***Error and debugging messages should always be written to* cerr**:
  - normally not redirected by the shell,
  - unbuffered so output appears immediately.
- Stream files other than 3 implicit ones require declaring each file object:

```
#include <fstream>   // required for stream-file declarations
ifstream infile( "myinfile" );          // input file
ofstream outfile( "myoutfile" );        // output file
```

- Type of the file, ifstream or ofstream, indicates whether the file can be read or written.

- Declaration **open**s a file making it accessible through the variable name, e.g., infile and outfile are used for file access.
- Check for successful opening of a file using the stream member fail, e.g., infile.fail(), which returns **true** if the open failed and **false** otherwise.
- Connection between the file name in the program and operating-system file is done at the declaration:
  - infile reads from file `myinfile`
  - outfile writes to file `myoutfile`

  where both files are located in the directory where the program is run.
- C++ I/O library overloads the bit-shift operators << and >> to perform I/O.
- C I/O library uses fscanf(outfile,. . .) and fprintf(infile,. . .), which have short forms scanf(. . .) and printf(. . .) for stdin and stdout.
- Parameters in C are always passed by value, so arguments to fscanf must be preceded with & (except arrays) so they can be changed.
- Both I/O libraries can cascade multiple I/O operations, i.e., input or output multiple values in a single expression.

2.8.1.1 Formats

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

oct	values in octal
dec	values in decimal
hex	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
fixed (default) / scientific	float-point values without / with exponent
setprecision(N)	fraction of float-point values in maximum of N columns
setw(N)	NEXT VALUE ONLY in minimum of N columns
setfill(′ch′)	padding character before/after value (default blank)
endl	flush output buffer and start new line (**output only**)
skipws (default) / noskipws	skip whitespace characters (**input only**)

- manipulators applies to all constants/variables after it, even 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′; only endl flushes for interactive output.

2.8.1.2 Input

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

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<<end } *// in/out closed implicitly*

- Input values for a stream file are C/C++ undesignated constants: 3, 3.5e-1,

etc., separated by whitespace.

- Except for characters and character strings, ***which are not in quotes***, so cannot read strings containing white spaces.
- Type of operand indicates the kind of constant expected in the stream, e.g., an integer operand means an integer constant is expected.
- Input starts reading where the last read left off, and scans lines to obtain necessary number of constants.
- Hence, the placement of input values on lines of a file is often arbitrary.
- Unlike Java, C/C++ must attempt to read ***before*** end-of-file is set and can be tested for.
- **End of file** is the detection of the physical end of a file; **there is no end-of-file character**.
- From a keyboard, <ctrl>-d (press the <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 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 constant OR no constant 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 constant), fail becomes **true** but eof is **false**.
- Generates infinite loop as invalid data is not skipped for subsequent reads.
- When bad data is read, ***stream must be reset and bad data cleared***:

```
#include <iostream>
using namespace std;
int main() {
    int n;
    cout << showbase;                                   // prefix hex with 0x
    cin >> hex;                                         // hex constants
    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 newlin
        } else {
            cout << hex << "hex:" << n << dec << " dec:" << n << end
        }
    }
    cout << endl;
}
```

- After an unsuccessful read, clear() resets the stream.
- ignore skips *n* characters, e.g., cin.ignore(5) or until a specified character.

- Alternatively, streams have a conversion to **void** `*`: if fail(), a null pointer; otherwise nonnull pointer.

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

- In C, routine feof returns **true** when eof is reached and fscanf returns EOF.
- 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 file
    }
}
```

2.8.1.3 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 supplies a list of formats and values, and output operation generates the strings:

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

- There is 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 is 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'; 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

2.8.2 Unformatted I/O

- **Unformatted I/O** transfers data without conversion, e.g., internal integer and real-floating values.
- Uses same mechanisms as formatted I/O to connect program to file (open/close).
- read and write routines transfer bytes without conversion from/to a file.

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

int main() {
    ofstream outfile( "xxx" );          // open output file “xxx”
    if ( outfile.fail() ) {             // successful open ?
        cerr << "Error!" << endl;
        exit( -1 );
    }
    double d = 3.0;
    outfile.write( (char *)&d, sizeof( d ) );  // coercion
    outfile.close();                    // close file before attempting read

    ifstream infile( "xxx" );           // open input file “xxx”
    if ( infile.fail() ) {              // successful open ?
        cerr << "Error!" << endl;
        exit( -1 );
    }
    double e;
    infile.read( (char *)&e, sizeof( d ) );  // coercion
    cout << e << endl;
    infile.close();
}
```

- read and write take a **char** * pointer and length.

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

- To pass any kind of pointer for unformatted I/O requires a **coercion**, which is a cast ***without*** a conversion.
- ***Coercion breaks the type system; use it very sparingly*** (and would be unnecessary if buffer type was **void** *).

2.9 Dynamic Storage Management

- Java is a **managed language**; C/C++ are unmanaged.
- C/C++ do not have **garbage collection** of dynamically allocated storage after a variable is no longer accessible.
- Instead, an additional dynamic storage-management operation is used to free storage.
- 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** a, b, c; } Foo p = **new** Foo(); p.c = 'R'; *// p garbage collected*	**struct** Foo { **char** a, b, c; }; Foo *p = (Foo *)malloc(**sizeof**(Foo)); p->c = 'R'; free(p); *// explicit free*	**struct** Foo { **char** a, b, c; }; Foo *p = **new** Foo(); p->c = 'R'; **delete** p; *// explicit free*

- Allocation has 3 steps:

  1. determine size/alignment of allocation,
  2. allocate heap storage of correct size/alignment,
  3. coerce undefined storage to correct type.

- Each step is explicit in C; C++ operator **new** performs all 3 steps implicitly.
- Parenthesis after the type name in the **new** operation are optional.
- Storage for dynamic allocation comes from an area called the **heap**.
- Before storage can be used, it must be allocated.

```
Foo *p;          // forget to allocate or initialize pointer
p->c = 'R';      // places 3 at some random location in memory
```

Uninitialized variables.

- 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:

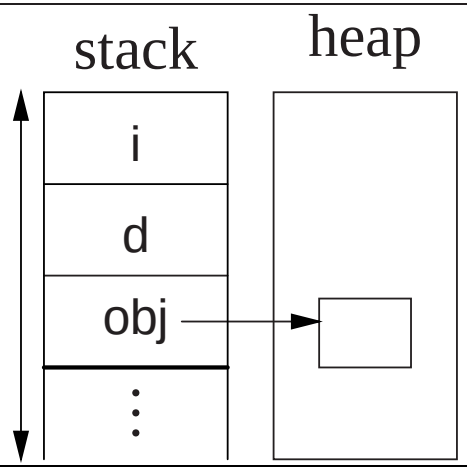
```
delete p;
p->c = 'R';           // result of dereference is undefined
```

  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:

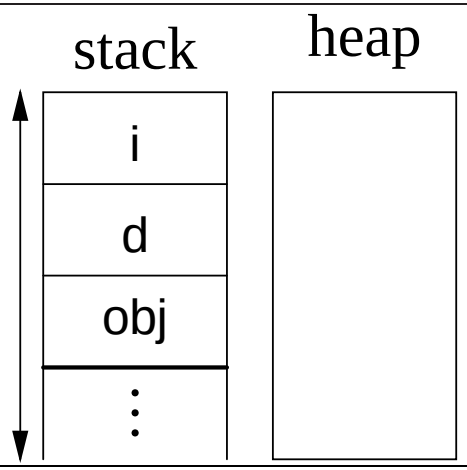
Java

```
{ // basic & reference
   int i;
   double d;
   ObjType obj =
         new ObjType();
   ...
} // garbage collected
```



C++

```
{  // all types
   int i;
   double d;
   ObjType obj;
   ...
} // implicit delete
```



- **Stack allocation eliminates explicit storage-management (simpler) and is more efficient than heap allocation — use it whenever possible.**
- Dynamic allocation in C++ should be used only when:
  - a variable's storage must outlive the block in which it is allocated:

```
ObjType *rtn(...) {
      ObjType *obj = new ObjType();
      ... // use obj
      return obj; // storage outlives block
} // obj deleted later
```

  - when each element of an array of objects needs initialization:

```
ObjType *v[10]; // array of object pointers
for ( int i = 0; i < 10; i += 1 ) {
    v[i] = new ObjType( i ); // each element has different initialization
}
```

- 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 often unspecified:

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

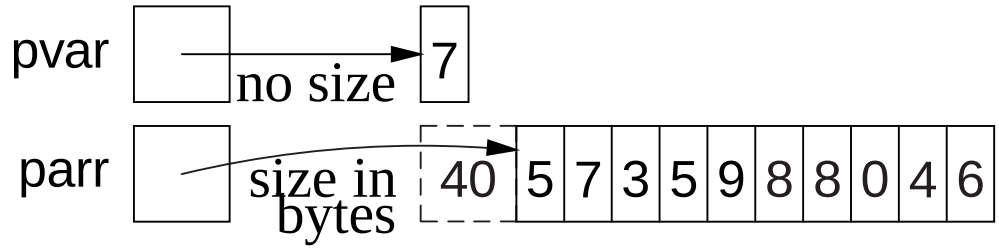
- Java notation:

```
int parr[] = new int[10];
```

cannot be used because **int** parr[] is actually rewritten as **int** parr[N], where N is the size of the initializer value.

- As well, no dimension information results in the following ambiguity:

```
int *pvar = new int;
int *parr = new int[10];   // parr[]
```



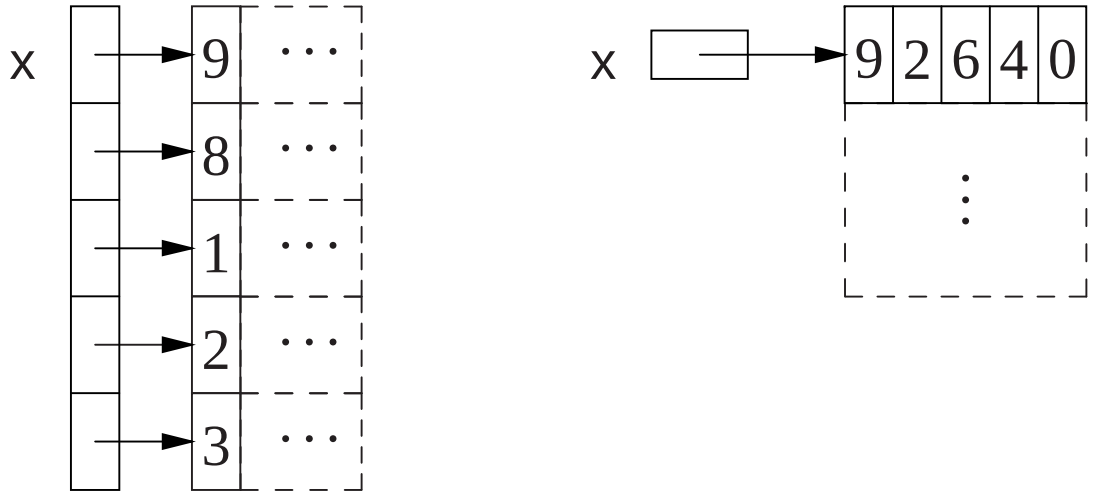
- Variables pvar and parr have the same type but one is an array, which poses a problem when deleting a dynamically allocated array.
- To solve the problem, special syntax is used to distinguish these cases:

```
delete pvar;        // single element
delete [] parr;     // multiple elements
```

- [] indicates multiple elements (but unknown number and size of dimensions) and array-size is stored with the array.
- **Never do this:**

```
delete [] parr, pvar; // => (delete [] parr), pvar;
```

  which is an incorrect use of a comma expression; var is not deleted.
- Declaration of a pointer to a matrix is complex in C/C++, e.g., **int** *x[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.
- For `*` and `[]` which applied first?
- Dimension is higher priority (as subscript, see Section 2.4, p. 83), so declaration is interpreted as **int** `(*(x[5]))` (left).
- Only the left example (above) of declaring a matrix can be generalized to allow a dynamically-sized matrix.

```
int main() {
    int *m[5];                                  // 5 rows
    for ( int r = 0; r < 5; r += 1 ) {
        m[r] = new int[4];                      // 4 columns per row
        for ( int c = 0; c < 4; c += 1 ) { // initialize matrix
            m[r][c] = r + c;
        }
    }
    for ( int r = 0; r < 5; r += 1 ) {          // print matrix
        for ( int c = 0; c < 4; c += 1 ) {
            cout << m[r][c] << ", ";
        }
        cout << endl;
    }
    for ( int r = 0; r < 5; r += 1 ) {
        delete [] m[r];                         // delete each row
    }
}                                               // implicitly delete array "m"
```

2.10 Command-line Arguments

- Starting routine main has exactly two overloaded interfaces.

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

- The second form is used by the shell to pass command-line arguments, where the command line string-tokens are transformed into C/C++ arguments.
- argc is the number of string-tokens on the command line, including the command name.
- ***With command name, number of tokens is one greater than in Java***.
- argv is an array of pointers to the 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) { **switch** (args.length) { **case** 0: . . . *// no args* **break**; **case** 1: . . . args[0] . . . *// 1 arg* **break**; **case** . . . *// others args* **break**; **default**: . . . *// usage message* System.exit(-1); } . . .	**int** main(**int** argc, **char** *argv[]) { **switch**(argc) { **case** 1: . . . *// no args* **break**; **case** 2: . . . args[1] . . . *// 1 arg* **break**; **case** . . . *// others args* **break**; **default**: . . . *// usage message* exit(-1); } . . .

- Arguments are processed in the range argv[1] through argv[argc - 1], i.e., starting one greater than Java.
- Process following arguments from shell command line:

cmd [infile-file = cin [outfile-file = cout [size = 20 [code = 5]]]]

- Note, dynamic allocation, strtol (atoi has no mechanism to check for errors), and **goto**; no duplicate code.

```
#include <iostream>
#include <fstream>
#include <cstdlib>                              // strtol, exit
#include <cerrno>                               // errno, ERANGE
using namespace std;

bool convert( int &val, char *buffer ) {        // convert C string to integer
    char *endptr;                               // buffer pointer
    val = strtol( buffer, &endptr, 10 );        // convert string to integer
    return errno != ERANGE && endptr != buffer && *endptr == '\0'; // val
} // convert

int main( int argc, char *argv[] ) {
    const unsigned int sizeDeflt = 20, codeDeflt = 5;
    istream *infile  = &cin;                    // default value
    ostream *outfile = &cout;                   // default value
    int size = sizeDeflt, code = codeDeflt;     // default value
```

```
    switch ( argc ) {
      case 5:
        if ( ! convert( code, argv[4] ) ) goto usage; // invalid integer ?
        // FALL THROUGH
      case 4:
        if ( ! convert( size, argv[3] ) ) goto usage; // invalid integer ?
        // FALL THROUGH
      case 3:
        outfile = new ofstream( argv[2] );
        if ( outfile->fail() ) goto usage;          // open failed ?
        // FALL THROUGH
      case 2:
        infile = new ifstream( argv[1] );
        if ( infile->fail() ) goto usage;           // open failed ?
        // FALL THROUGH
      default:                                      // all defaults
        break;
      usage:
        cerr << argv[0] << " [ infile-file [ outfile-file [ size = "
             << sizeDeflt << " [ code = " << codeDeflt << " ] ] ] ]" <<
        exit( -1 );                                 // TERMINATE
    }
    // do something
    if ( infile != &cin ) delete infile;            // close file, do not delete cin
    if ( outfile != &cout ) delete outfile;         // close file, do not delete co
}
```

2.11 Routine

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

- C++ routines are not part of aggregation (not combined in an object), e.g., routine main is not defined in a type.
- A routine is either a **procedure** or a **function** based on the return type.
- A procedure does NOT return a value that can be use in an expression, indicated with return type of **void**:

```
void proc( ... ) { ... }
```

- A procedure can return values through the argument/parameter mechanism.

- A procedure terminates when control runs off the end of routine body or a **return** statement is executed:

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

- A function returns a value that can be use 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.
- A 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.
- Routines cannot be nested in other routines.
- All routines are embedded in the global (external) level in a source file.
- **Global scope** contains types, variables and routines:

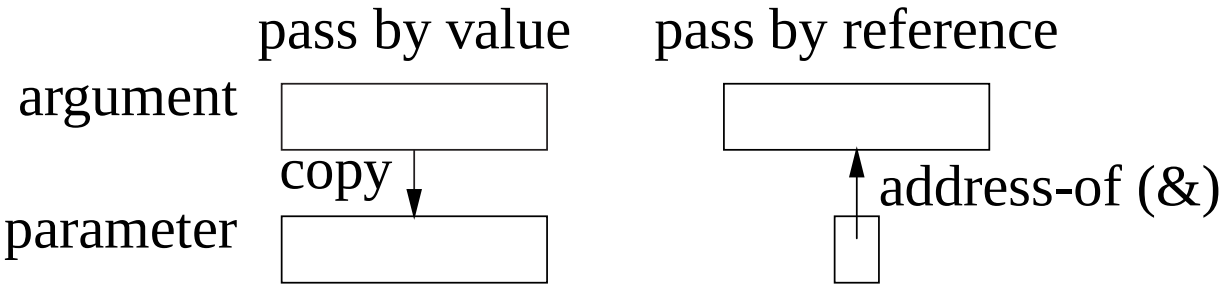
```
// global scope
enum Colour { R, G, B };      // type
Colour colour = B;            // variable
int main() {                  // routine
      colour = R;
      Colour colour = G;      // local scope, hides previous variables
}
```

- Global variables are allocated in declaration order and deallocated in reverse order at program exit ***per file but no order among files***.
- Gobal area is a separate memory from the stack and heap.

2.11.1 Argument/Parameter Passing

- Arguments are passed to parameters by:
  - **value**: parameter is initialized by the argument (usually bit-wise copy).

- **reference**: parameter is a reference to the argument and is initialized to the argument's address.



- 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 pushed on the stack to become the corresponding parameter, ***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 to become the corresponding reference parameter.

```
struct Complex { double r, i; };
void r( int i, int &ri, Complex c, Complex &rc ) {
        ri = i = 3;
        rc = c = (Complex){ 3.0, 3.0 };
}
int main() {
        int i1 = 1, i2 = 2;
        Complex c1 = { 1.0, 1.0 }, c2 = { 2.0, 2.0 };
        r( i1, i2, c1, c2 );
}
```

- Which arguments change?
- What if routine call is changed to:

```
r( i1, 3, c1, c2 );                 // fails!
r( i1, i1 + i2, c1, c2 );           // fails!
```

  Cannot change a constant or temporary variables!

- Value passing is most efficient for basic and small structures because the values are accessed directly in the routine.
- Reference passing is most efficient for large structures and arrays because the values are not duplicated in the routine.
- Use type qualifiers to create read-only reference parameters so the

corresponding argument is guaranteed not to change:

```
void r( const int &i, const Complex &c, const int v[5] ) {
      i = 3;          // assignments disallowed, read only!
      c.r = 3.0;
      v[0] = 3;
}
r( 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 because argument cannot change, and allows constants 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 r( int i, double g, char c = '*', double h = 3.5 ) { . . . }
r( 1, 2.0, 'b', 9.3 );          // maximum arguments
r( 1, 2.0, 'b' );               // h defaults to 3.5
r( 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.

2.11.2 Array Parameter

- Array copy is unsupported so arrays cannot be passed by value only by reference.
- Therefore, all array parameters are implicitly reference parameters, and hence, do not have a reference symbol.
- 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 programming practice uses the middle form because it clearly indicates the variable is going to 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;
}
```

2.11.3 Overloading

- **Overloading** occurs when a name has multiple meanings in the same context.
- Most languages have some overloading.
- E.g., most built-in operators are overloaded on both integral and real-floating operands, i.e., the + operator is different for 1 + 2 than for 1.0 + 2.0.

- Overloading requires the compiler to disambiguate among identical names based on some criteria.
- The normal criterion is type information.
- In general, overloading is done on operations not variables:

```
int i;                      // variable overloading disallowed
double i;
void r( int ) {}            // routine overloading allowed
void r( double ) {}
```

- ***Power of overloading occurs when type of a variable changes: operations on the variable are implicitly reselected to 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 types of the parameters ***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
```

- Implicit conversions between arguments and parameters can cause problems:

```
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
```

- 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.***

2.11.4 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, it is necessary to pass code as an argument, which is executed within the routine body.
- Most programming languages allow a routine pointer (Java does not) for further generalization and reuse.
- 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., when inverting a matrix, the matrix may not be 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 possible recover and continue with a correction (e.g., modify the matrix):

```
int singularDefault( /* info about error */ ) { return 0; }
int invert( int *matrix[], int rows, int cols,
            int (*singular)( /* info about error */ ) = singularDefault ) {
    ...
    if ( determinant( matrix, rows, cols ) == 0 ) {
        // compute correction to continue the computation
        correction = singular( /* info about error */ );
    }
    ...
}
```

- 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 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 supplying a call-back routine, and each widget calls its supplied call-back routine(s) when the widget changes state.
- Normally, a widget passes the new state of the widget to each call-back routine so it can perform an appropriate action, e.g.:

```
int callback( /* info about event */ ) {
    // examine event information and perform appropriate action
    // return status of callback action
}
...
registerCB( closeButton, callback );
```

- Call-back programming become difficult if it depending on the number of times it is called or previous argument values.

2.12 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 the structure by allowing routine members.

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*

- ***Each object provides both data and the operations necessary to manipulate that data in one self-contained package.***
- Routine member 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() { return sqrt( this->re * this->re + this->im * this->im ); }
```

(this should be a reference rather than a pointer.)

- Except for the syntactic differences, the two forms are identical.
- ***Like Java, the use of implicit parameter* this, *e.g.*, this->f, *is seldom necessary in C++.***
- 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.
- Add arithmetic operations:

```
struct Complex {
    ...
    Complex add( Complex c ) {
        Complex sum = { re + c.re, im + c.im };
        return sum;
    }
};
```

- To sum x and y, write x.add(y).

- Because addition is a binary operation, add needs a parameter as well as the implicit context in which it executes.
- Like Java, C++ allows overloading members in a type.

2.12.1 Operator Member

- It is possible to use operator symbols for routine names:

```
struct Complex {
    ...
    Complex operator+( Complex c ) {
        return (Complex){ re + c.re, im + c.im }; // remove sum
    }
};
```

- Addition routine is called +, and x and y can be added by x.**operator**+(y) or y.**operator**+(x), which is only slightly better.
- For convenience, C++ implicit 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;
cout << "sum:" << sum.re << "+" << sum.im << "i" << endl;
```

2.12.2 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.
- Constructor without parameters is the **default constructor**, for initializing a new object to a default value.

```
Complex x;
Complex *y = new Complex;
```
implicitly rewritten as
```
Complex x;   x.Complex();
Complex *y = new Complex;
             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***.
- 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 parenthesis to invoke default constructor for local declarations.***

```
Complex x(); // routine with no parameters and returning a complex
```

- ***Once a constructor is specified, structure initialization is disallowed:***

```
Complex x = { 3.2 };          // disallowed
Complex y = { 3.2, 4.5 };     // disallowed
```

- Replaced using overloaded constructors with parameters:

```
struct Complex {
    double re, im;
    Complex() { re = 0.; im = 0.; }
    Complex( double r ) { re = r; im = 0.; }
    Complex( double r, double i ) { re = r; im = i; }
    ...
};
```

- 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();
Complex y; y.Complex(1.0);
Complex z; z.Complex(6.1, 7.2);
```

- Dynamic allocation is same as Java:

```
Complex *x = new Complex(); // parenthesis optional
Complex *y = new Complex(1.0);
Complex *z = new Complex(6.1, 7.2);
```

- ***If only non-default constructors are specified, an object cannot be declared without an initialization value:***

```
struct Foo {
    Foo( int i ) { ... }
};
Foo x;          // disallowed!!!
Foo x( 1 );     // allowed
```

Must create a default constructor to allow first declaration.

- Unlike Java, constructor cannot be called explicitly at start of another constructor, so constructor reuse 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 ); }` `};`

2.12.2.1 Constant

- Constructors can be used to create object constants (like g++ type-constructor constants):

```
Complex x, y, z;
x = Complex( 3.2 );              // complex constant with value 3.2+0.0i
y = x + Complex(1.3, 7.2);       // complex constant with value 1.3+7.2i
z = Complex( 2 ); // 2 widened to 2.0, complex constant with value 2.0+0
```

- Previous operator + for Complex is changed because type-constructor constants are disallowed for a type with constructors:

```
Complex operator+( Complex c ) {
    return Complex( re + c.re, im + c.im ); // create new complex value
}
```

2.12.2.2 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 constants 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.
- Implicit constructor conversion is turned off with qualifier **explicit**:

```
struct Complex {
    ...
    explicit Complex( double r ) { re = r; im = 0.; } // turn off
    ...                                               // implict conversion
};
```

- However, this capability fails for commutative binary operators.
- 1.3 + x, fails 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 infixed 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	`operator+(x, y)`
`1.3 + x;`	rewritten as	`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?

2.12.3 Random Numbers

- **Random number**s 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:
  - online gambling, computer simulation, cryptography, computer graphics, 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 output is predictable.
- All **pseudo random-number generator**s (PRNG) involve some technique for 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., generate a ***fixed*** sequence of LARGE random values that repeats after 2^{32} values (but might repeat earlier):

```
class PRNG {
    uint32_t  seed_;      // 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_MA
        seed_ = 36969 * (seed_ & 65535) + (seed_ >> 16); // scramble
        return seed_;
    }
    uint32_t operator()( uint32_t  u ) {                    // [0,u]
        return operator()() % (u + 1);
    }
    uint32_t operator()( uint32_t l, uint32_t u ) {         // [l,u]
        return operator()( u - l ) + l;
    }
};
```

- Creating a member with the function-call operator name, (), (functor) allows these objects to behave like a routine.

```
PRNG prng;
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( getpid() );
```

2.12.4 Copy Constructor / Assignment

- There are multiple contexts where an object is copied.

1. declaration initialization (`ObjType obj2 = obj1`)
2. routine call (argument $\Rightarrow$ parameter)
3. assignment (`obj2 = obj1`)

- Cases 1 & 2 involve a newly allocated object with undefined values (unless a member has a constructor).
- Case 3 involves an existing object that may contain previously computed values.
- C++ differentiates between these situations: initialization and assignment.
- Constructor with a **const** reference parameter is used for initialization (declarations and parameters), 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 initialization:

```
Complex rtn( Complex a, Complex b );
Complex x, y;
rtn( x, y )
```

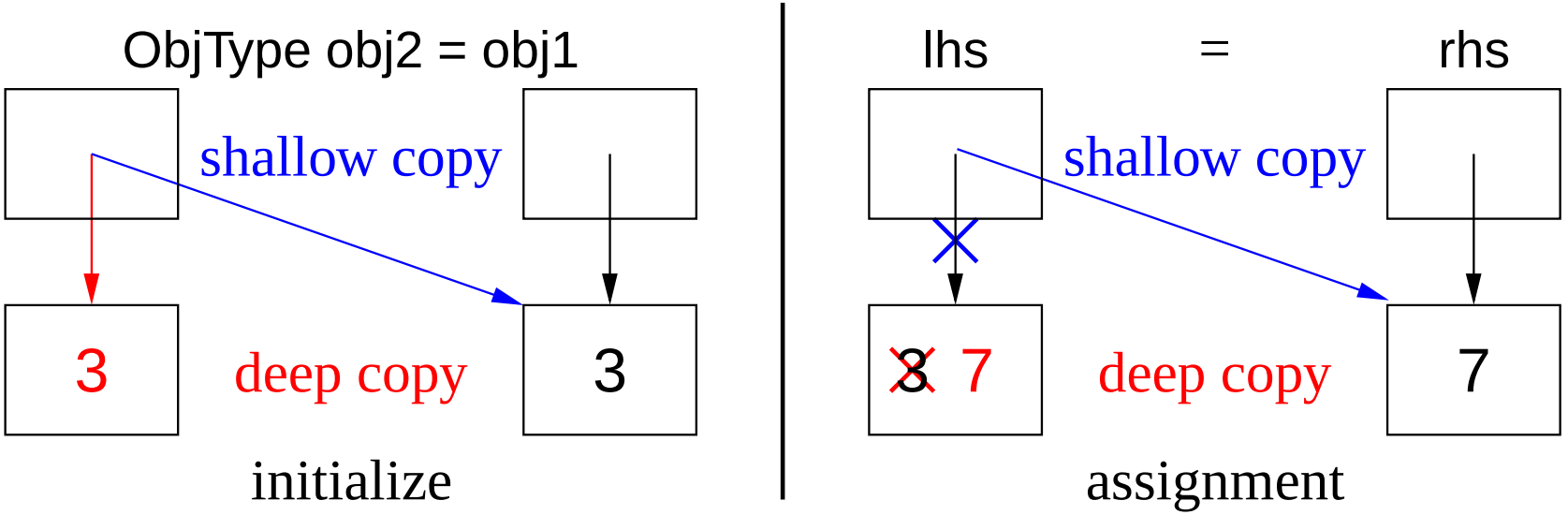
- call results in the following implicit action in rtn:

```
Complex rtn( Complex a, Complex b ) {
    a.Complex( x ); b.Complex( y ); // initialize parameters with argume
```

- If a copy constructor is not defined, an implicit one is generated that does a **shallow copy** (**bit-wise copy**), i.e., copies the object including pointers.
- Assignment routine is used for assignment:

```
Complex &operator=( const Complex &c ) { . . . }
```

  - value on the right-hand side of "=" is argument to assignment operator.
  - usually most efficient to use reference for parameter and return type.
- If an assignment operator is not defined, an implicit one is generated that does a shallow copy.
- When an object type contains pointers, it is often necessary to do a **deep copy**, i.e, copy the contents of the pointed-to storage rather than the pointers.



2.12.5 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 { Bar(**int** i) {. . .} *// no default constructor* } bar(3); **struct** Foo { **const int** i **= 3**; Bar * **const** p **= &bar**; Bar &rp **= bar**; Bar b**(7)**; } x;	**struct** Bar { Bar(**int** i) {. . .} *// no default constructor* } bar(3); **struct** Foo { **const int** i; Bar * **const** p; Bar &rp; Bar b; } x = { 3, &bar, bar**, 7** };

- Left: not allowed because fields cannot be directly initialized.
- Right: not allowed because cannot supply argument to b using this syntax.
- Try using a constructor:

Constructor/assignment	Constructor/initialize
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* } };	**struct** Foo { **const int** i; Bar * **const** p; Bar &rp; Bar b; Foo() : *// declaration order* i(3), p(&bar), rp(bar), b(7) { } };

- Left: not allowed 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 {
    ...
    int k;
    Foo() : ..., k( 14 ) {      // initialize k
        k = 14                  // or assign k
    }
};
```

2.12.6 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 depends/changes its environment***, e.g., opening/closing files, allocating/freeing dynamically allocated storage, etc.
- An **independent object**, like a Complex object, requires no destructor.
- 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 dependences***, 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 dependent 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
    ...
};
```

- 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.

2.13 Type Nesting

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

```
struct Foo {
    enum Colour { R, G, B };          // nested type
    int g;
    int r() { ... }
    struct Bar {                      // nested type
        Colour c;                     // Ok, static reference
        int s() { g = 3; r(); }       // fails, dynamic reference
    };
};
Foo::Colour colour = Foo::R;          // must qualify
Foo f;   f.g;   f.r();
Foo::Bar b;   b.c;   b.s();           // must qualify
```

- References inside the nested type do not require qualification.
- ***However, nesting aggregate types only imply static scoping not dynamic***.
- Hence, references in s to members g and r in Foo fail because no dynamic scope relationship between types Bar and Foo.
- References outside the object must be qualified with type operator "::".
- C++ selection operator ".", e.g., Foo.Colour, cannot be used because it requires an object not a type.

2.14 Declaration Before Use

- C/C++ have **Declaration Before Use** (DBU), e.g., a variable declaration must appear before its usage in a block:

```
{
        cout << i << endl;
        int i = 4;              // declaration after usage
}
// prints 4
```

- A compiler can handle some DBU situations, but there are ambiguous cases:

```
int i = 3;
{
        cout << i << endl;      // which i?
        int i = 4;
        cout << i << endl;
}
// prints 3 4
```

- 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 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.

```
void g( int i );         // forward declarations
void f( int i );
int main();
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 (usually) require DBU within a type:

Java	C++
// any g must be nested in a class **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* **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 DB* 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, 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 fails because DBU on 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.

2.15 Abstraction/Encapsulation

- **Abstraction** is the separation of interface and implementation allowing an object's implementation to change without affecting usage, which is essential for reuse and maintenance.
- E.g., a user of type Complex should not have or need direct access its implementation to perform operations:

```
struct Complex {
    double re, im;    // implementation data
    ... // interface routine members
};
```

- Possible to change from Cartesian to polar coordinates and user interface remains constant.
- ***Developing good interfaces for objects is important.***
- **Encapsulation** is hiding the implementation for security or financial reasons (**access control**).
- **Abstract data-type** (ADT) is a user-defined type that practices abstraction and encapsulation.

- ***Abstraction and encapsulation are neither essential nor required to develop software.***
- E.g., users could follow a convention of not directly accessing the implementation.
- However, relying on users to follow conventions is dangerous.
- 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.
- Like Java, C++ provides 3 levels of visibility 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.
- Only the object type can access the private members, ***so implementation members are normally private***.
- Public members define an object type's **interface**, i.e., what a user can access.
- While a user can see private and protectedmembers, they cannot be accessed, preventing user code from violating abstraction.
- **struct** has an implicit **public** inserted at the beginning, i.e., all members are public.
- **class** has an implicit **private** inserted at the beginning, i.e., 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 abstraction to preclude object copying by hiding copy constructor and assignment operator:

```
class Foo {
      Foo( const Foo & );              // undefined
      Foo &operator=( Foo & );         // undefined
   public:
      ...
};
Foo x, y;
rtn( x );        // fails for pass by value
x = y;           // fails
```

- Useful to prevent object forgery (lock, boarding-pass, receipt) or copying that does not make sense (file, database).

- Encapsulation introduces a new problem for routines outside of an object used to implement binary operations for an object.
- An outside routine may need to access an object's implementation, but it cannot access private members.
- C++ provides a mechanism to state that an outside routine is allowed access to its implementation, called **friendship** (similar to package visibility in Java).

```
class Complex {
    friend Complex operator+(Complex a, Complex b);
    ...
};
Complex operator+(Complex a, Complex b) { ... }
```

- The **friend** prototype indicates a routine with the specified name and type may access this object's implementation:

```
class Complex {
    friend Complex operator+(Complex a, Complex b);
    friend ostream &operator<<( ostream &os, Complex c );
    double re, im;
  public:
    double abs() { return sqrt( re * re + im * im ); }
    Complex() { re = 0.; im = 0.; }
    Complex(double r) { re = r; im = 0.; }
    Complex(double r, double i) { re = r; im = i; }
};
Complex operator+( Complex a, Complex b ) { ... }
ostream &operator<<( ostream &os, Complex c ) { ... }
```

2.16 Separate Compilation

- Like Java, C/C++ use **source file**s to provide another mechanism for encapsulation.

file.java

```
enum Colour { R, G, B };      // export
class C {                     // export
    private static int i;     // private
    private static void f() {} // private
    public static int j;      // export
    public static void g() {} // export
}
class D {                     // export
    private int i;            // private
    private void f() {}       // private

    public int j;             // public
    public void g() {}        // public
}
```

file.cc

```
enum Colour { R, G, B }; // local

static int i;            // private
static void f() {}       // private
int j;                   // export
void g() {}              // export

class D {                // local
    int i;               // private
    void f();            // private
  public:
    int j;               // public
    void g();            // public
}
```

- Like Java, C/C++ implicitly exports variables and routines from a source file.
- In C/C++, to encapsulate global variables and routines in a source file, the declaration must be qualified with **static**.
- Unlike Java, C/C++ do NOT implicitly export types from a source file.
- Java implicitly looks in *.**class** files for exported content.

- C/C++ require the use of the preprocessor and forward declarations to access exported content.
- Programmer must explicitly divided program into interface and implementation in two (or more) files.
- Interface is composed of the prototype declaration(s) (but possibly some implementation).
- Implementation is composed of actual declarations and code.
- Interface is entered into one or more include files (.h files), and the implementation is entered into one or more source files (.cc files).

file.java

```
enum Colour { R, G, B };      // export
class C {                     // export
    private static int i;     // private
    private static void f() {} // private
    public static int j;      // export
    public static void g() {} // export
}
class D {                     // export
    private int i;            // private
    private void f() {}       // private

    public int j;             // public
    public void g() {}        // public
}
```

file.h—file.cc

```
enum Colour { R, G, B }; // public



extern int j;            // public
extern void g();         // public

class D {                // public
    int i;               // private
    void f();            // private
  public:
    int j;               // public
    void g();            // public
}
```

```
static int i;            // private
static void f() {}       // private
int j;                   // public
void g() {}              // public
void D::f() {}           // private
void D::g() {}           // public
```

- **extern** qualifier means the actual variable or routine definition is located elsewhere.
- **extern** on routine prototypes is optional, but good documentation.
- ***Static class-variables must be declared once (versus defined) in a .cc file.***

.h	.cc
class C { **static** **char** c; *// defn* . . .	**char** C::c = ′a′; *// decl*

- Encapsulation is provided by giving a user access to the include file(s) and the compiled source file(s), but not the implementation in the source file(s).
- Most software supplied from software vendors comes this way.
- E.g., Complex prototype information is placed into file complex.h, which users include in their programs.

```
#ifndef __COMPLEX_H__
#define __COMPLEX_H__          // protect against multiple inclusion
#include <iostream>            // access: ostream
// inject no names, use qualification
extern void complexStats();    // interfaces
class Complex {
    friend Complex operator+( Complex a, Complex b );
    friend std::ostream &operator<<( std::ostream &os, Complex c );
    double re, im;             // exposed implementation
  public:
    Complex();
    Complex( double r );
    Complex( double r, double i );
    double abs();
};
extern Complex operator+( Complex a, Complex b );
extern std::ostream &operator<<( std::ostream &os, Complex c );
#endif // __COMPLEX_H__
```

- Complex implementation information is placed in file complex.cc.

```
#include "complex.h"            // do not copy interface
#include <cmath>                // access: sqrt
using namespace std;            // inject names
// global, private declarations
static int cplxObjCnt = 0;      // private, defaults to 0
// interface declarations
void complexStats() { cout << cplxObjCnt << endl; }
Complex::Complex() { re = 0.; im = 0.; cplxObjCnt += 1; }
Complex::Complex( double r ) { re = r; im = 0.; cplxObjCnt += 1; }
Complex::Complex(double r, double i) { re = r; im = i; cplxObjCnt += 1;
double Complex::abs() { 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";
}
```

- **.cc *file includes the* .h *file so that there is only one copy of the constants, declarations, and prototype information.***
- cplxObjCnt is qualified with **static** to make it a private variable to this source file.

- No user can access it, but each constructor implementation can increment it when a Complex object is created.
- Users call complexStats to print the number of Complex objects created so far in a program.
- Notice, all the member routines of Complex are separated into a forward declaration and an implementation after the object type, allowing the implementation to be placed in the .cc file.
- Note, by reading the .h file, it may be possible to determine the implementation technique used, so there is only partial encapsulation.
- To provide complete encapsulation requires abstract type and (more expensive) references:

```
#ifndef __COMPLEX_H__
#define __COMPLEX_H__          // protect against multiple inclusion
#include <iostream>            // access: ostream
// inject no names, use qualification
extern void complexStats();    // interfaces
class Complex {
    friend Complex operator+( Complex a, Complex b );
    friend std::ostream &operator<<( std::ostream &os, Complex c );
    struct ComplexImpl;        // hidden implementation, nested class
    ComplexImpl &impl;         // indirection to implementation
  public:
    Complex();
    Complex( double r );
    Complex( double r, double i );
    ~Complex();
    Complex( const Complex &c );  // copy constructor
    Complex &operator=( const Complex &c ); // assignment operator
    double abs();
};
extern Complex operator+( Complex a, Complex b );
extern std::ostream &operator<<( std::ostream &os, Complex c );
#endif // __COMPLEX_H__
```

```
#include "complex.h"                // do not copy interface
#include <cmath>                    // access: sqrt
using namespace std;                // inject names
// global, private declarations
static int cplxObjCnt = 0;          // private, defaults to 0
struct Complex::ComplexImpl {       // actual implementation, nested class
    double re, im;
};
// interface declarations
void complexStats() { cout << cplxObjCnt << endl; }
Complex::Complex() : impl(*new ComplexImpl) {
    impl.re = 0.; impl.im = 0.; cplxObjCnt += 1;
}
Complex::Complex( double r ) : impl(*new ComplexImpl) {
    impl.re = r; impl.im = 0.; cplxObjCnt += 1;
}
Complex::Complex( double r, double i ) : impl(*new ComplexImpl) {
    impl.re = r; impl.im = i; cplxObjCnt += 1;
}
Complex::~Complex() { delete &impl; }
Complex::Complex(const Complex &c) : impl(*new ComplexImpl) {
    impl.re = c.impl.re; impl.im = c.impl.im; cplxObjCnt += 1;
}
```

```
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.im );
}
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 must be used because complex objects now contain a reference pointer to the implementation.
- E.g., copying the reference pointer can result in two complex objects pointing at the same complex value and both may eventually attempt to delete it (dangling pointer).
- As well, overwriting a reference pointer may lose the only pointer to the storage so it can never be freed (memory leak).
- An encapsulated object is compiled using the -c compilation flag and subsequently linked with other compiled source files to form a program:

```
g++ -c complex.cc
```

- Creates file complex.o containing a compiled version of the source code.
- To use an encapsulated object, a program specifies the necessary include file(s) to access the object's interface:

```
#include "complex.h"
#include <iostream>
using namespace std;
int main() {
    Complex x, y, z;
    x = Complex( 3.2 );
    y = x + Complex( 1.3, 7.2 );
    z = Complex( 2 );
    cout << "x:" << x << " y:" << y << " z:" << z << endl;
}
```

- Then links with any necessary executables:

```
g++ usecomplex.cc complex.o # other .o files if necessary
```

- Notice, iostream is included twice, once in this program and once in complex.h, which is why each include file needs to prevent multiple inclusions.

2.17 Inheritance

- Object-***oriented*** languages provide **inheritance** for writing general, 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.

2.17.1 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 compose and debug a new object type.
- One way to understand this technique is to model it via explicit inclusion:

Inclusion	Inheritance
struct Base { **int** i; **int** r(...) { ... } Base() { ... } }; **struct** Derived { Base b; *// explicit inclusion* **int** s(...) { b.i = 3; b.r(...); ... } Derived() { ... } } d; d.b.i = 3; *// inclusion reference* d.b.r(...); *// inclusion reference* d.s(...); *// direct reference*	**struct** Base { **int** i; **int** r(...) { ... } Base() { ... } }; **struct** Derived : **public** Base { *// implicit in* **int** s(...) { i = 3; r(...); ... } Derived() { ... } } d; d.i = 3; *// direct reference* d.r(...); *// direct reference* d.s(...); *// direct reference*

- Inclusion implies explicitly creating an object member, b, to aid in the implementation.
- Object type Derived inherits from Base type via "**public** Base" clause.
- Inheritance implicitly:
  - creates an anonymous object member

- ***opens*** 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 inclusion.

```
                                  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 hides (overrides) outer-scope name.

- Still possible to access outer-scope names using "::" qualification to specify the particular nesting level.

Java	C++
class Base1 { **int** i; } **class** Base2 **extends** Base1 { **int** i; } **class** Derived **extends** Base2 { **int** i; **void** s() { **int** i = 3; **this**.i = 3; ((Base2)**this**).i = 3; *// super.i* ((Base1)**this**).i = 3; } }	**struct** Base1 { **int** i; }; **struct** Base2 : **public** Base1 { **int** i; *// hides Base1::i* }; **struct** Derived : **public** Base2 { **int** i; *// hides Base2::i* **void** r() { **int** i = 3; *// hides Derived::i* Derived::i = 3; *// this.i* Base2::i = 3; Base2::Base1::i = 3; } };

- E.g., Derived declaration first create an invisible Base object in the Derived object, like inclusion, 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.
- In general, routines provide smaller units for reuse than entire objects.

2.17.2 Type Inheritance

- Type inheritance extends name equivalence to allow routines to handle multiple types, called **polymorphism**, e.g.:

```
struct Foo {
    int i;
    double d;

} f;
void r( Foo f ) { ... }
r( f );    // valid call
r( b );    // should also work
```

```
struct Bar {
    int i;
    double d;
    ...
} b;
```

- Since types Foo and Bar are identical, 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) even though b is structurally identical to f.
- ***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.
- Now variables of type Complex are redeclared to Mycomplex, and member calls returns the current number of calls to abs for any Mycomplex object.
- Implementation inheritance provides reuse ***inside*** an object type; type inheritance provides reuse ***outside*** the object type by allowing existing code to access the base type.
- I.e, any routine that manipulates the base type also manipulates the derived type.
- 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, the result type from **operator**+ is Complex, not Mycomplex.
- Assignment of a complex (base type) to Mycomplex (derived type) fails 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.

2.17.3 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.
- Order is 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 the 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) {. . .} };

2.17.4 Overloading

- Overloading a member routine in a derived class hides 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() {}    // hides both versions of mem in base class
};
```

- Hidden base-class members can still be accessed:
  - Selectively provide explicit s 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; // bring all base mem routines into this interface
};
```

- Use explicit qualification to call members (violates abstraction).

```
Derived d;
d.Base::mem( 3 );
d.Base::mem( 'a' );
d.mem();
```

2.17.5 Abstract Class

- **Abstract class** combines type and implementation inheritance for structuring new types.

Java

```
abstract class Shape {
      private int colour;

      public abstract void move( int x, int y );
}
abstract class Polygon extends Shape {
      private int edges;

      public abstract int sides();
}
class Rectangle extends Polygon {
      private int x1, y1, x2, y2;

      public void move( int x, int y ) {...}
      public int sides() { return 4; }
}
class Square extends Rectangle {

      public void move( int x, int y ) {...}
}
```

C++

```
class Shape {
      int colour;
   public:
      virtual void move( int x, int y ) = 0;
};
class Polygon : public Shape {
      int edges;
   public:
      virtual int sides() = 0;
};
class Rectangle : public Polygon {
      int x1, y1, x2, y2;
   public:
      void move( int x, int y ) {...}
      int sides() { return 4; }
};
class Square : public Rectangle {
   public:
      void move( int x, int y ) {...}
};
```

- Strange initialization to 0 means this member *must* be defined by any derived type.

- ***Cannot instantiate objects from an abstract class, but can declare pointer/reference to it.***

2.17.6 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

```
interface Polygon {

    public int sides();
    public void move( int x, int y );
}
interface Rectilinear {

    final public int angle = 90;
}
class Rectangle implements Rectilinear,
                           Polygon {
    private int x1, y1, x2, y2;

    public void move( int x, int y ) {}
    public int sides() { return 4; }
}
class Square extends Rectangle {

    public void move( int x, int y ) {}
}
```

C++

```
class Polygon {
  public:
    virtual int sides() = 0;
    virtual void move( int x, int y ) = 0;
};
class Rectilinear {
  public:
    enum { angle = 90 };
};
class Rectangle : public Polygon,
                  public Rectilinear {
    int x1, y1, x2, y2;
  public:
    void move( int x, int y ) {}
    int sides() { return 4; }
};
class Square : public Rectangle {
  public:
    void move( int x, int y ) {}
};
```

- ***Restrict multiple inheritance to one* public *type and one or two* private *types.***

2.17.7 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 the routine defined in the referenced object, qualify the member routine with virtual.
- To invoke the routine defined by the type of the pointer/reference, do not qualify the 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 { **public void** f() {} *// virtual* **public void** g() {} *// virtual* **public void** h() {} *// virtual* } **class** Derived **extends** Base { **public void** g() {} *// virtual* **public void** h() {} *// virtual* } **final** Base bp = **new** Derived(); bp.f(); *// Base.f* ((Base)bp).g(); *// Derived.g* bp.g(); *// Derived.g* ((Base)bp).h(); *// Derived.h* bp.h(); *// Derived.h*	**struct** Base { **void** f() {} *// non-virtual* **void** g() {} *// non-virtual* **virtual void** h() {} *// virtual* }; **struct** Derived : **public** Base { **void** g() {}; *// non-virtual* **void** h() {}; *// virtual* }; Base &bp = *__new__ Derived(); *// polymorphic ass* bp.f(); *// Base::f, pointer type* bp.g(); *// Base::g, pointer type* ((Derived &)bp).g(); *// Derived::g, pointer type* bp.Base::h(); *// Base::h, explicit selection* bp.h(); *// Derived::h, object type*

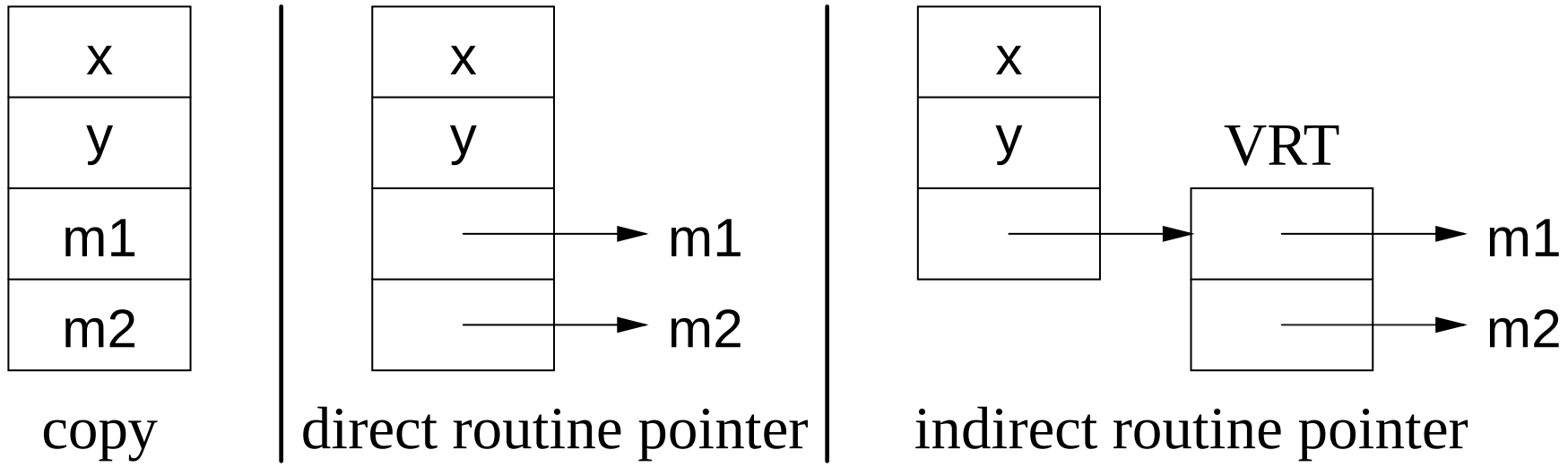
- Java casting does not provide access to base-type's member routines.
- ***Virtual members are only necessary to access derived members through a base-type reference or pointer.***
- If a type is not involved in inheritance (final class in Java), virtual members

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 programming 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:



2.17.8 Down Cast

- Type inheritance can mask the actual type of an object through a pointer/reference.
- Like Java, C++ provides a mechanism to dynamically determine the actual type of an object pointed to by a polymorphic pointer/reference.
- The Java operator instanceof and the C++ operator **dynamic_cast** 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.***

2.17.9 Abstraction

- 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; };      // y and z allowed
};
int main() {
    Derived d;
    d.x; d.y; d.z;               // z allowed
}
```

2.18 Template

- Inheritance provides reuse for types organized into a hierarchy that extends name equivalence.
- 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 abs( int val ) { return val >= 0 ? val : -val; }
double abs( double val ) { return val >= 0 ? val : -val; }
```

- Template routine eliminates duplicate code by using types as compile-time parameters:

```
template<typename T>  T abs( T val ) { return val >= 0 ? val : -val; }
```

- **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 << abs( 1 ) << " " << abs( -1 ) << endl; // T -> int
cout << abs( 1.1 ) << " " << abs( -1.1 ) << endl; // T -> double
```

- 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 N = 10> struct Stack {
    T elems[N];  // maximum N elements
    int size;
    Stack() { size = 0; }
    void push( T e ) { elems[size] = e; size += 1; }
    T pop() { size -= 1; return elems[size]; }
};
```

- Type parameter, T, declares the element type of array elems, and return and parameter types of the member routines.
- Integer parameter, N, denotes the maximum stack size.
- For template types, the compiler cannot infer the type parameter, so it must be explicitly specified:

```
Stack<int, 20> si;              // stack of int
Stack<double> sd;               // stack of double
Stack< Stack<int> > ssi;        // stack of stack of int
si.push(3);
sd.push(3.0);
ssi.push( si );
int i = si.pop();
double d = sd.pop();
si = ssi.pop();
```

- ***There must be a space between the two ending chevrons or >> is parsed as operator>>.***
- ***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.***

2.18.1 Standard Library

- C++ Standard Library provides different kinds of containers: vector, map, list, stack, queue, deque.
- In general, nodes are either copied into the container or pointed to from the container.

- Copying implies node type must have default and/or copy constructor so instances can be created without having to know constructor arguments.
- ***Standard library containers use copying and requires node type to have a default constructor.***
- Most containers use an **iterator** to traverse its nodes so knowledge about container implemented is hidden.
- Iterator capabilities depend on container, e.g., a singly-linked list has unidirectional traversal, doubly-linked list has bidirectional traversal, etc.
- Containers provides iterator(s) as a nested object type, e.g., list<Node> has list<Node>::iterator.
- Iterator operator "++" moves forward to the next node, until ***passed*** the end of the container.
- For bidirectional iterators, operator "--" moves in the reverse direction to "++".

2.18.1.1 Vector

- Like Java array, vector has random access, length, subscript checking (at), and assignment; vector also has dynamic sizing.

std::vector<T>	
vector() vector(**int** n)	create empty vector create vector with n empty elements
int size() **bool** empty() T **operator**[](**int** i) T at(**int** i)	vector size size() == 0 access ith element, NO subscript checking access ith element, subscript checking
vector &**operator**=(**const** vector &) **void** push_back(**const** T &x) **void** pop_back() **void** resize(**int** n) **void** clear()	vector assignment add x after last element remove last element add or erase elements at end so size() == n erase all elements

- vector is alternative to C/C++ arrays.

```
#include <vector>
int i, elem;
vector<int> v;                          // think: int v[0]
for ( ;; ) {
      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
```

- Dynamic sizing implies vector's elements are allocated on the heap.
- 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:

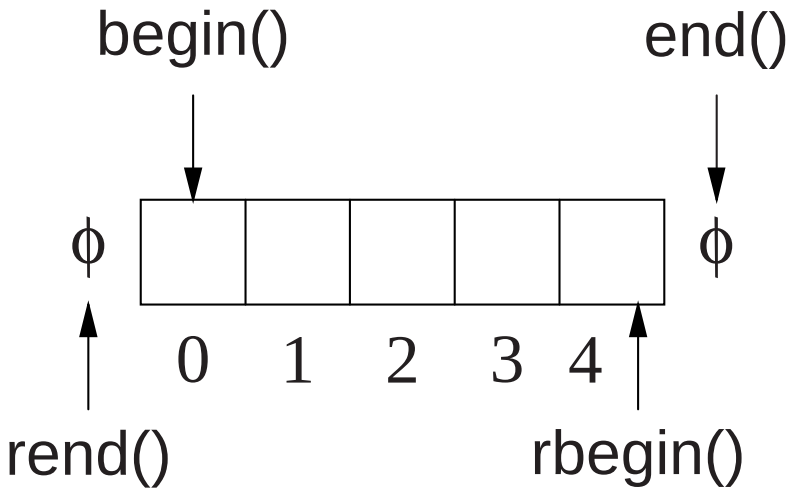
```
vector< vector<int> > m;
```

- Again, it is more efficient to dimension, when size is known.

```
#include <vector>
vector< vector<int> > m( 5 ); // 5 rows
for ( int r = 0; r < m.size(); r += 1 ) {
    m[r].resize( 4 );         // 4 columns per row
    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;
}
```

- Cannot specify number of columns at declaration, so each row is zero sized.
- Before values can be assigned into a row, a row can be dimensioned to a specific size, m[r].resize(4).
- All loop bounds are controlled using dynamic size of the row or column.
- Iterator is necessary for management operations (versus iterating using subscripting).

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



- **erase *and* insert *should take subscript so iterator is unnecessary!***
- Iterator returns a pointer to an element.

```
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

// find position of value 4 using iterator (versus subscript)
vector<int>::iterator f;
for ( f = v.begin(); f != v.end() && *f != 4; f ++ );
v.insert( f, 33 );                    // insert before position with value 4

// print reverse order using iterator (versus subscript)
vector<int>::reverse_iterator r;
for ( r = v.rbegin(); r != v.rend(); r ++ )
      cout << *r << endl;
```

- ***Cannot insert or erase during iteration using an iterator.***

2.18.1.2 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() **bool** empty() T **operator**[ ](**int** i) **int** count(Key key)	map size size() == 0 access ith element 0 $\Rightarrow$ no key, 1 $\Rightarrow$ key
map &**operator**=(**const** map &) insert(pair<Key,Val>(k, v)) erase(Key k) **void** clear()	map assignment insert pair erase key k erase all elements

- First subscript for key creates an entry and initializes it to default or specified value.

```
map<string, int> m, c;          // Key => string, Val => int
m["red"];                       // create, set to 0 for int
m["green"] = 1;                 // create, set to 1
m["blue"] = 2;                  // create, set to 2
m["green"] = 5;                 // overwrite 1 with 5
cout << m["green"] << endl;
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 )
m.clear();                      // remove ALL elements
```

- Iterator to search and return values in key order.

std::map<T>::iterator / std::map<T>::reverse_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 find(Key &k)	find position of key k
iterator insert(iterator posn, **const** T &x)	insert x before posn
iterator erase(iterator posn)	erase element at posn

- Iterator returns a pointer to an element 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;
```

2.18.1.3 Single/Double Linked

- If random access is not required, use more efficient single (stack/queue/deque) or double (list) linked-list container.
- Examine list, stack/queue/deque are simpler.

std::list<T>	
list() list(**int** n)	create empty list create list with n empty elements
int size() **bool** empty()	list size size() == 0
list &**operator**=(**const** list &) T front() T back() **void** push_front(**const** T &x) **void** push_back(**const** T &x) **void** pop_front() **void** pop_back() **void** clear()	list assignment first element last element add x before first element add x after last element remove first element remove last element erase all elements

- Iterator returns a pointer to a node.

std::list<T>::iterator / std::list<T>::reverse_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

```
#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
    Node n( 'a'+i, i, i+0.5 );          // node to be added
    dl.push_back( n );                  // copy node at 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
}
```

2.18.1.4 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>
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 by constructing a “function object”, called a **functor**, using the routine-call operator.

- 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.

2.19 Namespace

- C++ **namespace** is used to organize programs and libraries composed of multiple types and declarations.

- 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 abstract/encapsulation (use .h/.cc files).
- Unlike Java packages, C++ allows multiple namespaces to be defined in a file, as well as, among files.
- 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* --- **package** Foo; *// file* **public** enum Y . . . *// export one type* *// local types / declarations* --- **package** Bar; *// file* **public class** Z . . . *// export one type* *// local types / declarations*	**namespace** Foo { *// types / declarations* } **namespace** Foo { *// more types / declarations* } **namespace** Bar { *// types / declarations* }

- Contents of a namespace can be 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 conditionally importing all of the namespace content.

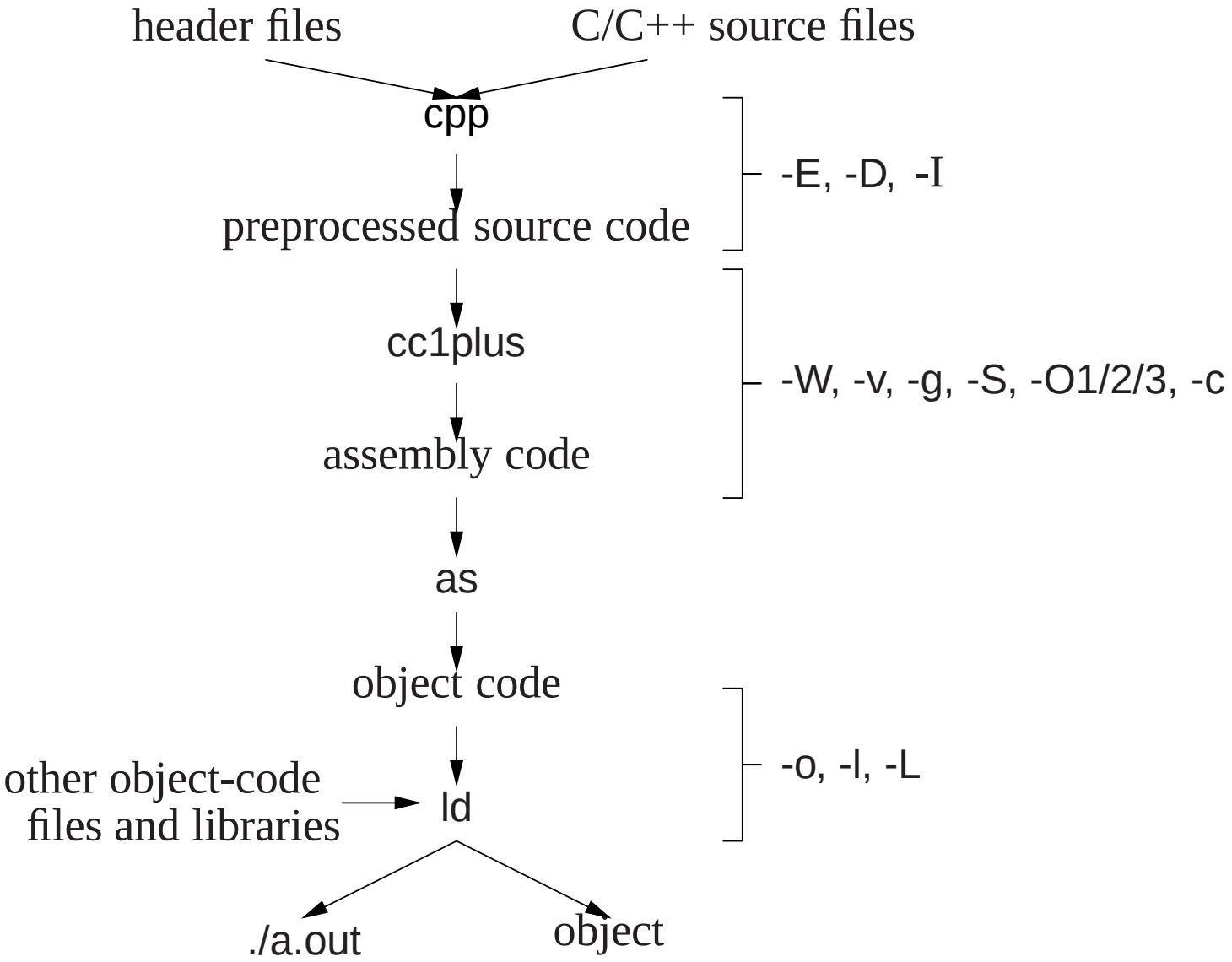
Java	C++
`import Foo.T;`	`using Foo::T;` *// import individual (conflicts)*
`import Foo.*;`	`using namespace Foo;` *// import all (non-conflicting)*

- Global variables are in an unnamed namespace accessible with unqualified "`::`".

```
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;
        int j = 5;
    }
}
int j = 0;                      // global
int main() {
    int j = 3;                  // local
    cout << j << endl;          // local
    cout << ::j << endl;        // global
    using namespace Foo;        // conditional import: Colour, i, C, Bar (not j)
    Colour c;                   // Foo::Colour
    C x;                        // Foo::C
    cout << i << endl;          // Foo::i
    using Foo::j;               // import: conflict
    cout << Foo::j << " " << Bar::j << endl;  // qualification
    using namespace Bar;        // conditional import: shrint (not j)
    shrint s = 4;               // Bar::shrint
}
```

3 Tools

3.1 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 ...
```

3.1.1 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
```

- If both -D and **#define** for same name, **#define** redeclares name.
- -I directory search directory for include files; can be referenced by name using <. . .>.

3.1.2 Compiler (cc1plus)

- Compiler (cc1plus) takes a preprocessed file and converts the C++ language into assembly language for the target machine.
- Options:
  - -Wkind generate warning message for this "kind" of situation.
    - -Wall print ALL warning messages.

- -Werror make warnings into errors so program does not compile until fixed.

- -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

3.1.3 Assembler

- Assembler (as) takes an assembly language file and converts it to object code (machine language).

3.1.4 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:
  - -o gives the file name where the combined object/ executable is placed.
    - If no name is specified, default name a.out is used.
  - -l library search library when linking, e.g., -lm for math library
  - -L directory search in directory for library

3.2 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.

3.2.1 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 (**#if**) 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 completely.
- When you go for help, either from your instructor or an advisor, you should have debug print statements in your program.
- In general, debug print statements never appear in the program you hand in for marking.

3.2.2 Assertions

- **Assertion**s enforce pre-conditions, post-conditions, and invariants, which document program assumptions.

- Macro assert provides a mechanism to perform a check, and if the check fails, to print the check and abort the program.

```
int main() {
    vector<int> a, b;
    // read values into a, b
    assert( ("must be the same size", a.size() == b.size()) );
    for ( int i = 0; ; i += 1 ) {
        assert( ("must have an unequal element", i < a.size()) );
      if ( a[i] != b[i] ) break;
        ...
    }
}
```

- Note, use of comma expression.
- When run with incorrect data produces:

```
% ./a.out
Assertion failed: ("must be the same size", a.size() == b.size()), file test
Abort (core dumped)
```

- Assertions can significantly increase a program's cost.
- Compiling a program with preprocessor variable NDEBUG defined removes all asserts.

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

3.2.3 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:

  - Forgetting a closing " or */, as the remainder of the program is *swallowed* as part of the character string or comment.
  - Missing a { or }, especially if the program is properly indented (editors can help here)

- **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 that caused the error is not usually 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 able to reason backwards. This is 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 account for all errors
- E.g., an infinite loop with nothing wrong with the loop; the initialization is wrong.

```
i = 10;
while ( i != 5 ) {
    ...
    i += 2;
}
```

- Difficult semantic errors are:
  - Forgetting to assign a value to a variable before using it in an expression.
  - Using an invalid subscript or pointer value.

- 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)

3.3 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.

3.3.1 GDB

- The two most common UNIX debuggers are: dbx and gdb.
- File test.cc contains:

```
1  void r( int a[] ) {
2      int i = 100000000;
3      a[i] += 1;     // really bad subscript error
4  }
5  int main() {
6      int a[10] = { 0, 1 };
7      r( a );
8  }
```

- 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.
- **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.

- **ba**cktrace command prints a stack trace of called **routine activation**s.

```
(gdb) backtrace
#0  0x000106f8 in r (a=0xffbefa08) at test.cc:3
#1  0x00010764 in main () at test.cc:7
```

- **p**rint command prints variables accessible in the current routine, object, or external area.

```
(gdb) print i
$1 = 100000000
```

- $1 is the name of a history variable (like history variables in a shell).
- Name $*N* can be used in subsequent commands to access previous values of i.

- 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
(gdb) p $3
$6 = 0
```

- **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
$7 = 7
(gdb) set var a[0] = 3
(gdb) p a[0]
$8 = 3
(gdb) p $3
$9 = 0
```

- 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 activation 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:7
7           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 required.
- **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 6.
(gdb) break test.cc:3
Breakpoint 2 at 0x106d8: file test.cc, line 3.
```

- Set breakpoint using routine name or source-file:line-number.
- If program is not compiled with -g flag, only the location is given.
- Command info breakpoints prints breakpoints currently set.

```
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x00010710 in main at test.cc:6
2   breakpoint     keep y   0x000106d8 in r(int*) at test.cc:3
```

- Breakpoints numbered consecutively from 1 and can be disabled, enabled or deleted at any time using commands:

```
(gdb) disable 1        temporarily disable breakpoint 1
(gdb) enable 1         re-enable disabled breakpoint 1
(gdb) delete 1         remove breakpoint completely 1
```

(Pretend none of these commands are entered.)

- 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:6
6           int a[10] = { 0, 1 };
(gdb) p a[7]
$10 = 0
```

- Once a breakpoint is reached, execution of the program can be continued in several ways.
- **s**tep [n] command executes the next n lines of the program and stop.

```
(gdb) step
7           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)
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.
- **<Return> *without a command repeats the last command.***
- If the next line is a routine call, control enters the routine and stops at the first line.

- **n**ext [n] command, like step, but routine calls are treated as a single statement, so control stops at the statement after the routine call instead of

the first statement of the called routine.

```
(gdb) run
...
Breakpoint 1, main () at test.cc:6
6           int a[10] = { 0, 1 };
(gdb) next
7           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
```

- **c**ontinue command continues execution until the next breakpoint is reached.

```
(gdb) run
...
Breakpoint 1, main () at test.cc:6
6           int a[10] = { 0, 1 };
(gdb) s
7           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) p i
$4 = 100000000
(gdb) set var i = 3
(gdb) c
Continuing.
Program exited normally.
```

- **fin**ish command finishes execution of the current routine and stops at the statement after the routine call.

```
(gdb) run
...
Breakpoint 1, main () at test.cc:6
6           int a[10] = { 0, 1 };
(gdb) c
Continuing.
Breakpoint 2, r (a=0xffbefa20) at test.cc:3
3           a[i] += 1;      // really bad subscript error
(gdb) set var i = 3
(gdb) fin
Run till exit from #0  r (a=0xffbefa20) at test.cc:3
main () at test.cc:8
8     }
(gdb) c
Continuing.
Program exited normally.
```

  - Print the value returned by the finished routine, if any.

- During debugging, it is useful to print variables each time the program stops at a breakpoint.
- Normally, requires typing a print commands each time the program stop.
- **displ**ay command is like the print command, with the addition of printing

each time the program stops.

```
(gdb) run
...
Breakpoint 1, main () at test.cc:6
6           int a[10] = { 0, 1 };
(gdb) display a[0]
1: a[0] = 67568
(gdb) s
7           r( a );
1: a[0] = 0
(gdb) s
r (a=0xffbefa20) at test.cc:2
2           int i = 100000000;
```

- Each displayed variable is numbered, in this case, a is numbered 1.
- Use number to stop displaying a variable via undisplay *n* command.
- If a variable goes out of scope, the display stops printing.

- **l**ist command lists source code.

```
(gdb) list
2           int i = 100000000;
3           a[i] += 1;
4       }
5       int main() {
6           int a[10];
7           r( a );
8       }
(gdb) list 3
1       void r( int a[] ) {
2           int i = 100000000;
3           a[i] += 1;
4       }
5       int main() {
6           int a[10];
7           r( a );
8       }
```

- 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:6
6           int a[10] = { 0, 1 };
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

3.4 Compiling Complex Programs

- Separate compilation has an advantage and disadvantage.
- Advantage: saves significant amounts of computer and people time by recompiling only the portions of a program that are changed.
- In theory, if an expression is changed, only that expression needs to be recompiled.
- In practice, the unit of compilation is much 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 (and doesn't work).
- So a TU should not be too big and not be too small.

- Disadvantage: TUs must depend on each other because a program shares many forms of information, especially types.
- Not a problem when all the code is in a single TU (except for DBU).
- As a program grows, the number of TUs grow, so does the dependencies among TUs.
- Now, when one TU is changed, it may require other TUs to change that depend on some or all of the shared information.
- ***For a large numbers of TUs, the dependencies turn into a nightmare with respect to recompiled.***

3.4.1 Dependences

- Dependences in C/C++ normally occur as follows:
  - executable depends on .o files
  - .o files depend on .C files
  - .C files depend on .h files

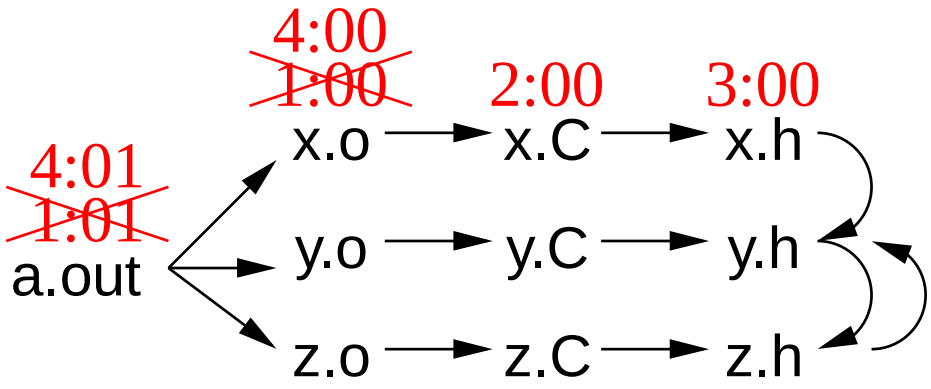
source tree

```
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"
```

dependencies



- The hierarchical **source tree** is compiled as follows:

```
% 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
```

- If a change is made to y.h, which files need to be recompiled? (all!)
- Does ***any*** change to y.h require these recompilations?
- There is no mechanism to know the kind of change made within a file, e.g., changing a comment, type, variable.
- So dependence is coarse grain, based on ***any*** change to a file.

- One way to denote file changes is with **time stamps**.
- UNIX stores in the directory the time a file is last changed, with second precision.
- Establishing dependencies means establishing a temporal ordering in the dependence graph so the root has the newest (or equal) time and the leafs the oldest (or equal) time.

3.4.2 Make

- **make** is a UNIX 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 express a relationship among sources, one that exists at the source-code level and is important.
- 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 is specified.
- Each target has a list of dependencies, and possibly a set of commands specifying how to re-establish the target.

```
variable = value
target : dependency1 dependency2 ...
        command1
        command2
        ...
```

- ***Commands must be indented by one tab character.***
- make is invoked with a target, which is a subnode or root of a dependence hierarchy.

- make builds the dependency graph and decorates the edges with time stamps for the specified files.
- If any of the dependency files (leafs) are newer than the target file (root), or if the target file does not exist, the commands are executed by the shell to update the target (generate 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 by:

```
% gmake -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
```

- -n only checks the dependencies and shows rules to be triggered (leave off to trigger 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)

- 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:

```
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

clean :
    rm -rf ${OBJECTS} ${EXEC}
```

- gmake *knows* how to construct simple rules when files have specific suffixes and when special variable names are used.
- These rules use variables ${CXX} and ${CXXFLAGS}.
- Target clean removes product files that can be rebuilt to save space.

```
gmake clean
```

- Eliminate 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}

clean :                                 # remove files that can be regenerated
    rm -rf ${DEPENDS} ${OBJECTS} ${EXEC}

-include ${DEPENDS}                     # copies files x.d, y.d, z.d (if exists)
```

- g++ flag -MMD generates a dependency graph for only user source-files.

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.

3.5 Source Code Management

- UNIX files only support the ***current*** version of the program.
- As a program develops/matures, it changes in many ways.
- UNIX files do not support this temporal notion of a program, i.e., history of program over time.
- Access to older versions of a program, supporting operations like backing out of changes because of design problems.
- Another issue is sharing program files among multiple developers each making independent changes.
- Current sharing allows damaging the contents of the files for simultaneous writes.
- Approaches:
  - Make copies of some or all of the project files before making changes.
    Wastes storage for unchanged files and burden of managing copied files.
  - Share files using group file permissions.
    Simultaneous access is unsafe and developers cannot test changes in isolation.

- Giving each developer a separate copy of the code base.
  Merging in changes from different developers is tricky and time consuming.

- To solve these problems, a **source control system** is used to manage cooperative work.

3.5.1 CVS

- **Concurrent Versions System** (CVS) is a source control system with the following features:
  - Master copy of all project files is kept in a **repository**.
  - Multiple versions of files are automatically stored in the repository.
  - Developers can check out a complete copy of the project.
  - Helpful integration back into the repository using text merging.
    Programmer has to deal with conflicts.

3.5.2 Repository

- Group members must add this line to their shell startup file:

sh:

```
% CVSROOT=/u/userid/cs246/cvsroot
% export CVSROOT
```

csh:

```
% setenv CVSROOT /u/userid/cs246/cvsroot
```

- For remote access:

```
CVS_RSH=ssh
export CVS_RSH
CVSROOT=userid@student.cs.uwaterloo.ca:/u/userid/cs246/cvsroot
export CVSROOT
```

- Shared repository is created at accessible location in the file system:

```
% cd cs246
% cvs init                          # make repository directory cvsroot
% chgrp -R cs246_75 cvsroot # set group on directory and subfiles
% chmod -R g+rwx cvsroot    # allow group members access to ALL files
% mkdir cvsroot/assn6               # specific project
```

- cvs int creates and initializes the repository.
- Other directories under cvsroot represent projects (can have any name).

3.5.3 Checking Out

- checkout command creates a working copy of the project:

```
% cvs checkout assn6          # checkout initial project
cvs checkout: Updating assn6
% cd assn6                    # move into project directory
% ls                          # administration directory CVS
CVS
```

- Creates project directory in current directory and under cvsroot.
- A checked out copy can be modify in any way without other developers seeing these changes until committed.
- Only check out once and continue working.

3.5.4 Adding

- add command schedules new files (in current directory) for addition into the repository:

```
% ... create files: Makefile  x.C  x.h  y.C  y.h  z.h z.C
% ls
CVS  Makefile  x.C  x.h  y.C  y.h  z.C  z.h
% cvs add *                          # add all files
cvs add: cannot add special file 'CVS'; skipping
cvs add: scheduling file 'Makefile' for addition
cvs add: scheduling file 'x.C' for addition
cvs add: scheduling file 'x.h' for addition
cvs add: scheduling file 'y.C' for addition
cvs add: scheduling file 'y.h' for addition
cvs add: scheduling file 'z.C' for addition
cvs add: scheduling file 'z.h' for addition
cvs add: use `cvs commit' to add these files permanently
```

- Addition only occurs on cvs commit.
- Forgetting cvs add is a common mistake.
- ***Do not put all files into repository, e.g., *.o, *.d, a.out.***

3.5.5 Checking In

- commit updates the repository with the changes made in checkout directory.

```
% cvs commit -m "initial files"
cvs commit: Examining .
RCS file: /u/userid/cs246/cvsroot/assn6/Makefile,v
done
Checking in Makefile;
/u/userid/cs246/cvsroot/assn6/Makefile,v  <--  Makefile
initial revision: 1.1
done
RCS file: /u/userid/cs246/cvsroot/assn6/x.C,v
done
Checking in x.C;
/u/userid/cs246/cvsroot/assn6/x.C,v  <--  x.C
initial revision: 1.1
done
RCS file: /u/userid/cs246/cvsroot/assn6/x.h,v
done
Checking in x.h;
/u/userid/cs246/cvsroot/assn6/x.h,v  <--  x.h
initial revision: 1.1
done
...
```

- If -m flag not used, cvs prompts for a change description using an editor.

- ***Always make sure that your code compiles and runs before committing.***
- It is unfair to pollute the source base with bugs.

3.5.6 Editting/Removal

- Edited files (in current directory) do not require any CVS command:

```
% vi y.h y.C                    # edit files y.h y.C
```

- Implicitly schedules files for update, which occurs on cvs commit.
- remove command tell CVS to remove existing files from the repository:

```
% rm z.h z.C                    # remove files z.h z.C
% cvs remove z.h z.C            # remove from repository
cvs remove: scheduling ‘z.h’ for removal
cvs remove: scheduling ‘z.C’ for removal
cvs remove: use ´cvs commit´ to remove these files permanently
```

- Schedules files for removal, which occurs on cvs commit.
- In fact, any removed file can always be retrieved from old versions.
- Commit edits and removals.

```
% cvs commit -m "changes to y.* and remove z.*"
cvs commit: Examining .
Checking in y.C;
/u/userid/cs246/cvsroot/assn6/y.C,v  <--  y.C
new revision: 1.2; previous revision: 1.1
done
Checking in y.h;
/u/userid/cs246/cvsroot/assn6/y.h,v  <--  y.h
new revision: 1.2; previous revision: 1.1
done
Removing z.C;
/u/userid/cs246/cvsroot/assn6/z.C,v  <--  z.C
new revision: delete; previous revision: 1.1
done
Removing z.h;
/u/userid/cs246/cvsroot/assn6/z.h,v  <--  z.h
new revision: delete; previous revision: 1.1
done
```

3.5.7 Update

- Cannot commit changes if other developers have checked in changes during a checkout.

- Changes must now be merged and then committed.
- update command merges changes into repository.
- Causes merged file in current directory to be updated.
- Merge algorithm is generally very good if changes do not overlap.
- Overlapping changes result in a conflict, which must be resolved manually.

```
% cvs commit
cvs commit: Examining .
cvs commit: Up-to-date check failed for ‘Makefile’
cvs [commit aborted]: correct above errors first!
% cvs update
cvs update: Updating .
RCS file: /u/userid/cvsroot/assn6/Makefile,v
retrieving revision 1.2
retrieving revision 1.3
Merging differences between 1.2 and 1.3 into Makefile
```

- Conflict is marked in Makefile:

```
CXX = g++                              # variables and initialization
<<<<<<< Makefile
CXXFLAGS = -g -MMD
=======
CXXFLAGS = -g -Wall
>>>>>>> 1.3
```

- ***You have to resolve the conflict.***

3.5.8 Versions

- Each time a file is committed, it receives a new version number.
- Version number is displayed during commit, and at other times.
- `cvs status` prints version information.
- Old versions are accessible using:

```
cvs update -p -r 1.2 Makefile      # -p prints to standard output
```

  which prints version 1.2 of Makefile to standard output.
- Differences between versions can be generated:

```
cvs diff -r 1.2 -r 1.1 Makefile
```

  which shows the differences between version 1.2 and version 1.1.

3.5.9 Tagging

- Version numbers are nondescript and often too low level (i.e., little changes here and there).
- It is possible to give a meaningful, symbolic name to a version, often at a stable point or before big changes.
- tag command adds a symbolic name to the current version of every file checked out:

```
cvs tag debug1                # name current version "debug1"
```

- Use symbolic name like version number:

```
cvs update -p -r debug1
```

- To compare named versions:

```
cvs diff -r debug1 -r debug2
```

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.
- E.g.: 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.
- E.g., designing by anthropomorphizing the computer is seldom a good approach (desktops/graphical interfaces).

- Software tools spend 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 normally emerge late, over budget, and do not work well.
- Today, hardware costs are nil, and people costs are great.
- 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.

4.2 Software Development

- Techniques for program development for small, medium, and large systems.
- Objectives:
  - plan and schedule software projects
  - 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.

4.2.1 Programming 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.

4.2.2 System Design

- In designing a system of any size it must be modularized.
- **Modularization** is the division of the system into smaller parts on some systematic basis.
- Modularization is necessary to:
  - make it easier to design and implement

- make it easier to read
- make it easier to maintain and modify
- abstract the data structures
- abstract the algorithms

- 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.

4.2.3 Top-Down Design

- Start at highest level of abstraction and break down problem into cohesive units.
- Then refine each unit further generating more detail at each division.
- This recursive process is called **stepwise refinement**.
- Each subunit is divided until a level is reached where the parts are comprehensible, and can be coded directly.
- 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 analysis/design process.

4.2.4 Factoring

- **Factoring** is the modularization of code in one module into multiple modules.
- Stop factoring when:
  - cannot find a well defined function to factor out
  - interface to the module would be as complicated as the module itself
- Factoring is done to:
  - 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

- 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
  - avoid high levels of nesting (3-5 levels is fine)

4.3 System Modelling

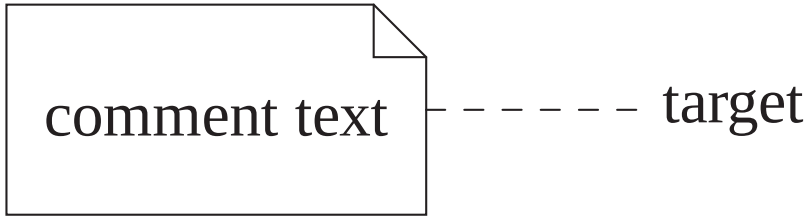
- **System modelling** involves modelling a complex system in an abstract way to provide a specific description of how the system works.

- Design grows from nothing to become a model of sufficient detail to be transformed into a functioning system.
- Design 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 all system design.
- System modelling has multiple viewpoints:
  - **class model** : describes static kinds and structure of system
  - **object model** : describes dynamic (temporal) behaviour of system objects
  - **interaction model** : describes the kinds of interactions among objects
- 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

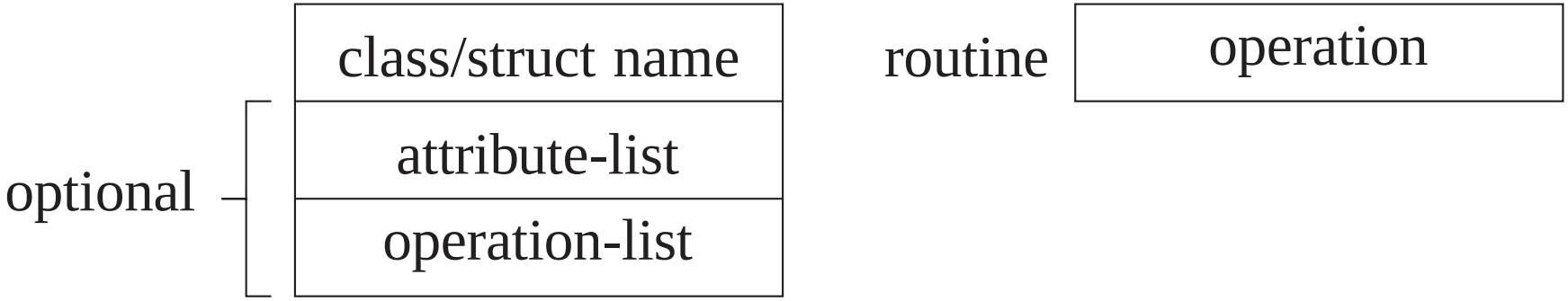
- Design tools can be used in various ways:
  - **sketch** out high-level design or complex parts of a system,
  - **blueprint** the entire system abstractly with high accuracy,
  - **generate** interfaces directly.
- Key advantage of design tool is the generic, abstract model of the system, which can be transformed into any format.
- Key disadvantage is the design tool is seldom linked to the implementation mechanism, so the two often differ
  (implementation = truth).
- As with design strategies, design tools have much in common and so only one is studied.

4.3.1 UML

- **Unified Modelling Language** (UML) is a graphical notation for describing and designing software systems, with emphasis on the object-oriented style.
- UML can handle class, object and interaction modelling. (focus on class modelling)
- Note/comment



- **Class diagram** collection of class templates and associated relationships.
- Class specifies a template for objects : name, attributes, operations.



- **attribute** : value description (field)

  [visibility] name [":" [type] ["[" multiplicity "]"]
  ["=" default] ["{" property-list "}"]]

  - visibility : access of attribute information by other classes
    $+ \Rightarrow$ public, $- \Rightarrow$ private, # $\Rightarrow$ protected, $\sim \Rightarrow$ package
  - name : required identifier for attribute (like field name in structure)

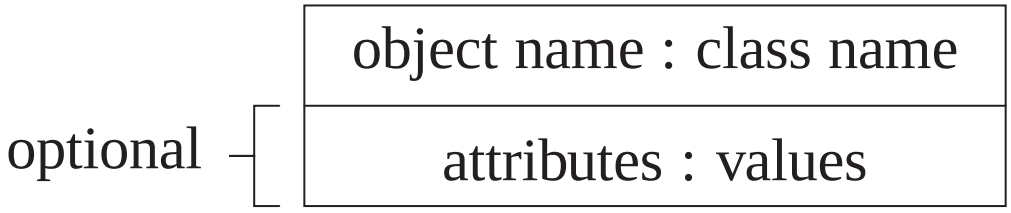
- type : restriction on kind of objects associated with attribute
  Boolean, Integer, Float, String, class-name
- multiplicity : restriction on number of objects associated with attribute
  $0..(N|*)$, from 0 to N or unlimited, N short for $N..N$, $*$ short for $0..*$
- default : value of newly created object
- property : additional aspects of attribute, e.g., { readonly }

• **operation** : action changing or returning object state (method)
[visibility] name ["(" [parameter-list] ")"] [":" return-type]
["[" multiplicity "]"] ["{" property-list "}"]

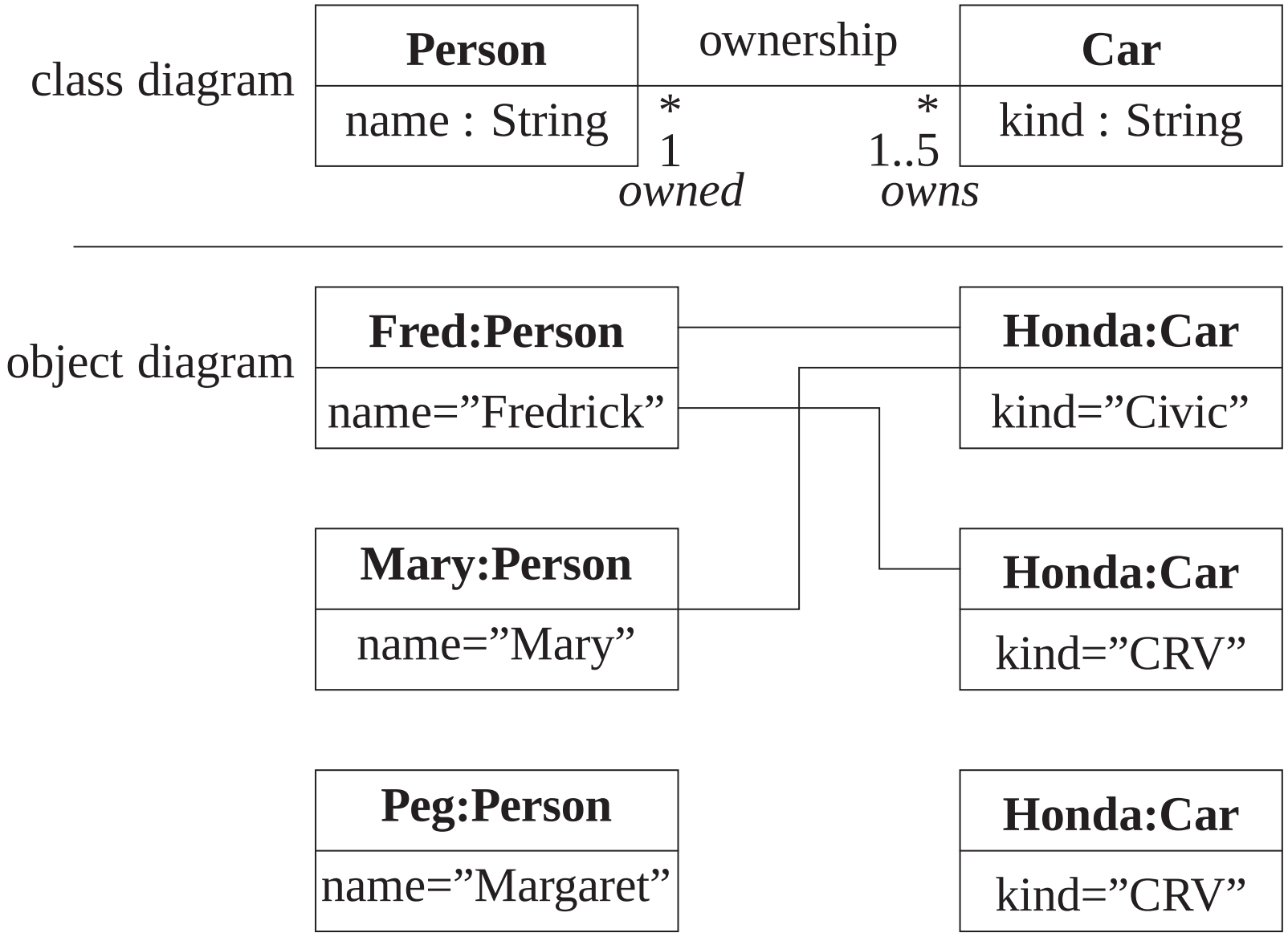
- visibility : access of attribute information by other classes
  $+ \Rightarrow$ public, $- \Rightarrow$ private, $\# \Rightarrow$ protected, $\sim \Rightarrow$ package
- name : required identifier for operation (like method name in structure)
- parameter-list : input/output values for operation
  [direction] parameter-name ":" type ["[" multiplicity "]"]
  ["=" default] ["{" property-list "}"]]
- direction : direction of parameter data flow
  "in" (default) | "out" | "inout"
- return-type : output from operation
- property-list : additional aspects of operation, e.g., { readonly }

	VendingMachine
attributes	- printer : Printer - nameServer : NameServer - Id : Integer - sodaCost : Integer - maxStockPerFlavour : Integer - stock : Integer [1..4]
operations	+ buy(in flavour : Flavours, inout card : WATCard) : Boolean + inventory : Integer [1..4] + restocked + cost : Integer + getId : Integer

- Include attributes defining model structure (no counters, temporaries, etc.)
- Leave out constructor operations as they do not contribute to the model.
- **Object diagram** : instance of a class.



- **Association** : a named conceptual/physical connection among objects.



- association is “ownership”

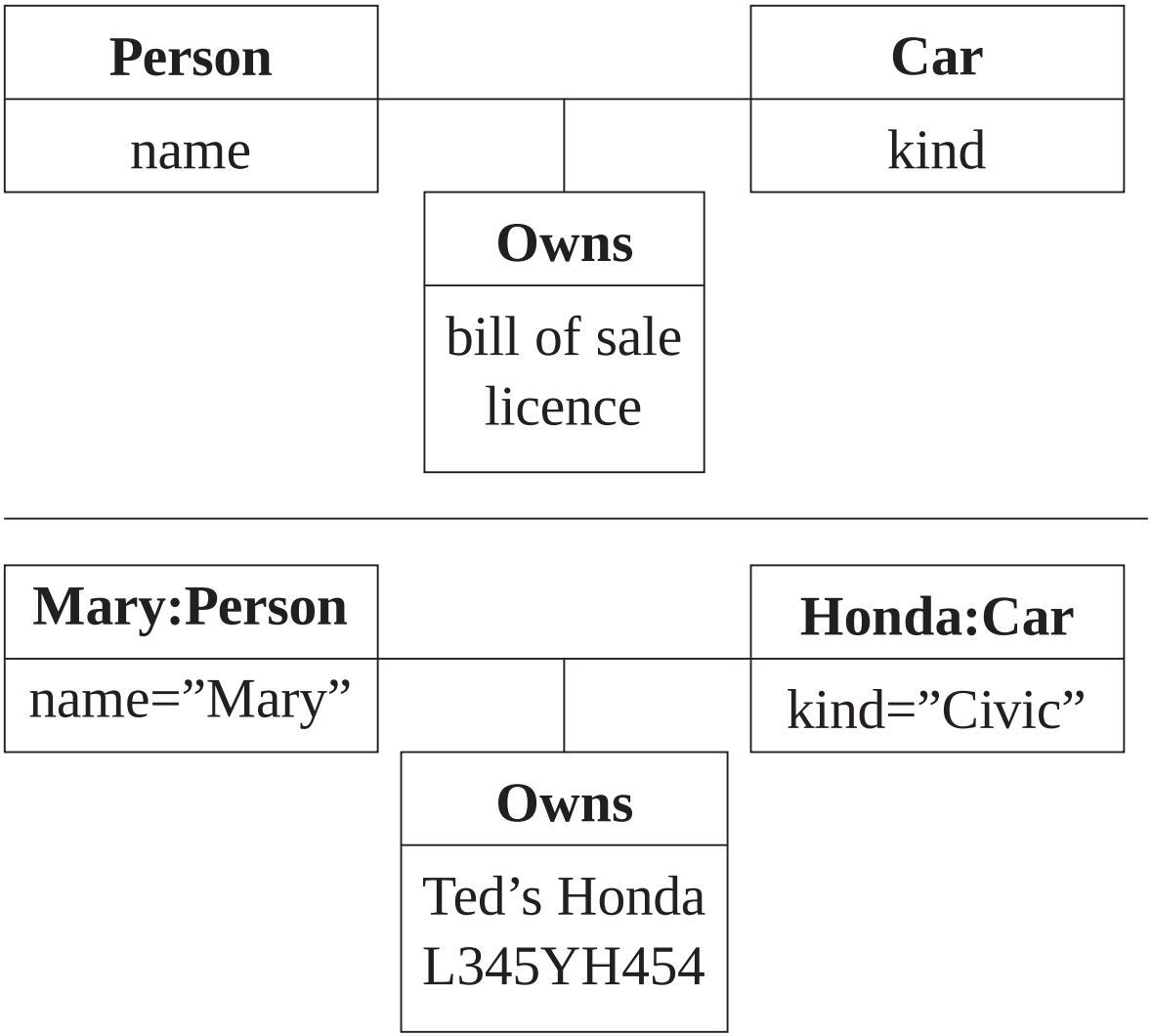
- person *owns* 0 or more cars (*)
  person *owns* 1 to 5 cars
- car is *owned* by 0 or more people (*)
  car is *owned* by 1 person

- Association is inherently bidirectional even if name implies a specific direction: employer | worksFor | employee
- Association can be represented as an attribute or a line.

Use attribute if many lines to a single class.

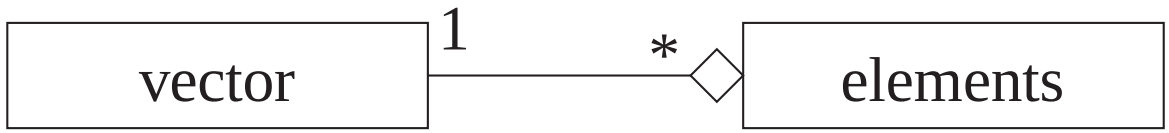
- Association may be implemented in a number of ways:
  - pointer from one object to another
  - related elements in arrays

- **Association Class** : association that is also a class



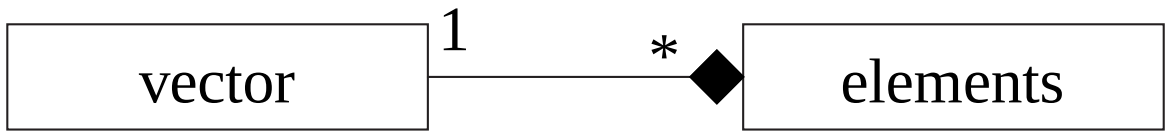
  - people without cars do not need “owns” fields
    cars without owners do not need “owns” fields
  - not real class because it cannot logically exist without association

- **Aggregation** is an association between an aggregate (collection) and its members.



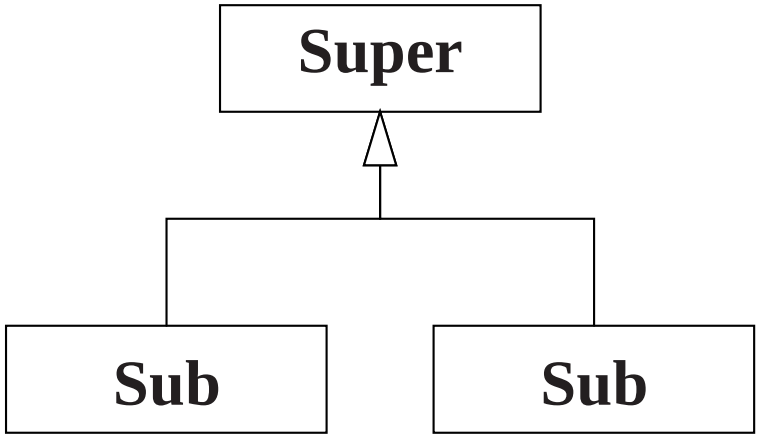
  - an aggregate is not complete without its members
  - but members exist outside of the aggregate (pointer to elements)

- **Composition** is stronger aggregation where components do not exit outside of composite.

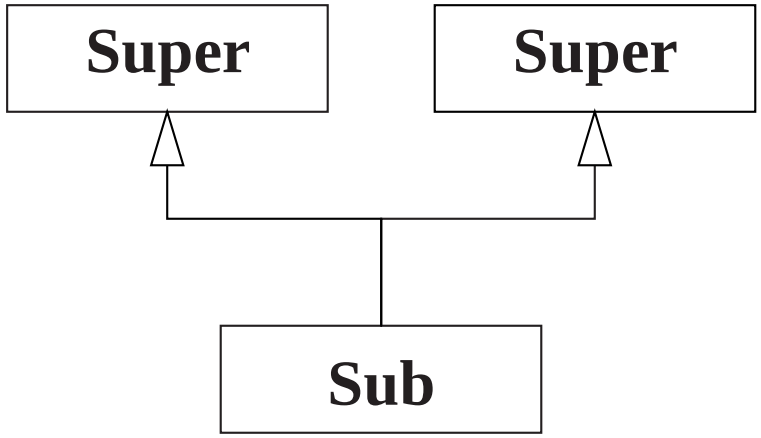


  - copy elements

- **Generalization** : reuse through form of inheritance.



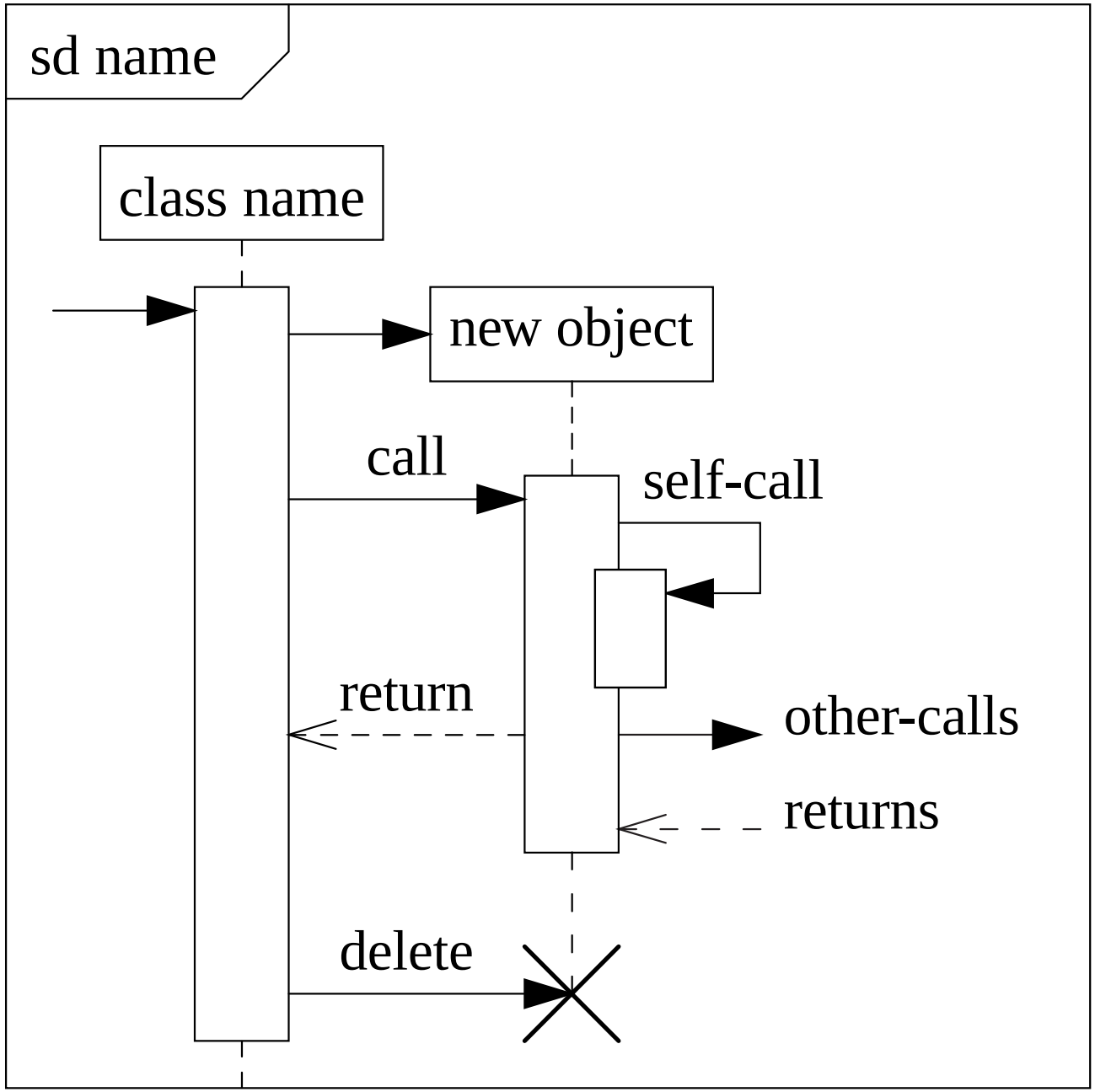
Inheritance



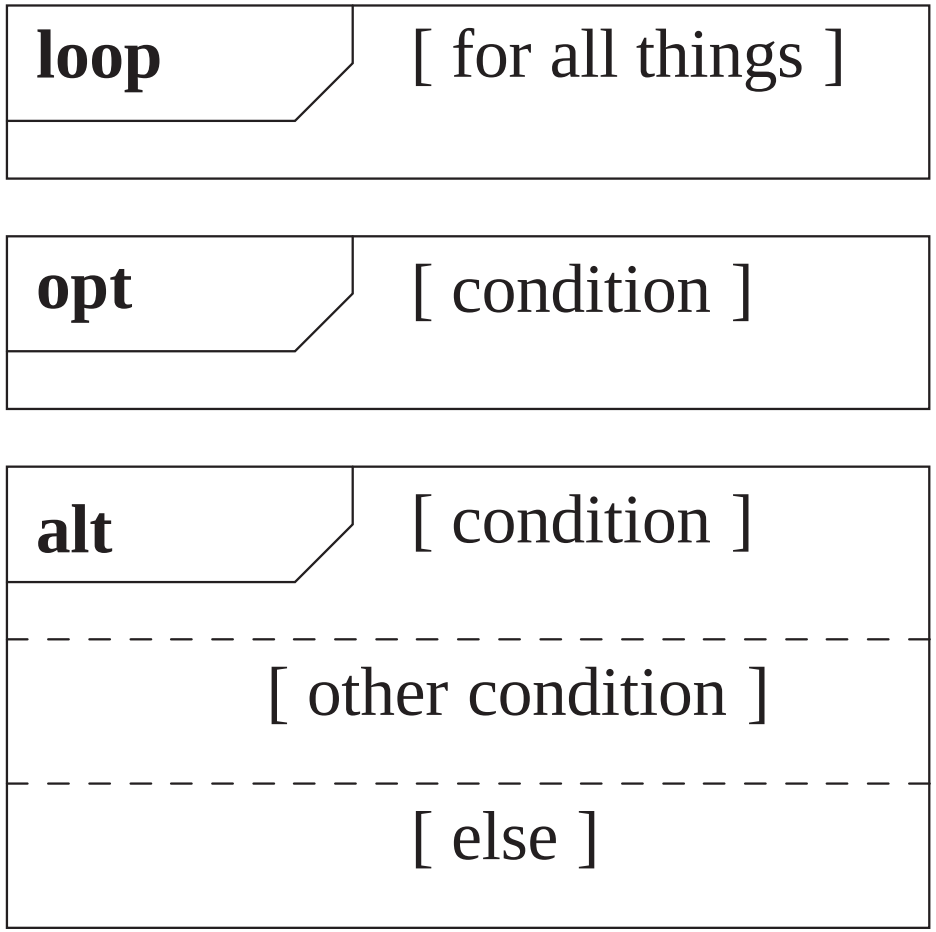
multiple inheritance

- Inheritance establishes “is-a” relationship on type, and reuse of attributes and operations.
- Association class can be implemented with forms of multiple inheritance (mixin).

- **Sequence diagram** : describes control-flow among objects with respect to particular scenario.
  - show static frame of program animation (call sequence).


sd name
class name
new object
call
self-call
return
other-calls
returns
delete

- show control flow



- complex and specific
- more concise to use pseudo-code (or actual code if it exists)
- use to show important/complex control flow sequences

- UML is significantly more general, supporting very complex descriptions of relationships among entities.

- VERY large visual mechanisms, with several confusing graphical representations.
- **Code = truth**

4.4 Programming Language Selection

- imperative, functional, logic
  - imperative : prescribes a sequence of actions directed by the state of variables, which are allowed to have multiple values (i.e., vary)
  - functional : like imperative, but variables are restricted to only one value (i.e., constant)
  - logic : series of logical expressions that are proven correct or incorrect through unification
- scripting : specialized languages (often only string type or dynamically typed) for specific purpose (shell, GUI, awk, Perl)
- interactive/interpreted : not compiled, can be typed and executed immediately (basic, shell)
- managed language : hide aspects of implementation to simplify programming, e.g., hide memory management via garbage collection,

execution via virtual machine

- static/dynamic type-system : variable types are fixed at compile time or allowed to vary at runtime.
- reification : manipulate program symbol-table and code at runtime, possibly with dynamic compilation.
- Useful language properties for SE:
  - abstraction/encapsulation : separate implementation from interface, and hide implementation
  - module/package : high-level bundling of types/variables/code with global initialization, e.g., container library
    - requires transitive closure of modules over program for initialization (cycles?)
  - class : aggregate data and code into single type
  - coroutines : retain control flow knowledge across routine call
  - concurrency : multiple simultaneous threads of execution (inherently difficult and complex)
  - polymorphism : generalization data/code across multiple types with similar structure and behaviour
  - libraries : error-free, efficient, reusable abstractions:

    - data structures, math, GUI, distributed/web
  - compilation/runtime errors : specific, comprehensible error messages
  - efficiency : after it works, after its good code, then make sure it is efficient
    - efficiency should never be an afterthought; it comes from good programming practice
    - nevertheless, programs have execution hot-spots that require extra attention
  - security : subscript checking, type checking, virtual machine, dynamic checking, etc.
- Java : imperative, managed, static typing (inconsistent builtin & object types), reification, abstraction/encapsulation, packages, class (strongly object-oriented), concurrency, medium polymorphism, large libraries, good error reporting, average to poor efficiency
- C++: imperative, not managed, static typing (consistent builtin & object types), abstraction/encapsulation, weak packages, class, routines, no concurrency, strong polymorphism, average libraries, poor error reporting, average to excellent efficiency
- Ada : imperative, many good features, but not used much anymore

- Cobol, Fortran, PL/I : legacy languages, updated but slowly disappearing
- Python/Ruby : scripting
- Haskell, Scheme, Erlang (Industrial) : functional

4.5 Development Processes

- There are different conceptual approaches for developing software, e.g.:

  waterfall : break down project based on activity and divide activities across a timeline
  - activities : (cycle of) requirements, analysis, design, coding, testing, debugging
  - 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 dynamically
- 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
- 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

4.6 Design Patterns

- **Design patterns** have existed since people/trades developed formal approaches.
- E.g., parent's raising children, 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, and their relationships, responsibilities, and collaborations
  - consequences : results and trade-offs of applying the pattern (alternative/implementation issues)

4.6.1 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
- Purpose : class/object creation issues, structural form, and behavioural interaction

- Class

  factory method/abstract : abstract class/template defining structure (and possibly some implementation) for creating other classes

```
struct F {                      // factory/abstract-class
    virtual void m1() = 0;
    virtual void m2() = 0;
};
struct P1 : public F {          // products
    void m1();
    void m2();
};
struct P2 : public F {
    void m1();
    void m2();
};
```

adapter/wrapper : convert interface into another

```
struct T1 {                          struct T2 {
    virtual void x(...);                 virtual void x(...);
    virtual void y(...);                 virtual void z(...);
};                                   };
struct T1T2 : public T1, private T2 {   // adapter/wrapper
    void x(...) { T2::x(...); }
    void y(...) { ... z(...); ... }
};
void p( T1 t1 ) { ... }
T1T2 t; // make use of T2 code with T1 routine
p( t );
```

template method : provide pre/post actions for subclass methods

```
class TM {
    virtual void doAction() = 0;
  protected:
    virtual void action() {
        pre-code    doAction();    post-code
    }
};
class AM : public TM {
    void doAction() {. . .}
  public:
    void action() { TM::action(); }
};
```

- Object

  adapter : convert interface into another

```
struct T1 {                          struct T2 {
    virtual void x(...);                 virtual void x(...);
    virtual void y(...);                 virtual void z(...);
};                                   };
struct T2toT1 : public T1 {          // adapter/wrapper
    T2 &t2;
    T2toT1( T2 &t2 ) : t2( t2 ) {}
    void x(...) { t2.x(...); }
    void y(...) { ... t2.z(...); ... }
};
void p( T1 t1 ) { ... }
T2 t2;
T2toT1 t( t2 );                      // any T2
p( t );
```

iterator : abstract mechanism to traverse container

```
list<Node>::iterator ni;
for ( ni = top.begin(); ni != top.end(); ++ni ) { // traverse list
    cout << "c:" << ni->c << " i:" << ni->i << endl;
}
```

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
```

proxy : frontend for another object to control access

```
struct T {
    void m1(...);
    void m2(...);
};
struct SProxyT : public T {                    // static
    void m1(...) { ... T::m1(...); ... }
    void m2(...) { ... T::m2(...); ... }
};
struct DProxyT : public T {                    // dynamic
    T *t;
    DProxyT() { t = NULL; }
    ~DProxyT() { if ( t != NULL ) delete t; }
    void m1(...) { if ( t == NULL ) t = new T; t->m1(...); ... }
    void m2(...) { ... don't need t ... }
};
```

decorator : attach additional responsibilities to an object dynamically

```
struct Abstract {
    virtual void m1(...) = 0;
    virtual void m2(...) = 0;
};
struct Concrete : public Abstract {
    void m1(...);
    void m2(...);
};
struct Decorator : public Abstract {        // generalize
    Abstract *parent;
    Decorator( Abstract &parent ) : parent( &parent ) {}
    void m1(...) { parent->m1(...); }       // forward
    void m2(...) { parent->m1(...); }       // forward
};
struct Decoratee1 : public Decorator {      // specialize
    ...
    Decoratee1( Abstract &parent, ... ) : Decorator( parent ), ... {}
    void m1(...) { decorate  Decorator::m1(...);  decorate }
    void m2(...) { decorate  Decorator::m2(...);  decorate }
};
struct Decoratee2 : public Decorator {...}  // specialize

Concrete c;
Decoratee1 d1( c );  Decoratee2 d2( c ); // decorate c two ways
d1.m1(...);  d2.m1(...);
```

observer : 1 to many dependency ⇒ change updates dependencies

```
struct Observee {                       // generalize
    Observer &oer;
    Observee( Observer &oer ) : oer( oer ) {}
    virtual void update() = 0;
};
struct Observer {
    list<Observee *> oees;              // list of observees
    static void perform( Observee *oee ) { oee->update(); }
    void attach( Observee &oee ) { oees.push_back( &oee ); }
    void deattach( Observee &oee ) { oees.remove( &oee ); }
    void notify() { for_each( oees.begin(), oees.end(), perform ); }
};
struct Oee : private Observee {  // specialize
    Oee( Observer &oer ) : Observee( oer ) { oer.attach( *this ); }
    ~Oee() { oer.deattach( *this ); }
    void update() { perform update action }
};
Observer oer;
Oee oee1( oer ), oee2( oer );       // register
oer.notify();                       // trigger updates
```

visitor : perform operation on elements of heterogeneous container

```
struct Visitor {
    void visit( N1 &n ) { perform action on node }
    void visit( N2 &n ) { perform action on node }
};
struct Node {
    virtual void action( Visitor &v ) = 0;
};
struct N1 : public Node {
    void action( Visitor &v ) { v.visit( *this ); } // overload
};
struct N2 : public Node {
    void action( Visitor &v ) { v.visit( *this ); } // overload
};
Visitor v;
list<Node *> l;
for ( int i = 0; i < 10; i += 1 ) {
    l.push_back( i % 2 == 0 ? (Node *)new N1 : (Node *)new N2 );
}
for ( list<Node *>::iterator it = l.begin(); it != l.end(); ++it ) {
    (*it)->action( v );
}
```

4.7 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.

4.7.1 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** looks for common problems:

  - 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

- **Desk checking** : single person "plays computer", executing program by hand.
- **Walkthrough** : team of people examine program by hand, often "grilling" the developer.

4.7.2 Machine Testing

- **Machine Testing** : systematic running of program using test data, which is designed to discover problems.
- 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.
- Start with the black-box approach and supplement with white-box tests.
- Black-Box Testing
  - **equivalence partitioning**
    - 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)
44, 45, 46     "
-1,  0,  1     "
```

  - **cause-effect graphing**
    - used to generate test cases representing combinations of conditions
    - construct boolean logic-graphs, which are converted to decision tables (describing test inputs and expected outputs)
  - **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

4.7.3 Testing Mechanics

- **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 large volumes of test data, possibly over long period of time.
- **Stress testing** : test if program handles extreme volumes of data over a short period of time, 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.

- If a problem is discovered, make up additional test cases to zero in on this particular issue.

4.7.4 Tester

- A program should not be tested by its writer, but in practice this often occurs.

- Testing can be very hard on the ego because you have to search out your own faults.
- Remember, the tester only tests what they thinks it should do.
- Any misunderstandings the writer had while coding the program are carried over into testing.
- Any system written for an end user must be tested by the end user to determine if it is acceptable.
- **Acceptance testing** : checking if the system satisfies what the user ordered.
- Points to the need for a written specification to protect both the end user and the supplier.

5 Conclusion

- Final exam is (largely) based on sections 2.12 (Objects) to end.
- Last 2 final exams and answers are available (see course web-site, seating)
- Last version of course note is up.
  - Send me any corrections you find during studying.
- assignment 6 extension : Sunday, Dec 6 @ 23:55
- Course topics:
  - 2 programming language : sh and C++, dangerous in both ;-)
  - Tools : compiler, debugger (maybe), make, CVS (maybe) : valgrind (memory errors)
  - SE : for the work place
- review (Erik), newsgroup, appointment
- Think like a computer to understand it and write good programs.
- Fran Allen's talk today at 2:00 in DC1302
  High Performance Computers and Compilers: A Personal Perspective
- Good luck on assignment 6 and the final exam.
  I want you all to succeed!